Softwaretechnik: Einige Grundlagen

Übersichtsmaterial zur Vorlesung "Softwaretechnik", 2020-04-08, Version 2

Prof. Dr. Lutz Prechelt, prechelt@inf.fu-berlin.de
Institut für Informatik, Freie Universität Berlin

Inhaltsverzeichnis:

1. Aufgaben und Inhalt dieses Dokuments

Dieses Dokument enthält nur einige wenige, **globale Informationen zur Vorlesung** (das meiste Material steckt in den Foliensätzen). Diese Informationen sollen vor allem zur besseren **Orientierung** im Aufbau und Inhalt der Vorlesung beitragen, denn die Softwaretechnik(vorlesung) ist ein recht umfangreiches und komplexes Gelände.

Die Orientierung wird dadurch gegeben, dass die Softwaretechnik auf unterschiedliche Weise in Teile zergliedert wird: In Produkt und Prozess (Abschnitt 3.2.1); in Definition, Realisierung, Validierung (Abschnitt 3.2.2) und in Probleme und Lösungen (Abschnitte 4.1 und 4.2). Außerdem unterscheiden wir verschiedene Situationen (Randbedingungen) unter denen SW-Entwicklung stattfinden kann (Abschnitt 4.3).

Es ist wichtig, an jeder Stelle der Vorlesung genau zu verstehen, warum und inwiefern etwas von Interesse ist und wie es sich in den Gesamtzusammenhang einordnet. Dieses Dokument liefert den Rahmen für diese Einordnung (Abschnitte 3 und 4), gibt Hinweise, wie man das Einordnen üben kann (Abschnitt 5), welche Prioritäten beim Erlernen der Softwaretechnik zu beachten sind (Abschnitt 6.1) und wie man das Hauptlernziel charakterisieren kann (Abschnitt 5.2).

2. Was ist Softwaretechnik/Software-Engineering?

2.1. Definitionsversuch

Ich verwende im Folgenden die Begriffe Softwaretechnik und Software-Engineering synonym.

Es gibt in der Literatur zahlreiche Definitionen für den Begriff "Software-Engineering", die verschiedene Schwerpunkte setzen. Wir betrachten hier nur eine einzige, die ich für einigermaßen gelungen halte. Sie stammt von Hesse aus "Ein Begriffssystem für die Softwaretechnik":

> "Softwaretechnik (synonym: Software Engineering): Fachgebiet der Informatik, das sich mit der Bereitstellung und systematischen Verwendung von Methoden und Werkzeugen für die Herstellung und Anwendung von Software beschäftigt."

Andere Definitionen erwähnen zusätzlich z.B. die folgenden Aspekte (und andere):

- Es geht um ingenieurmäßiges Vorgehen (d.h. eine Disziplin auf Basis wissenschaftlicher Erkenntnisse), evtl. ist auch "Quantifizierbarkeit" verlangt.
- Software besteht aus Programmen und Dokumentation.
- Zur "Herstellung" gehört auch die Ermittlung der Anforderungen, die Inbetriebnahme und die Pflege und Fortentwicklung.
- Typischerweise ändern sich bereits während der Entwicklung die Anforderungen.

Diese Vorlesung betrachtet Werkzeuge kaum, weil die leicht vom Wesentlichen ablenken.

2.2. Grobzerlegung der Softwaretechnik

Fundamental zum Lernen jedes komplexen Wissensgebietes ist eine Grobunterscheidung seiner wichtigsten Teile. Man braucht eine solche Zerlegung, um sich in dem Gebiet überhaupt orientieren zu können und um einen ersten (wenn auch ungenauen) Überblick zu gewinnen.

In der Softwaretechnik sind für eine solche Grobzerlegung **zwei Betrachtungsweisen üblich**, von denen meist eine im Vordergrund steht: **Entweder** die fundamentale Unterscheidung von **Produkt** (also den Arbeitsergebnissen) **und Prozess** (also den Arbeitsverfahren und -vorgängen) **oder** die **Zerlegung nach den groben zu lösenden Aufgaben.** Die folgenden Abschnitte stellen diese beiden herkömmlichen Grobzerlegungen kurz vor,

diskutieren ihre Vor- und Nachteile und führen dann eine **dritte Zerlegung** ein, die normalerweise nur als untergeordneter Gesichtspunkt verwendet wird: **Probleme und Lösungsansätze**.

2.2.1. Produkt und Prozess

Die wohl fundamentalste Unterscheidung von Grundkonzepten in der Softwaretechnik ist die zwischen Produkt und Prozess.

Produkt ist die Gesamtheit aller greifbaren Arbeitsergebnisse bei der Softwarekonstruktion, oft *Dokumente* oder *Artefakte* genannt: Programmcode, Benutzerdokumentation, interne technische Dokumentation, Pläne, Protokolle, etc.

Prozess ist die Gesamtheit einerseits aller konkreten Abläufe in einem Softwareprojekt und andererseits der zu Grunde liegenden allgemeinen Konzepte und Verfahrensweisen: Rollen, Aktivitäten, Methoden, Werkzeuge, etc.

Wenn man eine Unterscheidung nicht weiter verfeinern dürfte als in zwei Teile, wäre diese wohl diejenige mit dem größten Nutzen, denn sie enthält die grundlegendste aller Einsichten der Softwaretechnik: Dass man sich zum erfolgreichen Bau großer Softwaresysteme ausdrücklich Gedanken über das Vorgehen (Methodik) machen muss.

2.2.2. Definition, Realisierung, Validierung

Die wohl am weitesten verbreitete Unterscheidung von Grundkonzepten in der Softwaretechnik ist die entlang der großen grundlegenden Aufgaben (Aktivitäten), die man zu erledigen hat. Diese werden, da man lange die naive Vorstellung hatte, sie sequentiell hintereinander zu erledigen, oft auch Phasen genannt.

Die gröbste mögliche Zerlegung unterscheidet hier drei Konzepte:

1. **Definition: Festlegen, WAS man eigentlich bauen will.** Teilaufgaben: Anforderungserhebung, Anforderungsbeschreibung (Spezifikation), Anforderungsanalyse, Anforderungsverwaltung.
2. **Realisierung: Festlegen, WIE man es bauen will und das dann umsetzen.** Teilaufgaben: Architekturentwurf, Technologieauswahl, Grobentwurf, Feinentwurf, Kodierung (Programmierung).
3. **Validierung: Überprüfen, ob man wirklich das erfunden hat, was man wollte** (bzw. wo eben nicht). Im (zu) engen Sinne gehört hierzu insbesondere der Softwaretest; im (zu) weiten Sinne ist dies die so genannte Qualitätssicherung.

Ein_e Softwareingenieur_in wird diese Einteilung als unvermeidlich und unmittelbar einleuchtend ansehen, weil sie ihr längst in Fleisch und Blut übergegangen ist. Jemandem der Softwaretechnik gerade erst lernt, leuchtet sie anfangs vielleicht überhaupt nicht auf Anhieb ein, weil sie ihre eigenen Beweggründe wenig sichtbar macht: Warum diese Einteilung richtig und wichtig ist, kann man erst erkennen, wenn man gesehen hat, was hinter den einzelnen Teilen wirklich steckt.

2.2.3. Probleme und Lösungen

Deshalb verwendet die nachfolgende Darstellung für die Annäherung eine andere Grundzerlegung, die unmittelbar einleuchtet: Es gibt einerseits **zu lösende Probleme** und andererseits **Lösungsansätze**, mit denen sie angegangen werden. Die Lösungsansätze repräsentieren zum Teil Versuche, ***radikales Vorgehen*** (also das Agieren außerhalb des Bereiches, über den man ausreichende Erfahrungen besitzt) zu unterstützen und seine Effizienz und Erfolgswahrscheinlichkeit zu verbessern. Andere Lösungsansätze repräsentieren ***normales Vorgehen*** (also das Vorgehen gemäß Erfahrungswissen und die Wiederverwendung solchen Wissens).

Diese Einteilung ist wenig geeignet, die ganze Softwaretechnik danach zu präsentieren, weil Probleme und zugehörige Lösungsansätze natürlich meist gemeinsam betrachtet werden sollten. Aber die Einteilung ist hervorragend geeignet, sich in der Softwaretechnik zu orientieren, Themen einzuordnen und zu zerlegen. Ein_e kompetente_r Softwareingeni-

eur_in ist jederzeit in der Lage, verschiedene mögliche Lösungswege daraufhin zu analysieren, für welche Aspekte des aktuellen Problems sie gut und für welche sie weniger gut geeignet sind und warum.

2.3. Ziele der Softwaretechnik, Qualitätsmerkmale

Das Ziel der Softwaretechnik besteht darin, die folgenden Parametergruppen eines Entwicklungsvorhabens zu optimieren (möglichst alle gleichzeitig):

- **Kosten:** Man möchte das Vorhaben für möglichst wenig Geld umsetzen.
- **Zeit:** Man möchte möglichst schnell fertig werden oder wenigstens eine Teillösung möglichst bald in Einsatz bringen können.
- **Funktionalität:** Die Software soll möglichst umfangreiche und leistungsfähige Funktionen aufweisen; es gibt immer etwas, das man zusätzlich gern hätte. Die Funktionen werden hierbei natürlich nach Nützlichkeit gewichtet.
- **Qualität:** In der Software sollen diverse interne und externe **Qualitätsmerkmale** möglichst stark ausgeprägt sein. **Externe** (also für die Benutzer sichtbare) sind z.B. Benutzungsfreundlichkeit, Zuverlässigkeit, Robustheit, Installierbarkeit, Effizienz u.v.a.m. **Interne** (also nur für die Softwareingenieur_innen direkt sichtbare, aber diese Unterscheidung ist oft wacklig) sind z.B. Codeverständlichkeit, Erweiterbarkeit, Testbarkeit, Portierbarkeit u.v.a.m.

Während Zeit und Kosten eng miteinander verwandt sind (und deshalb meist ganz gut in Einklang gebracht werden können), stehen die anderen beiden Faktoren damit tendenziell im Widerspruch: Sehr hohe Qualität kostet mindestens kurzfristig mehr Zeit und Geld, breitere Funktionalität kostet fast immer mehr Zeit und Geld.

Das gilt aber nicht unbedingt: Insbesondere die internen Qualitätsmerkmale tragen oft sogar zur Senkung von Zeit- und Kostenaufwand (bzw. zur Steigerung der Funktionalität) bei. Zum Beispiel bedeutet eine gute Erweiterbarkeit (für die man Aufwand treiben muss und die deshalb etwas kostet) ja gerade, dass eine Erweiterung tendenziell billiger zu machen ist. Wie viel, wo und wie sollte man also in Erweiterbarkeit investieren? Das hängt ganz davon ab, welche Erweiterungen später wirklich gebraucht werden.

Als wie geeignet irgendeine Methode der Softwaretechnik in einer bestimmten Situation einzuschätzen ist, hängt davon ab, wie sie sich auf die jeweils als am wichtigsten erachteten Zielfaktoren auswirkt. Das lässt sich (ebenso wie die sinnvolle Gewichtung der Zielfaktoren selbst) niemals pauschal beurteilen und genau diese Urteilskraft macht vor allem eine_n gute_n Softwareingenieur_in aus (siehe auch Abschnitt 6 "Was muss man wissen?").

3. Taxonomie: Die Welt des Software-Engineering

Diese Unterteilung dient zur Orientierung innerhalb der Softwaretechnik. Hiermit lassen sich alle Themen in ihre Bestandteile zerlegen und diese dann einordnen. Das erleichtert die Beurteilung und die Konzentration auf das Wesentliche.

Die Einteilung ist nicht kanonisch; man könnte sie auch ganz anders machen. Aber sie ist nützlich. Die Einteilung ist außerdem höchst unvollständig. Sie erwähnt nur, was ich für am Wichtigsten halte und beschränkt sich auf die obersten Ebenen der Zerlegung.

Wir besprechen hier der Reihe nach:

- Die Welt der Problemstellungen: Mit welchen Phänomenen muss die Softwaretechnik fertig werden?
- Die Welt der Lösungsansätze: Welche ganz allgemeinen und immer wiederkehrenden Herangehensweisen hat die Softwaretechnik dafür entwickelt?

3.1. Welt der Problemstellungen

Die Welt der Problemstellungen umfasst all das, was das Programmieren-im-Großen (also den Anwendungsbereich der Softwaretechnik) erheblich schwieriger werden lässt als das Programmieren-im-Kleinen. Etwas hundertmal so Großes zu bauen ist nämlich nicht nur

hundertmal so aufwendig, sondern viel mehr, denn es kommen massenhaft ganz neue Probleme hinzu, die neben dem Aufwand auch die Schwierigkeit und das Risiko in die Höhe treiben.

Ganz wichtig: Alle nachfolgenden Probleme sind **essenzielle Probleme** der Softwaretechnik, d.h. sie lassen sich durch Fortschritte in Methodik oder Technologie oder durch Auswahl besonders talentierten Personals nicht lösen oder beseitigen, sondern nur abmildern.

3.1.1. Produkt: Komplexitäts-Probleme

Komplexität (lat.: Verflochtenheit) bezeichnet die Eigenschaft eines Systems (a) viele Einzelteile zu haben und zugleich (b) emergente Eigenschaften zu haben. Als ***emergent*** (lat. emergere, auftauchen, hervortreten) bezeichnet man ein Phänomen, das man nicht verstehen kann, indem man die verursachenden Teile nur einzeln betrachtet, sondern das aus deren Zusammenwirken hervorgeht. Einfach gesagt bedeutet komplex so viel wie "schwierig aufgrund vielfältiger Zusammenhänge".

Als Komplexitäts-Probleme bezeichne ich all diejenigen Fragestellungen, die aus der eigentlichen Aufgabe selbst und aus ihrer Größe resultieren und der entstehenden Software zuzuordnen sind. Hier geht es also vorwiegend um das Softwareprodukt, weniger um den Herstellungsprozess.

Der Aufbau der nachfolgenden Abschnitte ist jeweils: Einleitung/Definition, Problemnennung, Erläuterung.

3.1.1.1. Anforderungen (Problemraum)

Im Rahmen der Softwarekonstruktion müssen zwei Dinge geklärt werden:

1. **Was** wollen wir bauen (d.h. was soll die Software tun)?
2. Und **wie** können oder wollen wir sie bauen?

Die Anforderungen beschreiben das Was, also den relevanten Punkt im Problemraum der jeweiligen Softwareentwicklung.

Das Problem dabei liegt darin, dass in den meisten Fällen überhaupt **nicht klar** ist, was gebaut werden soll und es auch **schwierig herauszufinden** ist. Hier die wichtigsten Teilprobleme:

- A1 Die Leute, die es im Prinzip wissen sollten ("Fachexpert_innen" aus der Anwendungsdomäne, also dem thematischen Bereich der geplanten Software), können es meist nicht klar genug **ausdrücken**;
- A2 es fehlt ihnen oft an **Fantasie**, sich überhaupt auszumalen, was Software für ihre Zwecke leisten könnte -- vor allem, wenn es solche Software noch gar nicht gibt;
- A3 es gibt so viele **verschiedene Gruppen** von Fachexpert_innen, die unterschiedliche und oft widersprüchliche Anforderungen einbringen, dass es schwierig wird, eine Gesamtschau herzustellen;
- A4 die Fachexpert_innen verwenden eine **Terminologie**, die die Technikexpert_innen (also die Softwareingenieur_innen) nicht verstehen;
- A5 die Technikexpert_innen verwenden eine **Terminologie**, die wiederum die Fachexpert_innen nicht verstehen;
- A6 die Technikexpert_innen halten die Anforderungen in einer **Form** fest, die die Fachexpert_innen schlecht verstehen;
- A7 die Anforderungen **verändern sich** im Laufe der Zeit und zwar manchmal ganz erheblich und recht schnell;
- A8 die Anwendung ist vielleicht so **innovativ**, dass es überhaupt keine Fachexpert_innen dafür gibt.

Das Problem wäre meist einigermaßen einfach zu lösen, wenn es ausreichen würde, dass sich zwei Personen (Fachexpert_in, Technikexpert_in) einige Stunden austauschen, um eine Verständigung zu erreichen. Bei größeren Vorhaben steht dem jedoch die Komplexität im Wege: Man braucht viele Beteiligte und längere Zeit, niemand hat am Ende alle Einzelheiten im Kopf, und die Klärung ist dementsprechend schwierig zu beherrschen.

3.1.1.2. Entwurf (Lösungsraum)

Entwurf ist die Tätigkeit, die die zweite obige Frage "**Wie** wollen wir die Software bauen?" beantworten soll und zwar im Hinblick auf die Struktur der Software (Produkt), nicht auf das Vorgehen (Prozess).

Als Entwurf bezeichnet man auch das Ergebnis dieser Tätigkeit, also die Beschreibung der Struktur oder anders gesagt einen Punkt im Lösungsraum einer Softwareentwicklung.

Das Problem besteht darin, dass es **unzählige Möglichkeiten** für diese Struktur gibt, die **unterschiedliche Eigenschaften** im Hinblick auf den Aufwand für Ihre Umsetzung und auf die verschiedenen resultierenden **Qualitätsmerkmale** haben. Es ist schwierig, unter diesen Möglichkeiten die brauchbaren zu **erkennen** und die günstigste aus diesen **auszuwählen**. Teilprobleme sind:

E1 **Welche Lösungsmöglichkeiten gibt es *überhaupt***, um die Anforderungen umzusetzen?

E2 Welche davon könnten **akzeptabel** sein (im Hinblick auf Kosten und Qualität)?

E3 Wie wirkt jede davon auf jede der **Qualitätseigenschaften** (und zwar unter den gegebenen Randbedingungen, wie z.B. dem vorhandenen Personal)?

E4 Welche **Prioritäten** sollen die verschiedenen Qualitätseigenschaften bekommen?

E5 Wie sind diese Prioritäten gegeneinander **abzuwägen**? (Die meisten Qualitätseigenschaften haben keine normierbaren Maßeinheiten. Die einzig gemeinsame Bewertungsgrundlage ist meistens eine Umrechnung in Kosten.)

E6 Wie **ändern sich die Randbedingungen** (z.B. Personal, wünschenswerte Technologie) im Verlauf des Projekts und nach seinem Ende?

Siehe: 3.3

3.1.2. Prozess: Psycho-soziale Probleme

Die psycho-sozialen Probleme sind jene, die daraus hervorgehen, dass erstens zur Konstruktion einer großen Software viele Menschen zusammenarbeiten müssen und dass zweitens diese Menschen eben Menschen sind. Sie betreffen den Vorgang des Bauens (also den Softwareprozess) und nur indirekt das resultierende Produkt.

3.1.2.1. Kognitive Beschränkungen

Der menschliche Denkapparat ist zwar beeindruckend flexibel und leistungsfähig, aber in mancher Hinsicht auch sehr beschränkt. Die aus Sicht der Softwaretechnik wichtigste Beschränkung ist wohl die **Kapazität** des so genannten Arbeitsgedächtnisses. Dieses besteht, grob gesagt, aus dem Kurzzeitgedächtnis und der Aufmerksamkeitslenkung und im **Kurzzeitgedächtnis** kann ein Mensch nur etwa 5 bis 9 Elemente (chunks, Bündel) speichern.

3.1.2.2. Mängel der Urteilskraft

In vielen Fällen kann niemand der an einem Projekt Beteiligten eine gewisse Frage gut genug beurteilen, um eine fällige Entscheidung richtig zu treffen. Oft können es nicht einmal alle Beteiligten zusammen. Es fehlt also an einer ausreichenden Urteilskraft. Dies betrifft vor allem zwei Bereiche:

- **Radikales Vorgehen**, bei dem sich die **emergenten Eigenschaften**, des Systems vorher sehr wenig abschätzen lassen.
- Das **Rückgängigmachen falscher Entscheidungen**, das von psychologischen Effekten (Wahrnehmungsverzerrungen) sehr erschwert wird.

3.1.2.3. Wissen, Kommunikation und Koordination

Auch wenn an einer Stelle in einem Softwareprojekt das **Wissen** über eine nötige Maßnahme, über ein drohendes Risiko oder für eine richtige Auswahlentscheidung durchaus vorhanden wäre, ist es oft schwierig, dieses Wissen korrekt und rechtzeitig zu den Personen zu **transportieren**, bei denen es benötigt würde. Dieses Problem betrifft sowohl die Weitergabe inhaltlicher Information als auch die zeitlich-organisatorische Abstimmung von Tätigkeiten. Ursachen für dieses Problem sind

- zum Teil kognitive Beschränkungen oder Mängel der Urteilskraft,
- ungünstige Formen der Organisation und
- zum Teil soziale Phänomene wie individuelle Abneigungen, etwa gegen Kommunikation allgemein (Introversion), gegen bestimmte Personen (Aversion), gegen die Kommunikation bestimmter Arten von Informationen (Überbringung schlechter Nachrichten) usw.

3.1.2.4. Gruppendynamik

Die Haltungen (Einstellungen) zu gewissen Fragen und dadurch auch die Entscheidungsprozesse in Gruppen unterliegen einer Dynamik, die weit von einem kühl-rationalen Vorgehen abweichen kann. Das verstärkt oft andere Probleme, insbesondere Mängel der Urteilskraft ("Groupthink") und Schwächen in der Kommunikation.

3.1.2.5. Verborgene Ziele (hidden agendas)

Nicht alle Beteiligten an einem Softwareprojekt arbeiten zu jeder Zeit vorrangig auf das offizielle Projektziel hin. (Schlimmstenfalls wurde überhaupt kein solches Ziel formuliert.) Stattdessen verfolgen alle Beteiligten zu vielen Zeitpunkten zumindest nebenrangig auch persönliche Ziele. **Beispiele** dafür können sein: Selbst viel lernen; Spaß bei der Arbeit haben; möglichst elegante Lösungen finden (auf die man selber stolz ist); Achtung bei den Kolleg_innen finden; Achtung bei den Vorgesetzten finden; Gehaltserhöhung bekommen; früh nach hause gehen; faul sein können; spät nach hause gehen; bestimmte Kolleg_innen schlecht aussehen lassen; bestimmten Kolleg_innen aus dem Weg gehen; u.a.m. Solche persönlichen Ziele werden selten offen sichtbar verfolgt, sondern mehr oder weniger heimlich, was dem Ablauf des Softwareprojekts erheblichen Schaden zufügen kann.

3.1.2.6. Fehler

Fehler bezeichnet die Tatsache, dass ein Mensch bei dem Versuch, X zu tun, häufig versehentlich Y tut, ohne es sofort zu merken und ohne dass Y ein geeigneter Ersatz für X ist. Y kann auch die leere Menge sein (Versäumnisse). Das Resultat eines Fehlers ist häufig ein **Defekt** in der Software, der die Qualität der Software senkt. Insbesondere können Defekte dazu führen, dass die Software **versagt**, also in einem gewissen Moment nicht das tut, was sie tun sollte.

An sich sind Fehler stets nur eine **Erscheinungsform** irgendeines der oben genannten fundamentalen psycho-sozialen Probleme, aber sie sind in der Softwaretechnik so wichtig, dass es sich lohnt, sie separat zu betrachten.

3.2. Welt der Lösungsansätze

Die Welt der Lösungsansätze umfasst all das, was Softwareingenieur_innen an Ideen entwickelt haben, um die Ziele aus Abschnitt 3.3 besser zu erreichen und um insbesondere mit den Problemen aus Abschnitt 4.1 besser fertig zu werden.

In dieser Taxonomie behandeln wir hier nur hoch abstrakt die ganz grundlegenden Ansätze.

3.2.1. Technische Ansätze

Als technische Ansätze kann man all jene ansehen, die sich bei ihrer Anwendung so stark konkretisieren und schematisieren lassen, dass sie in wohldefinierte technische Formen wie Notationen oder Softwarewerkzeuge umgesetzt werden können.

3.2.1.1. Abstraktion

Abstraktion bedeutet die Darstellung eines Gegenstands oder Sachverhalts X in vereinfachter Form so, dass diejenigen Aspekte von X, die von Interesse sind, erhalten bleiben, während viele sonstige Aspekte (Details) entfallen.

Abstraktion ist das **Kernprinzip der gesamten Informatik**. Sie hat dementsprechend auch in der Softwaretechnik viele Ausprägungen. Gemeinsam ist ihnen, dass sich die Abstraktion in der Bildung eines **Begriff**s (**Konzept**s) ausdrückt und durch die Vergabe einer **Bezeichnung** benutzbar gemacht wird.

Solche Begriffsbildung verwendet die Softwaretechnik **für alle möglichen Zwecke**, insbesondere zur Beschreibung von Produkten (durch ihre Aufgaben oder deren Lösungen, also Anforderungen oder Entwürfe), Vorgehensweisen und Vorgängen.

Es gibt **allgemeine Begriffe** (beispielsweise für Konstrukte von Spezifikations- und Programmiersprachen, allgemeiner: definierte **Notationen** aller Art) und Begriffe, die **speziell für ein bestimmtes Vorhaben** erfunden werden (beispielsweise **Module**: Elemente zur Beschreibung eines Entwurfs als Zusammenwirken von Systemteilen (eben der Module), die einen starken inneren Zusammenhalt (Kohäsion) aufweisen, aber nur eine geringe Kopplung zu den restlichen Modulen. Die Abstraktion besteht darin, dass ein Modul auf Struktur und Verhalten seiner **Schnittstelle** reduziert wird.)

Abstraktion kann grundsätzlich (mit unterschiedlichen Graden von Wirksamkeit) als Lösungsansatz für alle der oben in 4.1 genannten Probleme eingesetzt werden.

3.2.1.2. Wiederverwendung

Den Lösungsansatz *Abstraktion* könnte man auch so beschreiben: Identifiziere eine nützliche Idee, gebe ihr einen Namen (um sich leichter auf sie beziehen zu können) und dann verwende die Idee immer wieder.

Die Wiederverwendung ist ein Spezialfall von Abstraktion, der noch einen Schritt weiter geht: Hier wird zusätzlich zu der Idee auch noch eine Ausprägung dieser Idee wiederverwendet, also eine gewisse Menge von Details.

Wiederverwendung kann nicht nur auf der Ebene von Code stattfinden, sondern **für alle Arten von Arbeitsprodukten und Prozesshilfsmitteln**. Beispiele für eventuell wiederverwendbare Arbeitsprodukte sind etwa Anforderungen, Anforderungsmuster, Architekturen, Teilentwürfe, Entwurfsmuster, Testfälle u.a. Beispiele für eventuell wiederverwendbare Prozesshilfsmittel sind Dokumentschablonen, Vorgehensbeschreibungen, Checklisten, Prozessmuster u.a.

Die Wiederverwendung und die Erzeugung wiederverwendbarer (Zwischen)produkte systematisch zu unterstützen ist insgesamt die wichtigste **Quelle von Produktivitätsverbesserungen** in der Softwaretechnik.

3.2.1.3. Automatisierung

Automatisierung in der Softwaretechnik ist quasi die Anwendung der Informatik auf sich selbst: Übertrage eine Tätigkeit, die im Rahmen des Softwareprozesses wiederholt erledigt werden muss, dem Computer, damit die menschlichen Beteiligten erstens Zeit (und damit Kosten) sparen und damit zweitens triviale Durchführungsfehler vermieden werden. Sind die Programme, die die Automatisierung erledigen, flexibel für unterschiedliche Situationen einsetzbar, so nennt man sie meist **Softwarewerkzeuge**.

Man kann Automatisierung als einen Spezialfall von Wiederverwendung ansehen.

3.2.2. Methodische Ansätze

Als methodische Ansätze zählen all jene, die nur schwach vorstrukturiert und auf menschliche Intelligenz und Kreativität angewiesen sind, um sie überhaupt verfolgen zu können.

Das scheint zunächst mal keine brauchbare Abgrenzung von den technischen Ansätzen zu sein: schließlich braucht man Intelligenz und Kreativität auch, um ein nützliches Diagramm in irgendeiner Notation zu zeichnen. Der grundsätzliche Unterschied besteht darin, dass sich dort aber zumindest ein Teil des Gesamtproblems sauber abspalten und universell lösen ließ (nämlich: eine hoffentlich nützliche Syntax und Semantik der Notation zu definieren). Die Abgrenzung bleibt zwar dennoch schwammig, hilft aber beim Nachdenken über Softwaretechnik.

3.2.2.1. Anforderungsermittlung

Kern von "Anforderungsermittlung" als einem methodischen Ansatz ist die Einsicht, dass man sich beim Bau von Software nicht einfach auf einen intuitiven Eindruck darüber verlassen darf, was gebaut werden sollte, sondern zuvor oder unterwegs die Anforderungen systematisch ermitteln sollte. Oberhalb dieses Kerns gibt es eine Reihe von Prinzipien, die

dabei zu beachten sind und oberhalb dieser Prinzipien folgen dann konkrete Methoden. Die Prinzipien lauten:

- **Erhebung**: Erhebe die Anforderungen **bei** möglichst vielen **verschiedenen** (Gruppen von) **Beteiligten**, denn keine einzelne Person überblickt alle Anforderungen.
- **Beschreibung**: Notiere die Anforderungen nach der Erhebung in einer Form, die die Beteiligten gut verstehen können
- **Validation**: Überprüfe die Anforderungen sodann nochmals bei und mit den Beteiligten, um Missverständnisse aufzudecken.
- **Spezifikation**: Überführe die Anforderungen anschließend evtl. in eine Form, die für die weitere Analyse und technische Realisierung besser geeignet ist.
- **Trennung von Belangen**: Nach Möglichkeit formuliert man Anforderungen so, dass sie möglichst **wenig** miteinander **gekoppelt** sind, so dass man sie leicht verstehen und bei Bedarf einzelne gut weglassen, verändern oder später wiederverwenden kann.
- **Analyse (Vollständigkeit)**: Versuche, durch geeignete (oft domänenabhängige) Kriterien, **Lücken** (Unvollständigkeit) in den Anforderungen aufzudecken und schließe sie dann durch Nacherhebung.
- **Analyse (Konsistenz)**: Versuche, durch geeignete (evtl. domänenabängige) Verfahren **Widersprüche** in den Anforderungen zu entdecken, die auf verbliebene Missverständnisse hindeuten, und löse sie auf.
- **Mediation**: Akzeptiere den Gedanken, dass manche Widersprüche nicht auf Missverständnissen basieren, sondern auf **Interessengegensätzen** zwischen den Beteiligten. Auch solche Widersprüche müssen **aufgelöst** werden (durch Entscheidung). Dies erfolgt idealerweise im Konsens, oft durch Kompromiss oder sonst durch Entdecken von Alternativen, die alle Beteiligten gutheißen können (**Win-Win**).
- **Verwaltung**: Immer, wenn sich während des Projektablaufs Änderungen an den Anforderungen ergeben, prüfe gründlich, ob sie **wichtig genug** sind, um die resultierende Störung des Ablaufs zu rechtfertigen und weise sie nötigenfalls zurück. Pflege ansonsten sorgfältig ein **stets aktuell gehalten**es Anforderungsdokument (oder eine Anforderungsdatenbank) und kommuniziere Veränderungen an die betroffenen Projektmitglieder.

3.2.2.2. Entwurf

Kern von "Entwurf" als einem methodischen Ansatz ist die Einsicht, dass man beim Bau von Software nicht einfach drauf los programmieren sollte, sondern sich zuvor Gedanken über die Struktur und Funktionsweise der Software machen sollte, um so erstens bessere Qualität (siehe 3.3) und zweitens arbeitsteilige Realisierung und Prüfung durch viele Softwareingenieur_innen zu ermöglichen.

Oberhalb dieser Grundidee gibt es eine Reihe von Prinzipien, die dabei zu beachten sind und oberhalb dieser Prinzipien folgen dann konkrete Methoden. Die Prinzipien lauten:

- **Trennung von Belangen** (separation of concerns): Verschiedene Belange der SW, insbesondere verschiedene Funktionen, sollten in der Umsetzungsstruktur klar voneinander getrennt sein, damit man sie separat von einander verstehen, realisieren und verändern kann. Dies führt zu einer Reduktion der Arbeitskomplexität.
- **Architektur**: Für Belange, die man nicht von anderen abzutrennen schafft, muss man **globale Überlegungen** anstellen, wie sie befriedigt werden sollen. Dies geschieht meist in Form einer Grobstruktur, genannt Architektur, die nur so detailliert beschrieben wird, dass man erklären kann, warum und wie die betreffenden Belange damit erfüllt werden können. Typische Beispiele für solche schwer abtrennbaren Belange sind nichtfunktionale Anforderungen (z.B. Geschwindigkeitsanforderungen, Sicherheitsanforderungen u.ä.).
- **Modularisierung**: Die Abtrennung von funktionalen Belangen erfolgt meist durch die Bildung so genannter Module. Ein Modul ist eine Programmeinheit, dessen Eigenschaften (vor allem das Verhalten) aus Sicht des restlichen Programms allein

durch seine **Schnittstelle** beschrieben werden, so dass die Schnittstelle viele Details der Realisierung vor dem Rest des Programms verbirgt (**information hiding**). Jede wichtige Entwurfsentscheidung, die sich möglicherweise später ändern könnte, sollte hinter einer Schnittstelle vor dem Rest des Programms verborgen werden. Die Entscheidung wird dadurch zu einem abgetrennten Belang und kann aufgrund der so erreichten **Lokalität** — die Entscheidung wirkt sich strukturell nur auf ein Modul aus — leicht revidiert werden. Ist die Realisierung des Moduls wesentlich komplizierter als die Schnittstelle (dies ist fast immer der Fall), dann führt Modularisierung außerdem zu einer Reduktion der Komplexität des restlichen Programms (z.B. beim Verstehen des Programms).

- **Wiederverwendung**: Für wiederkehrende Teilprobleme sollte man nicht jedes Mal einen Entwurf neu erfinden, sondern sich auf bekannte Lösungen oder Lösungsansätze abstützen: **Standard- und Referenzarchitekturen**, **Entwurfsmuster**.
- **Dokumentation**: Neben der Beschreibung des entstandenen **Entwurf**s selbst (in Form der **Modulschnittstellen**) sollten unbedingt auch alle **Entwurfsentscheidungen** samt ihrer Begründungen dokumentiert werden, damit die intendierte Systemgestaltung tatsächlich erfolgreich an die Entwickler_innen (jetzige und spätere) kommuniziert werden kann. Zu den Entwurfsentscheidungen gehört jeweils (a) der mit dem betroffenen Belang erreichte **Zweck**, (b) die Beschreibung der erwogenen **Alternativen** und (c) die Argumentation, **warum** man sich für welche dieser Alternativen entschieden hat.

3.2.2.3. Qualitätssicherung

Kern von "Qualitätssicherung" als einem methodischen Ansatz ist die Einsicht, dass man beim Bau von Software sehr leicht Fehler macht, die oft zu schwerwiegenden Mängeln im Produkt führen. Qualitätssicherung bezeichnet Maßnahmen, die mangelhafter Software vorbeugen wollen.

Oberhalb dieses Kerns gibt es eine Reihe von Prinzipien, die dabei zu beachten sind und oberhalb dieser Prinzipien folgen dann konkrete Methoden. Die Prinzipien lauten:

- **Konstruktive Qualitätssicherung:** Ergreife **vorbeugend**e Maßnahmen, die vermeiden helfen sollen, dass überhaupt erst etwas falsch gemacht wird.
- **Analytische Qualitätssicherung:** Verlasse dich nie ganz auf die Vorbeugung, sondern verwende zusätzlich **prüfend**e Maßnahmen, die entstandene Mängel aufdecken sollen, damit man sie beheben kann.

Die vorbeugenden Maßnahmen werden manchmal unter den Begriffen **Qualitätsmanagement** (das schließt evtl. die prüfenden mit ein) oder **Prozessmanagement** zusammengefasst. Zu den prüfenden Maßnahmen gehören eine Reihe recht verschiedener Techniken, insbesondere der **Softwaretest** (dynamische Prüfung) und die **Durchsichten** (manuelle statische Prüfung). Der Übergang zwischen vorbeugenden und prüfenden Maßnahmen ist fließend.

3.2.2.4. Projektmanagement

Kern von "Projektmanagement" als einem methodischen Ansatz ist die Einsicht, dass der Bau einer größeren Software ein Vorhaben ist, das nach zielgerichteter Leitung und Koordination verlangt. Projektmanagement ist die Lehre von den entsprechenden Maßnahmen.

Oberhalb dieses Kerns gibt es eine Reihe von Prinzipien, die dabei zu beachten sind und oberhalb dieser Prinzipien folgen dann konkrete Methoden, auf die wir hier nicht näher eingehen. Die Prinzipien lauten:

- **Zielsetzung**: Sorge dafür, dass die Ziele des Projekts (und deren Prioritäten) allen Beteiligten jederzeit klar bekannt sind und sie sich gut damit identifizieren.
- **Stabile Anforderungen**: Sorge dafür, dass sich die Anforderungen im Verlauf des Projekts nicht zu schnell verändern.
- **Iteration**: Sorge dafür, dass das Projekt in kurzen Abständen wohldefinierte Ergebnisse hervorbringt (Meilensteine). Idealerweise sind dies einsetzbare Versionen des Endprodukts.

- **Planung und Koordination**: Sorge dafür, dass alle Beteiligten zu jedem Zeitpunkt eine konkrete Aufgabe haben, sie diese verstehen, es ein sinnvolles zeitliches Ziel für die Erledigung gibt und die Einhaltung der Zeitplanung projektweit überwacht und sichtbar gemacht wird.
- **Kommunikation**: Sorge dafür, dass alle im Projektverlauf von jemandem benötigten Informationen diese Person zu einem geeigneten Zeitpunkt erreichen.
- **Konflikt**: Sorge dafür, dass im Projektverlauf (oder davor) entstehende Konflikte zwischen Beteiligten auf zufrieden stellende und zielführende Weise gelöst werden.
- **Risikomanagement**: Identifiziere vorbeugend die plausiblen unerwünschten Tendenzen und Ereignisse (Risiken), die den Erfolg des Projekts erheblich bedrohen. Entwickle für jedes Risiko entweder einen **Kontingenzplan** für den Fall des Eintretens oder leite **vorbeugende Maßnahmen** ein, die das Eintreten unwahrscheinlich machen. Überprüfe Risiken, Pläne und Maßnahmen regelmäßig, da sich Risiken im Projektverlauf ändern.
- **Normales versus radikales Vorgehen**: Halte Dich bei allen Aspekten des Projekts (Anforderungen, Entwurf, Vorgehensweise, Technologie) wo immer möglich an Ansätze, für die das Projektteam bereits ausreichende **Erfahrungen** besitzt (normales Vorgehen). Vermeide es, den Bereich solcher Erfahrungen zu verlassen (radikales Vorgehen), so lange nicht der davon erwartete Nutzen sehr hoch ist.

3.3. Arten von Softwaretechnik-Situationen

Die meisten Techniken und Methodiken der Softwaretechnik eignen sich nicht in jeder Situation in gleicher Weise und in gleichem Maß. Deshalb ist es wichtig, ein Verständnis für die grundlegend verschiedenen Zusammenhänge und Situationen zu haben, in denen Softwaretechnik betrieben wird. Hier folgt eine ganz knappe Beschreibung der wichtigsten Aspekte solcher Unterscheidungen.

3.3.1. Nähe und Anzahl der Kund_innen

- Fall 1: Individualsoftware
  - Es gibt nur eine_n Kund_in. Diese_r gibt die SW in Auftrag.
  - Die Kund_in ist konkret und für Auskünfte über die Anforderungen (mehr oder weniger gut) zugänglich
  - Beispiel: Betriebliches Informationssystem (das gibt es aber auch als Standardsoftware!)
- Fall 2: Standardsoftware
  - Es gibt (potentiell) viele Kunden. Eventuell Millionen.
  - Diese sind sehr verschieden in Bezug auf Wissen, Wünsche, Vorlieben, Bedürfnisse, Geduld, Risikofreude, Lern- und Innovationsbereitschaft.
  - Es ist schwierig, dieses Spektrum gut genug zu verstehen, um die "richtigen" Anforderungen für die SW zu ermitteln.
  - Beispiel: Videoschnitt-Software für Hobbyanwender

Die wichtigsten Unterschiede zwischen diesen beiden Fällen liegen bei Anforderungsermittlung (wie oben erwähnt) und Freigabe (release) der Software. Bei Individualsoftware erfolgt die Freigabe meist aufgrund eines expliziten Akzeptanztests. Bei Standardsoftware kann der Hersteller nach Gutdünken beschließen, in welchem Zustand er die Software auf dem Markt bringen will, büßt dann aber voreilige Freigaben mit einer Flut von Defektmeldungen und Unterstützungsanfragen und einem Ansehensverlust, der seine Marktstellung schwächt.

3.3.2. Art der Benutzer, Art der Benutzung

Art der Benutzer_innen:

- Fall 1: Geschulte, professionelle Benutzer_innen
  - Beispiel: SW-Ingenieur_innen (für ein API-basiertes SW-Produkt)

    - Beispiel: Fluglots_innen (für ein Flugsicherungssystem)
- Fall 2: Gering oder gar nicht ausgebildete Benutzer_innen
    - Beispiel: MS Word
    - Solche SW ist viel schwieriger gut hinzubekommen

Art der Benutzung:

- Fall A: Interaktiv und routinemäßig
    - Benutzbarkeit und Robustheit gegen Bedienfehler sind wichtig!
    - SW ist nur durch allmähliches Verbessern perfektionierbar
        - Benutzbarkeitsprüfung, Usability Engineering
- Fall B: Selten, unter Heranziehung von Dokumentation
    - Hohe Bedienkomplexität ist eher akzeptabel

In der Praxis gilt: Geschulte Benutzer_innen darf man immer nur ansatzweise als geschult betrachten; meist gelten die höheren Ansprüche an die Benutzbarkeit (also für wenig ausgebildete Benutzer_innen) mit nur punktuellen Abstrichen. Analog kommt auch der Fall B, dass es akzeptabel ist, zur Benutzung jedes Mal Dokumentation heranziehen zu müssen nur selten und dann nur punktuell vor.

3.3.3. Größe/Komplexität der SW und des Projekts

- Fall 1: Klein
    - z.B. zwei Entwickler_innen für wenige Wochen
    - Viele Methoden der SWT sind unnötig oder gar kostenschädlich
    - Verletzung der nötigen Methoden macht den Erfolg zwar mühselig, aber nicht unmöglich
- Fall 2: Groß
    - z.B. Hunderte (oder Tausende) Entwickler_innen über mehrere Jahre
        - Kosten z.B. 10.000 Personenjahre, entspricht ca. 1 Milliarde EUR
    - Beispiele: Microsoft Windows, SAP ERP
    - Kompetent angewandte SWT ist die einzige Chance für Erfolg
    - Eine riesige Palette von Methoden ist zu erwägen
    - Projektmanagement wird extrem bedeutsam

Hier gibt es natürlich ein Kontinuum zwischen diesen beiden Extremen. Man könnte sagen, Softwaretechnik ist die Lehre von der Bewältigung **nicht-kleiner Vorhaben**. Verwandt mit dem Faktor Größe sind auch einige andere Faktoren, die typischerweise mit der Größe korrelieren. Z.B. hat sehr große Software meist auch eine lange Lebensdauer (Jahrzehnte) und macht deshalb insgesamt viele Veränderungen durch; sie muss also gut wartbar sein und immer wieder besser wartbar gemacht werden.

3.3.4. Wie kritisch ist (nichttechnisches) Domänenwissen ?

- Fall 1: Eine rein softwaretechnische Domäne
    - Beispiel: Betriebssystemkern, Compiler
    - Hier verstehen die SW-Ingenieur_innen ohne fremde Hilfe die Anforderungen recht gut
- Fall 2: Eine Domäne mit komplexer Fachlichkeit
    - Beispiel: SW zur Berechnung des Kreditausfallrisikos einer Bank
    - Hier sind die SW-Ingenieur_innen bei den Anforderungen naiv
    - Die Anforderungen müssen von Fachspezialist_innen geliefert werden
        - Kommunikationsproblem!

Eine Warnung: Softwareingenieur_innen unterschätzen fast immer die fachliche Komplexität einer Domäne. Und selbst bei scheinbar rein softwaretechnischen Domänen ist häufig

eine verborgene Fachlichkeit vorhanden, z.B. die Fachlichkeit der Bedienvorgänge, wenn die Software eine interaktive Bedienschnittstelle hat.

3.3.5. Müssen Näherungslösungen verwendet werden?

- Fall 1: Nein, nirgends
  - Man kann die funktionalen Anforderungen genau erfüllen (zumindest im Prinzip)
  - Das ist bislang der Normalfall
- Fall 2: Niemand weiß, wie man das Problem wirklich lösen kann
  - Beispiele: Spracherkennung, Video-nach-Text-Transkription, alles was "intelligent" ist und z.B. Machine Learning einsetzt
  - Solche Aufgaben sind viel schwieriger, insbesondere bei der Qualitätssicherung, und verlangen andere Herangehensweisen

Der Fall 1 ist bislang der häufigere, aber Fall 2, also "intelligente" Software, nimmt an Bedeutung kontinuierlich zu. Bei Fall 2 ist während der Entwicklung die Urteilskraft (siehe 4.1.2.2) über die richtige Vorgehensweise noch stärker gefordert.

3.3.6. Wie kritisch ist Effizienz?

- Fall 1: Hauptspeicher, Massenspeicher, Netzbandbreite und CPU-Leistung sind "im Überfluss" vorhanden
  - Beispiel: Textverarbeitungsprogramm auf modernem PC
- Fall 2: Ressourcen sind knapp und müssen sparsam eingesetzt werden
  - Beispiel: SW für ein mobiles Gerät (z.B. Smartphone: Batterie sparen!)
  - Beispiel: SW für eine numerische Simulation (z.B. Wetter, Klima)
  - Beispiel: SW für Echtzeitanwendungen (z.B. Computerspiel, Fabrikationssteuerung)
  - Beispiel: SW für massiv verteilte Anwendungen (z.B. peer-to-peer)

Fall 2 bedeutet eine starke Verschärfung nichtfunktionaler Anforderungen und eine dementsprechend erweiterte Rolle von Software-Architektur.

3.3.7. Wie kritisch ist Verlässlichkeit?

Unter Verlässlichkeit verstehen wir hier die Kombination aus Zuverlässigkeit (Auswirkung der Korrektheit: Wie oft versagt die Software?), Schutz (security: Wie gut wird verhindert, dass ein_e Angreifer_in etwas Unerwünschtes mit oder an dem System anstellt?), Verfügbarkeit (Wie häufig und wie lange ist das System faktisch nicht dienstbereit?) und Sicherheit (safety: Wie gut wird verhindert, dass das System etwas Unerwünschtes in seiner Umgebung bewirkt?)

- Fall 1: Wenn die SW versagt, kann ein Mensch das abfedern
  - Beispiel: Fehlerhafte Silbentrennung in der Textverarbeitung
  - Beispiel: Unsinnige Vorschläge eines Routenplaners
- Fall 2: Wenn die SW versagt, geschieht etwas Schlimmes
  - Beispiel: Autopilot eines Verkehrsflugzeugs
  - Beispiel: Fahrassistenzsystem im Auto
  - Beispiel: Kontoführungssoftware einer Bank
  - Beispiel: Steuerung einer automatischen Insulinpumpe

Diese Unterscheidung gilt ausnahmslos immer nur graduell: Auch bei weitgehend-Fall-1 ist ein Schaden bei mangelnder Verlässlichkeit immer anzunehmen (und sei es nur ein Schaden am Ruf des Herstellers) und auch bei überwiegend-Fall-2 muss das Gesamtsystem immer Möglichkeiten vorsehen, wie ein seltenes Versagen zumindest potentiell von Menschen noch aufgefangen werden kann (siehe die Beispiele in der Vorlesung "Auswirkungen der Informatik").

4. Lerntipp: ABC-Klassifkation der Folien

Die obige Taxonomie von Problemen und Lösungen ist ein Wegweiser zur Orientierung innerhalb der Softwaretechnikvorlesung (und des Softwaretechnikwissens ein Leben lang). Allerdings verlangt es Übung, eine Frage oder Aussage in dieser Taxonomie richtig einzuordnen. Zugleich ist das Klassifizieren und Einordnen eine ganz zentrale Qualifikation für eine_n Softwareingenieur_in, sollte also ohnehin fleißig geübt werden.

Eine Gelegenheit dazu bieten die Vorlesungsfolien. Man denke sich zu jeder Folie mit Sachaussagen eine Kennzeichnung, die eine Grobeinordnung beschreibt wie folgt: Die Kennzeichnung ist ein Buchstabencode, z.B. "AdL", und besteht aus drei Teilen A, B, und C (in dieser Reihenfolge).

- **A (Allgemeinheit):** Auf der Folie geht es entweder um sehr allgemeine **Prinzipien (P)** oder um halbwegs **allgemeine Aussagen** der Softwaretechnik **(A)** oder um ein spezielles **Beispiel (B)**. Manche Folien sind **Metafolien (M)**, sagen also nichts über die Softwaretechnik, sondern etwas über andere Folien (z.B. Inhaltsübersichten, Fragen).
- **B (Bereich):** Die Aussagen beziehen sich vorwiegend auf den **Prozess (z)** oder das **Produkt (d)**.
- **C (Charakter):** Die Aussagen beschreiben überwiegend **Probleme/Aufgaben (A)** oder **Lösungen/Lösungsansätze (L)**.

Der genannte Code "AdL" könnte also Beispielsweise auf einer Folie stehen, die ein *Entwurfsmuster* beschreibt, denn das Muster ist allgemein (nicht nur illustrativ), also A, es beschreibt eine Lösung, also L, und bezieht sich auf die entstehende Software, also d, nicht auf das Vorgehen. Nicht jede Folie lässt sich auf diese Weise klar einordnen, aber der Versuch es zu tun schult das konzeptuelle Denken ungemein.

5. Was muss man wissen?

Die Menge an Wissen, über die ein_e Softwareingenieur_in idealerweise verfügen sollte, ist beängstigend groß. Mich deprimiert diese Tatsache auch nach Jahren immer noch jedes Mal, wenn ich eine einschlägige Zeitschrift aufschlage und merke, was ich alles nicht weiß.

Aber natürlich ist nicht alles gleich wichtig. Manches Wissen (das vielleicht noch nicht einmal schwierig oder umfangreich ist) ist wesentlich wirksamer für eine gute professionelle Leistung als anderes Wissen. Und darauf sollte folglich die Aufmerksamkeit in dieser Vorlesung zuerst gerichtet sein.

5.1. Prioritäten in dieser Vorlesung

Hier die wichtigsten Überlegungen für die Prioritäten, die man setzen sollte, wenn und weil man nicht alles sofort wissen kann. Wichtiger Hinweis: Diese Prioritäten beziehen sich nur auf **diese Vorlesung** und sind im Sinne einer **zeitlichen Priorität (also Lernreihenfolge)** zu verstehen. Eine Aussage z.B. der Art "Methodenwissen ist wichtiger als Technologiewissen" ist unsinnig, weil man für erfolgreiches Arbeiten in der Praxis natürlich beides braucht und die "Mengen" solchen Wissens nicht vergleichbar sind.

Im Rahmen dieser Vorlesung können und sollten Sie aber vor allem solches Wissen erwerben, dass Sie später kaum je wieder explizit vermittelt bekommen — viele **Firmen** schicken Ihre Mitarbeiter_innen gelegentlich auf Kurse, in denen technologisches Wissen vermittelt wird, aber selten auf solche, in denen methodisches, und kaum je auf solche, in denen allgemeines oder grundlegendes Wissen vermittelt wird.

5.1.1. Grundwissen vor Vertiefungswissen

Für eine gute Urteilskraft (siehe 4.1.2.2) braucht man vor allem ein ganzheitliches Verständnis eines Problems. Um ein ganzheitliches Verständnis zu erreichen, braucht man ein (ungefähres) Verständnis für jeden relevanten Teilbereich.

Deshalb sollte der Erwerb von Vertiefungswissen im Bereich X möglichst nie auf Kosten des Erwerbs von Grundwissen im Bereich Y gehen, wenn man erwarten muss, bei seinen Urteilen Aspekte von Y mit zu benötigen.

An die Vertiefung geht es also erst, wenn man in allen relevanten Bereichen sein Grundwissen beisammen hat.

5.1.2. Allgemeinwissen vor Spezialwissen

Mit genau derselben Überlegung ist auch der Erwerb von Wissen in eher allgemeinen Bereichen im Konfliktfall oft vorzuziehen gegenüber dem Erwerb von Wissen in mehr spezialisierten Bereichen.

An die Spezialisierung (selbst mit Grundwissen!) geht es also möglichst erst, wenn man in allen relevanten Allgemeinbereichen ein Grundwissen beisammen hat.

Achtung: Spezialwissen ist regelmäßig nötig, um normales Vorgehen (siehe 4.2.2.4) möglich zu machen; Allgemeinwissen allein reicht nicht!

5.1.3. Methodenwissen vor Technologiewissen

In den meisten Bereichen der Informatik wird man erst richtig handlungsfähig, wenn man eine erhebliche Menge von Wissen über die jeweils eingesetzte Technologie hat. Deshalb neigen manche Informatik-Studierende dazu, technologisches Wissen für das einzig relevante zu halten. Diese Haltung ist für Softwaretechnik ungünstig aus folgenden Gründen:

- Sie ist selbst bei den technologielastigsten Themen meist übertrieben, weil ein gutes Verständnis der zu Grunde liegenden Konzepte für die korrekte Verwendung der Technologie entscheidend ist.
- Die Softwaretechnik ist eine Methodenlehre; Technologie hat in diesem Zusammenhang immer nur eine Hilfsfunktion. Wer also die Methodik ignoriert, hat die Softwaretechnik ihres Gehaltes weitgehend beraubt.
- Außerdem hat das Lernen auf der Methodenebene besonders gutes Kosten-Nutzen-Verhältnis, weil methodisches Wissen weitaus weniger schnell veraltet als technologisches.
- Nur wenige der fehlschlagenden Softwareprojekte scheitern aus technischen Gründen! (Sondern? Siehe 3.1.2)
- Schließlich gilt das schon oben angeführte Argument, dass technologische Fortbildung alltäglich ist, methodische Fortbildung sich aber meist auf reines Erfahrungslernen beschränkt. Erfahrungslernen ist zwar wirksam, aber ineffizient: Sie können sich kaum leisten, etwas dadurch zu lernen, dass sie ein paar Hunderttausend Euro Investition in den Sand setzen.

5.2. Prioritäten für die Praxis

5.2.1. Methodisches Wissen

Sie sollten über alle grundlegenden Wissensbereiche der Softwaretechnik (also Anforderungen, Entwurf, Qualitätssicherung, Projektmanagement) **genügend** methodisches Wissen besitzen, **um in mäßig komplexen Projekten** der Ihnen vertrauten Domäne(n) **sicher vernünftig agieren** zu können. Dazu ist meist nur ein sehr kleiner Teil allen methodischen Wissens nötig, weil das Meiste in einer bestimmten Situation ohnehin nicht sinnvoll ist – hüten Sie sich aber davor, *zu viel* Methodik fallen zu lassen.

5.2.2. Technologisches Wissen

Sie sollten über die von Ihnen regelmäßig benutzten Technologien zwei Arten von Wissen besitzen:

- **Breites und sehr solides Wissen über die konzeptuellen Grundlagen** der Technologie. Moderne Technologien sind sehr komplex und ohne solches Grundwissen können Sie an vielen Stellen nur herumstümpern anstatt professionell zu agieren.

- **Ausreichendes Wissen über diejenigen speziellen Details** der Technologie, die für Ihre Arbeit aktuell von Belang sind. Wenn Sie etwas nicht wissen und erfragen oder nachlesen müssen: Lernen Sie es richtig und samt Hintergrundwissen. Geben Sie sich nicht damit zufrieden, dass Ihnen jemand ohne Begründung sagt, was Sie minimal tun müssen, um Ihr Problem zu lösen, sondern **klären Sie, was wirklich dahinter steckt**.

5.2.3. Domänenwissen

Wertschöpfung entsteht aus softwaretechnischem Wissen erst, wenn man es einsetzt, um damit Lösungen für relevante Probleme zu schaffen. Diese Probleme existieren nicht im luftleeren Raum und sind auch nicht ohne Weiteres zu erkennen und zu verstehen. Deshalb brauchen Sie über die Domäne, in der Sie arbeiten, **genügendes Wissen, um betroffene Probleme** oft selbständig **erkennen und** nach Erläuterung stets zuverlässig **verstehen zu können**.

5.2.4. Eigene Grenzen erkennen und sich fortentwickeln

In allen drei Bereichen – Methodik, Technologie und Domäne – wird Ihr Wissen immer beschränkt bleiben und immer mal wieder nicht ausreichen, um mit der aktuellen Situation angemessen umzugehen. Sie sollten die Fähigkeit entwickeln zu bemerken, wenn dies eintritt, sowie den Drang, dann (also bei konkretem Anlass) Ihr Wissen entsprechend ein Stück fortzuentwickeln. Aus solchem Verhalten entsteht das berühmte lebenslange Lernen und dadurch eine dauerhaft gefestigte Fähigkeit, Werte zu schöpfen.

5.3. Softwaretechnik als Lehre vom Abwägen

Es gibt noch einen anderen Blickwinkel, aus dem man die Frage "Was muss man wissen?" beantworten könnte. Man könnte sagen, dass die Softwaretechnik davon handelt, in jeder Situation aus den vielen Möglichkeiten, wie man etwas machen kann, die beste (oder jedenfalls eine günstige) auszuwählen. Aus dieser Perspektive ist also der im Abschnitt 3.1.2.2 besprochene Mangel an Urteilskraft das Kernproblem der Softwaretechnik. Um es zu lösen, muss man geeignete Abwägungen zwischen den Vor- und Nachteilen einer Möglichkeit im Vergleich zu allen übrigen Möglichkeiten machen.

Was muss man also wissen? Genug, um in jeder Situation die Vor- und Nachteile so weit zu überblicken, dass man eine brauchbare Abwägung hinbekommt – keine optimale, gut genug reicht. Als Voraussetzung gehört dazu natürlich, dass man überhaupt erst mal die Handlungsmöglichkeiten kennt.

Ich wünsche viel Spaß beim Überblick-Gewinnen und beim Abwägen-Lernen!

6. Übungen zu diesem Dokument

1. Lesen und verstehen Sie dieses Dokument
2. Identifizieren Sie mindestens 5 Stellen, an denen Ihr Verständnis des Dokuments Ihnen ungenau erscheint. Beschreiben Sie, worin die Ungenauigkeit besteht.
3. Identifizieren Sie mindestens 3 Stellen, an denen das Dokument falsche oder zumindest zu pauschale Aussagen macht.
4. Identifizieren Sie mindestens 3 Stellen, an denen das Dokument vermeidbar mehrdeutige Aussagen macht.
5. Geben Sie zu jedem Aufzählungspunkt in Abschnitt 3.2 ein Beispiel an.
6. Warum sind in Abschnitt 3.1.1 nicht auch Validierung und Wartung als Probleme aufgeführt?