CLEAN CODING COSMOS: TEIL 1: KOSMOLOGIE FÜR SOFTWAREENTWICKLER

Die Kosmologie ist die naturwissenschaftliche Disziplin, die das „Big Picture" unserer Welt zeichnet: eine vierdimensionale Raumzeit, in der verschiedene Kräfte auf Materie einwirken. Unser kosmologischer Blick auf die Softwareentwicklung ergibt ein ganz ähnliches Bild. Es gibt vier Dimensionen (Prozess, Wissen, Handwerk und Motivation), in denen Kräfte (die Meinungen der Akteure) auf die Materie (Software) einwirken und diese formen. In diesem Artikel wollen wir die vier Dimensionen und ihre Aspekte möglichst vollständig vorstellen, ohne uns in Details zu verlieren. Es geht uns um einen Überblick, um das „Big Picture" in der Softwareentwicklung.

Jeder Akteur in der *Softwareentwicklung (SE)*, z. B. Anforderungsspezialist, Entwickler, Stakeholder von der Fachseite oder aus dem Betrieb, Endanwender, hat eine Vorstellung, wie die „Materie" Software sein soll oder nicht sein darf. Diese Meinungen wirken in verschiedenen Dimensionen als anziehende oder abstoßende Kräfte auf die „Materie" ein und formen sie.

Hinter dem Begriff „Software" verbergen sich zwei sehr unterschiedliche Aspekte, die aber stark miteinander verwoben sind (siehe Abbildung 1):

- Zum einen gibt es einen für den Kunden/Anwender sichtbaren Teil der Materie (die Anwendung).
- Zum anderen gibt es einen verborgenen Teil – den Softwareentwicklungsprozess im weitesten Sinne (siehe Kasten 1).

Beide Aspekte sind gleich wichtig und in allen Dimensionen den abstoßenden und anziehenden Kräften ausgesetzt. Zwei dieser Dimensionen (Handwerk und Wissen) sind typische *Hard Skills*. Die menschliche Dimension „Motivation" widmet sich den *Soft Skills*. Da die Festlegung und Einhaltung von Prozessen stark vom Faktor Mensch geprägt ist, besitzt die zeitliche Dimension „Prozess" einen Mischcharakter.

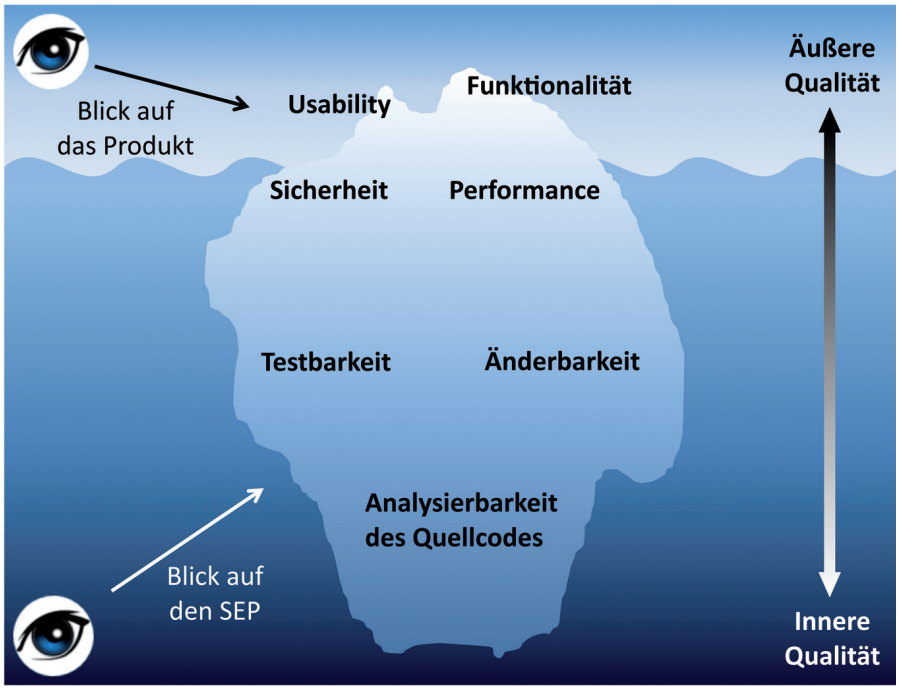

Abb. 1: Software ist wie ein Eisberg. Das Softwareprodukt (SP) ist die Spitze des Eisbergs, die aus dem Wasser ragt. Einige Merkmale sind noch knapp unter der Wasseroberfläche erkennbar. Der größte Teil aber, der die Eisbergspitze trägt, ist nur dem Insider ersichtlich. Der SEP muss für das SP mächtig genug sein, sonst geht der Eisberg unter.

Die zeitliche Dimension „Prozess"

Der Lebenszyklus einer Anwendung wird durch das *Application Lifecycle Management (ALM)* organisiert (vgl. [Wil12]). Doch wann beginnt dieser Zyklus? Beginnt er mit der ersten Produktionsversion, mit dem Kick-Off des Entwicklungsprojekts oder mit dem ersten Aufwand, der im Vorfeld in den Entwurf und in die Projektplanung investiert wird?

Die Existenz einer Anwendung beginnt virtuell mit der Idee in der Vorstellung eines Menschen (siehe Abbildung 2). Ist diese attraktiv, zieht sie weitere Ideen an und eine Vision beginnt, ihre Leuchtkraft zu entfalten. Ein neuer Stern ist im Entwicklerkosmos sichtbar geworden. Die Anwendung wächst virtuell in der Anforderungsanalyse zu einem Entwurf. Schließlich beginnt sie, mit der Implementierung auch in der realen Welt zu existieren. Während ihrer produktiven Phase im Betrieb reift die Anwendung. Wie ein echter Stern endet ihre Lebenszeit entweder schnell und laut, wenn sie ersetzt wird, oder

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.

- **Im weitesten Sinne** steht der **SEP** für den Lebenszyklus einer Anwendung bzw. das ALM (siehe Abbildung 2). Die Akteure aller Phasen tragen letztlich zur Entwicklung der Software bei.
- **Im engsten Sinne** steht der **SEP** für das Kerngeschäft der Softwareentwickler: die Implementierung (Phase 3 in Abbildung 2)
- In einem **nicht weiten und nicht engen Sinn** steht der **SEP** für das, was der Softwareentwickler zu leisten hat: Mit dem Anforderungsspezialisten die Anforderungen prüfen und vervollständigen, die Anwendung implementieren und zusammen mit dem Betrieb die Software einsatzfähig machen. In diesem Sinn schließt der SEP auch einen Teil von Phase 2, 4 oder 5 in Abbildung 2 mit ein. In Sonderfällen kann der SEP auch die Phasen 1 oder 5 komplett beinhalten.

Kasten 1: Bedeutungsbreite des Begriffs „Softwareentwicklungsprozess“ (SEP).

sie erfährt ein langsames stilles Ende, wenn sie immer seltener gebraucht wird. Der Akteur, der an den meisten Phasen beteiligt ist, ist der Softwareentwickler (siehe Kasten 1).

Das ALM ist der Basisprozess, in den alle Teilprozesse rund um die Anwendung eingebunden sind. Er beinhaltet oft komplexe Prozesse, die miteinander verzahnt sind und die aus einfacheren Teilprozessen bestehen (siehe Kasten 2).

ALM und Traceability

Problematisch beim ALM ist der Medienbruch im Informationsfluss zwischen den Akteuren, der typischerweise beim Phasenwechsel im ALM auftritt: Wie stellt der Visionär sicher, dass der Anforderungsspezialist alle wichtigen Ideen berücksichtigt? Wie stellt der Anforderungsspezialist sicher, dass der Entwickler alle Anforderungen richtig umsetzt? Wie stellt der Entwickler sicher, dass der Betrieb alle wichtigen Informationen zur korrekten Inbetriebnahme hat? Bei dem Informationsfluss über die Phasengrenzen ist es wichtig, Informationen zurückzuverfolgen, das heißt festzustellen, wo Informationen herkommen und ob sich Informationen geändert haben. Um diese Rückverfolgung (*Traceability*) sicherzustellen, wäre ein einheitliches Repository sinnvoll, das die Akteure aller Phasen gemeinschaftlich nutzen. Außerdem ist eine entsprechende Tool-Unterstützung notwendig (siehe unten „Traceability und SLI“).

Die technische Dimension „Wissen“

Diese Dimension fasst all das Know-how zusammen, das als SE-Theorie bezeichnet werden könnte. Dazu zählen technologisches Know-how sowie das Verständnis von Architektur und Implementierung (siehe Kasten 3).

Architekturwissen

Architektur hat sehr viele Aspekte (vgl. [Sta09], Kapitel 6 und 7). Für eine saubere Architektur halten wir aus Clean-Coding-Sicht drei Qualitätsmerkmale von Software für besonders wichtig: Erweiterbarkeit, Analysierbarkeit und Testbarkeit. Diese Qualitätsmerkmale haben auch einen Implementierungsaspekt (unverständlicher Quellcode mindert die Qualität aller drei Merkmale). In den drei folgenden Abschnitten gehen wir nur auf den architektonischen Aspekt ein.

Erweiterbarkeit

Erweiterbare Architektur beruht auf modularen Strukturen (z. B. komponentenorientierte Architektur, vgl. [Kno12]). Allerdings haben monolithische Strukturen den Vorteil, dass die Entwickler ohne Rücksicht auf

- **Projektmanagement:** Das Entwicklerteam organisieren.
- **Issue-Tracking:** Neue Anforderungen oder Fehler verwalten und bearbeiten.
- **Release-Management:** Die nächste Version genau mit den benötigten Softwareänderungen ausliefern.
- **Definition of Done:** Die Kriterien festlegen und sicherstellen, dass die Implementierungsaufgabe vollständig abgeschlossen ist.
- **Qualitätssicherung in der Entwicklung:** Sicherstellen, dass bereits bestehende Softwarefunktionen nicht ungewollt verändert wurden.
- **Auslieferung:** Die fertige Software für die Installation auf dem Zielsystem bereitstellen.
- **Qualitätssicherung im Betrieb:** Sicherstellen, dass die auf dem Zielsystem installierte Software richtig funktioniert.

Kasten 2: Typische Teilprozesse/Workflows im SEP.

Modulschnittstellen implementieren können. Eine Vielzahl von Schnittstellen und Abhängigkeiten zwischen Modulen erhöht nämlich die (Modul-)Komplexität von Software. Jedoch haben monolithische Strukturen den Nachteil einer größeren (Klassen-)Komplexität. Diese Form von

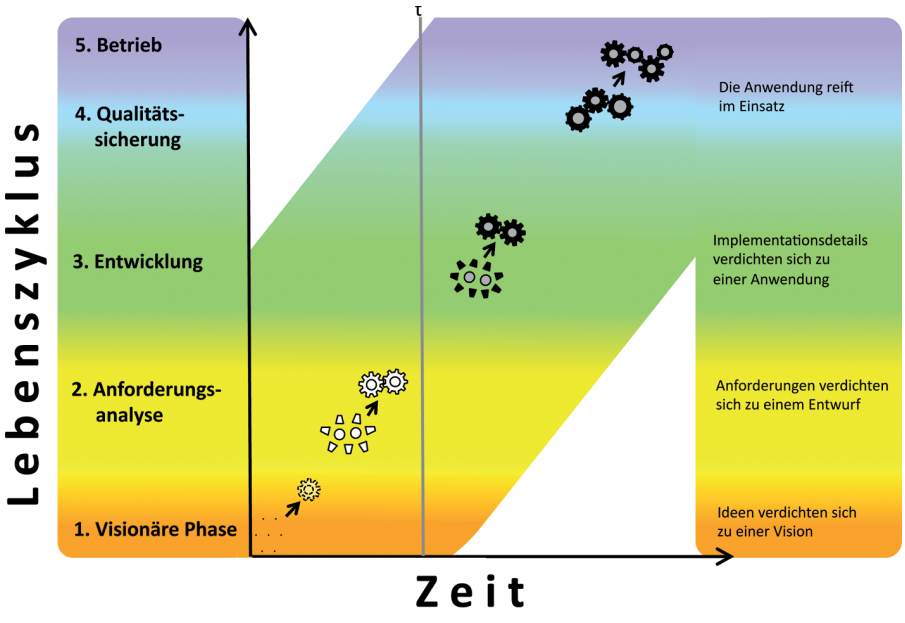

Abb. 2: Phasen im Lebenszyklus einer Anwendung. Zu einem Zeitpunkt können gleichzeitig verschiedene Phasen aktiv sein. Das ALM organisiert diese Phasen.

- **Architektur** ist das Treffen und Dokumentieren von Entscheidungen im Bauplan und Ablaufplan mit systemweiter (globaler) Tragweite. Solche Tragweite haben Entscheidungen, die allen Entwicklern einen einheitlichen Rahmen für ihre Arbeit geben und für ein gemeinsames Verständnis der Software essenziell sind.
- **Design** ist das Treffen von Entscheidungen mit mäßiger (regionaler) Tragweite. Plakativ lässt sich formulieren: „Architektur ist Grobdesign". Die Umkehrung „Design ist Feinarchitektur" ist wegen mangelnder Tragweite falsch. Außerdem hat Design einen stärkeren Bezug zur Implementierung und ist in der Regel stärker fachlich orientiert.
- **Implementierung** beinhaltet das Treffen von Entscheidungen mit lokaler Tragweite. Solche Entscheidungen haben auf die Arbeit anderer Entwickler wenig Einfluss.

Kasten 3: Architektur vs. Design vs. Implementierung.

Komplexität ist der große Nachteil eines Monolithen, die ihn unflexibel oder gar unwartbar macht. Sie wird durch eine modulare Architektur minimiert, wobei ein Teil der Komplexität von der Klassen- auf die Modulebene verschoben wird. Die Verschiebung von Komplexität auf die Modulebene ist vorteilhaft, weil sie sowohl die Klassen- als auch die Gesamtkomplexität minimiert – zumindest, wenn die Module technisch sauber und sinnvoll voneinander abgegrenzt sind (vgl. [Kno12], [Sta09]). Die Abgrenzung von Modulen (die Schnitte durch den Monolithen) können fachlich und/oder technisch erfolgen (siehe Abbildung 3).

Analysierbarkeit im Betrieb

Eine Anwendung, die ausdrucksstarke Informationen durch Logging, Protokollierung und Monitoring-Funktionen liefert (siehe Kasten 4), ist zur Laufzeit gut analysierbar. Für die technische Realisierung dieser Informationsdienste muss die Architektur einen einheitlichen Rahmen zur Verfügung stellen.

Testbarkeit

Testen ist das Sammeln aller Arten von Informationen über das Testsystem (vgl. [Sch11]). Ohne Tests ist es praktisch unmöglich, sichere Aussagen zur Softwarequalität zu machen. Wird nach der TDD-Methode (*Test First*) entwickelt, wird vor der Implementierung des Produktivcodes zuerst der Testcode implementiert (vgl. [Bec02]). Auf diese Weise ergeben sich automatisch Architektur- und Designstrukturen, die neben der Testbarkeit auch eine bessere Erweiterbarkeit gewährleisten. Ohne die TDD-Methode werden häufig unbewusst Entscheidungen getroffen, die das nachträgliche Implementieren von Tests praktisch unmöglich machen (vgl. [Fre10]).

Tabelle 1 stellt die drei typischen Testebenen vor: Unit-, Integrations- und Systemtests. Der Teststrategie in Tabelle 1 folgend, sollten alle Tests regelmäßig ausgeführt werden, um Fehler möglichst frühzeitig zu finden. Das ist jedoch nur bei automatisierten Tests wirtschaftlich möglich. Diese können allerdings in der Wartung – durch Umbaumaßnahmen im Quellcode oder Funktionserweiterungen – sehr aufwändig sein. Aber ohne diese Tests muss man davon ausgehen, dass bei komplexen Systemen die Analyse und Behebung von Fehlern noch viel teurer werden als das Implementieren und Pflegen des Testcodes (vgl. [Fea11]).

Automatische Tests sind „unkreativ". Deshalb sind – neben den automatischen Tests, die schnell und fehlerfrei ablaufen – immer zusätzlich manuelle Tests von kreativen Menschen nötig, die neue Testvarianten finden und die Software im Einsatz erleben.

Wissen, wie man verständlich implementiert

Sauberer Code ist verständlicher Code. Verständlicher Code bedeutet effektive SE. Jahrzehnte lange Erfahrung haben in der SE eine Reihe von Best Practices (*Patterns*) deutlich gemacht. Außerdem gibt es einige typische Fehler, die immer wieder gemacht werden (*Anti-Patterns*). Zum Teil sind diese „Dos and Don'ts" von der verwendeten

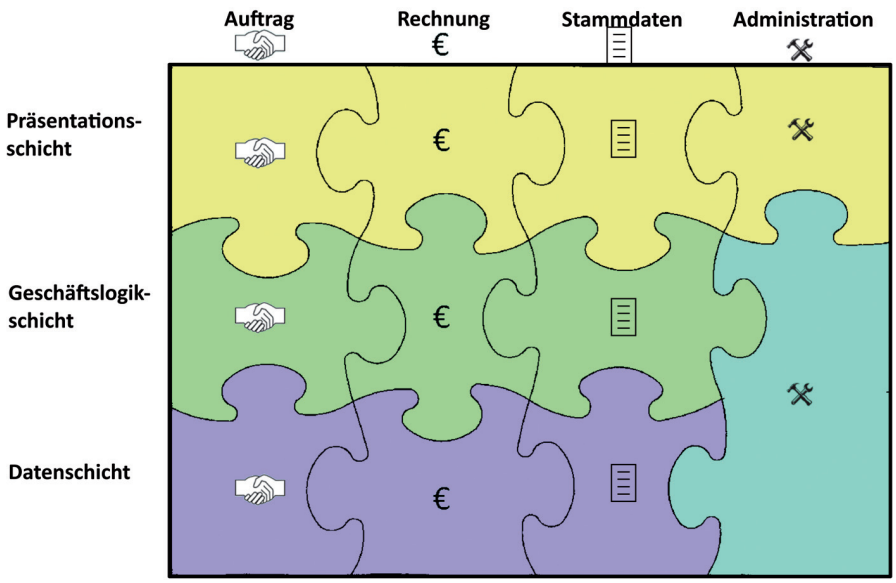

Abb. 3: Mögliche technische und fachliche Schnitte durch eine Anwendung. Vorrang haben fachliche Schnitte (vgl. [Par72]).

- **Logging (Tracing):** Speichern von Informationen über die im System ablaufenden Funktionen und Prozesse in speziellen Logdateien – geeignet für technische Informationen.
- **Protokollierung (Reporting):** Häufig wird darunter das Speichern von technischer und/oder fachlicher Information verstanden (z. B. [Sta09]). Wir schlagen vor, unter diesem Begriff nur das Speichern von fachlichen Informationen zu verstehen – vorzugsweise in strukturierte Form in Datenbank-Tabellen. So kann dieser fachliche Aspekt besser vom Logging abgegrenzt werden.
- **Monitoring:** Automatische regelmäßige Prüfung von Sollzuständen und ein definierter Alarm, wenn ein Sollzustand nicht erfüllt ist.

Kasten 4: Logging vs. Protokollierung vs. Monitoring.

		Empfohlene Teststrategie		Aussagekraft		Aufwand	
	Testebene	**Testab-deckung**	**Ausführung**	**Qualität**	**Testtiefe**	**Initialer Aufwand**	**Ausführung**
Unit-Tests	Programmzeilen, Funktionen, Klassen	Hoch (> 90%)	Nach jeder Code-Änderung im Versionskontroll-system	Testet die innere Qualität der Software	Flach, aber dafür sehr breit	Gering	Schnell
Integrations-Tests	Module, Komponenten, Subsysteme	Mittel (ca. 50%)	Möglichst nach jeder Änderung, mindestens einmal pro Tag	Etwas teils innere, teils äußere Qualität	Gemischte Strategie	Mittel	Mäßig
System-Tests	Vollständiges System	Gering (ca. 10%)	Einmal pro Tag, mindestens einmal pro Woche	Testet äußere Qualität der Software	Tief (durch alle Schichten)	Hoch	Langsam

Tabelle 1: Vergleich der verschiedenen Testebenen.

Programmiersprache und Technologie abhängig, zum Teil aber auch nicht. Die *Clean Code Developer (CCD)* haben die allgemeingültigen Best Practices zu einem Regelwerk gebündelt, dessen Anwendung zu sauberer SE führt. Tabelle 2 zeigt, dass es neben den CCD-Regeln aber noch weiteres Wissen gibt, das für die Verständlichkeit von Quellcode wichtig ist.

Wissen, wie man effektiv dokumentiert

Dokumentation lohnt sich, wenn sie entweder mit wenig Aufwand hilfreiche Information liefert (Notiz-Funktion) oder wenn sie vollständig, korrekt und verständlich ist (Lexikon-Funktion). Ein Großteil von Dokumentation liegt dazwischen, d. h. großer Umfang, aber unvollständig und/ oder unverständlich – letzteres häufig wegen fehlenden Spezialwissens der Leser. Außerdem ist es sehr schwierig, unmissverständlich zu formulieren. Für eine gute Dokumentation außerhalb des Codes helfen dabei Formulierungsregeln und Standard-Templates (vgl. [Rup10], [Rup09]). Für Dokumentation im Code gilt: „When you feel the need to write a comment, first try to refactor so that any comment becomes superfluous“ (vgl. [Fow99], [Mar08]).

Wissen über Technologien

Die grundlegende Technologie für einen Entwickler ist seine *Programmiersprache* und deren Grundfunktionalität. Zusätzlich gibt es eine große Vielfalt von *Third-Party-Libraries*, welche die Grundfunktionalität der Programmiersprache drastisch erweitern. Darüber hinaus existieren mächtige *Frameworks*, die dem Entwickler die Komplexität für eine umfangreiche oder schwierige Funktion abnehmen. Ein guter Entwickler hat auf allen drei Ebenen ein breites und an ausgewählten Stellen auch ein tiefes Wissen.

Die praktische Dimension „Handwerk“

SE ohne Werkzeugunterstützung ist heute undenkbar. So hängt die Produktivität stark vom Einsatz der *passenden* Werkzeuge und deren *Beherrschung* ab. Neben diesen Tool-Fertigkeiten ist es für den Softwareentwickler ebenfalls wichtig, Nutzen aus modernen SE-Techniken zu ziehen.

Die Werkzeuglandschaft in der SE

Ein Problem ist die kaum überschaubare Auswahl von Tools – ja sogar deren Kategorisierung wächst stetig. Ohne Anspruch auf Vollständigkeit versucht Abbildung 4, Ordnung in die Werkzeuglandschaft zu bringen (vgl. auch [Hru12]). Viele Werkzeuge werden zu immer weiter wachsenden Tool-Chains kombiniert (vgl. [Wiki-c]).

Traceability und SLI

Ein weiteres Problem betrifft die Kooperationsmöglichkeiten der verschiedenen Tools im ALM untereinander (vgl. [Wil12]). Die letzten Jahre waren gekennzeichnet von einer zunehmenden Spezialisierung. Viele Werkzeuge sind geschlossen und können nur wenig oder gar keine Informationen untereinander austauschen. Das führt zu Medienbrüchen im ALM (siehe oben, „ALM und Traceability“). Für einen Teil der Tools ist es üblich, die Integrationsmöglichkeiten einer *Integrated Development Environment (IDE)* zu nutzen. Die IDE ist das Cockpit des Entwicklers. Neben der Grundausstattung verfügen moderne IDEs über zahlreiche Erweiterungsmöglichkeiten mit Hilfe von Plug-Ins. Damit sind IDEs von Natur aus offene Systeme, an die sich viele andere Tools ankoppeln lassen: Versionsverwaltung, Build-Systeme, Datenbanken, Applikationsserver, Issue-Tracker usw.

Allerdings sind auch hierbei die Möglichkeiten beschränkt. Eine durchgängige *Traceablitiy* von der Vision bis zum Betrieb ist auch mit einer IDE bislang nicht gelungen. Zum Beispiel findet man kaum Verbindungen (Links) zwischen Quelltexten und den dazugehörigen Anforderungen und umgekehrt. Für diesen Zweck wurden Tools für AML-Komplettlösungen entwickelt. Dieser Ansatz, Tools aus dem gesamten ALM zusammenzuschließen, heißt *SLI (Software Lifecycle Integration)* (vgl. [Tas]). Die großen Hersteller haben das Problem erkannt und bieten Realisierungen dafür an (siehe Tabelle 3). Ein offensichtlicher Nachteil besteht darin, dass diese Lösungen nicht herstellerübergreifend funktionieren. Das kann sich jedoch noch ändern:

- *OSLC:* 2008 wurde der offene Standard *OSLC (Open Services for*

	Allgemein	Technologie-spezifisch
Pattern	[Cle] [Mar08]	[Ale04] [Blo08] [Codep]
Anti-Pattern	Kosmologische Suche nach Softwareentwicklungsschmutz (Teil 2 dieses Artikels).	[nsz10] [All02] [Dev08]

Tabelle 2: Was ist zu beachten, um verständlich zu implementieren?

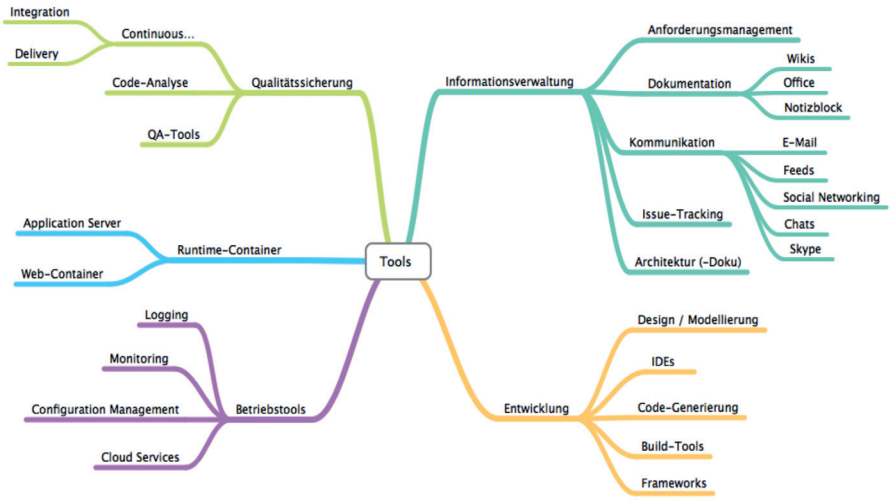

Abb. 4: Übersicht über die Tool-Landschaft im SEP im weitesten Sinne.

Lifecycle Collaboration) ins Leben gerufen mit der Zielsetzung, Tools sämtlicher Kategorien und vor allem auch unterschiedlicher Hersteller auf einfache Weise integrieren zu können (vgl. [You]).

- *Atlassian:* Die Firma Atlassian bietet selbst keine komplette ALM-Lösung an, sondern präferiert ein offenes Plug-In-Konzept (IDE-ähnlich). Mittlerweise gibt es eine beträchtliche Anzahl (> 600) von Plug-Ins. Auch die Anbindung von Drittanbieter-Tools ist bereits weit fortgeschritten.
- *Microsoft:* Der „Team Foundation Server" zeichnet sich ebenfalls durch eine große Ausbaustufe der ALM-Komplettlösung aus, allerdings nur im .NET-Umfeld.

Mit diesen „Superwerkzeugen" ist zu erwarten, dass der SEP im weitesten Sinne deutlich effizienter wird. Wir werden sehen, ob die SLI-Welle wirklich dazu führt.

SE-Techniken

Es gibt verschiedene Kategorien von SE-Techniken: Zum einen Techniken für das IT-Management im SEP im weitesten Sinne (z. B. [Boh12]), zum anderen für das Projektmanagement (z. B. RUP, XP, Scrum, Kanban, TCC) und schließlich für den SEP im engsten Sinne (z. B. OOD, TDD, DDD, BDD). Alle diese Techniken werden häufig nicht streng formalisiert angewandt. Vielmehr ist es hilfreich, ihre Daseinsberechtigung zu verstehen, ihre Vor- und Nachteile abwägen zu lernen und zu entscheiden, welche ihrer Aspekte in welchem Kontext wichtig sind. *TCC (Team Clean Coding)* ist eine Technik, die wir in Teil 3 dieses Artikels vorstellen werden.

Die menschliche Dimension „Motivation"

Die vierte Dimension ist von spezieller Natur, denn hier steht der „Faktor Mensch" im Mittepunkt. Antrieb allen menschlichen Handelns ist die Motivation. Steckt dahinter nur „Zuckerbrot und Peitsche"?

Die klassische Sicht

Im Zeitalter des Konsums schuf man den so genannten „Homo Oeconomicus" (vgl. [Wiki-a]): ein Menschenbild, das neben seinen Grundtrieben (Hunger, Fortpflanzung und soziale Bindung) als weiteres vorherrschendes Ziel die Gewinnmaximierung (für den Produzenten) bzw. die Nutzenmaximierung (für den Konsumenten) anstrebt. Dementsprechend herrschte im Management lange Zeit Einigkeit darüber, dass Arbeit an sich keinen Spaß mache und die Motivation, sie zu erledigen, folglich äußerer Reize bedürfe (vgl. [Pin10]). So könne der Arbeiter nur durch Belohnung (Lob, Boni, Karriere) bzw. Bestrafung (Tadel, Abmahnung) zur Ausführung bewegt werden: „Zuckerbrot und Peitsche" eben (vgl. [Wiki-d]).

Berufsgruppen wie Künstler und Wissenschaftler passten nicht so recht in dieses Bild, galten aber als Minderheit und konnten so diesem Weltbild nichts anhaben. Das änderte sich spätestens mit dem Aufkommen und der beträchtlichen Verbreitung der Open-Source-Bewegung. Softwareentwickler, die unentgeltlich an solchen Projekten mitarbeiten, stellen beileibe keine Randgruppe dar. Künstler, Wissenschaftler und Open-Source-Entwickler sind stark intrinsisch (siehe unten) motiviert.

Ursprünge von Motivation

Ihrem Ursprung nach können zwei Formen von Motivation unterschieden werden: intrinsisch und extrinsisch (vgl. [Wiki-b]). Intrinsische Motivation stammt zu 100 % aus dem Menschen selbst. Er handelt, weil sein Handeln für ihn einen Selbstzweck hat – einfach, weil er es für sich selbst will. Extrinsische Motivation beruht auf Faktoren seiner Umgebung (Stichwort „Zuckerbrot und Peitsche").

Eine neue Sichtweise

Ein wichtiges Ergebnis moderner Forschung lautet, dass der klassische Weg („Zuckerbrot und Peitsche") sich schädlich auf die intrinsische Motivation auswirken kann (vgl. [Pin10]). Werden beispielsweise Boni bzw. Mali ausschließlich unter *vorher festgelegten* Bedingungen vergeben, verschiebt sich das Interesse von „der Sache an sich" schnell zur alleinigen „Frage nach der Belohnung bzw. Bestrafung": Ein Homo Oeconomicus ist entstanden. Ein Lob, Dank oder Obolus bei guter Arbeit *im Nachhinein* hat hingegen stets einen positiven Effekt und passiert leider viel zu selten. Werden „Zuckerbrot und Peitsche" schonend eingesetzt, bleibt wahrscheinlich die intrinsische Motivation neben der extrinsischen erhalten, sodass sie sich sogar gegenseitig ergänzen.

Hersteller	ALM-Komplettlösung
HP	HP ALM, Quality Center, Performance Center
IBM	IBM Rational, Jazz Plattform
Microsoft	Visual Studio ALM, Team Foundation Server (TFS)
Oracle	Oracle Team Productivity Center
SAP	Solution Manager

Tabelle 3: SLI-Realisierung verschiedener Hersteller.

Motivation, besser zu werden, z. B.:
- Suchen und lesen.
- Reden und zuhören
- Tagungen, Fortbildungen und Veranstaltungen besuchen.
- Ausprobieren und testen.

Motivation, zu kommunizieren, z. B.:
- Sich präzise ausdrücken und aktiv zuhören.
- Fremde Meinungen akzeptieren lernen.
- Helfen beim Einrichten und Verbessern von Prozessen.

Motivation, verantwortungsvoll zu handeln, z. B.:
- Konsequent Regeln und Prozesse einhalten.
- Mitdenken und hinterfragen.
- Der Sache auf den Grund gehen.

Motivation, über sich selbst zu reflektieren, z. B.:
- Ist meine Meinung für andere nachvollziehbar?
- Ist mein Verhalten für andere akzeptabel?

Kasten 5: Grundlegende Ziele von Motivation in der SE.

Motivationsziele

Neben ihrem Ursprung gehört zu jeder Motivation auch ihr Ziel. In der SE sehen wir vier Ziele von Motivation als grundlegend an (siehe Kasten 5). Sie haben unserer Meinung nach fundamentalen Einfluss auf den beruflichen Werdegang (siehe auch Abbildung 5):

- Der Mensch startet neugierig und unvoreingenommen und beginnt zu lernen (*Phase 1*). In dieser Phase ist er in der Regel stark intrinsisch motiviert.
- Nach Jahren intensiver Berufserfahrung manifestiert sich eine Meinungsbildung (*Phase 2*). Dabei erlebt er, dass Rahmenbedingungen wie Budget und Termine oder spezielle Projektsituationen (vgl. [Voll] und Teil 2 dieses Artikels) den Projektalltag stark beeinflussen.
- Ein Teil der Mitarbeiter resigniert – manche so stark, dass sich eine Null-Bock-Stimmung manifestiert (*Phase 3a*). Diese Gruppe ist gar nicht mehr motiviert, weder in- noch extrinsisch.
- Ein anderer Teil in der Phase 2 bleibt engagiert und ist von sich überzeugt, „die Lage zu beherrschen“. So kennt man Fachlichkeit, Technik, hat vieles gelernt und versucht dogmatisch (zum Teil unerbittlich), mit seinen Vorstellungen und Ideen andere zu überzeugen. Für diese Gruppe besteht eine Frustrationsgefahr primär darin, bei den Teammitgliedern auf Widerstand zu stoßen. Mangelnde Anerkennung und eine subjektiv empfundene schlechte Projektsituation führen häufig zu Resignation (*Phase 3b*) – häufig verbunden mit Zynismus.
- Eine gute Entwicklung nehmen die Menschen, die lernen, dass sowohl für den persönlichen Erfolg als auch für den des Projekts zwei weitere Motivationen immer wichtiger werden: Die Motivation zu kommunizieren und die, über sich selbst zu reflektieren. Zusammen führen sie zur agnostisch-pragmatischen Phase (vgl. [Lin10]), in der eine Lösung nicht nur technisch pragmatisch, sondern auch psychologisch pragmatisch gefunden wird (*Phase 3c*).

In diesem Stadium hat der Entwickler Folgendes erkannt:

- Bei hinreichender Komplexität gibt es die einzig richtige Lösungen nicht, weil fast alles Vor- und Nachteile hat.
- Andere Meinungen sind zu tolerieren, selbst wenn man selbst überzeugt ist, eine bessere Lösung gefunden zu haben.
- Nachhaltig gute Lösungen lassen sich meist nur unter Berücksichtigung vieler Einflussfaktoren (d. h. mit viel Kommunikation) ermitteln.
- Veränderungen werden am effektivsten mit der Unterstützung von Gleichgesinnten erreicht (d. h. mit viel Kommunikation).
- Das gezielte Vermeiden von Kommunikation stellt eine große Gefahr für Ineffizienz und Unproduktivität dar und kann zu großen wirtschaftlichen Schäden führen.

Hier angelangt weitet sich der Blick, der dann die Perspektiven unterschiedlicher Stakeholder einschließt. Ohne Dogma-Fesseln können Entscheidungen gefunden werden, die eine breitere Akzeptanz finden. Kasten 5 soll helfen, diese Entwicklungsstufe, die wir mit diesem Artikel zur Diskussion stellen, zu erreichen.

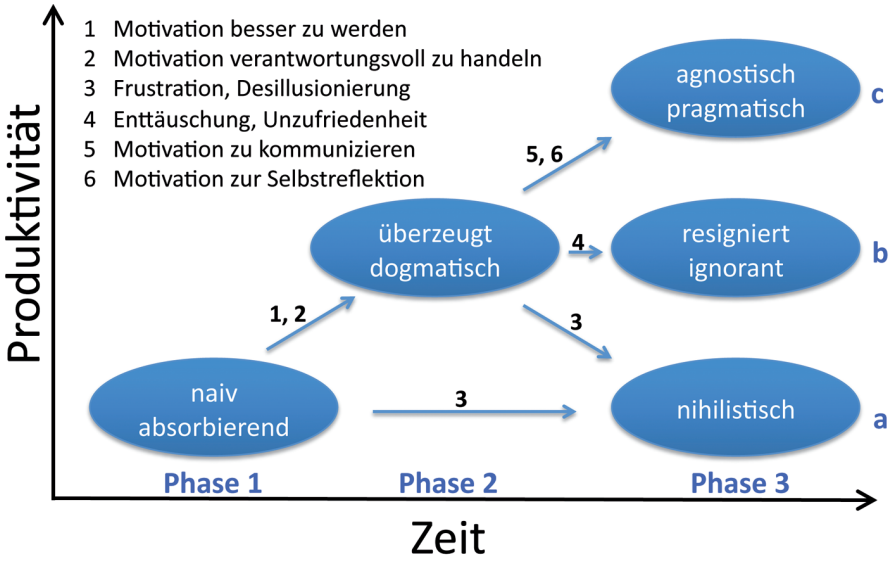

Abb. 5: Phasen im beruflichen Werdegang von Menschen, die mit komplexen Systemen arbeiten (vgl. [Lin10]).

Fazit

In der Naturwissenschaft ist die Suche nach der „Grand Unified Theory“ (der Weltformel) gescheitert. Die Weltformel liegt wohl auch außerhalb der menschlichen Möglichkeiten. Gleiches können wir auch für die kosmologische Weltformel der SE annehmen, denn die Kräfte, die in den verschiedenen Dimensionen herrschen, sind zu verschieden, um sie zu vereinheitlichen. Deshalb suchen wir in der SE nicht nach dem allgemeingültig richtigen Weg. Stattdessen halten wir es für wichtig, den Überblick zu erlangen und zu bewahren, welche Einflussfaktoren

die tägliche Arbeit in der SE bestimmen. Diese Einflussgrößen lassen sich in vier verschiedene Dimensionen einordnen:

Klar definierte, rundlaufende Prozesse, breites und tiefes technisches Wissen, gute handwerkliche Fertigkeiten im Umgang mit Tools und Techniken sowie positiv motivierte Teammitglieder: Wer in allen vier Dimensionen seine Kräfte ausgewogen einzusetzen vermag, wird die Materie „Software" erfolgreich formen und davon auch persönlich profitieren.

Im zweiten Teil dieses Artikels möchten wir zeigen, wie die Raumzeit der SE durch Softwareschmutz gekrümmt und verbogen wird. ■

Literatur & Links

[Ale04] A. Alexandrescu, H. Sutter, C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, Addison-Wesley Professional 2004

[All02] E. Allen, Bug Patterns in Java, Apress 2002

[Bec02] K. Beck, Test Driven Development: By Example, Addison-Wesley Professional 2002

[Blo08] J. Bloch, Effective Java: A Programming Language Guide, Addison-Wesley Professional 2008

[Boh12] M. Bohlen, Softwarearchitektur auf die schlanke Art, Amazon Media, Oktober 2012

[Cle] Clean Code Developer, Die Clean Code Developer Grade, siehe: http://www.clean-code-developer.de/Grade.ashx

[Dev08] DevAdvice.com, C# Nuggets, 2008, siehe: http://aspadvice.com/blogs/rbirkby/archive/2008/05/12/Code-Smells-in-C_2300_.aspx

[Fea11] M.C. Feathers, Effektives Arbeiten mit Legacy Code, mitp 2011

[Fow99] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman 1999

[Fre10] S. Freeman, N. Pryce, Growing Object-Oriented Software, Guided By Tests, Addison-Wesley Professional 2010

[Hru12] P. Hruschka, G. Starke, Kolumne: Knigge für Softwarearchitekten, in: Javamagazin 11/2012

[Kno12] K. Knoernschild, Java Application Architecture: Modularity Patterns with Examples Using OSGi, Addison Wesley 2012

[Lin10] J. Lind, A. Knecht, Der Weg zur Erleuchtung: Untersuchung zur Ontogenese des Software-Architekten, in: OBJEKTspektrum 3/2010

Man04] T. Manjaly, C# Coding Standards and Best Programming Practices, 2004, siehe: http://www.codeproject.com/Articles/8971/C-Coding-Standards-and-Best-Programming-Practice

[Mar08] R.C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall 2008

[nsz10] nsz, Dark side of C++, 2010, siehe: http://port70.net/~nsz/16_c++.html

[Par72] D.L. Parnas, On the Criteria To Be Used in Decompsition Systems into Modules, Carnegie-Mellon University, Dezember 1972, Vol 15, No 12

[Pin10] D.H. Pink, Drive: Was sie wirklich motiviert, Ecowin Verlag 2010

[Rup09] C. Rupp & die SOPHISTen, Requirements-Engineering und -Management, Carl Hanser Verlag 2009 (5. Auflage)

[Rup10] C. Rupp, R. Joppich, C. Wünch, Molekulares Requirements-Engineering (M-RE) – Der Bauplan einer perfekten Anforderung, 2010, siehe: http://www.ap-verlag.de/Online-Artikel/20100708/20100708o%20Sophist%20Molekulare%20Anforderungen.htm

[Sch11] H. Schaefer, Illusions about software testing, 2011, siehe: http://www.es.sogeti.com/PageFiles/173/Illusions about software testing_Hans Schaefer.pdf

[Sta09] G. Starke, Effiziente Software Architekturen. Ein praktischer Leitfaden, Carl Hanser Verlag 2009

[Tas] Tasktop Technologies, Software Lifecycle Ingegration, siehe: http://www.tasktop.com/softwarelifecycleintegration

[Voll] Informatikbüro Jörg Vollmer, Umfrage „Was stört Sie im IT-Alltag am meisten?", siehe: http://umfrage.clean-coder.de

[Wiki-a] Wikipedia, Homo oeconomicus, siehe: http://de.wikipedia.org/wiki/Homo_oeconomicus

[Wiki-b] Wikipedia, Motivation, siehe: http://de.wikipedia.org/wiki/Motivation

[Wiki-c] Wikipedia, Toolchain, siehe: http://de.wikipedia.org/wiki/Toolchain

[Wiki-d] Wikipedia, Zuckerbrot und Peitsche, siehe: http://de.wikipedia.org/wiki/Zuckerbrot_und_Peitsche

[Wil12] A. Willert, Application Lifecycle Management – Tool oder Vorgehen?, in: OBJEKTspektrum Sonderbeilage CALM, Winter 2012, siehe: http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/os/2012/Sonderheft_CALM/SH_CALM_2012_gesamt.pdf

[You] Youtube, Open Services for Lifecycle Collaboration (OSLC), siehe: http://www.youtube.com/watch?v=B2vqL8fujgE