Extreme Programming

Ein Bericht über den Vortrag von Herrn Prof. Dr. Andreas Zeller

Von Jan Hendrik Dithmar

Prof. Dr. Andreas Zeller eröffnet die Ringvorlesung „Perspektiven der Informatik“ mit seinem Vortrag über „Extreme Programming“. Zunächst stellt er kurz heraus, dass es grundsätzlich zwei Arten bei der Programmentwicklung gibt: die Arbeit allein oder die Arbeit im Team.

Für die Arbeit stellt er den Studentinnen und Studenten den „Code-and-Fix-Zyklus“ vor, bei dem es im wesentlichen drei Schritte gibt:

1. Code schreiben und testen
2. Code „verbessern“ (Fehlerbeseitigung, Erweiterung, Effizienz, ...)
3. wieder bei Punkt 1 beginnen

Dafür gelten aber ganz spezielle Voraussetzungen. Denn das Problem muss ganz klar spezifiziert sein und eine Person sollte das Programm allein erstellen und überschauen können. Treffen diese zwei Punkte zu, so ist gegen die „Code-and-Fix-Zyklus“-Methode nichts einzuwenden. Diese Programmiermethode hat allerdings auch gewaltige Nachteile, denn man kann keine Prognose über die Kosten oder die Qualität – schon gar nicht über einen möglichen Fertigstellungstermin – machen. Des weiteren nimmt die Qualität des jeweiligen Produkts mehr und mehr ab, je länger man daran bastelt. Irgendwann ist man an dem Punkt angelangt, an dem es schwer fällt, das Programm zu warten bzw. an dem die Wartung gar nicht mehr möglich ist. Da diese Programmiermethode de facto nur für 1-Personen-Projekte geeignet ist, kommt auf eine Firma ein großes Problem zu, wenn der verantwortliche Programmierer kündigt. In dieser Situation kann der Extremfall eintreten: dass der Quellcode ist nicht mehr zu gebrauchen ist, da außer dem Programmierer diesen keiner richtig durchschaut. Wenn man daraus die Konsequenzen zieht, stellt man fest, dass es unausweichlich ist, große Software-Projekte sorgfältig zu planen und zu koordinieren. Sobald ein Programmierer für einen Anwender eine Software erstellen soll, sind Meinungsverschiedenheiten, was den Funktionsumfang etc. betrifft, schon vorprogrammiert. In diesem Fall steht der Informatiker vor einer ernstzunehmenden Herausforderung: er muss herausfinden, was der Anwender eigentlich will.
Im Jahre 1968 führten diese Probleme zur sogenannten „Software-Krise“.

Als Konsequenz der Software-Krise entwickelte man das sogenannte „Wasserfallmodell“. Dieses Modell ist in fünf Phasen gegliedert:

- Anforderungen,
- Entwurf,
- Codierung,
- Test und
- Produktion.

In der ersten Phase (Anforderungen) wird festgelegt, was das Programm leisten soll. Dabei wird nicht näher darauf eingegangen, wie es das tut, denn dies ist nicht im Interesse des Auftraggebers. Endprodukt dieser ersten Phase ist ein *Pflichtenheft*, welches alle relevanten Merkmale der Software bezüglich Funktionsumfang und Qualität enthalten und präzise beschreiben, sowie vollständig, konsistent und für Auftraggeber und Auftragnehmer verständlich sein soll.
Allerdings ist Forlgendes anzumerken: man wird es generell nicht schaffen, ein Pflichtenheft präzise und – zumindest für den Auftraggeber – verständlich zu formulieren.

Die zweite Phase beschäftigt sich mit zwei Fragen die Systemarchitektur betreffend: Welche Module gibt es und wie stehen sie untereinander in Beziehung? Sind die Fragen bezüglich der Systemarchitektur geklärt, so entsteht eine *Entwurfsbeschreibung*. Die Entwurfsbeschreibung bildet die Grundlage für Phase 3: die Codierung des Programms.

Nun beginnt das, was sich viele Leute unter der Arbeit des Informatikers vorstellen: die Implementierung (im Volksmund: Programmierung). Liegt ein sorgfältig ausgearbeiteter Entwurf vor, so ist die dritte Phase recht einfach und macht lediglich 10-15% der eigentlichen Arbeit an der Software aus. Als Ergebnis dieser Phase kann man nun den *Programmcode* bewundern.

Nun kann Phase 4 beginnen: die Testphase. Das Testen (Validieren) der Software kann sehr gut durch Gegenlesen eines zweiten Programmierers geschehen (Reviews). Eine weitere Möglichkeit ist das Vorhaben, bewusst Fehler zu produzieren. Bei kritischer Software, z.B. bei Flugzeugen oder Satelliten, wird man allerdings keine der beiden Methoden verwenden. In diesem Fall versucht man die Validierung des Produkts durch Programmbeweise.
Ein Programmierer zeigt meist eine gewisse Beziehung zu seinem Programm. Da die Validierung ein destruktiver (= zerstörerischer) Prozess ist, sind es sehr oft Dritte, die diesen Prozess vornehmen. In der Regel will der Programmierer selbst seine Schöpfung nicht zerstören. Wurde das Programm ausführlich getestet, so kann man in die fünfte Phase übergehen: die Produktion.

Bei der Produktion wird die fertige Software beim Kunden installiert und danach gewartet. Wartung von Software ist ein wichtiger Teil und macht 50% der Kosten aus.

Das Wasserfallmodell brachte klare Vorzüge, wie z.B. strukturiertes Vorgehen, klares Benennen einzelner Phasen und ihrer Anforderungen, als auch die Möglichkeit des ersten Messens und Abschätzens des Entwicklungsaufwandes. Allerdings hat dieses Modell nicht nur Vorzüge. Durch Verfeinerungen, wie z.B. der Durchführung früher Tests – sobald etwas implementiert wurde – und der Tatsache, dass der Kunde Prototypen des Produkts sehen und ausprobieren kann (Rückkopplung), konnte der Prozess der Softwareentwicklung optimiert werden.

Ein weiterer Punkt im Vortrag: die „Prozessmodellierung“. Die dahinter steckende Grundidee ist folgende: Entwicklungsprozesse sind auch nur Software. Auf Grund dieser Idee kamen in den 90ern immer mehr programmierte Entwicklungsumgebungen auf. Folge der Prozessmodellierung war eine automatisierte, sehr systematische Programmentwicklung. Doch kommen dabei auch Fragen auf: Bringt eine systematische Entwicklung auch systematische Programme vor? Lassen sich Programmierer gerne automatisieren?

Man nimmt an, dass die Änderungskosten exponentiell ansteigen. Dies heißt für die Softwareentwicklung, dass man schon bei den Anforderungen und dem Entwurf sehr sorgfältig arbeiten muss. Je später eine Änderung erfolgt, desto mehr muss man zurückgehen und die Informationen auf den neusten Stand bringen (angefangen bei den Anforderungen, über den Entwurf,...). Nun stellt sich die Frage: wie viel Aufwand sollte man dann für den Entwurf betreiben? Auf der einen Seite ist ein großer Aufwand beim Entwurf gut, jedoch stellen sich dem Informatiker einige Probleme. Die Erstellung der Anforderungen und des Entwurfs können sehr viel Zeit in Anspruch nehmen. In dieser Zeit können sich die Einsatzbedingungen längst geändert haben. Des weiteren können manche Anforderungen nur beim Einsatz eines Prototypen geklärt werden. Da ein Prototyp frühestens nach der Codierungsphase zur Verfügung steht, bedeutet eine Änderung sehr viel Aufwand und auch hohe Kosten. Außerdem ist eine frühe Verfügbarkeit der Software wünschenswert und kann sogar über die Zukunft einer Firma entscheiden.
Es wäre also ein sehr großer Vorteil, die Änderungskosten niedrig und kostant zu halten und gleichzeitig flexibel zu sein. In diesem Fall greift das Modell des *Extreme Programming* oder kurz XP. Extreme Programming ist ein Vorgehensmodell für kleine bis mittelgroße Teams, deren Anforderungen vage sind und/oder sich schnell ändern können. Extreme Programming beruht auf bewährten Techniken und setzt diese in höchstem (extremem) Maße ein.

Ein wichtiger Vorteil ist das Programmieren in Paaren. Somit wird schon während der Programmentwicklung kontinuierlich der Quellcode auf seine Richtigkeit und Konsistenz überprüft. Ein weiterer Punkt ist das automatische bzw. automatisierte Testen von Komponenten. Kontinuierliches Design und Redesign der Software sowie kurze Release-Zyklen, die eine kontinuierliche Rückmeldung und ein ständiges Einbeziehen des Auftraggebers gewährleisten sollen, sind weitere Vorteile.
In den 90er Jahren kam man zu der Erkenntnis, dass das Gegenlesen von Programmen der beste Weg ist, die Qualität zu steigern. Im Extreme Programming findet dieser Vorteil Anwendung. Deshalb wird in Paaren programmiert. Das bedeutet: ein Monitor und eine Tastatur, aber zwei Programmierer. Vorteil hierbei ist (wie schon oben erwähnt): es findet ein ständiger Dialog zwischen den beiden statt, weil der eine schreibt und der andere gegenliest. Außerdem findet hier keine Spezialisierung auf Rollen (z.B. „Programmierer“ und „Tester“), sondern eine häufiger Rollen- und damit verbundener Tastaturtausch statt.

Als Beispiel, wie erfolgreich das Programmieren in Paaren ist, kann man das „C3 Chrysel Payroll System“ von 1997 anführen. Teile dieses Projekts wurden zum einen in Einzelarbeit, zum anderen in Paararbeit erstellt. Tatsache ist, dass nicht nur das System termin- und ressourcengerecht installiert wurde, sondern dass nahezu alle in den ersten 5 Monaten des Einsatzes aufgetretene Fehler aus der Einzelarbeit stammten.
Als zweites Beispiel kann man ein Experiment der Universität in Utah aus dem Jahre 1999 anführen: 13 Einzelprogrammierern (Studenten) und 14 Programmiererpaaren wurden 4 Aufgaben in 6 Wochen gestellt. Danach wurden die eingereichten Programme getestet. Als Ergebnisse konnte man damals festhalten, dass Programmiererpaare zum einen bessere Qualität produzieren und zum anderen schneller fertig sind. Paar-Programmierer finden außerdem, dass die Arbeit in Paaren ihr Vertrauen in das Programm erhöht (96 %) und mehr Spass macht als Einzelarbeit (94 %). Ein weiteres Argument für die Arbeit in Paaren ist folgendes: bei der Arbeit im Programmiererteam beschränkt man sich auf die wesentlichen Dinge der Programmerstellung. Das wichtigste bei der Arbeit im Programmiererteam ist, dass die Chemie zwischen den Partnern stimmen sollte, ja sogar stimmen muss.

Ein weiterer wichtiger Punkt ist das kontinuierliche (automatische) Testen der Programme. Ziel ist es, jederzeit die Funktionsfähigkeit des Systems zu gewährleisten. Auch diese Art der Validierung hat ihre Vor- und Nachteile. Im Extreme Programming werden z.B. lediglich Testfälle definiert, bevor eine Implementierung stattfindet. Gut dabei ist, dass der Test die Modulfunktion aus exemplarischer Sicht beschreibt und dass die Tests automatisch ablaufen (selbst radikale Änderungen können so validiert werden). Des weiteren sind Testfälle formaler als Spezifikationen und können auftretende Fehler dokumentieren: für jeden neuen Fehler wird sofort ein reproduzierender Testfall erstellt. Jedoch gibt es auch einen sehr großen Nachteil: die Testfälle als Spezifikation decken nur Beispiele ab.

Für das Software-Praktikum der Universität Saarbrücken bedeutet dies, dass das Verhalten des Systems durch Szenarios beschrieben wird und dass diese Szenarios in ausführbare Testfälle gegossen werden, bevor implementiert wird. Testbarkeit verbessert die Systemstruktur erheblich: die Subsysteme werden deutlich mehr entkoppelt und es findet eine deutliche Trennung von Funktionalität und Präsentation statt.
Konkret heißt das, dass man durch konsequente Testbarkeit selbst größere Änderungen am System vornehmen kann, ohne die Qualität oder Stabilität zu gefährden. Diese Methode findet im XP Anwendung, damit das System stets so einfach wie möglich gehalten wird. Man kann vier Punkte aufzählen, wann ein Design richtig ist:

- alle Testfälle müssen korrekt behandelt werden,
- es dürfen keine Redundanzen enthalten sein,

- alle wichtigen Absichten müssen klar ausgedrückt sein und
- das System muss mit einer minimalen Anzahl von Klassen und Methoden auskommen.

Refactoring (wörtl. „Refaktorisieren“) beinhaltet das Umstrukturieren von Software in weitgehend unabhängige Faktoren. Beim XP finden sogenannte Refactoring-Verfahren beim systematischen Redesign Anwendung. Refactoring-Verfahren sind z.B. die *Extract Method* (Code zu einer Methode zusammenfassen), die *Move Method* (Bewegen einer Methode von Klasse zu Klasse), ... Erste Werkzeuge realisieren bereits solche Methoden.
Der Vorteil bei der Extract Method: das System ist und bleibt stets wohlstrukturiert. Nachteile sind, dass man fast nichts der Arbeit außerhalb des Projekts wiederverwenden kann.

Wichtiger Unterschied ist, dass beim XP der Auftraggeber die Anforderungen über Stories beschreibt. Jede Story beschreibt einen konkreten Vorfall, der für den Betrieb des Systems notwendig ist. Beim Extreme Programming werden keine Entwurfsphasen benötigt; das Programmieren steht im Mittelpunkt. Die Funktionalität der Software wird in Stories zusammengefasst und jede Story in einen Testfall umgesetzt. Dieser wird dann implementiert. Die Implementierung selbst ist abgeschlossen, wenn alle Testfälle bestanden sind. Sollten bei der Abnahme weitere Fragen auftreten, so werden neue Stories formuliert. Das Spiel beginnt von vorne. Sehr vorteilhaft dabei ist, dass der Kunde ständig in die Entwicklung einbezogen ist. Der Projektfortschritt wird gemessen an den erfolgreich getesteten Stories.

Das Modell des Extreme Programming wird allerdings auch kritisiert: das Vorgehensmodell sei unvollständig, denn Stories alleine reichten nicht aus, ein System zu beschreiben. Ein weiterer Kritikpunkt besteht darin, dass Extreme Programming keine Aussage über Qualität und die Integration von Qualitätsstandards macht und dass man die Kosten der Entwicklung nicht abschätzen kann. Als einzig logische Folge hat man also Extreme Programming mit bewährten Verfahren und Modellen ergänzt.