M. Tichy, E. Bodden, M. Kuhrmann, S. Wagner, J.-P. Steghöfer (Hrsg.): SE 2018,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2018 217

Erklärungen (nur) bei Bedarf

Kurt Schneider[1]

1 Was muss erklärt werden?

Software ist kompliziert. Es ist schwierig, zu verstehen, wie sie funktioniert und wie sie bestimmte Resultate erzielt oder wie sie zu einer automatisierten Entscheidung gelangt. In diesem Beitrag wird die These vertreten, dass es dennoch nicht sinnvoll ist, möglichst viele Informationen für mögliche spätere Erklärungen zu sammeln oder zu generieren, denn ein großer Teil davon wird nie benötigt. Erklärbarkeit mag ein wichtiger Softwarequalitätsaspekt sein; sie ist aber dennoch eine „Sekundärtugend". In sie sollte man nur Aufwand und Zeit investieren, wenn der absehbare Nutzen dies auch rechtfertigt. Es ist wichtig, zunächst den angestrebten Nutzen genau zu verstehen.

Die erste Frage bei der Gestaltung verbesserter Erklärbarkeit lautet daher: Auf welche Arten von Erklärungen soll die Software vorbereitet sein? Denn es gibt verschiedene Typen von Fragen, zu denen man eine Erklärung sucht: (a) Wie wurde das Resultat aus einer Regel (einem Algorithmus, einem Neuronalen Netzwerk usw.) abgeleitet? (b) Wie lautet die dort umgesetzte Regel anschaulich und verständlich? (c) Wieso wurde gerade diese Regel gewählt? Wer eine Antwort auf (b) oder (c) sucht, dem ist mit einem detaillierten Trace eines komplexen Algorithmus (a) nicht gedient.

2 Analogie: Wissen aus Erfahrungen und Wissen für Erklärungen

Vor einigen Jahren hat Basili die Experience Factory vorgeschlagen. Unternehmen versuchten, systematisch aus ihren eigenen Erfahrungen zu lernen. Erfahrungsnutzung und zukünftige Nutzung von Wissen in Erklärungen haben einiges gemeinsam: In Basilis Experience Factory sammelte eine Gruppe in Unternehmen Messdaten und quantitative Erfahrungen, prüfte die Anwendbarkeit auf neue Projekten und lieferte übertragbare Erfahrungen. Zunächst versuchte man, mit Werkzeugen das Wissen aus Erfahrungen effizient zu verwalten und zugänglich zu machen. Das war aber ein Irrweg.

Die eigentliche Herausforderung lag nämlich darin, solche Erfahrungen zu identifizieren und zu erfassen, *die später auch wirklich gebraucht wurden*. Es gibt einen Lebenszyklus für Erfahrungen, an dem bemerkenswert ist, dass Erfahrungsträger erst einmal erkennen müssen, welche ihrer Erfahrungen für andere wichtig sein könnten. Übertragen auf die erklärbare Software sind wohl Mechanismen nötig, um komplizierte Abläufe zu verfolgen (tracen); es könnte aber viel wichtiger sein, diese detaillierten Abläufe zu verdichten

[1] Leibniz Universität Hannover, FG Software Engineering, Welfengarten 1, 30167 Hannover
kurt.schneider@inf.uni-hannover.de

cc by sa

und nach ihrem Sinn zu hinterfragen. Das erfordert vielleicht nicht so detailliertes Tracing, dafür aber komplementäre Begründungen (Rationale) und Anforderungen. Sie sind für Fragen der obigen Typen (b) und (c) unverzichtbar. Beobachtungen, Traces und Wissen können zudem – wie Erfahrungen - nicht „roh“ genutzt werden, sondern müssen zunächst geeignet aufbereitet werden, bevor sie anderswo und von anderen Personen genutzt werden können. Man darf sich nicht darauf beschränken, das zu sammeln, was l zugänglich ist, wie beispielsweise Traces – sondern das, was tatsächlich gebraucht wird. Das hängt auch von der Aufgabe dessen ab, der die Erklärung sucht.

3 Vision: Pragmatischer Ansatz - mit Grenzen

Ein *idealistischer Ansatz* könnte darin bestehen, alle Veränderungen am Code zu verfolgen und seinen Ablauf durch Instrumentierung zu tracen Ich plädiere jedoch stattdessen für einen *pragmatischen Ansatz*, in dem Aufwand gezielt reduziert und erst dann aufgewendet wird, wenn der Bedarf entsteht. Beispielsweise würde man also generierbare Erklärungen erst dann erzeugen, wenn sie abgerufen werden. Hier kann man von der systematischen Erfahrungsnutzung lernen und unorthodoxe vorgehen: Zunächst gibt es keine Erklärung; wer sich zuerst fragt, wieso hier ein bestimmter Algorithmus eingesetzt wird, muss seine Erkenntnisse nachher dokumentieren. Denn in diesem Fall war die Information ja schon einmal nützlich, und es lohnt sich, sie in diesem Kontext für künftige Fälle bereitzuhalten. Wird sie dagegen nie gebraucht, muss man auch keine Erklärung vorhalten. Es wäre unverhältnismäßig aufwändig, jede denkbare Erklärung schon bei der Entwicklung abzufragen oder abzuleiten, solange man noch gar nicht weiß, welcher Typ von Erklärung (z.B. a, b, c), für wen und für welchen Zweck gebraucht werden wird.

Dabei zeigen sich jedoch die Grenzen auch des pragmatischen Ansatzes: Oft stecken die gewünschten Informationen gar nicht in der Software und ihrem Verhalten selbst. Es handelt sich vielmehr um eine Kombination aus Anforderung, Entwicklungsprozess und Entscheidungen, die nur zusammen mit dem Algorithmus verständlich sind. Algorithmen werden im Verlauf von Änderungen in Anforderungen und Umfeld schrittweise angepasst, Fragen nach Sinn und Begründung antworten typischerweise auf Fragen der Typen (b) oder (c). Eines der größten Probleme für die Erklärbarkeit dürfte darin liegen, genau an dieses Hintergrundwissen heranzukommen, ohne dabei in eine übertriebene und teure Sammelwut abzugleiten. Man muss Hintergrundwissen „retten“, bevor die Wissensträger es vergessen oder nicht mehr im Unternehmen arbeiten. Man muss neue Mechanismen entwickeln, um das Wissen aus dem System von Mensch und Software zu retten. Es lässt sich sicher nicht nur aus der Software allein ableiten.

Ein Rezept für eine perfekte Balance gibt es noch nicht. In diesem Vortrag werden exemplarisch einige praktische Vorschläge gemacht, wie man mit Hintergrundwissen umgehen und es (erst) zum geeigneten Zeitpunkt erheben kann. Dazu gehört die effiziente Wissenserschließung aus Prototypen oder die Verwendung von Visionsvideos zur anschaulichen Aufbereitung von Sinn und Zweck von Anforderungen.