Memorandum

Wo steht, wohin geht Software-Engineering in Wissenschaft & Wirtschaft, in Theorie & Praxis?

Workshop am 6./7. Februar 2012
am Institute for Advanced Study
der Technischen Universität München
in Garching

Vorbemerkung

Der Begriff und das Gebiet *Software-Engineering* wurden 1968 begründet, auf einer internationalen Tagung in Garmisch. Mehr als 40 Jahre danach ist es sicherlich angemessen, das Erreichte zu bewerten, zu fragen, ob das Gebiet in seiner wissenschaftlichen Entwicklung womöglich ausgereizt ist, wie weit es in der Praxis gekommen ist, welche Herausforderungen offen sind und somit Ziele der künftigen Entwicklung in Theorie und Praxis sein sollten.

Diesen grundsätzlichen Fragen widmete sich ein Workshop, den die Ernst-Denert-Stiftung für Software-Engineering an der Technischen Universität München veranstaltet hat. In einem kleinen Kreis von Experten aus Wissenschaft und Wirtschaft wollten wir – das Kuratorium der Stiftung − einen Versuch wagen, die Position unseres Fachs zu klären. Mit diesem Ziel traf sich Anfang Februar 2012 eine von uns eingeladene Gruppe von Wissenschaftlern und Praktikern. Der Workshop hatte 20 Teilnehmer (Namensliste siehe unten) und dauerte zwei halbe Tage. Fünf Teilnehmer haben auf Bitte der Veranstalter umfangreichere Status-quo-Darstellungen vorgetragen. Alle Teilnehmer konnten mit kurzen Impuls-Statements ihre Position zu ihnen wichtigen Themen darstellen und diskutieren.

Das vorliegende Memorandum dokumentiert die gewonnenen Einsichten. Mit diesem Ergebnis wendet sich die Gruppe an die Fachöffentlichkeit, an Software-Entwickler, an das Management von Software-Unternehmen, an CIOs, an Software-Wissenschaftler sowie an Technologie- und Wissenschaftspolitiker.

München, im November 2012

Manfred Broy
Ernst Denert
Eike Jessen
Florian Matthes

Inhaltsverzeichnis

1 Software und ihre Entwicklung

Bedeutung von Software

Software durchdringt alle Bereiche des Lebens: Arbeit, Freizeit, Gemeinwesen. Sie dient der Organisation praktisch aller Unternehmen und unterstützt deren Prozesse, steuert technische Geräte, ist Teil unserer Infrastrukturen, Assistenzwerkzeug und selbst Produkt. Neuartige Produkte, Verfahren, Wissenssammlungen erhalten ihren innovativen Charakter in hohem Maße durch Software-realisierte Funktionen. Software ist eine junge Technik mit einer enormen Innovationsgeschwindigkeit. Sie hat ein weithin akzeptiertes, genügend hohes Qualitätsniveau erreicht und genießt Vertrauen, auch wenn dies nicht immer gerechtfertigt ist. Software hat eine herausragende Rolle als Wirtschaftsfaktor.

Deshalb kommt es sehr auf die Fähigkeit an, Software professionell zu entwickeln. Trotz beträchtlicher Fortschritte auf diesem Gebiet ist dessen Stand noch nicht in jeder Hinsicht befriedigend. Es sind weitere Anstrengungen erforderlich, um einerseits mit bereits erforschten und bewährten Verfahren die Software-Erstellung in der Praxis zu verbessern und um andererseits Methoden und Werkzeuge weiter zu erforschen und zu entwickeln. Nur so kann die Anforderung einer Informations- und Wissensgesellschaft erfüllt werden, Software in genügendem Umfang, preiswert und mit hoher Qualität verfügbar zu haben.

Wesen der Software-Entwicklung

Softwaretechnik ist eine junge Disziplin, verglichen mit anderen Fächern wie etwa den klassischen Ingenieurwissenschaften oder der Medizin. Im Gegensatz zu diesen ist Softwaretechnik losgelöst von direkten physikalischen oder biologischen Randbedingungen und hat einen entsprechend größeren Gestaltungsfreiraum.

Entwicklung und Wartung von Software sind kostspielig und werden oft unzureichend beherrscht. Die Unterschiedlichkeit ihrer vielen Methoden und Werkzeuge, deren Nutzen oft nicht einhellig und erst recht nicht objektiv beurteilt wird, erschwert die Zielsetzung und Bewertung in Forschung, Entwicklung und Ausbildung.

Anders als in verwandten Gebieten koexistieren und interagieren Softwaresysteme oft lange Zeit mit überkommenen, aber der Veränderung unterworfenen System-Umgebungen; vielfach ist die Anpassbarkeit daran ein wichtiges Gütekriterium für Software.

Die weitgehende Gestaltungsfreiheit bei der Software-Entwicklung verleitet dazu, immer wieder neue Lösungen zu realisieren, statt auf frühere Produkte als verlässliche Bausteine zurückzugreifen. Dadurch wird wertvolle Erfahrung nicht genutzt, und die Disziplin lernt nicht wirkungsvoll. Es fehlt an Beständen erprobter, wohl dokumentierter Software-Bausteine. Obwohl es einen weitgehend anerkannten Satz von Methoden und Werkzeugen gibt, wird deren Anwendung oft nicht strikt durchgesetzt, weil Entwickler, Manager oder Kunden ein informelles Vorgehen ausreichend finden, oft mit gravierenden langfristigen Problemen. Ferner führen diese Kenntnis- und Führungsschwächen dazu, dass Erfahrungen in Methoden und Werkzeugen nicht systematisch aufgebaut und weitergegeben werden. Nicht zuletzt deshalb wird das Gebiet immer wieder von Moden heimgesucht, die unter diesen Umständen nicht schnell bewertet und gezielt verworfen oder nutzbar gemacht werden können.

Eine weitere Eigentümlichkeit in der Entwicklung von Software ist die Heterogenität der Ausbildung. Der Anteil von Mitarbeitern mit Software-spezifischer Ausbildung, etwa aus der Informatik, steigt zwar, aber nur allmählich. Indessen differenziert sich das Software-Engineering fachspezifisch, und es werden Fachleute der Anwendungsgebiete mit einer ausreichenden Ausbildung in Software-Engineering gebraucht. Softwareprojekt-Management erhält vielfach nicht den erforderlichen Stellenwert in der Ausbildung.

Software-Engineering

Software-Engineering ist − wie der Name schon sagt – eine Ingenieurdisziplin und befasst sich mit den Vorgehensweisen, Methoden, Techniken und Werkzeugen zum Bau von Softwaresystemen, es umfasst Softwaretechnik und Management. Es bedarf auch eines ausreichenden Wissens über die fachlichen Erfordernisse der Anwendungsdomänen.

Über die Entwicklung von Software in Projekten hinaus gehören zum Software-Engineering ihre langfristige Evolution und Wartung, der Betrieb von Softwaresystemen sowie die strategische Planung und Gestaltung von Software und Softwarelandschaften in Unternehmen.

Es wurde und wird gigantisch viel Software in aller Welt entwickelt, mit großem Nutzen und erheblichen Auswirkungen. Das ging und geht nicht ohne ein brauchbares Software-Engineering. Dieses Gebiet ist in seinen gut 40 Jahren weit gekommen. Das Wort von der „Softwarekrise“ war lange Zeit in aller Munde, heute kommt es nicht mehr vor. Es wurde viel erreicht, europäische Beiträge, auch deutsche, spielten eine wesentliche Rolle.

2 Software-Entwurf

Unter Software-Entwurf verstehen wir alle Tätigkeiten, Überlegungen und deren Dokumentation, die vor der Realisierung der Software durch Programmcode erfolgen. Dabei unterscheiden wir die Spezifikation der Anforderungen an die Software, also die Außensicht (das „Was“) des Auftraggebers bzw. Anwenders, und ihre technische Gestaltung, die Innensicht (das „Wie“).

2.1 Anforderungen

Es ist allgemein anerkannt, dass die möglichst zutreffende und präzise Festlegung der Anforderungen an eine Software von entscheidender Bedeutung ist für den Erfolg ihrer Entwicklung und Einführung. Diese Einsicht ist im Laufe der Zeit gewachsen, nicht zuletzt aufgrund schlechter Erfahrungen und wirtschaftlicher Verluste im Fall ihrer Missachtung. Die *Anforderungsanalyse* und *-spezifikation* (Requirements-Engineering) ist zu einer der Schlüsseldisziplinen des Software-Engineerings geworden.

Man unterscheidet *funktionale* und *nichtfunktionale Anforderungen*. Erstere definieren die Funktion der Software im Wechselspiel mit ihrer Umgebung, sei es Mensch oder Maschine. Nichtfunktionale Anforderungen beschreiben quantitative Leistungsmerkmale (Reaktionszeit, Durchsatz), Sicherheit, Zuverlässigkeit, Benutz- und Wartbarkeit etc.

Für die Spezifikation funktionaler Anforderungen gibt es heute gut entwickelte Methoden, Techniken und Werkzeuge (man könnte in bestimmten Fällen fast von einer Standardisierung sprechen):

- Entity/Relationship-Datenmodellierung
- Modellierung mit objektorientierter Methodik
- Szenarien (Use-Cases), Sequenzdiagramme im objektorientierten Entwurf
- Workflowmodelle
- UML als Notation dafür, unterstützt durch zahlreiche Werkzeuge
  (UML hat die zuvor bestehende Vielfalt von OO-Notationen beendet)
- Schnittstellenbeschreibungen

Probleme der gängigen Praxis:

- Das Wechselspiel zwischen den Stakeholdern und die Zuständigkeiten sind nicht klar geregelt.
- Die Anforderungsanalyse ist oft eng verschränkt mit dem technischen Entwurf oder gar der Implementierung (vor allem bei den agilen Methoden).
- Anwendungsexperten stimmen den Anforderungen zu, auch wenn sie diese nicht richtig verstanden haben, etwa weil sie zu technisch formuliert sind.
- Der Fokus liegt meist nur auf den (funktionalen) Anwenderanforderungen, nichtfunktionale Anforderungen werden vernachlässigt.
- Entwurfsentscheidungen werden oft nicht dokumentiert.
- Die Systematik beschränkt sich auf Vorwärtsentwicklung.

– Die Wiederverwendung von Softwarebausteinen (Komponenten) schwankt zwischen Extremen − alles neu und selber machen vs. Komponenten zusammenstöpseln − und hat keine allgemeingültigen Richtlinien.

Die Bedeutung der Anforderungsdefinition ist weithin anerkannt. Allerdings wird man ihr in der Praxis oft nicht gerecht. Deshalb sollte ihr noch mehr Geltung und Unterstützung zuteil durch

– effektive Methoden, wie man ermittelt, was entwickelt werden soll,
– mehr Aufwand in der Ermittlung der Anforderungen (bis zu 40% des Gesamtaufwands),
– Spezialisierung auf Anwendungsdomänen,
– Unterstützung beim Festlegen nichtfunktionaler Anforderungen,
– systematisches Berücksichtigen von Anforderungen aus allen Lebenszyklusphasen,
– Verständnis und Akzeptanz dafür schaffen, dass es sich lohnt, Anforderungs- und Entwurfsmängel und -fehler durch Qualitätssicherung so früh wie möglich aufzudecken,
– systematische Trade-off-Entscheidungen zwischen Kosten und Qualitätsanforderungen,
– Werkzeugunterstützung bei der Aufzeichnung von Verfolgbarkeitsinformation (Traceability),
– neue Modelle entwickeln für das Requirements-Engineering von Produktlinien.

2.2 Architektur, Modularisierung

Architektur im Zusammenhang mit Software ist zu einem Begriff mit vielerlei Bedeutung geworden. Oft ist unklar, was genau damit gemeint ist. Es gibt zahlreiche Interpretationen, jede bedarf einer präzisen Definition. So gibt es Begriffe wie Funktionsarchitektur, Fachentwurf, Software-Architektur, Prozessarchitektur, Anwendungslandschafts-Architektur, Grob- und Feinentwurf und auch technische Architektur, die allesamt strukturelle Eigenschaften aus unterschiedlichen Perspektiven beschreiben.

Für *betriebliche Informationssysteme* hat sich über Jahrzehnte eine Standardarchitektur aus drei Schichten etabliert:

- Benutzerschnittstelle zur Gestaltung und Steuerung der Interaktion mit dem Anwender
- Anwendungskern für die Fachlogik
- Zugriff auf die Datenbank

Das MVC-Konzept (Model-View-Controller) ist eine Ausprägung davon.

Die Idee der Drei-Schichten-Architektur entstand zunächst für Mainframe-Anwendungen und wurde für verschiedene Umgebungen (Client/Server-Systeme, Web-Anwendungen, mobile Endgeräte) weiter entwickelt. Zustandslose Web-Server in beliebiger Parallelität mit vorgeschaltetem Lastverteiler sind vorteilhaft hinsichtlich Effizienz, Skalierung, Zuverlässigkeit und Robustheit. Neuere Ansätze setzen auf SOA (Service-orientierte Architektur).

Software-Architektur als grundlegende Disziplin ist in einigen wenigen Unternehmen gut etabliert, oft aber reduziert auf den Einsatz von Produkten bzw. Werkzeugen. Zudem entwickelt sie sich in der Praxis auf verschiedenen „Kontinenten“ (JEE, .NET, SAP u.a.), zwischen denen die Kommunikation nicht besonders intensiv ist. Konvergenz findet innerhalb der Kontinente statt, übergreifende Integration ist dagegen ein Problem, und zwar mehr sozial als technisch.

Bei eingebetteter Software gibt es eine Reihe von Architekturansätzen in den unterschiedlichen Domänen. Beispiele sind *Autosar* in der Automobilindustrie oder *Modulare Avionik*.

Die Architektur eines Softwaresystems bestimmt wesentlich dessen *Modularität*, d.h. seinen Aufbau aus eigenständig entwickelbaren Moduln bzw. Komponenten, die über wohldefinierte Schnittstellen interagieren. Allerdings ist der Komponentenentwurf weniger ausgeprägt als die Schichtenbildung. Hier mangelt es an Standards und Werkzeugen.

Für die Modularisierung ist die *Objektorientierung* seit Jahrzehnten das methodische Grundprinzip. Als Programmiertechnik in Norwegen begonnen, zur Spezifikations- und Entwurfsmethodik weiterentwickelt, mit UML als standardisierter Notation, ist sie heute die Methodik der Software-Entwicklung schlechthin.

Die Design Patterns (Entwurfsmuster) von Gamma et al haben einen wichtigen Beitrag zur Standardisierung von Architekturelementen und Professionalisierung der Software-Entwicklung geleistet.

Im letzten Jahrzehnt ist die Analyse von sog. Anwendungslandschaften vorangetrieben worden. Es sind vor allem Methoden entstanden, mit denen Funktionalität und Schnittstellen der oftmals zahlreichen und vielfältigen Systeme eines Unternehmens übersichtlich dargestellt werden können. Sie schaffen eine Grundlage für die Weiterentwicklung und ggf. Sanierung der Unternehmens-IT.

Eine Architektur soll Wiederverwendung und Wartbarkeit fördern. Sie ist am ehesten bei nichtfachlichen Komponenten möglich, denn es gibt − vor allem im JEE-Bereich – zahlreiche leistungsfähige Bibliotheken. Wiederverwendung fachlicher Komponenten ist eines der großen offenen Probleme, besonders hinsichtlich der Frage, wann sich welche Generalisierung lohnt. Dies ist nicht zu verwechseln mit der Verwendung von Softwareprodukten für Anwendungen, etwa SAP, die schon lange erfolgreich praktiziert wird.

Ebenso ermöglichen ESB-Produkte (Enterprise Service Bus) effiziente und flexible technische Architekturen. Allerdings ist die Technologie-übergreifende Integration oft schwierig. Das gilt auch für die fachliche Architektur von Systemverbunden, etwa mit einer systemübergreifenden SOA. Es wird viel darüber berichtet, richtig gute und allgemein akzeptierte Lösungen gibt es jedoch nicht.

Weitere Defizite in puncto Software-Architektur sind:

- Ein systematisches Verständnis für die Auswirkungen von Entwurfsentscheidungen auf nichtfunktionale Eigenschaften. Für deren Planung und Herbeiführung existiert zwar etwas Theorie, in der Praxis aber überwiegen Versuch und Irrtum.
- Behandlung von Fehlern und Ausnahmen − auch hier gilt: vorhanden sind einige Anti-Muster, aber zu wenige Muster.

- Umgang mit Parallelität und Sperren: einige Anti-Muster, wenige Muster.
- Diagnose kranker Systeme/Systemverbunde: wenig Theorie, Software-Diagnose-werkzeuge für statische und dynamische Analysen gewinnen an Bedeutung.

Einige interessante Entwicklungen:

- Software-Architektursimulation: Software-Entwürfe können hinsichtlich nicht-funktionaler Eigenschaften wie Performanz oder Zuverlässigkeit simuliert werden und damit Alternativen frühzeitig bewertet, der Ressourcen-Bedarf für den Betrieb abgeschätzt sowie die Skalierbarkeit untersucht werden.
- Massendatenverarbeitung mit Hadoop, Zookeeper & Co: Virtualisierung von CPU und Dateien auf Anwendungsebene mit leistungsfähigen und benutzbaren Biblio-theken.
- NoSQL-Datenbanken (z.B. SOLR) liegen auf halbem Weg zwischen relationalen Datenbanksystemen und Business Intelligence. Sie sind leichtgewichtig, skalierbar, schnell.
- Design for Diagnosibility: Überwachung und dynamische Analyse von Software-Systemen.

Die Spezialisierung des Software-Engineerings auf unterschiedliche Anwendungsarten (z.B. betriebliche und eingebettete Software) steckt noch in den Anfängen. Sie ist besonders bei der Software-Architektur nötig, denn unterschiedliche Anwendungs- und Technikdomänen finden ihren Niederschlag in jeweils andersartigen Architekturen. In der Forschung sollten dazu – in enger Zusammenarbeit mit der Praxis – Publikationen entstehen, die – konkret und gut verständlich geschrieben – als Vorlage für spezielle Standard-Architekturen dienen können. Dazu gehört u.a.

- die Einführung und Durchsetzung eines akzeptierten Komponentenbegriffs,
- die deutliche Erkennbarkeit der fachlichen Funktionalität in der Architektur und des Zusammenhangs mit der Anforderungsdefinition und deren Begriffen,
- die von der fachlichen Funktionalität abgegrenzten Konzepte der technischen Realisierung, auch Technologie-übergreifend im Fall des Einsatzes mehrerer verschiedener Systemplattformen,
- Sammlungen verlässlicher Software-Bausteine und -Patterns mit fachlicher und nichtfachlicher Funktionalität,
- eine Darstellung, wie sich die Architektur im Programmcode niederschlägt,
- die Erstellung von Analysen, Simulationen und Prototypen für Machbarkeitsana-lysen und Performancetests,
- die Evolution von Software-Systemen und Anwendungslandschaften (Systeme von Systemen, einschließlich großer Altsysteme),
- Sicherheitsarchitektur: Schutz der Software gegen Angriffe.

Durch die Ausarbeitung „artgerechter“ Software-Architekturen würde auch der Architek-turbegriff konkretisiert und präzisiert – eine überfällige Klärung und eine Voraussetzung für die Entwicklung von Werkzeugen für alle Lebenszyklusphasen der Software.

3 Programmierung

Die größten Fortschritte für die Software-Entwicklung wurden über die Jahrzehnte bei den Programmiersprachen und in der Programmiertechnik erzielt:

- *Höhere Programmiersprachen*
  Von Fortran und Cobol über Algol, Pascal, Ada, C zu Java und darüber hinaus: Mit Java haben wir in der Praxis zum ersten Mal eine Sprache, die auch wissenschaftlichen Ansprüchen genügt, und müssen nicht mehr unter den Schwächen von PL/1, Cobol und C++ leiden.
- *Objektorientierte Programmierung*
  Sie ist seit über zwanzig Jahren als bedeutendste Programmiertechnik etabliert, unterstützt durch diverse Programmiersprachen und -umgebungen. Unit-Tests und Entwurfsmuster fördern ihre methodisch angemessene Anwendung.
- Ein Programmierstil, der zu gut lesbarem Code führt, ist heute selbstverständlich, beispielsweise im Gegensatz zu den nichtssagenden, durchnummerierten Variablen in Zeiten der Cobol-Batch-Programmierung.

Eine aktuelle Entwicklung in der Hardwaretechnik wird die Programmierung vermutlich wesentlich beeinflussen: die *Multikern-Prozessoren*. Das Leistungspotential, das in ihnen steckt, kann nur genutzt werden, wenn die mögliche Parallelverarbeitung auch programmiert werden kann. Das ist im Allgemeinen nicht einfach, liegt jedoch bei bestimmten Anwendungen nahe. Es ist deshalb eine lohnende Forschungsaufgabe, die Multikern-Programmierung so zu entwickeln, dass sie möglichst nicht viel schwieriger wird als die sequenzielle Programmierung. Des Weiteren sollten Techniken und Werkzeuge entstehen, welche die (halb-) automatische Parallelisierung von Bestandssoftware ermöglichen.

Der Boom der Apps für mobile Geräte nährt den Wunsch nach einer einfachen Programmiertechnik für Jedermann. Es ist jedoch nicht klar, ob man brauchbare Apps ohne Ausbildung entwickeln kann.

Software-Engineering wird häufig definiert in Abgrenzung zur Programmierung. Das wertet das Programmieren tendenziell ab und führt dazu, dass gute Absolventen und Fachkräfte im Lauf der Karriere von der Programmierung weg wollen und Architektur- und Managementpositionen anstreben. Deshalb: Rehabilitation des Programmierens als zentrale Tätigkeit des Software-Engineerings mit guten Karrierechancen, Spaß an der Programmierung, anstatt sie zu vermeiden. Programmieren muss wieder sexy werden! Denn nur der Code ist das nutzbare Softwareprodukt.

4 Software-Plattformen und Entwicklungswerkzeuge

Die Vielfalt und -zahl der *Plattformen* (Betriebs-, Datenbank- und Kommunikationssysteme), für die und auf denen Software entwickelt wird, hat enorm zugenommen. Anfangs waren es einige wenige monolithische Systeme, seien es Mainframes oder mittlere Rechner sowie Prozess- und Steuerungssysteme. Der PC als Arbeitsplatzgerät hat zu Client/Server-Systemen geführt und den Zugang zum Internet geöffnet. Heute haben wir es zudem mit hochgradig verteilten und immer mehr mobilen Systemen zu tun. Dazu kommen vielfältige Prozessoren für eingebettete Software als Teil einer Maschine.

Die heutige Allgegenwart und Mächtigkeit der IT-Anwendungen in vielen Lebensbereichen, in Technik und Wirtschaft basiert auf der Funktionalität und Leistungsfähigkeit von Plattformen, auf denen Software läuft. Um nur wenige herausragende Beispiel zu nennen:

- *relationale Datenbanken*, ein Durchbruch Mitte der 1970er Jahre für die Entwicklung von Informationssystemen,
- *graphische Benutzeroberflächen,* die ab Ende der 1980er Jahre die Benutzung des Computers grundlegend verändert haben,
- *Browser,* die in der ersten Hälfte der 1990er Jahre das World Wide Web eröffnet haben.

Anders als früher, als Unternehmen die gesamte Plattform aus Hard- und Software oft von einem einzigen Hersteller geliefert und betreut bekamen, z.B. von IBM, haben sie es heute mit einer Vielzahl von Anbietern zu tun, auch Open Source, deren Systeme sie selbst integrieren müssen.

Auch bei den *Entwicklungsumgebungen* und -*werkzeugen* gab und gibt es erhebliche Fortschritte, die zu einer unübersehbaren Fülle geführt haben. Die Produktivität der Software-Entwicklung ist dadurch deutlich gestiegen und steigt weiter, in erster Linie durch Automatisierung, aber auch durch Wiederverwendung (Frameworks).

Mit Werkzeugen gut abgedeckt sind einige Bereiche der Software-Entwicklung, darunter

- Editieren, Übersetzen und Binden von Programmcode
- Bearbeiten von Modellen, z.B. in UML und Mathlab
- Versionenkontrolle (Configuration Management, Build Management, Continuous Delivery)
- Verwalten von Fehlermeldungen

Die heutigen Entwicklungsumgebungen sind zu sehr auf das Programmieren und die technische Plattform konzentriert. Sie bieten zu wenig für die Modellierung von Anwendungsdomänen, die Erfassung von Anforderungen, die Architektur- und Komponentenbildung, für Unit-Tests, Projekt-Management und den Umgang mit Altsystemen (200 Mrd LoC in Cobol weltweit). Wir brauchen Entwicklungsumgebungen, die

- die logische Verbindung zwischen Software-Artefakten (und nicht nur Programmen) halten,
- Brücken zwischen Technik- und Anwendungsdomänen bauen,
- Softwaresysteme zu überblicken helfen (etwa mithilfe von Architekturansichten),
- verteiltes Arbeiten an verschiedenen Standorten erleichtern.

Für eine erfolgreiche, zielgerichtete Weiterentwicklung von Methoden und Werkzeugen werden dringend analytische Modelle und ein ganzheitlicher Wirtschaftlichkeitsbegriff benötigt.

Die Bedeutung von Werkzeugen für die Software-Entwicklung ist evident. Ohne sie sind in größeren Teams weder effizientes Arbeiten möglich noch durchgängige Struktur und Qualität zu erzielen. Dazu kommt, dass über Umgebungen und Werkzeuge methodisches Wissen transportiert wird.

5 Qualitätssicherung

Auch wenn der Workshop, der diesem Text zugrunde liegt, sich nicht auf Qualitätssicherung fokussiert hat, ist die Bedeutung dieses Themas unbestritten.

Die in der Praxis bei weitem wichtigste Maßnahme zur Qualitätssicherung ist das Testen. Der Stand der Methoden und Werkzeuge dafür wird dessen Bedeutung jedoch nicht gerecht. Die Entwicklung von leistungsfähigen Tools zur effizienten Erstellung von wartbaren, automatisch ausführbaren Tests ist eine Aufgabe für die Zukunft.

Immerhin sind Unit-Tests − und deren Erstellung vor und parallel zur Programmierung − eine der Errungenschaften der letzten 20 Jahre. Sie ergänzen den klassischen, von der Benutzerschnittstelle aus getriebenen Test, der meist keine ausreichende Code-Überdeckung erreicht. Zudem ersetzen Unit-Tests in der Praxis auch fehlende oder mangelhafte Spezifikationen. Das ist nicht ideal, aber besser als nichts.

Vorangetrieben werden sollten Nutzung sowie Erforschung und Entwicklung effektiver und effizienter Methoden und Verfahren für die analytische Qualitätssicherung, d.h. für Reviews und Inspektionen, Verifikation und Test, Code- und Performanceanalyse, etc.

6 Vorgehen

6.1 Vorgehensmodelle

Über die Jahrzehnte sind viele Konzepte für das Vorgehen in Softwareprojekten publiziert und praktiziert worden. In der Regel folgen sie dem klassischen ingenieurmäßigen Schema: Planen – Konstruieren – Fertigen – Prüfen – Installieren – Warten − Weiterentwickeln, ggf. mit Korrekturschleifen. Für dieses Vorgehen in Phasen sind diverse Bezeichnungen anhand graphischer Darstellungen geprägt worden: Wasserfall-, Spiral-, V-Modell. Diese Metaphern sind sachlich wenig erhellend, aber Gegenstand einer Art weltanschaulicher Auseinandersetzung. In großen Organisationen, insbesondere auch staatlichen, gibt es eine Tendenz zu formal-bürokratischer Ausformung von Phasen- und Reifegradmodellen. Das bekannteste Beispiel ist das Capability Maturity Model (CMM) des SEI in den USA. Auf der anderen Seite und als Gegenbewegung sind vor gut zehn Jahren die agilen Methoden entstanden, beginnend mit Extreme Programming (XP). Ihnen mangelt es an ingenieurmäßiger Methodik. Sie beanspruchen dennoch, überlegen zu sein, vor allem dadurch dass sie den Teams mehr Freiraum lassen.

Eine Spezialisierung des Software-Engineerings ist auch für Vorgehensmodelle zweckmäßig. Sie sollten den unterschiedlichen Arten von Anwendungen angepasst sein.

In jeder Art Vorgehensmodell sollte berücksichtigt sein:

- Software-Controlling: eine kontinuierliche, umfassende Analyse (statisch und dynamisch) der in Entwicklung befindlichen Software, beginnend am ersten Tag der Programmierung, mit dem Ziel, negative Trends zu erkennen und Gegenmaßnahmen einzuleiten.
- Stufenweise Einführung (Deployment) von Software, statt „Big Bang".

6.2 Management

Software-Engineering umschließt auch Fragen der Organisation und des Managements von Software-Entwicklungsprojekten sowie des Betriebs und der Wartung von Softwaresystemen.

Das Wissen über Software im Top-Management etlicher Unternehmen lässt zu wünschen übrig. Es mangelt an Verständnis für ihre Entwicklung und ihren Einsatz, generell für ihre Bedeutung im Geschäft, sei es als Organisationsmittel oder als Teil des Produkts. „Software-Governance" über alle Führungsebenen wäre wünschenswert, ist aber nur in wenigen Unternehmen zu erkennen. Ein besseres Verständnis von Software würde u.a. helfen,

- gutes Software-Engineering in den unternehmenseigenen Teams zu etablieren und durchzusetzen;
- das Beauftragen und Abnehmen von Software, die von Dritten geliefert wird, besser zu beherrschen, und zwar in unter verschiedenen Gesichtspunkten wie Vollständigkeit, Korrektheit, Wartbarkeit, Eleganz;
- zu erkennen, welche Software-Engineering-Kenntnisse der Kern-Belegschaft für Offshore-Outsourcing wesentlich sind;
- das Software-Produkt-Management in den Unternehmen ganzheitlich anzugehen. Es ist ein praktisch relevantes Gebiet, das auch mehr wissenschaftliche Aufmerksamkeit verdient.

6.3 Empirie im Software-Engineering

Methoden und Werkzeuge der Softwaretechnik entsprechen nicht unbedingt den Anforderungen der prospektiven Nutzer, und es ist nicht von vornherein klar, ob sie deren Arbeit qualitativ und quantitativ ausreichend verbessern. Empirische Untersuchungen könnten das prüfen. Sie wurden jedoch bis vor 10-15 Jahren kaum durchgeführt, erst in jüngster Zeit wurde ihre Bedeutung für Softwareforschung und -praxis klar. Der Faktor Mensch spielt dabei eine wesentliche Rolle.

Mittlerweile gibt es eine große Anzahl empirischer Untersuchungen zur Wirksamkeit neuer Techniken. Das wird deutlich in einer Reihe spezieller Zeitschriften und Tagungen, aber auch in allgemeinen Publikationen werden sie erwartet. Außerdem ist zu den bekannten empirischen Methoden eine neue hinzugekommen, nämlich die Untersuchung von Software-Depots, die die komplette Historie von Softwareprojekten enthalten.

Leider unterlassen es Forscher häufig, ihre Techniken mit professionellen Softwareentwicklern zu untersuchen, sondern arbeiten mit (Informatik-)Studenten. Diese sind für

Voruntersuchungen durchaus geeignet, aber sie unterscheiden sich in kritischen Aspekten von professionellen Entwicklern.

Eine rühmliche Ausnahme ist das Experiment von Arisholm et al[1] zur Analyse der Paarprogrammierung mit 295 professionellen Entwicklern aus drei Ländern. Die Kosten betrugen ca. 250.000 €, hauptsächlich für die Entlohnung der Teilnehmer. Es erforderte eine lange Vorbereitung und wiederholte Durchführungen. Auch die Rekrutierung so vieler Teilnehmer war ein Problem.

Generell dauern Experimente mit menschlichen Teilnehmern lange, die Erkenntnisse sind oft gering, die Kosten hoch. Hinzu kommt, dass es sehr schwierig ist, Störfaktoren zuverlässig auszuschließen, die das Ergebnis verfälschen können.

Informatiker wollen meist etwas konstruieren und nicht die Erfindungen anderer im Nachhinein analysieren, zumal die Ergebnisse oft negativ und nicht konstruktiv sind. Daraus resultiert ein Nachwuchsproblem bei „reiner" empirischer Forschung. Besser ist daher die begleitende Erprobung bei der Entwicklung neuer Softwaretechniken. Dabei empfiehlt es sich, öfter und in kleinen Schritten zu experimentieren, auch ohne Testpersonen.

Ein probater Weg sind Benchmarks. Das sind Mengen von Problemen mit einer Qualitätsmetrik für Lösungen. Unabhängige Forscher wenden ihre „Löser" an und vergleichen die Ergebnisse. Damit entsteht ein Wettbewerb, der die Spreu vom Weizen trennt. Problemsammlungen haben einen wesentlichen Vorteil gegenüber Versuchen mit menschlichen Teilnehmern: Sie können beliebig oft benutzt werden, mit minimalen Kosten.

Weiterhin könnten Fallstudien zur nachträglichen (auch quantitativen) Bewertung von Softwareprojekten wertvolle Erkenntnisse bringen.

Metriken und Messwerkzeuge erscheinen wünschenswert, um Software zu bewerten und beurteilen, ebenso Qualitätsmaße und die Messung bzw. Bewertung nichtfunktionaler Eigenschaften. Wert und Nutzen von Metriken sind jedoch nicht wirklich erwiesen.

Die Empirie im Software-Engineering ist schwach entwickelt. Experimente zum Erforschen des Nutzens von Methoden, Techniken und Verfahren sind teuer und zeitaufwändig, ihr Ergebnis ist nicht immer zweifelsfrei interpretierbar. Dennoch sollte dieses Gebiet weiter entwickelt werden.

[1] Erik Arisholm, Hans Gallis, Tore Dyba, Dag Sjoberg, „Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise", IEEE Trans. on Software Engineering, Vol 33, no 2, Feb. 2007, 65-85.

7 Forschung und Lehre

7.1 Forschung

Wie jede andere Disziplin braucht Software-Engineering eine wissenschaftliche Grundlage. Diese darf nicht nur daran gemessen werden, was unmittelbar in die Praxis umsetzbar ist, sondern ist auch unverzichtbarer für das Selbstverständnis eines Faches – auch dafür, dass es bewertbar und lehrbar ist. Ein angewandtes Fach wie Software-Engineering muss sich aber auch fragen lassen: Was ist aus der Forschung in der Praxis angekommen?

Es gibt Resultate der Forschung, die die Praxis erheblich beeinflusst haben, beispielsweise höhere Programmiersprachen (Algol, Pascal u.a.), strukturierte Programmierung, Information Hiding, abstrakte Datentypen und Objektorientierung, Grundlagen der Betriebssysteme (Multics ⇨ Unix) und relationalen Datenbanken, einige Werkzeuge (z.B. Versionsverwaltung: RCS ⇨ CVS). Einige wichtige Beiträge wurden auf der Konferenz „Software-Pioniere“ in Bonn dokumentiert[2].

Als wenig ergiebig für die Praxis hat sich die Forschung über formale Methoden erwiesen, auch wenn sie eine Zeit lang beansprucht hat, dafür nützlich zu sein.

Mit Abstand die wichtigste Form des Transfers ist der Übergang von Menschen aus der Forschung in die Praxis. Allein schon dieser „Hirntransfer“ rechtfertigt den Aufwand für die staatliche Software-Engineering-Forschung.

Sehr einflussreich waren die Forschungszentren der Industrie, bevor sie praktisch weltweit Kürzungen zum Opfer fielen. Bekannte Beispiele sind Xerox-PARC, IBM Yorktown Heights, die Bell Labs, das SCR von Digital Equipment, Microsoft Research und Aktivitäten einzelner Koryphäen wie Barry Boehm. Ebenfalls sehr einflussreich waren einzelne Personen, die als Unternehmer oder Privatleute neue Ideen umgesetzt haben: Steve Jobs (Apple), Tim Berners-Lee (www), Linus Torvalds (Linux), Larry Page und Sergey Brin (Google) u.a.

Das heißt: Das Bild der heutigen Software-Landschaft ist außer durch die Wissenschaft stark geprägt durch Entwicklungen und Innovationen, die außerhalb der staatlichen Forschungseinrichtungen stattgefunden haben.

In der Software-Engineering-Forschung wird das Gros der Dissertationen und der übrigen Publikationen oft nicht zur Grundlage weiterer Arbeiten. Dies hat vermutlich zwei Ursachen:

- Wir trauen den publizierten Resultaten in der Regel nicht und würdigen sie zu wenig. Eine wirklich kritische Auseinandersetzung mit neuen oder veränderten Methoden, Sprachen, Werkzeugen ist die Ausnahme.
- Fast alle Ergebnisse sind an spezielle Prämissen geknüpft, die andernorts nicht gegeben sind. Wenn beispielsweise jemand den Erfolg eines bestimmten Prozesses in seiner Firma dokumentiert, wissen wir weder, ob den Aussagen zu trauen ist, noch ob die Resultate auf eine andere Umgebung übertragbar sind.

[2] Broy/Denert (Eds.), *Software Pioneers,* Springer 2001

Die Software-Forschung muss die relevanten Probleme der Praxis erkennen, mit Praktikern diskutieren und ihre Ergebnisse (nicht nur Methoden und Werkzeuge, sondern auch Konzepte) der Validierung unterziehen, oder wo das die Mittel sprengt, die Validierung durch die Praxis als Teil des Forschungsergebnisses vorplanen.

Wissenschaftliche Einrichtungen sollten Arbeitstagungen zur kritischen Analyse von Software-Projekten, -Methoden, -Werkzeugen, -Management durchführen. Diese könnten auch der Formulierung von Anforderungen an Ausbildung, Forschung und deren Förderung dienen.

Die Forschung sollte folgende Prämissen beachten:

- Jede neue Idee im Software-Engineering bedarf nicht nur der technischen Realisierung und Demonstration, sondern auch der Erprobung und der funktionierenden Einführungsstrategie. Wir müssen diesem Aspekt wesentlich mehr Aufmerksamkeit widmen und ihn entsprechend hoch gewichten.
- Kleine, solide Beiträge sollten als wesentliche Bausteine der Wissenschaft akzeptiert werden.

7.2 Ausbildung

Die Informatik-Lehre an den Universitäten und Fachhochschulen ist im Allgemeinen gut. Ihre Grundlagen und eine eventuelle Vertiefung in Software-Engineering sind für die wirtschaftliche Praxis der Software-Entwicklung sehr nützlich. Zahlreiche gute Lehrbücher tragen dazu bei.

Die mehrfach angesprochene Spezialisierung des Software-Engineerings sollte auch in der Lehre ihren Platz haben, allerdings ohne wichtige allgemeine Kenntnisse und Fähigkeiten zu kurz kommen zu lassen, gewissermaßen den Weg vom Generalisten zum Spezialisten vorbereiten.

Wir sollten einen Kanon der Ausbildung zum Software-Ingenieur erarbeiten, als Teil des Informatik-Studiums, in ggf. reduzierter Form auch als Teil anderer Studiengänge, ebenso zur Fortbildung von Software-Entwicklern. Dabei sollte Software-Engineering nicht in Abgrenzung zu einer „bloßen Programmierung“ definiert werden. Für die Praxis der Software-Entwicklung wäre ein Kanon von Patterns verschiedener Granularität (für Anforderungen, Design, Architektur, Code, Qualitätssicherung) wünschenswert, die jeder Entwickler kennen und beherrschen muss. Das Software-Projektmanagement sollte im Software-Engineering als vollwertiges Teilfach anerkannt sein.

Ein Katalog für die Minimal-Qualifikation eines Software-Ingenieurs sollte abgeleitet und von uns offensiv vertreten werden.

Spezielle offene Fragen sind:

- Wie können Berufstätige von über 35 Jahren weiter qualifiziert werden?
- Sollte dem Erwerb der Fähigkeiten zur Programmierung und konzeptuellen Modellierung in der schulischen, beruflichen und akademischen Ausbildung zukünftig ein deutlich höherer Stellenwert eingeräumt werden?

8 Resümee

Spezialisierung

Software-Engineering wird immer noch zu umfassend und allgemein aufgefasst. Es sollte jedoch stärker spezialisiert werden, um den diversen Anwendungs- und Technikdomänen besser gerecht zu werden. Die Software-Entwicklung für betriebliche Informationssysteme beispielsweise unterscheidet sich stark von jener für eingebettete Software in Maschinen. Bei Web-Anwendungen sind anderes Vorgehen und andere Techniken gefragt als bei der Programmierung von Prozessoren im Automobilbau. Oder bei der Realisierung von ubiquitärer Intelligenz für Anwendungen in persönlicher Umgebung.

Selbstverständlich verdienen bestimmte grundlegende Prinzipien Beachtung bei jeglicher Spezialisierung.

Die Spezialisierung ist in allen Teilbereichen des Software-Engineerings nötig: in Anforderungsdefinition und Architektur, in der Programmierung, in Qualitätssicherung, Test und Integration, etc. Und sie muss in Lehre und Forschung ihren Niederschlag finden.

Die Spezialisierung führt sicher auch zu einem besseren Verständnis für das Wechselspiel zwischen den Modellen und Methoden des Software-Engineerings und denen der jeweiligen Fachdisziplinen (z.B. Ökonomie, Psychologie, Soziologie, Automatisierungs- oder Elektrotechnik).

Best-Practice in der Praxis umsetzen

Fortschritte in der industriellen Praxis des Software-Engineerings sind weniger aus neuer Forschung zu erwarten als vielmehr aus der konsequenten Anwendung des Know-hows, das sie bereits in den letzten Jahrzehnten geliefert hat. Denn was wir im Prinzip können, beherrschen wir in der Praxis noch lange nicht. Es kommt vor allem darauf an, die in der Spitze erreichte Best-Practice als allgemein anerkannten Stand der Technik in der Breite zu etablieren. Es hapert in der Praxis immer wieder an der Softwaretechnik und vor allem am Projekt-Management. Zu Schwierigkeiten führen oftmals auch unrealistische Anforderungen und/oder Terminziele, die, vom Auftraggeber vorgegeben, vom Auftragnehmer hingenommen werden, warum auch immer.

Weiterentwicklung des Fachs

Getragen und getrieben von dem immer noch atemberaubenden Fortschritt in der Mikroelektronik wird die Nachfrage nach Software in etablierten und neuen Anwendungen weiter anhalten, ja noch zunehmen. Funktionalität, Leistung, Ressourcenbedarf und die damit verbundenen Kosten werden weiter steigen. Umfangreiche Software-Systeme sind und werden über drei bis vier Jahrzehnte im Einsatz sein und eine aufwändige Evolution und Wartung erfordern. Das fordert das Fach Software-Engineering heraus, seine Sprachen, Modelle, Methoden und Verfahren weiter zu entwickeln und in die Praxis zu bringen, stärker noch als bisher auf der Basis solider – auch empirischer – Erkenntnisse.

9 Teilnehmer des Workshops

Dr. Boris **Bokowski**, Google Germany GmbH, München
Dr. Uwe **Dumslaff**, Capgemini Deutschland GmbH, München
Prof. Dr. Martin **Glinz**, Universität Zürich
Prof. Dr. Jochen **Ludewig**, Universität Stuttgart
Prof. Dr. Oscar **Nierstrasz**, Universität Bern
Dr. Markus **Pizka**, itestra GmbH, München
Prof. Dr. Klaus **Pohl**, Universität Duisburg-Essen
Perry **Prust**, IVU Traffic Technologies AG, Berlin
Dr. Thomas **Reiss**, SAP AG, Walldorf
Prof. Dr. Ralf **Reussner**, Karlsruher Institut für Technologie
Prof. Dr. Gunter **Saake**, Otto-von-Guericke-Universität, Magdeburg
Dr. Christian **Schlögel**, Wincor Nixdorf GmbH, Paderborn
Prof. Dr. Johannes **Siedersleben**, QAware GmbH, München
Dr. Reinhard **Stolle**, BMW AG, München
Stefan **Teuchert**, MAN Nutzfahrzeuge AG, München
Prof. Dr. Walter **Tichy**, Karlsruher Institut für Technologie

Veranstalter
Prof. Dr. Manfred **Broy**, Technische Universität München
Prof. Dr. Ernst **Denert**, Ernst Denert-Stiftung, Grünwald
Prof. Dr. Eike **Jessen**, Technische Universität München
Prof. Dr. Florian **Matthes**, Technische Universität München