http://dx.doi.org/10.1109/MS.2013.101
http://www.infoq.com/articles/measuring-architecture-sustainability
http://de.wikipedia.org/wiki/Digitale_Nachhaltigkeit

Nachhaltige Softwareentwicklung: Mehr als API-Ökonomie und Anpassbarkeit

Viele Softwareprodukte stecken längst in der Wartungsfalle fest, da das Produkt oft auf Plattformen mit Werkzeugen entwickelt wurde, die heute gar nicht mehr existieren. Damit eine Software den aktuellen fachlichen und technologischen Anforderungen genügt, muss sie permanent angepasst werden. Das Prinzip der Nachhaltigkeit kann, um unnötige Ressourcenverschwendung und teure Migrationsprojekte zu vermeiden, auch auf die Softwareentwicklung übertragen werden. In diesem Artikel werden die wesentlichen Aspekte, die eine nachhaltige Softwareentwicklung unterstützen, vorgestellt.

Nachhaltigkeit: Keine wilde Softwarezucht

Wer Unternehmensanwendungen entwickelt, möchte diese über Jahre hinweg nutzen. Die Erfahrungen der letzten Jahre haben gezeigt, dass die Kosten für die Wartung die eigentlichen Entwicklungsaufwendungen übersteigen und damit der Einsatz der Software oft nicht mehr wirtschaftlich ist. Hier kommt der Begriff der Nachhaltigkeit ins Spiel.

Als Folgen des Dreißigjährigen Krieges herrschte im 17. Jahrhundert in ganz Europa ein großer Holzmangel. Deswegen erstellte der Oberberghauptmann Hans Carl von Carlowitz (1645–1714) aus Freiberg in Sachsen in seinem Buch „Sylvicultura oeconomica oder Anweisung zur wilden Baum-Zucht“ vor über 300 Jahren eine Anleitung zur kontinuierlichen, beständigen und nachhaltenden Nutzung des Holzes. Gleichzeitig beschrieb er damit als Erster das Prinzip der Nachhaltigkeit.

Ähnlich wie der Wald ist die Softwareentwicklung heute keine Monokultur mehr, sondern ein komplexes Ökosystem mit vielen Beteiligten. Angestoßen durch den Brundtland-Bericht (vgl. [Bru87]) der Vereinten Nationen von 1987 („Development that meets the needs of the present without compromising the ability of future generations“) wurde das Konzept der digitalen Nachhaltigkeit entwickelt, das die langfristig orientierte Herstellung und Weiterentwicklung von digitalen Wissensgütern ermöglicht.

Die knappe Ressource für uns heute ist der Mensch mit seinem Wissen sowie seinen Erfahrungen und Fähigkeiten. Um diese dauerhaft zu nutzen und zu entwickeln, muss sich die Art ändern, in der wir Softwareentwicklung betreiben. Dabei müssen die Interessen von Ökonomie, Ökologie und Sozialem miteinander in Einklang gebracht werden (**siehe Abbildung 1**).

Je mehr Software in die Wirtschaftsprozesse eines Landes eingebunden ist, umso wichtiger wird der Faktor Open-Source (vgl. [Sow12]). Ohne Open-Source-Komponenten sind heute weder das WWW, noch eine kommerzielle Softwareentwicklung wirtschaftlich denkbar. Gerade wenn die Software in industriellen Produktionsprozessen oder embedded eingesetzt wird, ist es entscheidend, nicht nur den Quellcode zu besitzen, sondern den gesamten Erstellungsprozess dauerhaft durchführen zu können. Im Rahmen des Produktlebenszyklus-Managements müssen dafür frühzeitig geeignete Maßnahmen getroffen werden.

Im Open-Source-Bereich werden speziell reservierte Versionen und Editionen (*Langzeit-Unterstützung – LTS*) (vgl. [Wik-b]) von Software mit einem deutlich verlängertem Release-Lebenszyklus erstellt. Auch wenn die Art und Häufigkeit von Software-Updates abnimmt, ist ein Mindestmaß an Unterstützung gewährleitstet. So gibt es für mehrere Linux-Versionen LTS-Editionen.

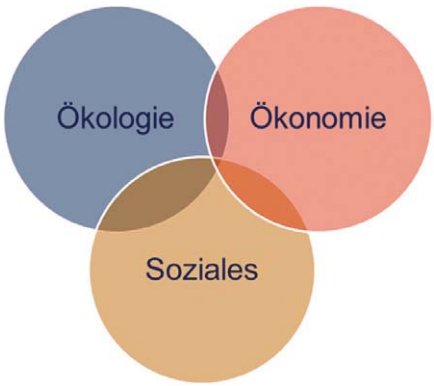

Abb. 1: Nachhaltigkeit in der Softwareentwicklung harmonisiert Ökologie, Ökonomie und Soziales.

Hier spielt der Begriff der Kompatibilität eine große Rolle. Dabei ist eine Abwärtskompatibilität leichter zu erreichen als eine Aufwärtskompatibilität. So ist zum Beispiel UTF-8 abwärtskompatibel zu 7-Bit-ASCII, um auch damit erstellte Dokumente verarbeiten zu können. Web-Browser stellen Aufwärtskompatibilität her, indem sie beispielsweise neue HTML-Tags, die sie nicht kennen können, ignorieren.

Es ist jedoch auch ein API-Lebenszyklus-Management (vgl. [Jac12], [Cut14]) nötig. Werden z. B. öffentlich genutzte APIs durch neuere abgelöst wird, sollten diese für einen ausreichend langen Zeitraum als solche gekennzeichnet werden. Inkompatibilitäten sollten an einer prominenten Stelle bekannt (z. B. durch Major-Release-Nummer) und am besten mit Werkzeugunterstützung überprüfbar gemacht werden.

Neben der Kompatibilität der Programmierschnittstellen (*Application Programming Interface – API*) ist hier auch die Kompatibilität der Binärschnittstellen (*Application Binary Interface – ABI*, vgl. [Wik-a]) wichtig, um eine Austauschbarkeit und Portabilität zu gewährleisten. Die ABI ist besonders dann wichtig, wenn sich die Produktionsumgebung hinsichtlich Betriebssystem- und Prozessorarchitektur geändert hat. Die ABI ist aber auch dann wichtig, wenn mit einem Quellcode mehrere unterschiedliche Plattformen versorgt werden sollen.

Nachhaltigkeit: Abschätzen und Messen

Eine gute Basis für die langfristige Nutzung von Software sind nicht-funktionale Qualitätsmerkmale. Nach der ISO-9126-Norm gehören dazu beispielsweise die folgenden Qualitätsmerkmale:

Abb. 2: ISO-Merkmale für eine nachhaltige Softwarequalität.

- Interoperabilität
- Fehlertoleranz
- Wartbarkeit
- Änderbarkeit
- Analysierbarkeit
- Modifizierbarkeit
- Stabilität
- Testbarkeit
- Übertragbarkeit
- Anpassbarkeit
- Austauschbarkeit

Für die Bereiche Wartbarkeit, Änderbarkeit, Anpassbarkeit und Modifizierbarkeit gibt es ausgereifte objektorientierte Metriken, um den Erfüllungsgrad zu messen und geeignete Korrekturmaßnahmen einzuleiten. Meist werden diese Metriken mit der Wiederverwendbarkeit in Verbindung gebracht, die jedoch nicht automatisch dadurch erreicht wird. Wichtiger als Wiederverwendbarkeit sind oft die Austauschbarkeit, Übertragbarkeit und die Interoperabilität von Softwarekomponenten für eine nachhaltige Softwareentwicklung (**siehe Abbildung 2**). Eine wichtige Voraussetzung dafür sind die Testbarkeit, Fehlertoleranz, Analysierbarkeit und Stabilität der verwendeten Software. Leider ist es hier schon schwieriger, allgemeine Metriken zu finden, sodass diese selbst definiert werden müssen.

Für die Übertragbarkeit, Analysierbarkeit und die Interoperabilität sind neben der Einhaltung von Standards auch die Verwendung von den einheitlichen Querschnittskonzepten, wie z. B. Logging, Internationalisierung (i18n) oder Lokalisierung (l10n) wichtig. Auf die Produktqualität hat auch die Prozessqualität einen großen Einfluss. Ein eingespieltes Entwicklungsvorgehen mit von allen gelebten Regeln, eine Kultur des Wissensaustausches und der Fehlertoleranz sind dabei eine wichtige Basis. Für einen guten Wartungsprozess muss das Wissen im Team über die Software, die dabei verwendeten Pflegeprozesse sowie die wichtigen Architekturprinzipien und -entscheidungen frisch gehalten werden. Nachhaltigkeit betrifft alle Phasen des Softwareentwicklungsprozesses. Anforderungsmanagement, Architektur, Design, Implementierung, Testen und Wartung sind dabei neben den unterstützenden Prozessen auf die Nachhaltigkeitsprinzipien ausgerichtet. Diese hat Heiko Koziolek aus der Projektpraxis bei ABB (vgl. [Koz13]) schön zusammengefasst (**siehe auch Abbildung 3**). Um die Anpassbarkeit oder Änderbarkeit, als Ziel auch zu erreichen, ist es hilfreich, hier explizite mögliche Wachstums- und Änderungsszenarien (**siehe Tabelle 1**) frühzeitig zu entwickeln und diese von Zeit zu Zeit bei Entscheidungen zu überprüfen oder bei geänderten Rahmenbedingungen explizit anzupassen. Dadurch können Entscheidungen besser kontrolliert und unter konkreteren Bedingungen getroffen werden. Viele Codemetriken und agile Vorgehensweisen greifen hier zu kurz und brauchen Ergänzungen (vgl. [Tat05], [Wik-d]). Qualitätsmetriken mit Qualitätsszenarien helfen, die Qualitätsmerkmale, wie Wartbarkeit, Erweiterbarkeit, Anpassbarkeit und Portabilität besser zu erreichen. Neben einer größeren Klarheit bei Architekturentscheidungen können durch Szenarien, je konkreter messbar diese sind, umso besser Tests oder Validierungen zur Risikominimierung durchgeführt werden. Ein einfaches Wachstumsszenario wäre: Wie wirkt sich eine Verdopplung der Benutzerzahl auf die Konfiguration und die Deployment-Topologie der Anwendung aus, wenn die Antwortzeiten gleich bleiben sollen?

Bei historisch gewachsenen Systemen hat man häufig einen Mix aus Technologien, Prinzipien und Praktiken. Ein kontinuierlicher und unabhängiger Qualitäts- und Kontrollprozess hilft hier mit automatisierten Metrikprüfungen, damit das Projekt in der Spur bleibt. Neben einem aktuellen System-Kontextdiagramm sollte nachvollziehbar sein, welche Anforderungen, warum und wie umgesetzt wurden. Die Wahrheit nur im Quellcode ist nicht nur mühsam, sondern blendet auch viele wichtige Aspekte aus. Dennoch ist bei den Codemetriken weniger oft mehr. Trotz immer besser werdender Werkzeugunterstützung bei der Auswertung und Visualisierung von Metriken sind wenige einfache, dafür aber konsequent angewandte Regeln vielen und ausgefeilten überlegen. Wichtiger als eine Zeitpunktbetrachtung ist die historische und kontinuierliche Ent-

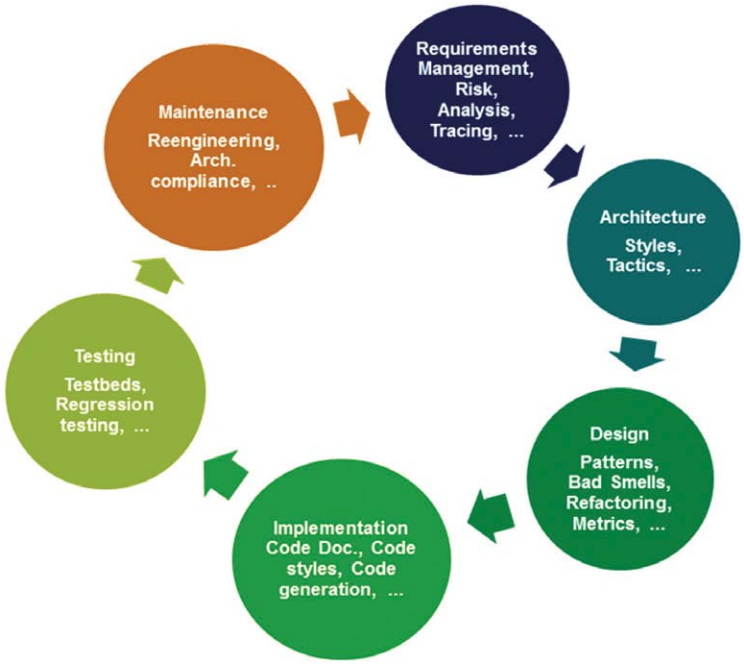

Abb. 3: Sustainability Guidelines for Long-Living Software Systems (Quelle: [Koz13]).

Art des Evolutionsszenariums	Name	Wahrscheinlichkeit
Wachstum	Ausbau der Produktionsumgebung auf 10 Knoten.	mittel
Änderung	GUI-Framework austauschen.	gering
Änderung	Fremdbibliothek auf Major-Version aktualisieren.	hoch
Wachstum	Benutzerzahlen verdoppeln sich.	mittel

Tabelle 1: Evolutionsszenarien.

wicklung der Metriken. Dadurch können Trends frühzeitig erkannt und mit Daten, z.B. aus dem Version- oder Bugtracking-System, verglichen werden.

Um Trends zu beobachten, schlägt Heiko Koziolek (vgl. [Koz13]) hier Metriken für Module, Methoden, Zustände, Modul-Nutzung, API-Nutzung und Schichtverletzungen vor (**siehe Abbildung 4**).

Beim Design einer extern angebotenen API sollte ein besonderes Augenmerk auf Einfachheit, Unabhängigkeit und Robustheit gelegt werden. Dies wird besonders für die in der „Public Cloud" angebotenen Web-APIs immer wichtiger, da deren Nutzung kontinuierlich zunimmt. Eine gute Richtschnur ist hier das 1981 von Jon Postel im RFC793 beschriebene „TCP Robustness Principle": „Be conservative in what you do, be liberal in what you accept from others".

Für nicht-fachliche Querschnittsfunktionen hat sich die Verwendung von Open-Source-Komponenten mit einer liberalen Lizenz, wie die von Apache, Eclipse oder MIT, bewährt. Damit werden nicht nur Entwicklungs-, Test- und Wartungsaufwände eingespart, sondern es wird auch die Stabilität und Robustheit gefördert, da sich diese bereits mehrfach im Einsatz bewährt haben. Dank ihrer Spezialisierung lassen sich Open-Source-Komponenten oft recht einfach integrieren oder auch leicht austauschen.

Ökologie trifft Ökonomie: Nutzen statt besitzen

Eine Grundsatzfrage, die man sich bei der Entwicklung und noch mehr bei der Weiterentwicklung der Software stellen sollte, lautet: Will oder muss ich bestimmte Funktionen selbst entwickeln oder kann ich bestehende Funktionen nutzen und integrieren. Um Ressourcen zu sparen, geht der Trend in der Konsumwelt des „Shareconomy" immer mehr vom Besitzen zum Nutzen. So können die oft konkurrierenden Ziele von Ökologie und Ökonomie miteinander in Einklang gebracht werden.

Selbst im Betrieb setzen sich immer mehr Cloud-Angebote durch. Dabei werden die vielen unterschiedlichen *PaaS*-Angebote (*Platform-as-a-Service*) inzwischen mit xPaaS zusammengefasst Das WWW ist längst zum größten Anbieter von maschinennutzbaren APIs geworden. Hier stehen weniger einzelne Bibliotheken als spezialisierte Dienste, wie z.B. zur Visualisierung, Suche, Zwischenspeicherung, Auswertung, Überwachung oder Nutzung von Datendiensten, etwa zur Validierung oder Geo-Lokation, im Vordergrund. Vor einer Nutzung muss man jedoch das Risiko abwägen, was passiert, wenn der Dienst nicht mehr oder nicht mehr in der genutzten Form zu Verfügung steht.

Abweichend von Bibliotheken, wo man häufig sogar den Quellcode einsehen kann, sind externe Dienste meist Black-Box-Systeme. Hier ist es hilfreich, wenn es Zusatzinformationen über die Servicegüte und die API-Evolution (vgl. [API09]) gibt. Für die Nutzung von externen entfernten APIs sollte man die in [Nyg07] beschriebenen Stabilitätsmuster und Antimuster von Michael T. Nygard beachten.

Stabile APIs und eine Versionierungsstrategie für größere API-Änderungen sind für die Langzeit-Unterstützung eines Softwareprodukts immer wichtiger. Im eigenen

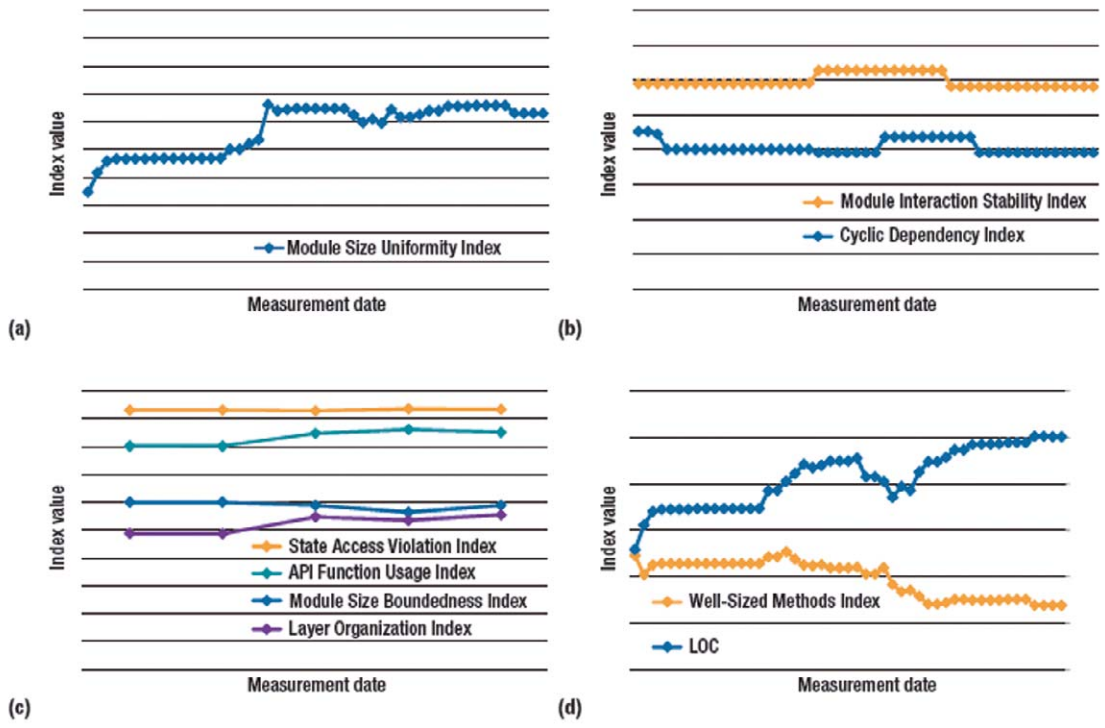

Abb. 4: Überwachen wichtiger Metriken für nachhaltige Architektur (Quelle: [Koz13]).

Projekt sollte man zwar immer nur eine Version einer Bibliothek verwenden. Doch kann es passieren, dass in der Ablaufumgebung oder in größeren Projekten mehrere Versionen zum Einsatz kommen. Um einen Parallelbetrieb von neuer und alter API zu erreichen und Konflikte möglichst zu vermeiden, sollte man sich einheitliche Versionierungsverfahren für API-Änderungen überlegen. Apache-Projekte verwenden hier unterschiedliche Versionierungsstrategien.

Eine Versionsnummer als Postfix verwendet das Apache-Derby-Projekt. Um Klassen, die für eine JDBC-4.x-Unterstützung zuständig sind, von Klassen, die ältere JDBC-Versionen unterstützen, zu unterscheiden, wird die Zahl 40 an den Klassenname angehängt. Diese Variante für Versionsnummern findet sich auch im Produkt „HttpComponents-Client for Android" von Apache, die in den ab Version 4.3 hinzugekommene Klassen ein Postfix HC4, also `org.apache.http.**.*HC4`, anhängt, um Konflikte mit älteren HttpComponents-Client-Klassen in der Android-Laufzeitumgebung zu vermeiden.

Eine andere Strategie fahren die Apache-Commons-Produkte „lang" und „math", die neuere API-Änderungen bereits im Paketnamen mit einer angehängten Versionsnummer, also `lang3` bzw. `math3` kenntlich machen.

Auf Änderungen einstellen

Alles hat seine Zeit. Deswegen sollte man sich rechtzeitig von Dingen trennen, die auf einer früheren falschen Annahme oder Umsetzung beruhen. Sollte sich beim eigenen Produkt mit der Zeit herausstellen, dass die Wartungsaufwendungen und Änderungszeiten unverhältnismäßig steigen, sollte man über Reparaturmaßnahmen nachdenken. Dabei sollte immer abgewogen werden, ob ein größerer Aufwand oder ein größeres Risiko für eine Reparatur gegenüber einem Austausch und Ersatz einer anderen Komponente gerechtfertigt ist.

Eine Frage, die man sich beim Weiternutzen bestehender Produkte stellen sollte, ist, ob diese auch in Zukunft den gewachsenen und veränderten Anforderungen standhalten. Können diese angepasst oder erweitert werden oder müssen sie grundsätzlich umgebaut und repariert werden? Wenn riskante und teure Umbauten nötig sind, sollte man überlegen, ob man die Funktionen oder Teile davon nicht ersetzen kann. Dabei hat sich gezeigt, dass Refacoring-Maßnahmen im Kleinen oft und erfolgreich, größere Architektur-Refactorings jedoch nur selten und gezielt angewendet werden.

Eine gezielte regelmäßige Aktualisierung der Software erleichtert spätere Aktualisierungen. Für einen bereits vorhandenen Continuous Delivery ist es einfach, diesen zu erweitern, um parallel zu den aktuell unterstützen, bereits zukünftig zu verwendende Versionen zu testen. Oft lassen sich damit größere Migrations- und Modernisierungsschritte vermeiden. Notwendige Änderungen sollen frühzeitig angegangen und besser isoliert betrachtet werden. Leichen im Design und Bomben im Code-Keller entfernt man sofort, bevor sie größeren Schaden anrichten. Dabei sollte man an den Satz von Marc Aurel denken „Beachte immer, dass nichts bleibt, wie es ist, und denke daran, dass die Natur immer wieder ihre Formen wechselt."

Soziale Strukturen beeinflussen das Ergebnis

Das Problem der Altsoftware sollte von seiner Entstehung her gesehen werden. Nach dem Gesetz von Conway (vgl. [Con68]) werden die Strukturen von Systemen durch die Kommunikationsstrukturen der sie umsetzenden Organisationen beeinflusst. In einer Microsoft-Research-Studie wird dieses Gesetz für Windows Vista bestätigt. Die dort entstandene Komplexität und Fehlerrate lassen sich direkt aus der dafür zuständigen Organisationseinheit bei Microsoft ableiten (vgl. [Nag08]).

Literatur & Links

[API09] APIDesignPatterns, 2009, siehe: `http://wiki.apidesign.org/wiki/APIDesignPatterns`

[Boo13] G. Booch, On Computing, 2013, siehe: `http://www.computer.org/portal/web/computingnow/oncomputing/-/blogs/in-defense-of-boring`

[Bru87] Brundtland-Bericht, 1987, siehe: `http://www.bne-portal.de/was-ist-bne/grundlagen/brundtland-bericht-1987`

[Car13] H.C. von Carlowitz, Sylvicultura oeconomica oder Anweisung zur wilden Baum-Zucht, 1713 (oekom 2013)

[Con68] M.E. Conway, How Do Committees Invent?, 1968, siehe:

[Cut14] Cutter Benchmark Review, Herding a Hurricane: Implementing and Managing API Programs, Vol. 14, No. 1, 2014

[IEE90] IEEE Standard Glossary of Software Engineering Terminology 610.12-1990, siehe: `http://dx.doi.org/10.1109/IEEESTD.1990.101064`

[Jac12] D. Jacobson, G. Brail, D. Woods, APIs: A Strategy Guide, O'Reilly 2012

[Koz13] H. Koziolek, D. Domis, T. Goldschmidt, P. Vorst, Measuring architecture sustainability, Software IEEE, 30(6), November 2013

[Nag08] N. Nagappan, B. Murphy, V. Basili, The Influence of Organizational Structure On Software Quality: An Empirical Case Study, 2008 siehe: `http://research.microsoft.com/apps/pubs/default.aspx?id=70535`

[Nyg07] M.T. Nygard, Release It: Design and Deploy Production-Ready Software, The Pragmatic Programmers 2007

[Pie11] F. Pientka, Unter Indianern: Dem Geheimnis der Apache Software auf der Spur, in: OBJEKTspektrum 10/2011

[Sow12] S.K. Sowe, G. Parayil, A. Sunami, Free and open source software and technology for sustainable development, United Nations University Press 2012

[Tat05] K. Tate, Sustainable Software Development: An Agile Perspective, Pearson Education 2005

[Wik-a] Wikipedia, Binärschnittstelle, siehe: `http://de.wikipedia.org/wiki/Bin%C3%A4rschnittstelle`

[Wik-b] Wikipedia, Long-term support, siehe `http://en.wikipedia.org/wiki/Long-term_support`

[Wik-c] Wikipedia, Digitale Nachhaltigkeit, siehe: `http://de.wikipedia.org/wiki/Digitale_Nachhaltigkeit`

[Wik-d] Wikibooks, Software Engineering with an Agile Development Framework/Whole process/Sustainability, siehe: `http://en.wikibooks.org/wiki/Software_Engineering_with_an_Agile_Development_Framework/Whole_process/Sustainability`

`www.melconway.com/Home/Conways_Law.html`