Clean Coding Cosmos: Teil 3: Kosmische Effizienz durch Team Clean Coding

Was bedeutet eigentlich clean? Unsere Antwort darauf lautet: effizient. Als clean wurde ursprünglich nur der Quellcode bezeichnet, der Begriff wird heutzutage aber viel breiter verstanden. Wir dehnen ihn auf den gesamten Entwicklerkosmos aus. Wir glauben weiter, dass ein Appell an die Professionalität bei den meisten Entwicklern nicht ausreicht, sondern dass es noch ein stärkeres Mittel gibt, um mehr Sauberkeit im Entwickler-Kosmos zu erreichen: nämlich Teamplay. Deshalb wollen wir in diesem Artikel die „Team-Clean-Coding"-Methode vorstellen und zeigen, mit welchen konkreten Maßnahmen diese Methode im Projektalltag eingeführt und so die Clean-Coding-Mentalität viel lebendiger werden kann.

Professionalität ist die Fähigkeit, durch spezielle Handlungen ein konkretes Ziel mit minimalem Aufwand zu erreichen. Professionell bedeutet also, etwas effizient d. h. *clean* zu tun (**siehe Kasten 1**). Dementsprechend haben die *Clean Code Developer (CCD)* – eine Initiative, die 2009 von Ralf Westphal und Stefan Lieser gegründet wurde (vgl. [CCD]) – eine Definition dieses Begriffs aufgestellt, der für effiziente *Softwareentwicklung (SE)* wichtig ist: Professionalität = Bewusstsein + Prinzipien. Unter „Bewusstsein" verstehen die CCD die ständige Selbstkontrolle eines Entwicklers, das Wertesystem auch unter widrigen Umständen konsequent anzuwenden. Damit appellieren die CCD (und wir auch) an den Berufsethos und den Ehrgeiz von Entwicklern, ein Profi zu sein oder einer zu werden. Dieser Appell richtet sich an die menschliche Vernunft eines jeden einzelnen Entwicklers.

Warum Team Clean Coding (TCC)?

Der Philosoph Immanuel Kant vertrat schon vor über 200 Jahren die Meinung, ein Mensch, der verstanden habe, was vernünftig ist, schöpfe aus dieser rationalen Einsicht die Motivation, das Vernünftige auch wirklich zu tun (vgl. [Wik-c]). Diese Philosophie hätte die Welt eigentlich verbessern müssen. Tatsächlich reicht die rationale Einsicht nur bei starken Idealisten als alleinige Motivationsquelle aus, vernünftig zu handeln. Eine zusätzliche, viel stärkere Motivationsquelle ist eine soziale Form von Kontrolle. Als sozialverträglich betrachten wir eine Kontrolle auf Augenhöhe in Form der TCC-Qualitätssicherung (**siehe Abbildung 1**). Sie wirkt folgenden Einstellungen entgegen:

- *„Aber Saubermachen kann ich das jetzt nicht mehr!"* Diese Haltung macht deutlich, dass der innere Schweinehund auf Dauer – vor allem unter Zeitdruck – stärker wird als die persönliche Überzeugung, dass Clean Coding wichtig ist.
- *„Nein, nein, so mach ich das aber nicht!"* Unterschiedliche Meinungen, die sich nicht annähern, stellen offene oder versteckte Konflikte dar, die Wildwuchs und damit Ineffizienz bewirken. TCC hilft, die Meinungsvielfalt zusammenzuführen.
- *„Meine Meinung zählt ja sowieso nicht!"* Die Durchsetzungsstarken im Team haben nicht automatisch die besseren Argumente. Durch TCC haben die Durchsetzungsschwachen ein Mittel, das ihre Position stärkt.
- *„Ist doch egal, wie mein Code aussieht, es interessiert sich ja doch keiner dafür!"* Gegen diese Einstellung hilft die

Clean ist alles, was intuitiv verständlich ist: (Quellcode-)Dokumente, Datenstrukturen, Konzepte, Regeln, Verfahren usw. Intuitiv verständlich ist, was mit wenig Spezialwissen in kurzer Zeit richtig verstanden wird. „Wenig" und „kurz" ist dabei relativ zur vorliegenden Komplexität zu sehen. Clean ist also alles, was die Softwareentwicklung beschleunigt. Das schließt die Korrektheit (vgl. [Wes12]), die Evolvierbarkeit (vgl. [Wes13-a]) und die Produktionseffizienz (vgl. [Wes13-b]) der Software ein.

Kasten 1: IT-Definition von „clean".

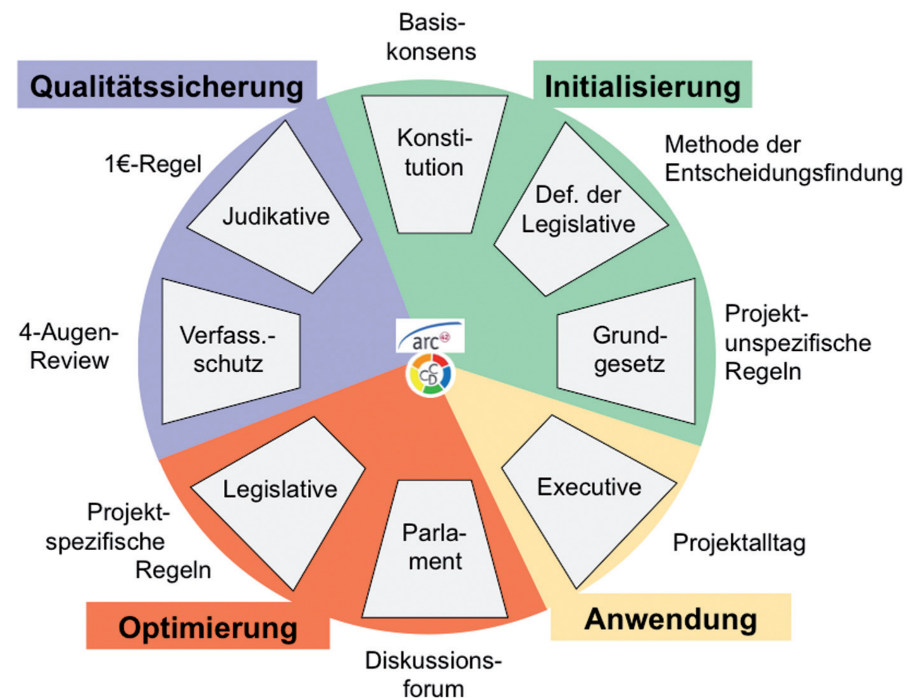

Abb. 1: Das TCC-Konzept (Kasten 3) besteht aus 4 Bereichen und acht Teilbereichen. Es hat das Ziel, Clean Code und Clean Architecting (Kasten 5) zu gewährleisten.

Clean Code ist ein alter Begriff, der den Blick auf den Quellcode fokussiert. Die CCD-Bewegung hat mit ihren Praktiken bereits deutlich gemacht, dass viel mehr als nur der Code sauber sein sollte. Um den Blick zu weiten und den ganzen Softwareentwicklungsprozess in den Fokus der Effizienzsteigerungen zu bringen, sprechen wir im *Clean Coding Cosmos (CCC)* vom *Clean Coding*.
Der CCC erweitert den Professionalitätsgedanken der CCD um weitere Aspekte des Applikations-Lebenszyklus-Managements (**siehe Abbildung 1** in [Obe13]). Diese Erweiterung schließt sauberes Requirements-Engineering, *Clean Architecting* (**siehe Kasten 4**), verantwortungsvolles Projektmanagement (siehe **Kasten 6** in [Obe14]) und effektive Zusammenarbeit mit dem Betrieb ein.
Als CCD unterscheiden wir Prinzipien, also Maßnahmen, die den Quellcode sauber halten, und Praktiken, d.h. eine bunte Sammlung von Maßnahmen, die ganz unterschiedliche Teile des Softwareentwicklungsprozesses betreffen. Im CCC verfolgen wir das ganzheitliche Ziel, alle Einflussgrößen auf die Software und den Softwareentwicklungsprozess nicht nur zu nennen, sondern sie so zu strukturieren, dass alle gleichzeitig besser im Blick behalten werden können. Zu diesem Zweck haben wir in [Obe13] die vierdimensionale Raumzeit des Softwareentwickler-Kosmos vorgestellt.
Im CCC betrachten wir nicht nur die schöne, saubere Seite der Softwareentwicklung, sondern auch die dunkle, schmutzige Seite. Das ist wichtig, um zu erkennen, in welchen Bereichen des Softwareentwicklungs-Kosmos am meisten Handlungsbedarf für Effizienzsteigerung besteht. Zu diesem Zweck haben wir in [Obe14] den kosmologischen Schmutz beleuchtet.
Der in [Obe14] gefundene Softwareentwicklungs-Schmutz macht deutlich, dass die Kommunikation, die in allen vier Dimensionen des CCC eine Rolle spielt, im Kontext von Effizienzsteigerung vernachlässigt wurde. Der Appell der CCD an die Professionalität ist gut, reicht aber oft nicht aus. Eine gegenseitige Qualitätskontrolle im Team ist dagegen viel wirkungsvoller. Diese funktioniert allerdings nur mit effektiver Kommunikation. Zu diesem Zweck gibt es im TCC zusätzliche Regeln, die das Teamplay betreffen. Wie das clean aussehen kann, stellen wir in diesem Artikel vor.

Kasten 2: Clean Code Developer (CCD) und Clean Coding Cosmos (CCC).

TCC-Kontrolle. Sie ist eine stärkere Motivation als das „Zuckerbrot und Peitsche"-Prinzip (vgl. [Obe13]) durch einen Code-Watch.

Das Team Clean Coding

TCC ist Teil des *Clean Coding Cosmos (CCC)*. Im CCC schließen wir uns der Philosophie der *Clean-Code-Developer (CCD)* an, gehen allerdings in einzelnen Punkten noch weiter (**siehe Kasten 2**).
Die „Ritter der Tafelrunde" verband das gemeinsame Grundverständnis, dass sie erstens alle Gleiche unter Gleichen waren und sie sich zweitens einem gemeinsamen Verhaltenskodex unterordneten. Die gleiche Idee liegt dem TCC zu Grunde. Auf diesem Grundverständnis basiert das TCC-Konzept, das aus folgenden vier Bereichen besteht:

1. Initialisierung
2. Ausführung
3. Optimierung
4. Qualitätssicherung

Jeder dieser Bereiche beinhaltet Teilbereiche. Die insgesamt acht Teilbereiche des TCC-Konzepts lassen sich gut durch die Staatsmetapher veranschaulichen (**siehe Abbildung 1 und Kasten 3**).

Initialisierung

1.1 Alle Teammitglieder (TM) verpflichten sich, Regeln und Prozesse anzuerkennen und einzuhalten, die für alle gleichermaßen verbindlich sind. Diese Regeln und Prozesse können bei Bedarf jederzeit angepasst werden.
1.2 Als erstes wird eine Methode zur Entscheidungsfindung festgelegt: Wie fällt das Team verbindliche Entscheidungen?
1.3 Als nächstes werden zu folgenden grundsätzlichen Fragen verbindliche Entscheidungen getroffen: a) *Dokumentation:* Welche Informationen werden wie festgehalten, persistiert und für alle TMs zugänglich gemacht? b) *Definition of Clean (DoC):* Was bedeutet für uns „Clean Code" und „Clean Architecting"? c) *Definition of Done (DoD):* Was genau muss erfüllt sein, damit eine Programmieraufgabe als „fertig" anerkannt werden kann? d) *Code Ownership:* Gibt es so etwas wie persönlichen Code oder darf jedes TM jede beliebige Codestelle ändern?

Anwendung

2. Alle TM wenden alle festgelegten Regeln und Prozesse bei ihrer täglichen Arbeit konsequent an.

Optimierung

3.1 Die TM kommen regelmäßig zusammen und tauschen Informationen aus, ob die bisherigen Regeln und Prozesse gut funktionieren oder ob sie angepasst bzw. erweitert werden müssen. Dazu stehen Daily-Standups, Journal Coding Club und Retrospektive-Meetings zur Verfügung. Darüber hinaus können Informationen auch durch Pair-Programming und Coding Notes ausgetauscht werden.
3.2 Bei Bedarf verabschiedet das Team mit der unter „Initialisierung", Punkt 1.3, festgelegten Methode neue Regeln oder ändert bestehende.

Qualitätssicherung

4.1 Nach Abschluss einer Programmieraufgabe durch ein TM stellt ein anderes TM sicher, dass die vereinbarte DoD erfüllt ist (Vier-Augen-Review). Erst wenn für beide TM die DoD vollständig ist, gilt die Programmieraufgabe offiziell als fertig.
4.2 Das Team kann Regeln benennen, deren Verstoß als schwerwiegend deklariert wird. Ein Verstoß gegen solche Regeln darf im Konsens aller TM mit einer Verpflichtung zu einer 1-Euro-Spende verknüpft werden. Das Team bestimmt später gemeinschaftlich die Verwendung der Spendenkasse.

Kasten 3: Formelle Beschreibung des TCC (siehe auch Abbildung 1).

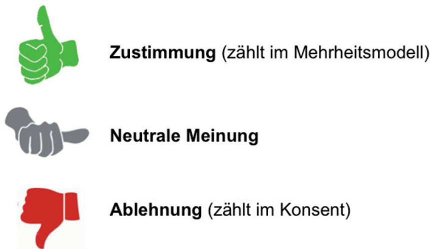

Abb. 2: Thumb Voting. Alle Teammitglieder werden um ein Handzeichen gebeten, bei dem diese drei Daumenpositionen zulässig sind.

Konstitution/Basiskonsens
Am Anfang steht die Einigung aller *Teammitglieder (TM)* auf den kleinsten gemeinsamen Nenner: den Basiskonsens (**Punkt 1.1 in Kasten 3**) – er ist das Fundament des TCC-Konzepts.

Definition der Legislative/Methode zur Entscheidungsfindung
Die (Meta-)Entscheidung, wie das Team zu einer Entscheidungsfindung kommt, ist für alle kommenden Entscheidungen von grundlegender Wichtigkeit. Als Werkzeug für die Entscheidungsfindung eignet sich das *Thumb Voting* (**siehe Abbildung 2**). Für die Meta-Entscheidung gibt es vier Wege:

> Sauberes Umsetzen einer Architekturaufgabe hat aus CCC-Sicht drei Aspekte:
>
> **1. Dokumentation**
> Die Beschreibung der Architektur beinhaltet alle relevanten Anforderungen, ist in sich schlüssig und für alle Stakeholder gut zugänglich und leicht verständlich.
>
> **2. Bewertung**
> Die gefällten Architekturentscheidungen ermöglichen, dass die wichtigen Anforderungen gut umgesetzt werden können und die Risiken minimiert werden.
>
> **3. Realisierung**
> Die beschriebene Architektur wird in der Implementierung umgesetzt und lässt sich im Code klar nachvollziehen. Um das zu erreichen, schlagen wir im TCC die Verwendung des „arc42 Templates“ vor (vgl. [Hru13-a], [Zör12]). Zu diesem Template gehört eine Methodik, die einen gewissen Prozess nahe legt (vgl. [Hru13-b]). Die Anwendung von Template und Prozess führt zu „Clean Architecting“.

Kasten 4: Clean Architecting.

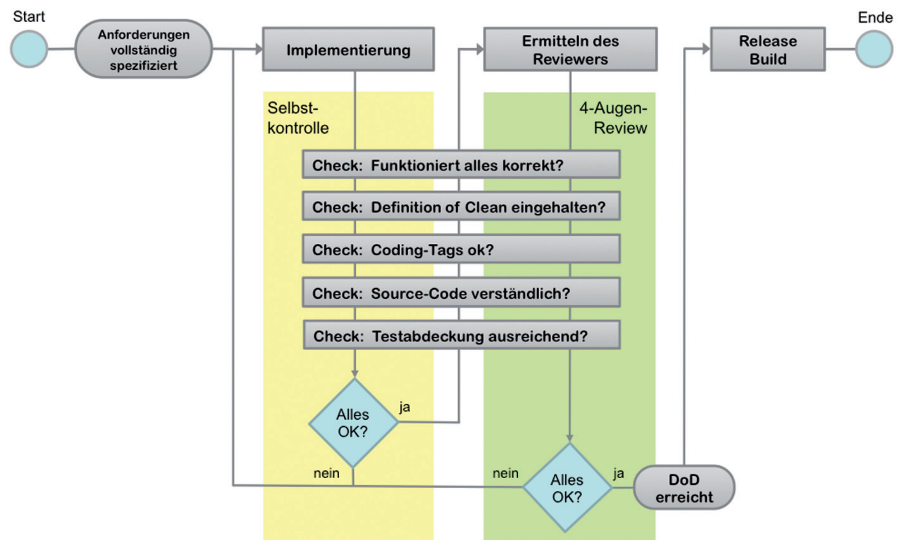

Abb. 3: Ablauf einer Programmieraufgabe im TCC.

- *Hierarchisches Modell:* Das Team überträgt einem Mitglied für einen bestimmten Bereich die Entscheidungshoheit, z.B. einem Mitglied mit viel Architekturerfahrung die Entscheidungshoheit über Architekturfragen.
- *Mehrheitsmodell:* Die Alternative mit der höchsten Zustimmung wird beschlossen.
- *Konsent:* Die Alternative mit der geringsten Ablehnung wird beschlossen (vgl. [Wik-a]).
- *Konsens:* Eine Alternative wird nur beschlossen, wenn sie keine Ablehnung erfährt (vgl. [Wik-b]). Sie muss eventuell so geändert werden, dass es keine Ablehnung mehr gibt. Bei einer Konsensrunde wird jeder Teilnehmer explizit befragt, welche Alternative ein (gegebenenfalls persönliches) Problem darstellt (vgl. [Wit11]).

Vorteil der ersten drei Möglichkeiten ist, dass Entscheidungen relativ schnell gefällt werden können. Ihr Nachteil besteht in der Gefahr, Teamkollegen zu zwingen, gegen ihre persönliche Überzeugung zu arbeiten. Das kann zu starken offenen oder versteckten Widersprüchen führen. Der Vorteil der vierten Möglichkeit ist genau die Vermeidung solcher Widersprüche im Team. Ihr Nachteil ist jedoch eine unter Umständen langwierige Entscheidungsfindung.

Grundgesetz/projektunspezifische Regeln
Letzter Schritt der Initialisierung ist das gemeinschaftliche Beantworten von Fragen, die grundsätzlich in jedem Projekt wichtig sind (**Punkt 1.3 in Kasten 3**). Dies stellt die zuvor beschlossene Meta-Entscheidung auf eine erste Probe und kann bei größeren Teams mehr als einen Nachmittag Aufwand bedeuten. Wir geben für die in **Kasten 3** gestellten Fragen folgende Empfehlungen:

- *Dokumentation:* Alle beschlossenen Regeln und Prozesse sowie alle vom Team getroffenen architektonischen Entscheidungen (**siehe Kasten 4**) sind in einem Wiki festzuhalten. Alle TM haben Lese- und Schreibrechte. Darüber hinaus werden ein zentraler Problem-Lösung-Katalog und eine zentrale Checklistensammlung gepflegt (**siehe Kasten 5**).
- *DoC:* Für CCD-Anfänger eignen sich die Regeln des roten Grades (siehe unten). CCD-Fortgeschrittene können unter Berücksichtigung der Kritikalität die für das Projekt geeigneten CCD-Regeln festlegen (**siehe Abbildung 6**).
- *DoD:* Die in **Kasten 3 unter 4.1** genannten Maßnahmen verbindlich festlegen (**siehe Abbildung 3** in [Obe14]). Im Zusammenhang mit der Testabdeckung empfehlen wir bei mittlerer Kritikalität circa 80 Prozent der Programmzeilen (**siehe Abbildung 6**). Außerdem halten wir mindestens einen Integrationstest für notwendig. Eine manuelle Untersuchung muss dabei sicherstellen, dass alle kritischen Pfade enthalten sind.
- *Code-Ownership:* Allen gehört alles, d.h. jeder Entwickler darf jede Codestelle ändern. Das ist nicht selbstverständlich und muss klar kommuniziert werden.

Don't Repeat Mistakes
Vieles in der Softwareentwicklung lässt sich nicht automatisieren und muss manuell erledigt werden. Dabei geschehen immer wieder Fehler. Damit diese Fehler nicht wiederholt auftreten, ist es wichtig festzuhalten, wie sie vermieden werden können. Für manuelle Arbeitsschritte eignet sich eine Checkliste, die jeden Schritt in der richtigen Reihenfolge benennt.

Don't Re-unravel Mysteries
Bei komplexen Systemen kommt es immer wieder vor, dass sich das System völlig unerwartet verhält. Die Ursachenfindung dauert manchmal lange und hält die Softwareentwicklung sehr auf. Nach Abschluss einer solchen Rätselsuche ist es deshalb wichtig, das Problem zusammen mit seiner Lösung zu dokumentieren. Dafür eignet sich ein Problem-Lösung-Katalog. Dieser kann mit der Zeit für ein spezielles Problem (Symptom) mehrere Lösungen beinhalten. Auf diese Weise muss ein Rätsel nur einmal gelöst werden.

Kasten 5: Die DoReMi- und DoReMy-Regel.

Exekutive/Anwendung
Der Projektalltag eines Entwicklers besteht im Wesentlichen aus der Analyse von Anforderungen, dem Umsetzen dieser Anforderungen und der Qualitätssicherung. Bei dieser Arbeit werden alle im Team abgesprochenen Regeln und Prozesse angewandt. Dabei gehört es zum Berufsethos, diese Regeln auch gegen widrige Umstände wie Termindruck und den inneren Schweinehund professionell zu verteidigen.

Parlament/Meinungsaustausch
Für eine ständige Optimierung des TCC braucht das Team ein Forum zum Diskutieren und zur Meinungsbildung. Im TCC stehen dafür drei verschiedene Formen von „Parlamentssitzungen" zur Verfügung:

- *Daily Standups:* Nach dem Vorbild von Scrum sehr kurze Meetings, die täglich zur selben Zeit im Stehen durchgeführt werden. Jedes TM beschreibt kurz seine aktuelle Tätigkeit und nennt gegebenenfalls Probleme, die es aufhalten. *Kosten:* Circa zwei Minuten pro Person und Tag, circa 0,5 *Personentage (PT)* pro Monat für ein fünfköpfiges Team. *Nutzen:* Aufdecken von Querverbindungen zwischen Programmieraufgaben, Hilfeangebote von TM, die Möglichkeit zur täglichen Optimierung von Regeln und Prozessen.
- *Journal Coding Club:* Wöchentliche oder 14-tägige Team-Meetings, die Fortbildungscharakter mit direktem Projektbezug haben. *Kosten:* Circa 2 PT pro Monat für ein fünfköpfiges Team. *Inhalt:* Besprechung von Fachliteratur, Code-Reviews, Vorstellung von aktuell oder zukünftig eingesetzten Technologien. *Nutzen:* Angleichung des Wissensstands im Team, einheitliche Vorstellung davon, „was sauber ist und was nicht", einheitliche Meinung, mit welcher Lösungsstrategie wiederkehrende Probleme angegangen werden.
- *Retrospektive-Meetings:* Nach dem Vorbild von Scrum monatliche oder quartalsweise zwei- bis dreistündige Team-Meetings. *Inhalt:* „Was lief gut, was schlecht?" sowie „Lessons Learned". *Kosten:* Circa 1,5 PT pro Monat für ein fünfköpfiges Team. *Nutzen:* Regelmäßige Reflexion der aktuellen TCC-Regeln und -Prozesse über einen längeren Zeitraum.

Diese drei Team-Meetings benötigen zusammen für ein fünfköpfiges Team ca. 4 PT pro Monat an Kommunikationsaufwand. Dieser Aufwand ist aber kein reiner Zusatzaufwand für TCC, denn Kommunikation findet auch ohne TCC statt. Bei Teams, die von sich aus viel kommunizieren, ist der Zusatzaufwand für TCC gering, die Kommunikation erfolgt nur in strukturierterer Form.
Parlamentsmitglieder tauschen auch außerhalb der Parlamentssitzungen Informationen aus. Für TCC-Mitglieder stehen dazu Pair-Programming (vgl. [Wik-f]) und Coding-Notes zur Verfügung:

- *Pair-Programming* ist besonders effektiv, wenn es gezielt bei kniffligen Problemen zum Einsatz kommt. Eine gute Lösung wird so meistens viel schneller gefunden als allein. Zugleich wird das Wissen über diese Lösung im Team verbreitet und unerfahrene Entwickler können dabei extrem gut von erfahrenen lernen.
- *Coding Notes* sind ein Mittel, um wertvolle Informationen zu verbreiten und auch zu konservieren. Die einfachste Form einer Coding Note ist die E-Mail an den Team-Verteiler. Auf diese Weise können Hinweise über neue Utilities im Code oder Links zu einer interessanten Webseite einfach mitgeteilt werden. Werden solche E-Mails mit einem speziellen Schlüsselbegriff im Betreff markiert, kann jeder Entwickler in seinem Postfach sehr einfach eine Wissensdatenbank aufbauen.

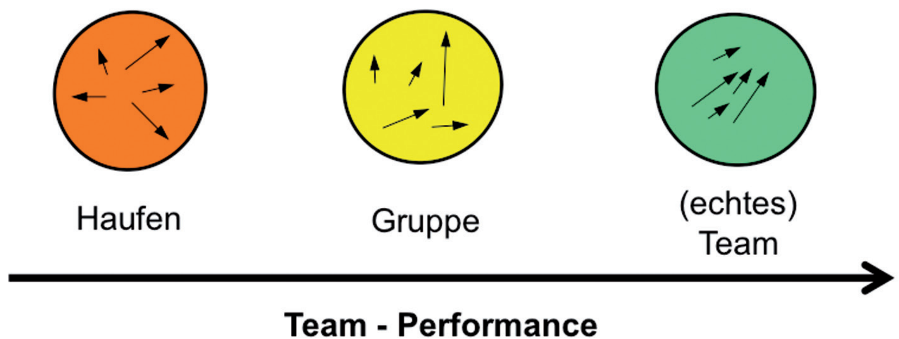

Abb. 4: Schematische Darstellung von Team-Typen ([Wiki-e]). Jeder Pfeil stellt die Wirkrichtung eines Entwicklers und dessen Stärke dar. Jeder Kreis stellt ein Team dar. Die Summe der Wirkrichtungen in einem Team stellt die Team-Performance (die Effizienz des Teams) dar. Nur in echten Teams ist die Performance hoch. Arbeiten die Teammitglieder nicht gut zusammen, handelt es sich eher um eine Gruppe oder sogar um einen Haufen. TCC ermöglicht, aus Haufen Gruppen und aus Gruppen echte Teams zu machen. Dazu müssen aber auch die Rahmenbedingungen stimmen.

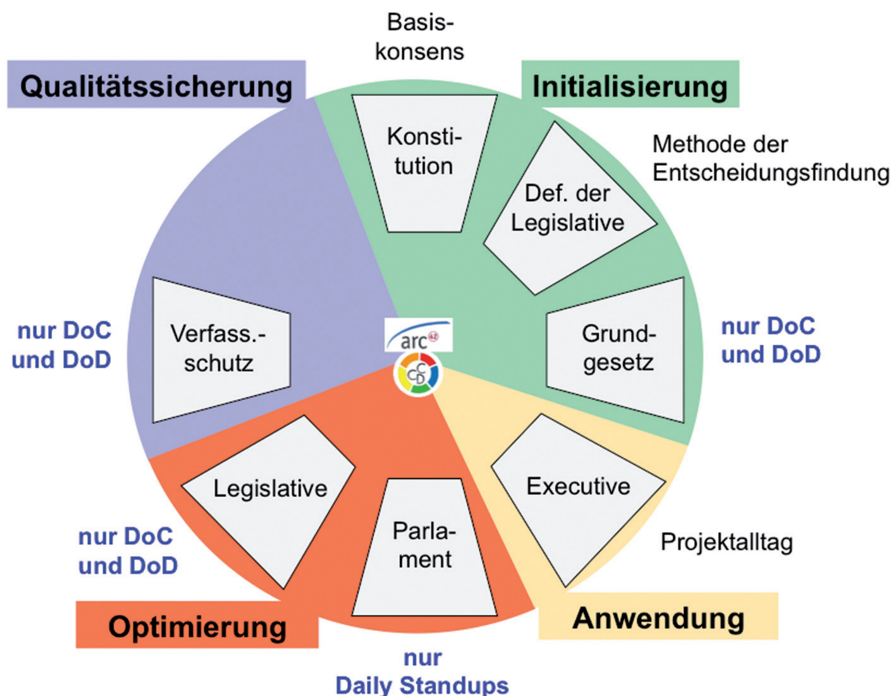

Abb. 5: Das TCC-Startup-Kid (DoD = Definition of Done, DoC = Definition of Clean).

Legislative/Weiterentwicklung des Regelwerks

In den frühen Phasen eines Projekts reichen die projektunspezifischen Grundregeln zunächst aus, um effektiv im Team zu arbeiten. Jedes Projekt stellt jedoch eine eigene kleine oder große Welt dar, die spezifische Umstände und Situationen mit sich bringt. Deshalb dauert es in der Regel nicht lange, bis die Notwendigkeit nach projektspezifischen Regeln und Prozessen deutlich wird. Es gibt immer wieder Phasen im Projekt, in denen die Komplexität wächst – und mit dem Projekt entwickeln sich auch die TM weiter. Aus diesem Grund ist es wichtig, die bestehenden Regeln und Prozesse ständig zu reflektieren und gegebenenfalls zu optimieren. Die im Abschnitt „Parlament/Meinungsaustausch“ angesprochenen Möglichkeiten zur Diskussion und zum Informationsaustausch führen bei den TM zu einer Meinungsbildung, die sich dann immer wieder in der Überarbeitung der bestehenden Regeln und Prozesse oder in der Formulierung neuer Regeln und Prozesse manifestiert.

Der Verfassungsschutz/ Vier-Augen-Review

Das Vier-Augen-Review prüft die Einhaltung der *Definition of Done (DoD)*, beruht auf dem Vier-Augen-Prinzip und stellt ein bewährtes Review-Verfahren dar: die Stellungnahme (vgl. [Poh10]). Betrachtet ein TM eine Programmieraufgabe (Bugfix oder neues Feature) als vollständig umgesetzt, bittet es ein anderes TM seiner Wahl, ein Vier-Augen-Review für diese Programmieraufgabe durchzuführen (siehe Abbildung 3). Der Reviewer prüft Folgendes:

- Alle funktionalen und nicht-funktionalen Anforderungen wurden umgesetzt.
- Alle im Team vereinbarten CCD-Regeln wurden eingehalten (DoC).
- Es gibt keine als unfertig markierten Codestellen mehr (siehe Kasten 6).
- Der Quellcode ist leicht verständlich (dazu zählt unter anderem die Code-Dokumentation).
- Die Testabdeckung ist ausreichend.

Mit folgender Syntax können Codestellen, die in irgendeiner Form noch eine Überarbeitung benötigen, markiert werden:

```
<Tag> <Datum> <Autor des Tags>
<Beschreibung>
```

Folgende Tags stehen zur Verfügung:

- `FIXME`: Diese Codestelle muss noch bearbeitet und der Tag entfernt werden, um die DoD zu erreichen.
- `TODO`: Diese Codestelle (und der Tag) dürfen für die DoD nur verbleiben, wenn dafür im verwendeten Issue-Tracker ein Issue angelegt wurde.
- `REFACTORME`: Diese Tags zeigen an, dass eine Codestelle nicht optimal gelöst wurde. Sie sind aber nicht DoD-relevant und sollten sparsam verwendet werden.

Kasten 6: Das TCC-Code-Tagging-System.

Erst nachdem auch der Reviewer sein OK abgegeben hat, gilt die Programmieraufgabe offiziell als erledigt. Bei Fehlern bzw. Mängeln, die im Nachhinein auftreten, trägt der Reviewer eine ebenso große Verantwortung wie der Autor des Codes. Unterstützen lässt sich diese Vorgehensweise durch das (leider nur auf GIT und Jenkins basierende) Tool „Gerrit“ (vgl. [Ger13]). Diese Vorgehensweise hat folgende fünf Vorteile:

1) Alle Softwarefehler, die der Reviewer findet, können sofort und zeitnah vom Autor behoben werden. Ohne das Review fallen diese Fehler meistens erst nach der nächsten Auslieferung auf – oft erst nach Wochen und Monaten. Das macht zusätzlichen Verwaltungsaufwand dieses Bugs in einem Issue-Tracker und eine häufig langwierige nachträgliche Analyse notwendig.
2) Trotz Einsatz von automatischen Code-Analysen bleibt der menschliche Code-Review das einzig sichere Mittel, die Verständlichkeit von Quellcode festzustellen.
3) Für Besuch macht man zu Hause in der Regel gründlicher sauber als für sich allein. Das Gleiche gilt beim Programmieren: Wer möchte schon, dass ein TM Schmutz im selbst geschriebenen Code findet?
4) Konstruktive Kritik ist für viele Entwickler motivierender als Desinteresse an geleisteter Arbeit.
5) Findet ein Reviewer wenig oder keine Kritikpunkte, stellt dieses Review-Ergebnis für den Autor ein unausgesprochenes Lob dar – eine wortlose Anerkennung für gute Arbeit. Das Lob darf aber auch gerne offen ausgesprochen werden.

Die Judikative/die 1-Euro-Regel

Fehler zu machen, ist menschlich. Fehler wiederholt zu machen, reduziert aber die Effizienz der Softwareentwicklung (siehe Kasten 5). Immer wiederkehrende Fehler, die andere behindern, schaffen schlechte Teamstimmung, die ebenfalls die Effizienz reduziert. Hier ein Beispiel: Zum Feier-

abend schließt ein TM seine Tagesarbeit mit einem gutem Gefühl ab und überträgt den aktuellen Entwicklungsstand ins zentrale Repository. Ein automatischer Prozess holt in seiner Abwesenheit die aktuellen Sourcen aus dem zentralen Repository, kompiliert sie, testet sie und baut die Software. Tritt dabei ein Fehler auf, steht ein anderes TM spätestens morgens früh vor einem fremden Hindernis. Diese Feierabend-Check-In-Mentalität ist ein typisches Beispiel für chronische Unsitten, gegen die das Team mit der im **Kasten 3** beschriebenen 1-Euro-Regel vorgehen kann. Die Erfahrung hat gezeigt, dass diese etwas skurril wirkende Regel für unangenehme Kratzer am eigenen Entwicklerstolz und damit eine erhöhte Motivation für die Einhaltung der Vereinbarungen sorgt. Nach einer längeren Projektphase kann das Team von den erworbenen Spenden zusammen essen gehen.

TCC-Rahmenbedingungen

Es gibt teils mehr, teils weniger klare Grenzen der Selbstbestimmung eines TCC-Teams. Deshalb hinkt der oben gezogene Vergleich zu den Rittern der Tafelrunde, denn die Ritter waren kaum fremdbestimmt, viele Entwicklerteams sind es aber maßgeblich – und zwar durch folgende Faktoren:

- Jedes Team arbeitet für einen Auftraggeber, der Rahmenbedingungen vorgibt. Dazu zählen z.B. die räumliche Infrastruktur, Hardware und Software.
- Ein Entwicklungsteam ist in der Regel nicht für den gesamten Lebenszyklus der Software verantwortlich (vgl. auch die Dimension „Prozess“ in [Obe13]). Die eigentliche Entwicklung ist meistens in einen größeren ALM-Prozess eingebunden. Deshalb muss das Team auf externe Faktoren, z.B. den Betrieb, der die Software in der Produktion pflegt, Rücksicht nehmen.
- Bei großen Unternehmen ist ein einzelnes Team häufig Teil einer ganzen Entwicklungsabteilung. Aus Sicht der Leitung der Gesamtentwicklung ist es ineffizient, falls die einzelnen Teams voneinander völlig losgelöst und autark einen Staat im Staate bilden. Um Wildwuchs und Ineffizienz aus team- und projektübergreifender Sicht zu vermeiden, muss ein TCC-Team Vorgaben der Gesamtentwicklung (z.B. bezüglich Qualitätssicherung) in ihrem Regelwerk berücksichtigen. Die Kunst des Entwicklungsleiters liegt in der Gratwanderung zwischen teamübergreifender Vereinheitlichung und Wahrung der Selbstbestimmung der einzelnen Teams.

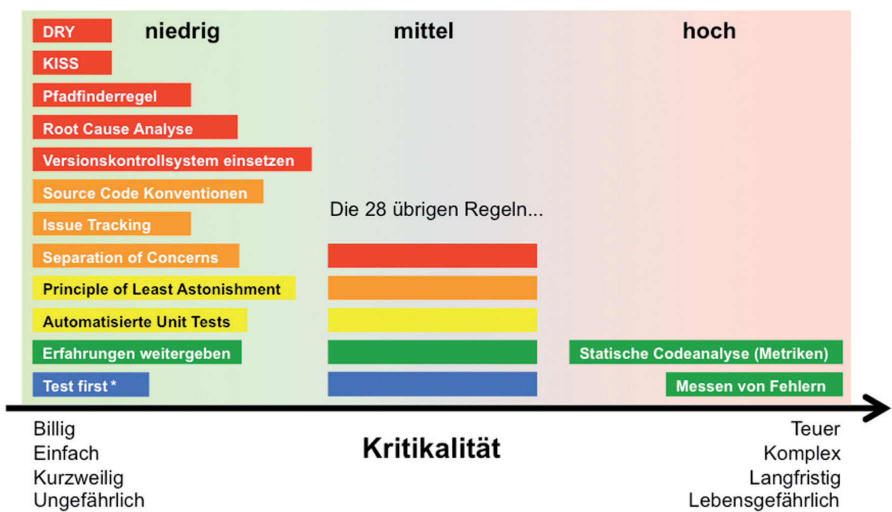

Abb. 6: CCC-Zuordnung von CCD-Regeln zur Kritikalität von Software. Die Farben zeigen den jeweiligen CCD-Grad der Regeln an. Die links aufgeführten Regeln schlagen wir im CCC-Kontext als CCD-Basisregeln vor.
** Im CCC-Kontext setzen wir Test First mit TDD gleich. Diese Entwicklungstechnik birgt gewaltiges Effizienz-Potenzial, ist aber sehr aufwändig im Erlernen. Für TDD-Neulinge sollte das Set der CCD-Basisregeln wenigstens eine TDD-Fortbildung beinhalten.*

Selbstorganisation des Teams unterstützen
Das Umfeld von TCC-Teams sollte die Selbstorganisation der Teams akzeptieren und unterstützen. Beispielsweise muss die benötigte Infrastruktur – wie ausreichende starke Software und Hardware für CI-Server und andere Qualitätssicherungsmaßnahmen – bereitgestellt werden.

Kommunikation im Team einplanen
Es muss ausreichend Zeit für die Kommunikation im Team eingeplant werden (siehe oben, Abschnitt „Parlament/Meinungsaustausch“), für ein fünfköpfiges Entwicklerteam etwa vier PT pro Monat.

Fortbildung der firmeneigenen Mitarbeiter fördern
Die Fortbildung der TM sollte unterstützt werden. In der schnelllebigen Welt der IT gilt: Stillstand ist Rückgang. Um von neuen modernen Lösungen zu profitieren, muss regelmäßig neues Know-how von außen in die Firma, in die Abteilung und in die Teams geholt werden. Außerdem erhöht die Fortbildung die Motivation vieler Entwickler auch in ihrem Entwickleralltag.

Teamstimmung berücksichtigen und fördern
Nicht selten werden neue TM, speziell externe Mitarbeiter, anhand so genannter *Skillsets* (Teil ihres Profils) ausgewählt und die persönlichen Eigenschaften und die menschliche Kompatibilität der TM werden vernachlässigt. Das kann schnell zu schlechter Teamstimmung, mangelnder Kommunikation und niedriger Performanz führen (**siehe Abbildung 4**). *Daily Standups* ermöglichen es, solche frühzeitig aufzudecken. Persönliche Konfliktsituationen kann das Team in der Regel nicht selbst lösen. In solchen Fällen ist ein Eingreifen des Projektleiters bzw. Scrum Masters gefragt. Ein nützliches Forum dafür stellen auch die Retrospektive-Meetings dar (siehe oben, Abschnitt „Parlament/Meinungsaustausch“). In einigen Fällen sollte eine neue Teamzusammenstellung in Betracht gezogen werden, was bei (Feature-getriebenen) agilen Teams ohnehin gebräuchlich ist.

TCC einführen

TCC ist eine Zusammenstellung von Maßnahmen, die alle nicht neu, aber im Teamplay bewährt sind. Die Motivation, diese Maßnahmen anzuwenden, muss – wie auch bei anderen Entwicklungsmethoden (wie Scrum, OOP oder TDD) – von den Entwicklern selbst kommen. Sie lassen sich nicht verordnen. Der Versuch, neue Arbeitsweisen in Teams einzuführen, ist deshalb grundsätzlich ein Problem. Insbesondere wenn es sich um ein noch nicht allgemein anerkanntes Verfahren handelt, sind viele Akteure sehr skeptisch. Um das so genannte *Not-Invited-Here-Syndrom*

(vgl. [Wik-d]) zu vermeiden, muss jedes TM bewusst an den Entscheidungsprozessen beteiligt werden.

TCC „unter der Haube" einführen
Der TCC-Maßnahmenkatalog ist kein monolithischer Prozess. Die einzelnen Maßnahmen des Katalogs können bei Bedarf einzeln angewandt werden. Die Motivation dafür stammt in der Regel aus Schmerzen, die eine ineffiziente SE verursacht. TCC kann also auch schrittweise *pain driven* eingeführt werden. Dabei ist jedoch der Basiskonsens grundsätzlich immer notwendig.

TCC im Start-Up-Kit einführen
Um zu testen, wie sich TCC anfühlt, ohne das volle Programm anzuwenden, kann ein Team einen reduzierten Maßnahmen-Katalog, das TCC-Start-Up-Kit, ausprobieren (siehe Abbildung 5). Nach etwa zwei Wochen stellt sich dann ein erstes TCC-Gefühl ein.

TCC ohne CCD-Kenntnisse einführen
Den Effizienzgewinn der gemeinsamen Anwendung von TCC- und CCD-Regeln halten wir für enorm.
Die CCD geben keine Prioritäten für ihre Regeln an und die Einteilung in die farbigen Grade ist eher eine Lernhilfe. Im CCC schlagen wir vor, die Priorität der CCD-Regeln mithilfe der Kritikalität der Software zu bewerten (siehe Abbildung 6). Vor dem Hintergrund der Kritikalität sehen wir einige einfache, leicht verständliche Basisregeln, die wirklich immer als Minimum-Muss angewendet werden sollten.
Teams, die planen, TCC ohne CCD-Vorkenntnisse einzusetzen, empfehlen wir wenigstens die in Abbildung 6 genannten Regeln für geringe Kritikalität zu berücksichtigen.

Fazit

„Team Clean Coding" ist eine Methode, um die Clean-Coding-Regeln im Entwicklerteam lebendig werden zu lassen. Dadurch wird der Softwareentwicklungsprozess erheblich effizienter, weil die Teammitglieder sich so gegenseitig motivieren, voneinander lernen und an einem Strang ziehen. Als Herzstück des Softwareentwicklungsprozesses wirkt das TCC in alle Dimensionen des Entwicklerkosmos hinein, die wir in Teil 1 dieses Artikels (vgl. [Obe13]) vorgestellt haben. Und es wirkt gegen alle Formen von kosmischem Schmutz, die wir in Teil 2 dieses Artikels (vgl. [Obe14]) erläutert haben. Es ist aber keine Weltformel – eher eine Einschwenkhilfe in einen günstigen Orbit, d.h. ein Wegweiser zu einem Softwareentwicklungsprozess, der zu einem bestimmten Stern, d.h. Projekt, und den dortigen Akteuren gut passt. Das TCC besitzt einige konkrete Startregeln, jedoch wenige allgemeingültige Regeln und lässt somit viele Anpassungsmöglichkeiten für spezielle Projektsituationen offen. ||

Literatur & Links

[CCD] Clean Code Developer, siehe: http://www.clean-code-developer.de/
[Hru13-a] P. Hruschka, G. Starke, arc42 – Ressourcen für Software-Architekten - Template, 2013, siehe: http://www.arc42.de/template/template.html
[Hru13-b] P. Hruschka, G. Starke, arc42 – Ressourcen für Software-Architekten – Prozess, siehe: http://www.arc42.de/process/process.html
[Obe13] R. Oberrath, J. Vollmer, Clean Coding Cosmos: Teil 1: Kosmologie für Softwareentwickler, in: OBJEKTspektrum 6/2013
[Obe14] R. Oberrath, J. Vollmer, Clean Coding Cosmos: Teil 2: Kosmologische Suche nach Softwareentwicklungsschmutz, in: OBJEKTspektrum 1/2014
[Poh10] K. Pohl, C. Rupp, Basiswissen Requirements Engineering (2. Auflage), dpunkt.verlag 2010
[Wes12] R. Westphal, Clean Code Developer: Korrektheit als Wert, in: OBJEKTspektrum 06/2012
[Wes13-a] R. Westphal, Clean Code Developer, Teil 3: Evolvierbarkeit als Wert, in: OBJEKTspektrum 01/2013
[Wes13-b] R. Westphal, Clean Code Developer, Teil 4: Produktionseffizienz als Wert, in: OBJEKTspektrum 02/2013
[Wik-a] Wikipedia, Soziokratie, siehe: http://de.wikipedia.org/wiki/Soziokratie
[Wik-b] Wikipedia, Konsens, siehe http://de.wikipedia.org/wiki/Konsens
[Wik-c] Wikipedia, Kritik der praktischen Vernunft, siehe: http://de.wikipedia.org/wiki/Kritik_der_praktischen_Vernunft
[Wik-d] Wikipedia, Not-invited-here-Syndrom, siehe: http://de.wikipedia.org/wiki/Not-Invented-Here-Syndrom
[Wik-e] Wikipedia, Teambildung, siehe: http://de.wikipedia.org/wiki/Teambildung
[Wik-f] Wikipedia, Pairprogramming, siehe: http://de.wikipedia.org/wiki/Paarprogrammierung
[Wit11] M. Wittwer, Entscheiden im Konsens, 2011, siehe: http://www.oose.de/blogpost/entscheiden-im-konsens-teil-1-was-ist-konsens/
[Zör12] S. Zörner, Softwarearchitekturen dokumentieren und kommunizieren: Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten, Hanser Verlag 2012

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.