Software Engineering Rationale: Wissen über Software erheben und erhalten

Kurt Schneider

Lehrstuhl Software Engineering
Leibniz Universität Hannover
Welfengarten 1, 30167 Hannover
Kurt.Schneider@inf.uni-hannover.de

Abstract: Damit Software lange genutzt werden kann, muss sie fortwährend korrigiert, erweitert und verändert werden. Dabei werden Entscheidungen und Annahmen getroffen, Domänenwissen wird in die Software eingearbeitet und in Modellen interpretiert. Programme zu warten, bei denen dieses Wissen fehlt, ist schwierig. Das ergänzende Wissen müsste eigentlich zusammen mit dem Programm aufgebaut, gepflegt und weitergegeben werden. Auf längere Sicht kann sich sogar das Programm selbst als weniger langlebig erweisen als das damit assoziierte Wissen. Dieser Beitrag betont den komplementären Charakter von Programm und Rationale, codiertem und ergänzendem Wissen und zeigt Forschungsherausforderungen auf dem Weg, diesem Charakter gerecht zu werden.

1 Einleitung

Von Anforderungsanalyse bis zu Test und Wartung werden unzählige große und kleine Entscheidungen gefällt und Informationen über Software in diese eingearbeitet. Verliert man das Wissen über die Gründe (das „Rationale“), so verliert man zunehmend auch die Fähigkeit, die Software zu pflegen. Durch innovative Techniken und Werkzeugen soll es gelingen, dieses Zusatzwissen zu sichern, ohne viel spürbaren Zusatzaufwand zu treiben. Entwurfsbegründungen, Erfahrungen und Feedback aus dem Betrieb helfen dann, die Software beweglich und nützlich zu erhalten.

Software altert relativ zu den veränderten Kontextbedingungen: Mitarbeiter gehen und nehmen ihr Wissen mit. Neue Mitarbeiter kommen und verändern die Software, ohne frühere Entscheidungen und ihre Grundlagen zu kennen. Nur wenn man schnell genug bemerkt, dass sich da eine Lücke auftut, kann man reagieren und die Veralterung aufhalten. Das gilt nicht nur in der Entwicklung, sondern auch später im Betrieb der Software. Zusammen mit dem *Programm* muss auch *Wissen* bewahrt werden.

Je nach Art und Granulat des bewahrten Wissens kann es in unterschiedlicher Weise später genutzt werden, wie die folgenden Beispiele zeigen.

Dabei geht es um ein weites Spektrum von Aspekten, nicht nur um Codefeinheiten:

- Es ist wichtig zu wissen, ob ein Programm eigentlich als „Prototyp“ gedacht war und dann doch weiter gewachsen ist. Das erklärt viele Besonderheiten und Schwächen; man glaubt ja anfangs, der Prototyp werde bald weggeworfen.
- Wenn eine Sicherheitsnorm wie ISO 61508 der Grund dafür war, sicherheitskritische von eher unkritischen Programmteilen zu trennen, so muss man das später wissen – sonst sind die darauf aufbauenden Architekturentscheidungen nicht nachvollziehbar und werden vielleicht verletzt.
- Begriffe, Regelungen und Terminologie aus einer Domäne können sich für spätere Entwickler „falsch“ anhören, weil diese die Domäne nicht kennen. Trotzdem sollte man sie beibehalten.
- Am codenahen Ende des Spektrums muss man wissen, wo und warum ein Design Pattern angewendet wurde, damit man es nicht später „kaputtoptimiert“.

IEEE Std 610.12 – 1990 definiert *Software* als „Computer programs, procedures and possibly associated documentation and data pertaining to the operation of a computer system”. Es kann geschehen, dass dieses ergänzende Wissen in Form von Modellen, Entscheidungen und Erfahrungen wichtiger wird als das Programm selbst. „Die Software“ lebt weiter, obwohl das ursprüngliche Programm ersetzt wird.

Mit einem Softwareprodukt muss ein Schatz technischer und umgebungsbezogener Informationen wachsen. Aufgebaut wird er mit möglichst wenig Zusatzaufwand, ja vielleicht sogar mit unmittelbarem Zusatznutzen schon während der Softwareentwicklung. Denn kaum ein Softwareentwickler ist bereit, Zeit und Mühe zu investieren, nur damit es die Nachfolger leichter haben. Eine der zentralen Herausforderungen für die Bewahrung des Softwarewissens steckt in dieser Tatsache: Oft wird beispielsweise ein scheidender Experte aufgefordert, sein Wissen „aufzuschreiben“. Das dauert viel zu lange und nützt nur dem Nachfolger! Wer aber einen Beitrag zur Wissensbewahrung leistet, muss selbst erkennbaren Nutzen irgendeiner Art haben. Gegen diese scheinbar so triviale Einsicht wird ständig verstoßen. Könnte der Experte sein Programmteil zum Beispiel einfach mündlich demonstrieren und erklären, wäre nicht nur sein Aufwand geringer (siehe FOCUS-Beispiel im Hauptteil). Seine Expertenrolle wird herausgehoben, was mit sozialer Anerkennung verbunden ist. Die ist oft viel wirksamer als etwas Geld.

In Abschnitt 2 werden Beispiele von assoziiertem Wissen gegeben, das im Zusammenhang mit Programmen – und als Teil von „Software“ – eine wichtige Rolle für Wartung und Langlebigkeit spielt. An diesen Beispielen wird gezeigt, was mit dem komplementären Charakter von Programm und Wissen gemeint ist. Auf dieser Basis kann Abschnitt 3 skizzieren, wie sich dieses Wissen auf die Langlebigkeit von Software im weiteren Sinne auswirkt. Abschnitt 4 beschreibt, worauf bei der Erhebung zu achten ist. Die Erhebung ist nicht als isolierte Wissensmanagement-Aufgabe konzipiert, sondern als Koevolution von Wissen und Programmartefakten angelegt. Damit wird die Wissenserhebung als integraler Bestandteil der Softwareentwicklung interpretiert. In so einer Konstellation werden sich „eigentliche Softwareentwicklung“ und „Wissensarbeit“ gegenseitig beeinflussen.

Da Wissen oft langlebiger ist als Programme, kann sich das anfängliche Verhältnis langfristig umkehren (Abschnitt 5). Dann wird das Wissen die treibende Kraft, an der das Programm hängt. Schließlich diskutiere ich in Abschnitt 6 einige wichtige Forschungsfragen, die sich aus diesem Ansatz für die Zukunft ergeben.

2 Rationale, Erfahrungen, Entscheidungen – Basis für Langlebigkeit

Ein wesentliches Postulat dieses Beitrags lautet:

> In Zusammenhang mit Softwareentwicklung wird viel Wissen zusammengetragen und erstellt, von dem sich nur ein kleiner Teil in den eigentlichen Programmen wiederfindet. Das ergänzende Wissen bleibt oft undokumentiert. In der Wartung fehlt es dann, und das verursacht Strukturbrüche und Fehler, macht die Altsoftware letztlich unbrauchbar.

Beim Übergang von Requirements Engineering zu Entwurf hat sich diese Einsicht seit langem durchgesetzt. Es gibt zahlreiche Arbeiten, die sich mit dem „Design Rationale" auseinandersetzt, also mit den Begründungen für Entwurfsentscheidungen [DMMP06]. Beides, die Entscheidungen und die Gründe dafür, stellen Meta-Informationen (in Anlehnung an Metadata) dar, die nicht erklären, *wie* zum Beispiel ein Algorithmus funktioniert – sondern *wieso* er vor dem Hintergrund der konkreten Anforderungen und *mit welcher Begründung* ausgewählt worden ist. Diese Begründung steckt nicht in den Programmzeilen und kann von dort auch nicht extrahiert werden (von den Kommentaren einmal abgesehen). Dagegen enthalten die Programmzeilen das ganze Wissen über die Funktionsweise des Algorithmus; sie sind ja dessen operative Implementierung. Es mag auch Aspekte geben, die sowohl im Code als auch im ergänzenden Wissen ihre Spuren hinterlassen. Das Entwicklungswissen steckt also in der *Kombination* von Programmcode und zugehörigem Rationale (Abbildung 1). In sofern verwundert es nicht, dass man den nackten Code ohne zugehöriges Wissen irgendwann nicht mehr verstehen und pflegen kann.

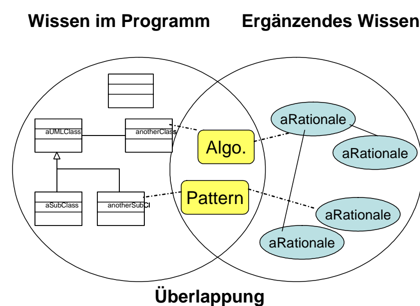

Abbildung 1: Wissen in der Softwareentwicklung landet im Programm oder wird zu ergänzendem Wissen. Manches trägt zu beidem bei (Prinzipdarstellung).

Die Rolle der Dokumentation im Software Engineering wird seit langem betont [Bo81]. Dabei ist jedoch nicht die Tatsache entscheidet, dass es möglichst viel Dokumentation gibt; dagegen wehren sich nicht nur die agilen Methoden mit gutem Recht [Be00, C02]. Es kommt vor allem darauf an, welche Inhalte dabei erfasst werden und wie man sie wieder nutzen kann. Die traditionellen Dokumente enthalten oft tautologische Beschreibungen von Programmstrukturen, die man leicht aus dem Code selbst extrahieren kann. Dabei handelt es sich also nicht um ergänzendes Wissen.

Verschiedene **Typen von ergänzendem Wissen** sind besonders wichtig, werden aber häufig nicht erfasst:

- **Annahmen** über die Randbedingungen. Ändern sich die Bedingungen, so können sich auch darauf aufbauende Entscheidungen ändern.
- **Domänenwissen**: Zusammenhänge im Anwendungsbereich schlagen sich einerseits direkt im Code nieder, andererseits bilden sie die Begründung für Entscheidungen.
- **Prüfergebnisse**: Review- und Testberichte können Informationen zur Bewertung von Code liefern. Diese sind nicht aus dem Code selbst abzulesen.
- **Feedback** aus der Programmnutzung: Rückmeldungen der Nutzer können Divergenzen zwischen Annahmen und realer Situation oder zwischen Anforderungen und Programmeigenschaften erkennen lassen. Sie sind ebenfalls naturgemäß nicht im Code repräsentiert.
- **Erklärungen über das Umfeld oder absehbare Änderungen**. Sie können sich teilweise in flexiblerem Code niederschlagen (z.B. durch den Einsatz von Design Patterns [GHJV95]).
- **Erfahrungen** verschiedener Art entstehen während der Entwicklung und Wartung eines Programms. Zum Beispiel kann man feststellen, dass ein Modul bisher besonders fehleranfällig war und bei neuen Fehlern zuerst geprüft werden sollte. Design Patterns sind typische Resultate von Erfahrungen.
- **Modelle**: Sie stellen gewisse relevante Merkmale der Software dar.

Die Liste ist sicher nicht vollständig. Sie nennt vor allem Informations- und Wissensarten, die Begründungen, Bewertungen und darauf aufbauende Schlussfolgerungen oder Entscheidungen enthalten. Daher ist es sinnvoll, kurz von „Software Engineering Rationale“ zu sprechen. Der Begriff schließt „Design Rationale“ ein, geht aber darüber hinaus.

Eine besonders wichtige Rolle für die Vermittlung zwischen einem Programm und dem Wissen über dieses Programm spielen zunehmend „Modelle“, oft in UML. Wie bereits von Stachoviak [St73] 1973 festgestellt, erweist ein Modell seine Nützlichkeit stets im Hinblick auf ein *Original*, einen *Modellzweck* und einen *Nutzer*. Das Original von UML-Entwurfsmodellen kann beispielsweise der später erzeugte Code sein. Das UML-Modell wird zum Zweck der Codeerstellung (manuell oder generiert) entwickelt, entweder für Entwickler oder für Generatoren.

In der MDA [PM06] oder der Modellgetriebenen Software-Entwicklung [SV05] wird Code aus den Modellen generiert. Damit enthält der Code keine zusätzlichen Informationen, die Modelle allein sind die Träger des Wissens.

An dieser Stelle sind zwei alternative Konsequenzen denkbar:

- Entweder werden die Modelle (anstelle des Codes) mit ergänzendem Wissen verbunden. Das Gesamtwissen der Software ist dann in zwei verschiedenen, jedoch miteinander verbundenen Modellen repräsentiert: im UML-Modell und im Wissensmodell.

- Oder der Modellzweck nach Stachoviak wird umdefiniert und erweitert: Das Modell dient dann nicht mehr nur der Codeerstellung, sondern außerdem der Sicherung ergänzenden Wissens. In diesem Fall wären Metamodelle zu erweitern. Neben Codegenerierung wären Wissensspeicherung und -pflege die Zwecke des Modells. Die Adressaten im Stachoviakschen Sinn sind dann nicht mehr allein die Erstentwickler, sondern alle, die im Lebenszyklus einer Software an dieser arbeiten.

Da sowohl UML-Modelle als auch Ontologien formal fassbar und sogar strukturell ähnlich sind, gibt es viele Möglichkeiten, sie miteinander zu verbinden. Abbildung 2 zeigt eine davon.

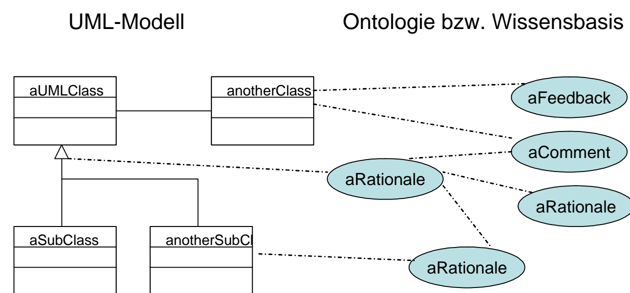

Abbildung 2: Verschiedene, aber verbundene Modelle für Produkt und Wissen bzw. Rationale

Diese Überlegung verdeutlicht, wie unterschiedlich technische Realisierungen für ergänzendes Wissen aussehen können. In den folgenden Abschnitten wird für die enge Integration und Verbindung argumentiert. Dafür bieten Metamodellerweiterungen mit UML-Profilen natürlich einen interessanten technischen Ansatz. Dieser Aspekt ist hinlänglich bekannt und soll daher hier nicht weiter ausgeführt werden.

Andererseits sollte man auch nicht aus den Augen verlieren, dass aus dem Wissens-management Konzepte wie Ontologien zur Verfügung gestellt werden, die den Aspekt der Wissensrepräsentation gezielter abdecken als ein kurzfristig erweitertes UML-Metamodell. Es wird eine der Forschungsfragen sein, hier eine Abwägung zu treffen.

3 Lebenszyklus von Software und “ergänzendem Wissen”

Software wird eingesetzt, so lange sie nützlich ist, also ihren Betreibern einen Nutzen verspricht. Weichen die Funktionen oder die nicht-funktionalen Eigenschaften zunehmend von den – möglicherweise veränderten – Erfordernissen ab, wird eine Möglichkeit gesucht, die Abweichung wieder zu verringern. In der Regel wird man versuchen, die Software an die neue Situation anzupassen. Wenn das zu aufwändig erscheint oder überhaupt nicht mehr geht, beginnt man, das eigene Verhalten und die eigenen Geschäftsprozesse an die Möglichkeiten der alternden Software anzugleichen. Das ist ein deutliches Zeichen für das nahende Ende des Software-Lebenszyklus. Symptomatisch ist auch, wenn keine Verbesserungen mehr gewagt werden. Die Software kann nur noch so verwendet werden, wie sie gerade ist. In dieser Phase des Lebenszyklus wird die Software als Last oder Legacy wahrgenommen, die nur deshalb weiter eingesetzt wird, weil es kurzfristig keine Alternative gibt.

Man kann sich die obigen Effekte aus den Wirkzusammenhängen zwischen dem Programm und dem ergänzenden Wissen erklären. In diesem Ablauf zeigt sich eine typische, aber nicht wünschenswerte Entwicklung, die man aufhalten müsste:

1. Während der Entwicklungsphase tragen die Kunden, Entwickler und anderen Projektbeteiligten die wichtigsten Informationen in sich. Es gibt persönliche Notizen, Protokolle und auch Modelle. Entscheidungen werden auf dieser Basis wohlüberlegt und gut informiert getroffen.

2. Durch die hohe Geschwindigkeit der Entwicklung und durch den meist hohen Zeitdruck werden die Aufzeichnungen nicht mehr systematisch erfasst und gepflegt. Manche Modelle sind nicht mehr aktuell, auch Anforderungen, die sich ändern, werden nicht bis in alle Konsequenz nachgeführt.

3. Nach und nach verlassen ursprüngliche Mitarbeiter und Stakeholder das Projekt. Sie nehmen ihr Wissen mit und lassen nur den Teil im Projekt, der sich inzwischen in Code oder expliziter Dokumentation niedergeschlagen hat. In der Praxis geht dabei viel ergänzendes Wissen verloren.

4. Unter den Effekten (2, 3) leidet die Software zunächst scheinbar nicht, da das Programm ja an und für sich „funktioniert“. Kommt es jedoch zu Änderungsbedarf (auf Grund von Fehlern, Weiterentwicklung oder Anpassung an die geänderte Umwelt), so fehlen immer mehr Informationen.

5. Daher wird oft versucht, nur kleine, lokale Änderungen vorzunehmen. Dies führt allerdings erst recht zu einer schwer durchschaubaren Struktur mit vielen Ausnahmen und möglicherweise Inkonsistenzen: die „Patches“ passen nicht mehr zueinander und zu der grundlegenden Systemstruktur. Die Struktur verwildert zusehends.

6. Nur im Idealfall werden die Änderungen ebenfalls dokumentiert und das so gesicherte Wissen mit dem zuvor gesammelten, ergänzenden Wissen integriert. In der späten Lebensphase eines (Legacy-) Systems, in der man kein grundlegendes Refactoring und keine Umstrukturierung mehr wagt, wird kaum mehr Energie in Wissenssammlung gesteckt werden. Ohnehin ist das System nun nur noch schwer zu verstehen, was die Wissensintegration weiter erschwert und zunehmend sinnlos erscheinen lässt.

7. Ein Teufelskreis ist in Gang gekommen, der letztlich zu Lücken und Inkonsistenzen im gesamten Wissen (bestehend aus Code, Modellen und ergänzendem Wissen) führt. Der Code hat sich durch die lokalen Patches von den ursprünglichen Plänen wegentwickelt. Die Pläne und Entwürfe selbst können mangels Überblick nicht mehr konsistent gepflegt werden.

Für die Entwicklung langlebiger Softwaresysteme müsste man an mehreren Stellen in den skizzierten Lebenszyklus eingreifen:

In Schritt (1) werden Anforderungen, Rationale und weiteres Wissen zusammen getragen. Wenn man sie in Schritt (2) wirkungsvoll erfassen könnte, dann könnte die nachteilige Wirkung der folgenden Schritte gemildert werden. In unseren Arbeiten nimmt Schritt (2) hier eine Schlüsselstellung ein: Die Phase, in der zwar die Entwicklungstätigkeit alle Energie an sich zieht, in der aber andererseits auch die Entscheidungen und das Rationale noch präsent sind, betrachten wir als geeigneten Einstiegspunkt für die Rettung von ergänzendem Wissen: Im „By-Product Approach“ [Sc06] ist beschrieben, wie man durch gezielte Computerunterstützung in dieser Situation Wissen einsammelt, das sonst flüchtig und „flüssig“ (in der Terminologie der Informationsflüsse [SSK08]) bliebe und rasch versickern würde. Es stünde dann in der Wartung nicht mehr zur Verfügung.

Wenn es zusätzlich gelingt, das gesammelte Wissen bei Änderungen nutzbar zu machen (Schritt 5), so steigt die Chance, die kombinierte Struktur länger zu achten. Wiederum sollte man aufwandsschonende Computerunterstützung nutzen, um auch bei Änderungen die Begründungen und Entwurfsentscheidungen einzusammeln. Dabei muss man wieder ganz konkret bedenken: auch bei Änderungen muss jeder Aufwand durch einen persönlichen Nutzen kompensiert werden.

Es ist kaum zu erwarten, dass man den Alterungsprozess ganz zum Stillstand bringen kann. Aber eine merkliche Verlangsamung wäre bereits ein wichtiger Erfolg.

4 In Zukunft: Programme, Modelle und Wissen verbinden

Es gibt mehrere Beispiele, wie das „By-Product“-Konzept umgesetzt werden kann. Als Beispiel wird hier das Werkzeug FOCUS verwendet, das die Konzepte besonders deutlich zeigt: Mit FOCUS kann man während Software-Vorführungen (Demos) Wissen extrahieren, das zuvor in dem Programm und seinem Entwickler steckte. Schon 1996 wurde eine Smalltalk-Version von FOCUS auf der ICSE vorgestellt [Sc96].

Die Ideen hinter FOCUS sind einfach, aber nicht ganz einfach umzusetzen:

- Während ein kenntnisreicher Entwickler seinen Prototypen vorführt, wird diese Demonstration aus mehreren Perspektiven aufgezeichnet: die Bildschirmanzeige und ein Trace (aufgezeichnete Sequenz) ausgeführter Methoden ergeben zwei, durch FOCUS miteinander verbundene Resultate.

- FOCUS kann nun mit Hilfe der Traces die technische Erklärung des Prototypen anleiten: Im Browser werden die Methoden der Reihe nach angezeigt, die während der Demo ausgeführt wurden. So lassen sich – von der Oberfläche bis zu Algorithmen, Architekturentscheidungen und sogar fragwürdigen Kompromissen – viele relevante Punkte erläutern, indem man den Traces folgt.

- Es steht dem Entwickler frei, kommentarlos weiterzuschalten – oder die Wirkungsweise einer Methode zu erklären. Wieder ergibt sich ein Pfad aus den Methoden, die erklärt wurden. Er folgt teilweise den Traces, weicht dann aber wieder ab.

- Das Verfahren kann mit mehreren Demonstrationen und mehreren Erklärungssitzungen angewendet werden und führt so – als Nebenprodukt – dazu, dass FOCUS jedes Mal einen „Ausführungspfad“ und einen „Erklärungspfad“ aufzeichnet und zu den bestehenden hinzufügt. Diese Pfade können sich überscheiden und parallel laufen, oder sie können divergieren.

- In späteren Sitzungen können Entwickler auch Erklärungspfaden folgen, die zuvor von anderen angelegt wurden. Da die Aufnahme der Pfade vollständig automatisiert durch FOCUS erfolgt, bieten die Pfade und alle darauf aufbauenden Mechanismen die Gelegenheit, weiteres Wissen einzusammeln. Hat der Entwickler dann das Projekt verlassen (wie in Schritt 6), so ist doch ein Netz aus Pfaden und Bildschirmvideos entstanden und im Projekt verblieben, das eng mit den Methoden (also dem Programmcode) verbunden ist.

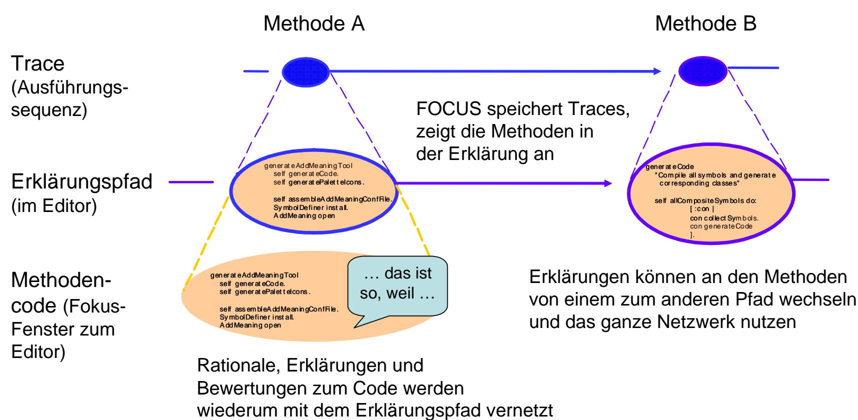

Abbildung 3: Traces und Erklärungspfade werden in FOCUS mit Code vernetzt

Selbstverständlich sind viele andere Beispiele vorstellbar, die durch Aufzeichnung, Beobachtung und Integration durch Indizierung nach dem gleichen Prinzip Wissen erfassen und strukturieren [Sc06]. In der Regel setzt die Anwendung des Prinzips jedoch eine spezifische Werkzeugunterstützung voraus, die tief in die Details der wissensrelevanten Arbeitsschritte integriert sein muss. Bei FOCUS müssen Traces erfasst, Browser damit gesteuert und gleichzeitig Erklärungen aufgezeichnet werden. Ein Trace ist die Folge ausgeführter Methoden von ausgewählten Klassen. Ferner müssen die Bildschirmaufzeichnung indiziert werden, damit man gezielt einzelne Sequenzen anspringen kann; beispielsweise die Erklärung einer komplizierten Methode.

Bei ganz anderen wissensrelevanten Aufgaben prägt sich das Prinzip auch anders aus: In der Projektplanung oder im Risikomanagement würde man anders vorgehen. Auch FOCUS wurde mehrere Jahre später noch einmal für Java als Eclipse plug-in realisiert [Sc06]. Dabei konnte FOCUS in Smalltalk auf sich selbst angewendet werden, um zumindest einige Grundkonzepte über Jahre hinweg nach Java zu retten.

5 Programme, Modelle und Wissen separat nutzbar machen

Soeben wurde das Netzwerk aus Pfaden geschildert, die in FOCUS mit dem Code über die Methoden feinmaschig verbunden sind. Ab Schritt (4) im Lebenszyklus beginnt jedoch eine neue Entwicklung, der man mit anderen Mitteln begegnen muss:

Teile des Codes müssen ersetzt oder portiert werden, oder sie funktionieren wegen Fehlern und Änderungen in der Umwelt nicht mehr wie zuvor. Dann kann es sinnvoll sein, das Verhältnis zwischen Code und ergänzendem Wissen (in FOCUS: Pfaden und Aufzeichnungen) umzudrehen. Das (ursprünglich: ergänzende) Wissen wird zur primären Wissensrepräsentation, die Bildschirmaufzeichnungen und Quellcodeauszüge werden zu Beispielen und Illustrationen für das Rationale.

In der ersten Version von FOCUS in Smalltalk [Sc96] erhielten die Pfadelemente durch Vererbung die Fähigkeit, das gesamte Netzwerk in der oben geschilderten Weise in html auszugeben und mit dem Group Memory der Abteilung zu verbinden. Bildschirmaufzeichnungen und der Quellcode einzelner Methoden wurden darin zur Veranschaulichung eingebettet. Die Pfade wurden durch Links abgebildet. Diese Darstellung ist gut lesbar, aber nicht ausführbar und stellt eindeutig einen Bruch im Sinne von Lebenszyklus Schritt (6) dar. Daher sollte so lange wie möglich mit der integrierten Variante gearbeitet werden. Andererseits ist die so entstehende Wissensdarstellung nicht mehr auf Smalltalk-Umgebungen angewiesen; ein Browser reicht.

Die Integration von Rationale und Wissen in den Code bedeutet übrigens nicht, dass in einem Softwareprodukt auch alle eingebetteten Wissenselemente an den Kunden ausgeliefert werden müssten oder sollten. Lediglich solche Mechanismen, die für die Erfassung von Feedback von den Bedienern gedacht sind, müssen dort verbleiben – was zu vielen weiteren interessanten Fragestellungen führt [LS06, LK07]. Es muss jedoch eine Referenzinstallation bei der Entwicklung geben, die das ergänzende Wissen enthält. Bei Bedarf wird eine neue Produktversion extrahiert und ausgeliefert.

6 Herausforderungen und Forschungsrichtungen

Der oben vorgestellte Ansatz, ergänzendes Wissen durch Computerunterstützung mit Programmartefakten zu integrieren, steht noch am Anfang seiner Entwicklung. FOCUS ist ein relativ codenahes und leicht erklärbares Beispiel für seine Umsetzung; viele andere Arten von Wissen können nach ähnlichen Prinzipien bewahrt werden.

Dabei ist die Idee, Wissenmanagement-Methoden auf Software Engineering anzuwenden, nicht neu: Es gibt sogar Tagungsreihen zu dem Thema (wie SEKE: Software Engineering und Knowledge Engineering, KBSE: Knowledge-Based Software Engineering) mit internationalem Journal dazu. Auch der Workshop LSO (Learning Software Organizations) bearbeitet seit Jahren verwandte Themen. Der Workshop ist manchmal einer Software-Engineering-Tagung, dann wieder einer Wissensmanagement-Veranstaltung angegliedert. Die Relevanz des Themas ist grundsätzlich schon lange bekannt. Allerdings ist der hier besprochene Aspekt der „nebenbei wirksamen", auf die Software-Engineering-Aktivitäten fein abgestimmten Strukturen und Werkzeuge noch nicht im Zentrum der Aufmerksamkeit. Ohne Werkzeuge – insbesondere ohne *akzeptierte* Werkzeuge – können Wissenserhebung und -nutzung aber nicht funktionieren.

Eine ganze Reihe von Forschungsfragen lässt sich aus diesem Ansatz ableiten. Sie bilden einen Kern dieses Beitrags, weil sie Möglichkeiten aufzeigen, den Problemen langlebiger Softwaresysteme einen Schritt näher zu kommen:

- *Welche Informationen und Wissensarten braucht man?* Zunächst müssen solche Wissensarten identifiziert werden, die in der Wartung (Lebenszyklus Schritt 4-6) gebraucht würden, aber oft fehlen. Diese können dann gezielt gesammelt werden. Die Beispiele in Abschnitt 2 werden dazu gehören.

- *Wie kann man ergänzendes Wissen nebenher erheben?* Der hier vorgestellte Ansatz sieht vor, durch spezifische Computerunterstützung aufzunehmen und zu indizieren. Dabei ist genau auf das Verhältnis von Aufwand und Nutzen zu achten.

- *Wie integriert man die Wissenserhebung und -verwendung in den Arbeits- und Lernprozess einer lernenden Software-Organisation [Bi06]?* Nach Kelloway und Baring [KB03] kommt Lernen am Arbeitsplatz nur in Gang, wenn dafür die (1) Fähigkeit des Einzelnen, die (2) Gelegenheit durch Aufgabe und Umfeld und (3) Motivation zusammenkommen. Wie das in der Softwareentwicklung zu gewährleisten ist, geht über das reine Software Engineering hinaus. Aber auch für das Software Engineering ergeben sich konkrete Konsequenzen, wenn wirksame Unterstützung angeboten werden soll.

- *Wie bringt man die Experten dazu, ihr Wissen für andere preiszugeben?* Diese Frage ist ein Spezialfall der vorigen und hat ihrerseits viele Facetten, von der Gestaltung von Incentives [DP00] bis hin zur Bedienbarkeit der eingesetzten Werkzeuge, einem Aspekt der Usability im Software Engineering [La05].

- *Was ist der primäre Wissensträger?* Bisher wird außer dem Code kaum etwas aufgehoben; Code ist dann der primäre Wissensträger, zu dem ergänzendes Wissen hinzukommen kann (in Ontologien, Büchern, FOCUS-Pfaden). Wie sieht das aber aus, wenn Code aus Modellen generiert wird?

- *Wie verhalten sich die Modelle des Softwaresystems und Modelle des ergänzenden Wissens zueinander?* Sind die Metamodelle integriert oder stehen sie nebeneinander (z.B. UML und Wissensschema)? Wie kann man sie trennen, wenn dies erforderlich wird?

- *Welche Mechanismen können bei der Wiederverwendung des Wissens eingesetzt werden?* Auch hier ist an eine aufwandsoptimierte Lösung aus Sicht der Beteiligten zu denken. Die Computerunterstützung beim Erfassen kann auch das Wiederverwenden erleichtern, wie das Beispiel der html-Ausgabe von FOCUS-Pfaden zeigt. Keinesfalls reicht es aus, Informationen „auf dem Internet verfügbar“ zu machen. Sie müssen aktiv viel näher an die Arbeitsaufgaben der Entwickler und der Wartungsingenieure gerückt werden.

Viele weitere Fragen aus dem Bereich des Wissensmanagement und der Psychologie können interdisziplinär angegangen werden. Im vorliegenden Beitrag sieht man, dass schon aus der reinen Software-Engineering-Perspektive viele interessante Fragen auftauchen [Sc09]. Für manche Aspekte haben wir bereits Vorarbeiten geleistet. Aber die meisten Fragen sind noch offen.

Manche der Vorarbeiten adressieren einzelne Fragen, während andere versuchen, eine durchgängige Lösung für ein spezielles Anwendungsproblem anzubieten [Vo06, Vr06]. Es entstehen auch praktisch einsetzbare Werkzeuge, die sich an die Modelle oder Code anlehnen, um Wissen zu erheben und zu erhalten. Dahinter sind semantisch weitergehende Aufbereitungen und Auswertungen möglich. Zunächst ist es aber am wichtigsten, an viel versprechenden Stellen in Entwicklung und Betrieb Feedback einzusammeln und Erfahrungen zu erheben – damit sie für langlebigere Konzepte und Software genutzt werden können.

7 Fazit

Dieser Beitrag hebt die Bedeutung von „ergänzendem Wissen“, insbesondere von Software Engineering Rationale, für langlebige Software hervor. Für das Software Engineering ist es wichtig, dieses Wissen wirksam zu sammeln und mit den Programmen oder Modellen so zu verbinden, dass es bei der Wartung an Ort und Stelle ist und nicht mühsam gesucht werden muss. Das verlangt nach neuen Konzepten, spezialisierten Techniken und Werkzeugen. Ihre Entwicklung ist schwieriger, als es zunächst erscheinen mag: Eine Reihe dabei relevanter Forschungsfragen wurde genannt. Es reicht nicht aus, sich einmal mit einer davon zu beschäftigen. Um Software langlebiger zu machen, muss man das ergänzende Wissen retten – und seine Beziehung zu den Programmen und Modellen ohne ständigen Zusatzaufwand aktuell halten.

Literaturverzeichnis

[Be00] Beck, K.: Extreme Programming Explained: Addison-Wesley (2000).

[Bi06] Birk, A., et al. (eds): Learning Software Organisation and Requirements Engineering: The First International Workshop. J.UKM Electronic Journal of Universal Knowledge Management (2006).

[Bo81] Boehm, B.: Software Engineering Economics, N.J.: Prentice Hall, Engelwood Cliffs (1981).

[Co02] Cockburn, A.: Agile Software Development: Addison Wesley (2002).

[DMMP06] Dutoit, A.H.; McCall, R.; Mistrik, I.; Paech, B. (eds.): Rationale Management in Software Engineering. Springer: Berlin (2006).

[DP00] Davenport, T., Probst, G..: Knowledge Management Case Book - Best Practises, München, Germany: Publicis MCD, John Wiley & Sons (2000).

[GHJV95] Gamma, E., Helm, R., Johnson, R., Vlissides, J..: Design Patterns - Elements of Reusable Object-Oriented Software: Addison-Wesley Publishing Company (1995).

[KB03] Kelloway E K, Barling, J.: Knowledge work as organizational behavior, International Journal of Management Reviews, 2000. 2(3). (2000) 287-304.

[La05] Lauesen, S.: User Interface Design - A Software Engineering Perspective, Harlow, London: Addison Wesley (2005).

[LK07] Lübke, D. and Knauss, E.: Dealing with User Requirements and Feedback in SOA Projects, in Workshop on Software Engineering Methods in Service Oriented Architecture. Hannover, Germany (2007).

[LS06] Lübke, D. and Schneider, K.: Leveraging Feedback on Processes in SOA Projects, in EuroSPI. Joensuu: Springer-Verlag Berlin-Heidelberg (2006).

[PM06] Petrasch, R. and Meimberg, O.: Model-Driven Architecture. Eine praxisorientierte Einführung in die MDA, Heidelberg: Dpunkt Verlag (2006).

[Sc06] Schneider, K.: Rationale as a By-Product, in [DMMp06] 91-109.

[Sc09] Schneider, K.: Experience and Knowledge Management in Software Engineering. Springer, Berlin (2009).

[Sc96] Schneider, K.: Prototypes as Assets, not Toys. Why and How to Extract Knowledge from Prototypes, in 18th International Conference on Software Engineering (ICSE-18). Berlin, Germany (1996).

[SSK08] Schneider, K., Stapel, K., and Knauss, E.: Beyond Documents: Visualizing Informal Communication, in Third International Workshop on Requirements Engineering Visualization (REV 08). Barcelona, Spain: IEEE Xplore (2008).

[St73] Stachoviak, H.: Allgemeine Modelltheorie, Wien, New York: Springer Verlag (1973).

[SV05] Stahl, T. and Völter, M.: Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management, Heidelberg: Dpunkt Verlag (2005).

[Vo06] Volhard, C.: Unterstützung von Use Cases und Oberflächenprototypen in Interviews zur Prozessmodellierung, Fachgebiet Software Engineering, Gottfried Wilhelm Leibniz Universität Hannover (2006).

[Vr06] Vries, L.d.: Konzept und Realisierung eines Werkzeuges zur Unterstützung von Interviews in der Prozessmodellierung, Fachgebiet Software Engineering, Gottfried Wilhelm Leibniz Universität Hannover (2006).