20 Jahre WSR(E) – ein persönlicher Rück- und Ausblick

Rainer Koschke
Universität Bremen
koschke@uni-bremen.de

1 Ein wenig Geschichtliches

Zwanzig Jahre soll es bereits her sein, dass ich beim allerersten WSR (damals noch das offizielle Akronym) mit Kollegen der Universität Stuttgart meinen ersten Beitrag [1] zum WSR eingereicht habe? Kaum zu glauben. Jürgen Ebert und Franz Lehner haben den WSR – tatkräfigt unterstützt von Andreas Winter, Volker Riediger und Bernt Kullbach – mit dem Ziel ins Leben gerufen, einen Treffpunkt zum wissenschaftlichen Austausch von Praktikern und Wissenschaftlern im Bereich Reengineering im deutschsprachigen Raum zu etablieren. Das ist ihnen vortrefflich und dauerhaft gelungen. Workshops kommen und gehen. Der WSR in Bad Honnef bleibt. Der WSR verstand sich immer als offener, inklusiver Ort des wissenschaftlichen Diskurses. Diesen Charakter hat sich der WSR auch nach seiner Migration zum WSRE erhalten. Kein Wunder. Wenn Spezialisten zur Wartung und Evolution von Software nicht in der Lage wären, einen Workshop am Leben zu erhalten – wer dann?

Die vorherrschenden internationalen Konferenzen im Bereich Wartung und Evolution waren damals und sind noch heute: WCRE, CSMR (die beide zu SANER wurden) und ICSM(E), die sich vor dem WSR(E) ein ergänzendes E gegönnt hat. Die Themen dieser wichtigen Konferenzen aus dem Jahre 1999, also vor 20 Jahren zur Geburt des WSR, waren Reverse Engineering von Anforderungen und Architektur, Tranformation und Migration (unter anderem auch von COBOL), Software-Visualisierung, das Jahr-2000-Problem, die Priorisierung und Generierung von Tests, der Test paralleler und komponentenbasierter Programme, Tracability, Erkennung von Entwurfsmustern und Cliche-Recognition (auch Planerkennung genannt), Wiederverwendung, Prozesse in Wartung und Evolution, Klonerkennung, Change-Impact-Analyse, Software-Metriken, Reverse Engineering von binären Programmen, Data Reverse Engineering und Programmanalyse. Das kommt Ihnen bekannt vor, selbst wenn Sie erst seit kurzer Zeit in dieser Wissenschaftsdomäne unterwegs sind? Alle diese Themen sind auch heute noch Überschriften in den Tagungsbänden dieser Konferenzen – abgesehen einmal vom Jahr-2000-Problem, das sich ja nur alle zwei Millennien zeigt. Auch COBOL-Programme werden dieser Tage weiter gepflegt, migriert und sogar neu verfasst.

Dass die Überschriften gleich bleiben, ist nicht weiter verwunderlich. Schließlich sind das die Kernthemen von Wartung und Evolution. Die Erkenntnisse und Schwerpunkte haben sich jedoch geändert. Das kann man sehr schön am Beispiel der Software-Klone – das heißt: duplizierten Codes – darstellen. War vor 20 Jahren noch die automatisierte Erkennung im Vordergrund (immer im Glauben, Klone seien schlecht und sollten deshalb gefunden und beseitigt werden), hat sich im Laufe der Zeit der Fokus auf empirische Untersuchungen zu den tatsächlichen Auswirkungen von Klonen auf die Wartung, ihre Ursachen und die Evolution von Klonen in langlebigen Systemen verlagert. Es geht nicht mehr allein um Algorithmen, auch der Mensch rückt immer weiter in den Blickpunkt. Heute ist die Einschätzung von Klonen sehr differenziert und ihre Erforschung trotzdem nicht abgeschlossen. Wir können auch heute noch nicht zuverlässig sagen, wie welche Art von Klon behandelt werden sollte.

2 Das Hier und Heute

Ist Wartung und Evolution von Software nach 20 Jahren nicht schon längst gelöst? Wenn man einmal die Anzahl der Einreichungen zu diesen Konferenzen als Maß nimmt, dann ist festzuhalten, dass das Thema immer noch brandaktuell sein muss. Deren Anzahl hat in den letzten zwei Dekaden enorm zugenommen. Dass nicht noch weit mehr Veröffentlichungen existieren, hat damit tun, dass die Annahmequoten der Konferenzen stark gesunken sind (was nicht notwendigerweise gut für die Idee einer Konferenz sein muss).

Inhaltlich beschäftigen wir uns noch immer mit den Gebieten von früher, wenn sich auch die Schwerpunkte, Details und Möglichkeiten verändert haben – wie oben am Beispiel der Software-Klone illustriert. Die statische Programmanalyse hat zum Beispiel eine hohe Reife erlangt, die sowohl neuen Algorithmen als auch ganz besonders der viel potenteren Hardware, die uns heute zur Verfügung steht, geschuldet ist. Auch in der Praxis ist sie angekommen. Viele kommerzelle und freie Werkzeugen implementieren unterschiedlichste Arten statischer Programmanalyse, von einfachen Metrikenwerkzeugen bis hin zu sehr elaborierten Tools für die Erkennung von Programmierfehlern.

Dank des Siegeszugs der Open-Source-Bewegung und die Verfügbarkeit öffentlicher Software-Versionskontrollsysteme sind wir als Forscher heute auch in

der Lage, die Evolution großer Programme nach allen Regeln der Kunst zu studieren. Das Software-Repository-Mining ist mit einer eigenen Konferenz gleichen Namens ein etabliertes Wissenschaftsgebiet. Da auch viele Unternehmen ihre Software open-source entwickeln, können wir auch Aussagen über industrielle Programme machen. Jedoch ist unklar, ob Erkenntnisse, die für Open-Source gewonnen wurden, auf Closed-Source übertragen werden können. Hier wäre eine größere Bereitschaft der Industrie, die closed-source entwickelt, uns ihre Daten (selbstverständlich anonymisiert) zur Forschungszwecken zur Verfügung zu stellen, sehr zu begrüßen. Am Ende des Tages fließen die auf diese Weise gewonnenen Erkenntnisse schließlich wieder zurück an die Industrie. Als Teilgebiet des Software-Engineerings sind wir ganz klar dem Engineering verpflichtet.

Besonders zu begrüßen ist der deutlich erkennbare Trend zu mehr empirischen Studien und aussagekräftigen Evaluationen in neuerer Zeit. Heute hat ein Artikel, der nicht auch überzeugende empirische Belege für seine Behauptungen liefert, kaum eine Chance auf Veröffentlichung. Der Mensch selbst rückt hierbei auch immer mehr in den Fokus der Untersuchung. Teilweise mag die Frage gestellt werden, ob bestimmte heute veröffentlichte Untersuchungen nicht besser von Sozialwissenschaftlern oder Kognitionspsychologen übernommen werden sollten. Nichtsdestotrotz wissen wir immer noch zu wenig darüber, welche Strategien Entwickler in der Wartung und Evolution anwenden, welche Informationsbedürfnisse sie wirklich haben und wie wir diese gewinnen und so aufbereiten können, dass sie auch wirklich zur richtigen Zeit verfügbar und nutzbar sind. Noch immer rezitieren wir die Wartungs-Mantra von Fjeldstad und Hamlen aus dem Jahre 1979 [3], dass Programmverstehen mindestens die Hälfte der Arbeitszeit der Entwickler ausmache, mangels neuerer Untersuchungen. Was machen die nur die ganze Zeit und warum dauert das so lange? Und könnte das bitte jemand einmal nachmessen? Diese Zahlen wurden von Fjeldstad und Hamlen nämlich allein durch Befragungen gewonnen.

Bedauerlicherweise ist auch festzustellen, dass viele der Techniken und Werkzeuge, an denen die Forschung entwickelt, keine Beachtung in der Praxis finden. Dazu gehört zum Beispiel auch die Software-Visualisierung. Ihr Versprechen, durch höhere Abstraktion für besseres Programmverstehen zu sorgen, ist noch nicht eingelöst. Die Nichtverbreitung unserer Ideen kann teilweise daran liegen, dass wir (akademische Forscher) noch nicht die richtigen Wege zur Kooperation mit Entwicklern aus der Praxis gefunden haben – und vielleicht auch daran, dass die Entwickler zu sehr mit ihrem Alltagsgeschäft beschäftigt sind, um die Zeit für den Austausch mit Forschern zu finden. Insofern ist es dem WSRE hoch anzurechnen, dass es ihm gelungen ist, immer auch Praktiker anzusprechen.

3 Und was machen wir morgen?

Vieles könnte besser werden, wenn wir uns einfach nur auf die Tugenden des Software-Engineerings besännen, die wir gerne in Vorlesungen unseren Studierende predigen, selbst aber in unserer Forschung bisweilen vernachlässigen.

Zunächst einmal sollten wir – wie im Software-Engineering verlangt – unsere Anforderungsanalyse richtig machen. Das heißt, wir brauchen ein besseres Verständnis der tatsächlichen Bedarfe der Entwickler für Wartungsaufgaben. Weitere empirische Grundlagenforschung ist hier von Nöten, auch gerne in Kooperation mit Psychologen, Kognitionswissenschaftlern und Soziologen. Diese Disziplinen bieten Expertise im methodischen Vorgehen und viele ihrer Erkenntnisse sollten auf unsere Domäne übertragbar sein.

Dann sollten wir viel stärker dazu übergehen, partizipatorisch zu entwickeln. Soll heißen, statt vorab viel Zeit in die Entwicklung einer neuen Technik oder eines neuen Werkzeugs zu investieren, um hernach festzustellen, dass Entwickler sich damit nicht zurecht finden, sollten wir unsere anvisierten Nutzer schon ganz früh im Design-Prozess mit einbeziehen. Iteratives Vorgehen und inkrementelle Entwicklung ist auch in der Entwicklung von Methoden und Werkzeuge für die Wartung anwendbar.

Es mangelt nicht selten auch am langen Atem, was möglicherweise der meist zeitlich sehr begrenzten Finanzierung von Forschungsprojekten geschuldet ist. Kann denn zum Beispiel die volle Wirksamkeit einer Software-Visualisierung erkannt werden, wenn man sie im Rahmen eines zeitlich eng gefassten Experiments eine halbe Stunde vorstellt und die Entwickler damit auf ihnen fremden Code loslässt? Eine Visualisierung muss gelesen werden können. Und sie muss einen Mehrwert liefern, das heißt, darf nicht nur mitteilen, was die Entwickler ohnehin schon über ihren eigenen Code wissen. Dringend notwendig sind aus meiner Sicht mehr Langzeitstudien, bei denen die Teilnehmer mit Einfluss auf Methoden und Werkzeuge nehmen, die die Forscher entwickeln – zum Beispiel nach dem Modell der Aktionsforschung [2].

Literatur

[1] J. Czeranski, T. Eisenbarth, H. Kienle, R. Koschke, and D. Simon. Wiedergewinnung von architekturinformationen: Ausblicke. In *Workshop on Software Reengineering*, number Nr. 8/2000 in Fachberichte Informatik, Universität Koblenz-Landau, May 1999.

[2] P. S. M. d. Santos and G. H. Travassos. Action research use in software engineering: An initial survey. In *International Symposium on Empirical Software Engineering and Measurement*, pages 414–417, Oct 2009.

[3] R. Fjeldstad and W. Hamlen. Application program maintenance study: Report to our respondents. In *Proceedings of the GUIDE 48*, Philadelphia, PA, 1979. The Guide Corporation.