„Software Aging“ in den DH: Kritik des reinen Forschungswillen

Bürgermeister, Martina
martina.buergermeister@uni-graz.at
ZIM-ACDH, Universität Graz, Österreich

Schneider, Gerlinde
gerlinde.schneider@uni-graz.at
ZIM-ACDH, Universität Graz, Österreich

Makowski, Stephan
stephan.makowski@uni-koeln.de
CCeH, Universität Köln, Deutschland

Jeller, Daniel
daniel.jeller@icar-us.eu
ICARUS, Wien, Österreich

Bigalke, Jan
JBigalke@smail.uni-koeln.de
CCeH, Universität Köln, Deutschland

Theisen, Christian
ctheise1@smail.uni-koeln.de
CCeH, Universität Köln, Deutschland

Vogeler, Georg
georg.vogeler@uni-graz.at
ZIM-ACDH, Universität Graz, Österreich

Einleitung

Dieser Beitrag behandelt die Frage, warum in den DH entwickelte und angewandte Software häufig schnell altert. Jede Software altert relativ zu der Umgebung, in der sie eingesetzt wird, unabhängig von der Qualität am Beginn ihrer Verwendung (Engels et al. 2009: 393). Wandeln sich Hardware, Infrastruktur oder Anforderungen an die Software, wird sie, um weiter brauchbar zu sein, angepasst. Je nach Beschaffenheit können sich diese Anpassungen positiv, oftmals aber auch negativ auf die Lebensdauer und Fitness einer Software auswirken.

Aus der Praxis behaupten wir, dass kontextuelle und inhaltliche Spezifika von DH-Software dazu führen, dass eine langfristige Lauffähigkeit und Brauchbarkeit erschwert werden. Unser Beitrag bringt allgemein die Bedeutung und Relevanz des Themas „Software Evolution“ (2) nahe, beschreibt Spezifika der Software Evolution aus der DH-Praxis (3) und zeigt welche konkreten Maßnahmen im Projekt *monasterium.net* (4) dahingehend gesetzt werden.

Software Evolution

Software Evolution umfasst alle Aktivitäten und Prozesse, die Software verändern (Godfrey/German 2008). Änderungen der Hardware, der Infomationsübermittlung sowie der Anforderungen sind Kräfte die auf diesen Evolutionsprozess wirken. Softwareentwicklungsprozesse werden seit den 1970er Jahren definiert und systematisiert, um die Qualität von Software zu steigern. Aus dieser Zeit stammt auch das Konzept des sogenannten Software Lifecycles und die Idee, diesen Zyklus zu managen (Lehman 1980). Unterschiedliche Methoden und Techniken dazu haben sich seither für alle Phasen im Lebenslauf von Softwaresystemen etabliert. Dank der intensiven Auseinandersetzung mit der Qualitätssteigerung in der Softwareentwicklung wurden die Fehlerquoten gesenkt (Thaller 2000: 6). Hochwertige Software ist nicht nur (nahezu) fehlerfrei, sondern auch kompatibel zur ihrer Umgebung. Verläuft die Evolution einer Software nicht in diesem Sinne, spricht man vom „Software Aging“ beziehungsweise sogar von deren Verfall (Parnas 1994). Demeyer et al. (2013: 4f.) fassen die Symptome veralteter Software wie folgt zusammen: Unvollständige oder keine Dokumentation, fehlende Tests, Ausstieg ursprünglicher Entwickler, verlorengegangenes Insiderwissen, fehlender Überblick über Gesamtsystem, zeitintensive Anpassungen, ständige Fehlerkorrekturen und Wartung damit verbundener Abhängigkeiten, lange „Build“-Zeiten und schlechter Code.

Parnas (1994: 280) erkennt zwei Hauptfaktoren für das Altern von Software. „Lack of movement“, also keine Änderungen an der Software vorzunehmen, und „Ignorant surgery“: Aus der Praxis weiß man, dass bei dringenden Korrekturen am Programmcode, die formale Kriterien für gute Software oftmals nicht eingehalten werden. Ein Beispiel ist das unreflektierte Copy-and-paste aus *Stack Overflow* [1]. Kurzfristige werden den besten Lösungen vorgezogen. Derartige Eingriffe und nicht-systematisches Vorgehen beschleunigen den Prozess der Softwarealterung. Es wird immer aufwendiger, Änderungen an der Software vorzunehmen.

Demzufolge werden Ideen zur systematischen und automatisierten Verjüngung von Software erforscht und erprobt: Refactoring-Tools, beispielsweise für Java in *Eclipse*, *Python Rope*, oder aber auch für HTML und CSS (Mazinanian/Tsantalis 2017, Harold 2008), wurden entwickelt. Sogenannte „Prediction“- Modelle werden ermittelt, um Softwareevolution besser verstehen zu können und vor allem dem Problem der „Legacy software“ zu begegnen (Goltz et al. 2015, Paech et al. 2016).

Software Herausforderungen in der DH Praxis

Diese teilweise schon seit Jahrzehnten bekannten Erkenntnisse aus dem Software Engineering haben für die DH eine besondere Relevanz, da die Projekte hier wesentlich kleinere Budgets, oftmals kurze Projektlaufzeiten und andere Unsicherheiten haben. Aus unserer Erfahrung wird Softwareentwicklung in den DH häufig sehr informell gehandhabt. Diesbezüglich nachhaltiger zu werden, haben unter anderem Czmiel (2017), Schrade (2017) oder Kasper/Grüntgens (2017) gefordert. Nicht nur der Entwicklungsprozess von DH-Software muss längerfristig gedacht werden (Hattrick 2016), auch der Kontext, in dem die Software entsteht und besteht, beeinflusst deren Entwicklung und Veränderung.

Erstens ist es nicht ungewöhnlich, dass Projekte in den DH von einer einzigen Person technisch umgesetzt werden, wie es etwa im Falle von Dissertationsprojekten typisch ist. Der entstandene Code ist bei Projektende lauffähig, es kann aber nicht vorausgesetzt werden, dass dieser auf einen langfristigen Einsatz ausgelegt ist und entsprechend gewissenhaft programmiert und dokumentiert ist. Forschungsergebnisse sind im Projektkontext meist wichtiger als die Qualität der entwickelten Software. Generell bedeutet ein Projektende nicht die Übergabe eines Produktes an einen Kunden, es bedeutet vielmehr: Die Finanzierung läuft aus und der/die Entwickler/in verlässt das Projekt. Was zurückbleibt, ist Software, die von anderen gewartet werden muss. Dazu ist es notwendig, die Dokumentation und Systemarchitektur zu verstehen, sich in den Fremdcode einzuarbeiten. Veränderungen am Code können oft nicht mehr ihrer ursprünglichen Intention entsprechend vorgenommen werden. Die Wartung wird aufwendig und zeitintensiv. Das heißt, die Organisationsstrukturen des Forschungsbetriebes beeinflussen die Alterung von Software.

Zweitens bringen die komplexen Anforderungen der Forschungsdaten nicht-klassische Lösungsansätze mit sich. Mit diesen Ansätzen vertraute Entwickler/innen sind schwer zu finden und zu halten, Einarbeitungsphasen dauern lange. Besonders augenfällig wird das am in den DH weit verbreiteten Gebrauch von X-Technologien. Sie werden immer mehr zur Nischenanwendung. Während die Definitionen von XSLT 1.0 und XPath 1.0 noch von einer größeren Breite von Softwareprodukten implementiert wurden, sogar Teil der Browser wurden, gibt es nur noch wenige Implementationen der Weiterentwicklungen XSLT 2.0 und 3.0. Auch die Menge verwendbarer XML-Datenbanksysteme ist heute geringer als noch vor einigen Jahren. In den DH entwickelte Softwarelösungen sind also speziell auf die Bedürfnisse des Gegenstandes ausgelegt und stellen keine Standardlösungen dar. Sie brauchen spezifisches Know-how, um gewartet werden zu können. Fehlt dieses, beziehungsweise ist es nur mangelhaft vorhanden, droht die Software zum unbrauchbaren Altsystem zu verkommen.

DH-Software verlangt drittens besondere Zuwendung, wenn der Code gleichzeitig die Forschungsergebnisse interpretiert. Wenn die Forschungsleistung also nicht allein in den Daten liegt, braucht es individuelle Wartungslösungen. Eine Digitale Edition kann beispielsweise als die Gesamtheit von Daten, Systemarchitektur, Anwendung und GUI verstanden werden (Andrews/Zundert 2018). Diese Interpretationsleistung als Teil der Forschung muss bei allen Phänomenen der Veränderung an der Edition mitbedacht werden. Die Gefahr ist groß, dass nach einiger Zeit das Argument durch Softwareanpassungen verwässert oder im schlimmsten Fall nicht mehr nachvollziehbar ist und für die Forschung unbrauchbar wird.

Zusammenfassend sehen wir in der nicht langfristigen Finanzierung, der hohen Fluktuation an Personen, der Notwendigkeit von Speziallösungen und im Forschungsgegenstand selbst erhöhten Bedarf an Maßnahmen, um unsere Softwareprojekte lauffähig zu halten.

Anti-Aging Maßnahmen im Projekt monasterium.net

Seit 2008 basiert die Urkundenplattform *monasterium.net* auf *eXist-db* als Applikationsserver und Datenbank. Die Plattform wurde hauptsächlich von drei aufeinanderfolgenden Hauptentwicklern programmiert. Um die Software zu modularisieren, wurde seit 2011 das *mom-ca*-Framework entwickelt, eine Webapplikation in XRX-Architektur (XQuery, REST, XForms). Die Architektur galt damals in Verbindung mit XML-Datenbanken als Empfehlung, wird allerdings in der modernen Webentwicklung kaum mehr eingesetzt. Mit Auslaufen eines Projektes 2014 verließ der letzte Entwickler mit Überblick über das Gesamtsystem das Projekt. Zuvor wurde der Gesamtcode in ein öffentliches Repository überführt. Wissen und Intentionen gingen jedoch verloren. Wir, als das aktuelle, größtenteils projektfinanzierte Entwicklerteam, beschäftigen uns nun aktiv damit, wie der derzeitige Code-Bestand unter unsteten Umständen wartbar und aktuell gehalten werden kann. Im Folgenden beschreiben wir vier Anti-Aging-Maßnahmen, die einerseits Refactoring (das Überarbeiten des Codes), aber auch ganz grundsätzliche Umstellungen des Entwicklungsworkflows betreffen.

Softwareverwaltung durch *Git* und Nutzung der Services von *GitHub*.

Sowohl Entwicklung als auch Dokumentation erfolgen über ein öffentliches *GitHub*-Repository[2]. Die dadurch verfügbaren Möglichkeiten der Versionsverwaltung, des

Bugtracking und des Code Review werden genutzt, um die Qualität des Codes zu verbessern und diesen transparent und nachvollziehbar zu entwickeln.

Einrichtung einer Testumgebung.

Jede Neuentwicklung wird, vor ihrer Übernahme in das Produktivsystem anhand eines festgelegten Testszenarios evaluiert. Durch die Spiegelung des Livesystems auf einem Testserver soll reales Systemverhalten reproduziert werden. Fehler können so vorzeitig entdeckt und behoben werden.

Refactoring von HTML und CSS.

Die Verwendung eines auf den Konzepten von Material Design[3] basierenden CSS-Frameworks garantiert ein konsistentes Gesamtdesign von *monasterium.net*. Teile des Benutzerinterfaces werden dadurch modularisiert und leichter anpassbar. Die Verwendung eines Präprozessors und das Einführen einer Namenskonvention sollen die Wartbarkeit, das Auffinden von Fehlern und die Umsetzung neuer Features erleichtern.

Entwicklung einer RESTful API zwischen Client und Datenbank.

Die zukünftige Kommunikation zwischen Client und Datenbank übernimmt eine neudefinierte REST-API. Die Datenabfrage aus der XML-Datenbank erfolgt noch per XQuery, zurückgeliefert werden wahlweise in XML oder JSON serialisierte Daten. Diese Form des Reengineerings gewährt eine definierte, standardisierte Verarbeitungsweise sowie die Weiternutzung und Kombination multipler Datenquellen. Die Abstraktion von Datenbank, Programmlogik und Benutzeroberfläche erleichtert so in Zukunft deren entkoppelte Anpassung oder Austausch.

Fazit

Softwarealterung ist nicht nur in der Softwareindustrie eine aktuelle und fordernde Problematik. Auch für DH-Forschungsinfrastrukturen ist diesbezüglich ein gezielter Umgang gefragt, um Software fit zu halten. Unwissenheit hinsichtlich der Wartung einer Software kann schlimmstenfalls zu einer zukünftigen Unbrauchbarkeit der Forschungsergebnisse führen. Eine dahingehende Bewusstseinsbildung kann über die empirische Betrachtung vorhandener Praktiken und Lösungswege geschehen.

Anhand von *monasterium.net* haben wir exemplarisch mögliche Verjüngungsmaßnahmen dargestellt. Das Projekt eignet sich als Fallbeispiel, da seine Software eine über zehnjährige Laufzeit aufweist. Geringes Projektbudget und häufiger Personalwechsel mit daraus resultierenden Wissensverlusten haben die Codebasis gezeichnet. Das Projekt zeigt, dass Nachvollziehbarkeit des Entwicklungsprozesses, systematisches und standardisiertes Vorgehen, Modularisierung von Softwarekomponenten sowie kontinuierliches Testing in die Evolution von Software gewinnbringend eingreifen können.

Die Verantwortung kann allerdings nicht allein bei den Entwickler/innen liegen. Um Wissensverluste vorzubeugen, müssen langfristige Strukturen aufgebaut und finanziell abgesichert werden. Es muss Teil der Förderungspolitik werden, die Unausweichlichkeit der Softwarealterung zu bedenken. Sollen Entwicklungen auch nach fünf Jahren noch benutzbar sein, muss der Aufwand der nachhaltigen Entwicklung und Wartung in der Antragsplanung verankert werden.

Fußnoten

1. Stack Overflow ist eine Online Community, zur gegenseitigen Unterstützung und zur Wissensgenerierung bei Fragen zur Softwareentwicklung: stackoverflow.com
2. github.com/icaruseu/mom-ca
3. material.io/guidelines/

Bibliographie

Andrews, Tara / Zundert, Joris van (2018): “What are you Trying to Say? The Interface as an Integral Element of Argument”, in: Bleier, Roman et al. (eds.): *Digital Scholarly Editions as Interfaces* (=Schriften des Instituts für Dokumentologie und Editorik). Norderstedt: Books on Demand.

Czmiel, Alexander (2017): “Funktionalität Digitaler Editionen“, in: *DHd 2017. Digitale Nachhaltigkeit. Konferenzabstracts.* Bern 138-141. http://www.dhd2017.ch/wp-content/uploads/2017/02/Abstractband_ergaenzt.pdf [letzter Zugriff 24. September 2017].

Demeyer, Serge / Ducasse, Stéphane / Nierstrasz, Oscar (2013): *Object-Oriented Reengineering Patterns*. Bern: Square Bracket Associates. http://scg.unibe.ch/download/oorp/OORP.pdf [letzter Zugriff 24. September 2017].

Engels, Gregor et al. (2009) "Design for Future: Legacy-Probleme von morgen vermeidbar?", in: *Informatik Spektrum* 32, 5: 393-397. https://doi.org/10.1007/s00287-009-0356-3 [letzter Zugriff 24. September 2017].

Godfrey, Michael W. / German, Daniel M. (2008): “The Past, Present, and Future of Software Evolution”, in: *Proceedings of the 2008 Frontiers of Software Maintenance.*

New York: IEEE 129-138. https://doi.org/10.1109/FOSM.2008.4659256 [letzter Zugriff 24. September 2017].

Goltz, Ursula et al. (2015): “Design for future: managed software evolution”, in: *Computer Science - Research and Development 30, 3-4*: 321-331. https://doi.org/10.1007/s00450-014-0273-9 [letzter Zugriff 24. September 2017].

Harold, Rusty Elliotte (2008): *Refactoring HTML. Improving the Design of Existing Web Applications*. Upper Saddle River, NJ: Addison-Wesley.

Hattrick, Simon (2016): Research Software Sustainability. Report on a Knowledge Exchange Workshop. JISC: http://repository.jisc.ac.uk/6332/1/Research_Software_Sustainability_Report_on_KE_Workshop_Feb_2016_FINAL.pdf [letzter Zugriff 24. September 2017].

Kasper, Dominik / Grüntgens, Max (2017): “Nachhaltige Konzeptionsmethoden für Digital Humanities Projekte am Beispiel der Goethe-Propyläen“, in: *DHd 2017. Digitale Nachhaltigkeit. Konferenzabstracts.* Bern 165-168. http://www.dhd2017.ch/wp-content/uploads/2017/02/Abstractband_ergaenzt.pdf [letzter Zugriff 24. September 2017].

Lehman, Meir M. (1980): “Programs, life cycles, and laws of software evolution”, in: *Proceedings of the IEEE 68, 9: 1060-1076.* https://doi.org/10.1109/PROC.1980.11805 [letzter Zugriff 24. September 2017].

Mazinanian, Davood / Tsantalis, Nikolaos (2017): “CCSDev: Refactoring duplication in Cascading Style Sheets”, in: *Proceedings of the 39th International Conference on Software Engineering Companion.* New York: IEEE 63-66. https://doi.org/10.1109/ICSE-C.2017.7 [letzter Zugriff 24. September 2017].

Paech, Barbara et al. (2016): “Empirische Forschung zu Software-Evolution”, in: *Informatik Spektrum 39, 3*: 186-193. https://doi.org/10.1007/s00287-015-0910-0 [letzter Zugriff 24. September 2017].

Parnas, David L. (1994): “Software Aging”, in: *Proceedings of 16th International Conference on Software Engineering.* New York: IEEE 279-287. https://doi.org/10.1109/ICSE.1994.296790 [letzter Zugriff 24. September 2017].

Schrade, Torsten (2017): “Nachhaltige Softwareentwicklung in den Digital Humanities. Konzepte und Methoden“, in: *DHd 2017. Digitale Nachhaltigkeit. Konferenzabstracts.* Bern 168-171. http://www.dhd2017.ch/wp-content/uploads/2017/02/Abstractband_ergaenzt.pdf [letzter Zugriff 24. September 2017].

Thaller, Georg E. (2000): ISO 9001: *Software-Entwicklung in der Praxis.* Hannover: Heise.