Nachhaltige Softwareentwicklung in den Digital Humanities. Konzepte und Methoden.

Schrade, Torsten
Torsten.Schrade@adwmainz.de
Akademie der Wissenschaften und der Literatur Mainz, Deutschland

Ausgehend von den umfangreichen Infrastrukturinitiativen der vergangenen Jahre existieren inzwischen vielfältige digitale Ressourcen, Werkzeuge und Dienste, die von einer lebhaften digitalen Forschungskultur in den Geisteswissenschaften zeugen. Arbeitsgruppen wie beispielsweise NESTOR oder auch die DINI-Initiative haben es sich zur Aufgabe gemacht, Empfehlungen und *best practices* für den gesamten Lebenszyklus digitaler geisteswissenschaftlicher Forschungsprojekte (Datenerfassung, Datenverwaltung, Datenpublikation, Datenarchivierung) zu entwickeln. Mit dem von DARIAH und TextGrid initiierten Memorandum zur nachhaltigen Bereitstellung digitaler Forschungsinfrastrukturen für die Geistes- und Kulturwissenschaften in Deutschland ist das Thema ‚Digitale Nachhaltigkeit' ganz besonders in den Fokus gerückt (s. http://dhd-blog.org/?p=6559).

Während das Bewußtsein für eine nachhaltige Erschließung kultureller Objekte durch den Einsatz entsprechender Datenformate und -standards inzwischen als hoch eingeschätzt werden kann, gelingt eine nachhaltige Integration von Softwarewerkzeugen in die konkrete Forschungswirklichkeit geisteswissenschaftlicher Projekte noch nicht immer. Die Gründe hierfür sind divers und reichen von einer immer noch existierenden, mangelnden Akzeptanz bzw. Berührungsangst der Geisteswissenschaftler_innen hinsichtlich informationstechnischer Verfahren bis hin zu einer nicht an den Projektzielen ausgerichteten Implementierung der benötigten Software seitens der technischen Partner eines Projektes.

Ein bisher wenig berücksichtigter aber ganz zentraler Grund ist jedoch, dass die Ebene der Softwareentwicklung in den Nachhaltigkeitsdiskussionen der Digital Humanities bisher kaum eine Rolle spielt. Erst seit diesem Jahr liegt ein erster Bericht zu generellen Voraussetzungen für die Nachhaltigkeit von Forschungssoftware vor (Hettrick 2016). Dieser kommt zu folgendem Schluss: „many researchers know how to code, but few understand the wider set of skills that are needed to develop reliable, reproducible and reusable software. […] software engineering should be incorporated […] at the very start of a research career." (Hettrick 2016, S. 14). Neben den sicherlich notwendigen Überlegungen zur Nachhaltigkeit geisteswissenschaftlicher Forschungsdaten sollte künftig mehr darauf geachtet werden, neben der reflexiven Ebene auch das konkrete entwicklerische Handwerkszeug in Digital Humanities Studiengänge einzubeziehen. Insbesondere müssen entwicklerische Leistungen als gleichrangige akademische Tätigkeit anerkannt werden (vgl. Hettrick, S. 13). Neben die Theorie sollte ein akademisch anerkanntes Digital Humanities "Craftsmanship" treten.

Die Gründe für eine nicht nachhaltige Entwicklung geisteswissenschaftlicher Software sind relativ einfach zu identifizieren und keineswegs spezifisch für das akademische Entwicklungsumfeld. Sie spielen genauso in der freien Wirtschaft oder der Open Source Szene eine Rolle. Zu den Hauptgründen einer mangelnden Software-Nachhaltigkeit können beispielsweise gehören:

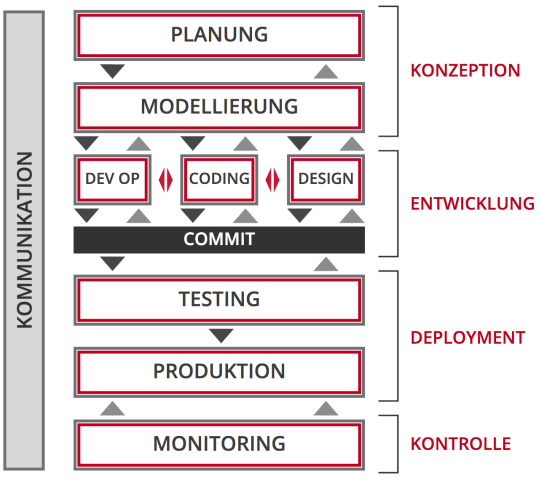
- Eine ausgelaufene Projektfinanzierung, wodurch der Weiterbetrieb der Software nicht mehr gewährleistet ist,
- Entwickler_innen, die dem Projekt nicht mehr zu Verfügung stehen, aber vor ihrem Weggang die ausschließlichen Wissensträger waren,
- eine veraltete und nicht mehr wartbare Infrastruktur,
- veralteter oder unverständlicher Programmcode, der von neu einsteigenden Entwickler_innen weitergeführt werden muss,
- Sicherheitslücken, die vermeidbar gewesen wären, jetzt aber einen Weiterbetrieb der Software verhindern,
- (schwerwiegende) Bugs in der Software, die erst im Produktivbetrieb auffallen, da vorher keine Softwaretests durchgeführt wurden,
- ein fehlendes Monitoring der geisteswissenschaftlichen Forschungsanwendung, wodurch Störfälle nicht oder erst spät auffallen.

Blickt man vor diesem Hintergrund in die freie Wirtschaft und Softwareindustrie und fragt nach aktuellen Projektmanagement-Methoden bzw. Herangehensweisen zur Steigerung der Qualität und Nachhaltigkeit einer Software, lässt sich sehr schnell feststellen, dass insbesondere die unter dem Stichwort „Agile Softwareentwicklung" gefassten methodischen Ansätze sich sehr gut eignen, um den genannten Herausforderungen entgegenzutreten (vgl. Ayelt 2014). Obwohl agile Entwicklungsansätze häufig unterschiedliche Teilaspekte eines Entwicklungs-Workflows adressieren (bspw. die Konzeptionsebene, die Entwicklungsebene, die Ebene des Testings oder des Deployments einer Software), legen alle doch den Schwerpunkt auf eine kontinuierliche Kommunikation aller Projektbeteiligten untereinander (von den Stakeholdern über die Entwickler_innen bis zu den Testnutzer_innen und Endnutzer_innen). Agile Softwareentwicklung sieht häufig in einer für alle nachvollziehbaren Kommunikation den entscheidenden Schlüssel für ein erfolgreiches und nachhaltiges Softwareprodukt.

Hiermit befinden wir uns aber wiederum sehr nahe an den Digital Humanities. Schon lange wird für Digitale Geisteswissenschaftler_innen eine kommunikative Schlüsselstellung reklamiert. Als Mediatoren mit Fachwissen aus zwei Welten sollen sie eine für alle Parteien gemeinsame, verständliche Sprache entwickeln und so die unterschiedlichen geistes- und informationswissenschaftlichen Konzepte eines Forschungsprojektes miteinander in Einklang bringen.

Das Team der Digitalen Akademie der Mainzer Akademie der Wissenschaften und der Literatur integriert bereits seit 2009 Konzepte aus dem Bereich der agilen Softwareentwicklung in die tägliche Forschungs-, Entwicklungs- und Projektarbeit. Hierbei werden auf verschiedenen Ebenen Konzepte angewendet, die sich über die Zeit als besonders geeignet für geisteswissenschaftliche Anwendungskontexte herausgestellt haben. Sowohl zur Steigerung der Softwarequalität, insbesondere aber auch zur Steigerung der Nachhaltigkeit der Forschungsapplikationen wurde mit der Zeit eine an den Prinzipien der „Continuous Delivery“ ausgerichtete Prozesskette aufgebaut (zum Begriff vgl. Wolff 2015).

Die nachfolgende Grafik gibt einen Überblick über die einzelnen Ebenen dieser Prozesskette.



Auf Ebene der Konzeption und Programmierung kommen zwei Herangehensweisen zum Einsatz: das sogennannte Domain-Driven Design (DDD) und das Behaviour-Driven Development (BDD). Domain-Driven Design ist dabei zum einen eine Herangehensweise an die Modellierung komplexer Software, zum anderen ein bestimmtes Denkkonzept zur Steigerung der Produktivität von Softwareprojekten im Umfeld komplexer fachlicher Zusammenhänge. Das Hauptaugenmerk fällt dabei auf die Einführung einer ubiquitären (allgemein verständlichen) Sprache, welche in allen Bereichen der Softwareerstellung von den Konzeptionsgesprächen mit den Fachwissenschaftler_innen bis hin zur Code-Ebene verwendet werden sollte. Domain-Driven Design ist unabhängig von Programmiersprachen, Tools und Frameworks (vgl. Evans 2013, S. 13). DDD eignet sich ausgezeichnet für eine nachhaltige und offene Modellierung geisteswissenschaftlicher Anwendungskontexte, da iterativ gearbeitet wird. Zu Beginn der Domänen-Modellierung ist in geisteswissenschaftlichen Forschungsprojekten die eigentliche Datengrundlage und der Funktionsumfang der zu erstellenden Software häufig nicht vollständig klar. Beides entsteht sukzessive in der Beschäftigung mit dem Forschungsgegenstand. Somit können während der eigentlichen Entwicklung häufig neue Gegenstände auftauchen, noch nicht bedachte Eigenschaften hinzukommen oder sich auch Teile der Applikationslogik grundlegend ändern. Durch regelmäßige Iterationen nach dem DDD-Prinzip kann die Software kontinuierlich mit der sich stetig wandelnden Projektrealität verändern. Die Codebasis bleibt dabei im Einklang mit der Konzeptions- bzw. Modellierungsebene.

Behaviour-Driven Development wiederum geht davon aus, dass sich die Funktionalität einer Anwendungsdomäne (und somit die Geschäftslogik eines Domänen-Modells) durch formalisierte Szenarien in einer allgemeinverständlichen Sprache beschreiben lässt. BDD ist ein „outside-in“-Ansatz, der von außen (also mit dem Blick der Geisteswissenschaftlerinnen) auf eine Software blickt und deren Funktionalität in ausführbaren Tests dokumentiert. Ursprünglich im Umfeld des Test Driven Development entstanden, achtet auch BDD darauf, dass die Nutzungszenarien einer Software vor der eigentlichen Programmierung der Software erstellt werden. Dadurch dass die Tests in natürlicher Sprache nach einem festen Dreischritt-Prinzip (Angenommen…, Wenn…, Dann…) formuliert werden, kann die oftmals komplexe und wenig nachhaltige Präsentationsschicht und Funktionslogik einer Software in direkter Zusammenarbeit mit den Fachwissenschaftler_innen gemeinsam beschrieben und nachhaltig dokumentiert werden. In der Umsetzung hat dies für die Entwickler_innen den Vorteil, dass exakt nur soviel Code geschrieben werden muss, bis die jeweiligen Tests erfolgreich ablaufen und die Software exakt wie geplant funktioniert.

Als dritte wichtige Säule in einem nachhaltigen Entwicklungsprozess ist die Virtualisierung und Automation der Infrastruktur nach ‚DevOps‘-Prinzipien (eine Zusammenfügung der beiden Begriffe Development und Operations) zu nennen. ‚DevOps‘ betrachtet ‚Infrastruktur als Code‘ und setzt entsprechende Werkzeuge ein, um eine vollständige Kapselung der Softwareschicht und gleichzeitige Reproduzierbarkeit der Gesamtapplikation einschließlich ihrer Infrastruktur zu erreichen. Der große Vorteil dieser Verfahrensweise liegt in der automatisch enstehenden Dokumention hochgradig spezialisierter Anwendungsumgebungen. Gleichzeitiger sind die „Baupläne“ dieser Anwendungsumgebungen in einem Versionskontrollsystem versionierbar.

Alle bisher genannten Konzepte streben eine Versionierbarkeit ihrer Outputs an, was die Nachvollziehbarkeit und somit die Nachhaltigkeit auf der Software-Ebene deutlich steigert. Auf diese Weise hergestellte Software legt nicht nur offen, wie sie funktioniert, sondern wie sie hergestellt wurde und das auf allen Ebenen, von der Konzeption über die Programmierung und das Testing bis hin zum Deployment. Insofern kommt dem ‚Commit', also dem wiederholten Einspielungsvorgang der jeweiligen Entwicklungsstände die Rolle des zentralen Dreh- und Angelpunktes einer nachhaltigen Softwareentwicklung zu. Zusammenfassend lässt sich im Abgleich zu den oben genannten Punkten festhalten, dass bei einer nachhaltigen Softwareentwicklung

- insbesondere die beständige Kommunikation aller Projektbeteiligten untereinander einen zentralen Faktor darstellt,
- ein gemeinsames Vokabular festgelegt werden und dies auf allen Ebenen konsequent angewendet werden muss (Konzeption, Datenschema, Code, Tests etc.).
- auf forschungsgetriebene, agile Entwicklungsmethoden gesetzt werden sollte,
- ein nachvollziehbarer Entwicklungsprozess durch Versionskontrolle gewährleistet werden muss,
- die Infrastruktur nach DevOps-Prinzipien virtualisiert und automatisiert werden sollte,
- Softwaretests vor jedem Live-Deployment durchzuführen sind,
- die Applikation im Produktivbetrieb kontinuierlich überwacht werden muss.

Innerhalb des Vortrags werden die dargelegten Konzepte und Methoden anhand von Projektbeispielen genauer illustriert und zur Diskussion gestellt. Der Beitrag versteht sich somit als ein Erfahrungsbericht aus der mehrjährigen Arbeit im Kontext der Digital Humanities Projekte der Digitalen Akademie der Mainzer Akademie sowie des Mainzer Zentrums für Digitalität in den Geistes und Kulturwissenschaften (mainzed).

Fußnoten

1. In Anlehnung an den Begriff des *software craftsmanship*.

Bibliographie

Komus, Ayelt (2014): *Status Quo Agile 2014*. Hochschule Koblenz. https://www.hs-koblenz.de/rmc/fachbereiche/wirtschaft/forschung-projekte-weiterbildung/forschungsprojekte/status-quo-agile/ [letzter Zugriff 26. August 2016].

Evans, Eric (2003): *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison Wesley.

Hettrick, Simon (2016): *Research Software Sustainability: Report on a Knowledge Exchange Workshop*. Edinburgh: The Software Sustainability Institute.

Wolff, Eberhard (2015): *Continuous Delivery: Der pragmatische Einstieg*. Heidelberg: dpunkt.verlag.

Vernon, Vaughn (2013): *Implementing Domain Driven Design*. Boston: Addison Wesley.