Seminar "Softwareevolution"
Fallstudie zur Entwicklung von Software

Peter Weißgerber

16. Mai 2002

1 Überblick

1.1 Die betrachtete Fallstudie

Im Rahmen des Seminars "Softwareevolution" habe ich mir eine Fallstudie zur Entwicklung eines Teilsystems der Lucent 5ESS Telefonweiche angeschaut[1]. Diese Studie beschäftigt sich mit der Entwicklung dieses Teilsystems über einen Zeitraum von 12 Jahren und versucht, die Wirkung von Parallelität in der Softwareentwicklung, vor allem bei umfangreicher Software, zu erforschen. Sie gehört zu einer Studienreihe namens "Code Decay Projekt" und soll eine Grundlage für darauf aufbauende, weitere Studien zu diesem Thema sein.

Parallelität wird hierbei von verschiedenen Ebenen beleuchtet. Einerseits wird gleichzeitig an verschiedenen Versionen des Systems gearbeitet, denn es wird für zwei unterschiedliche Märkte entwickelt und alte Versionen werden auch noch weiter gepflegt. Außerdem können zum Beispiel verschiedene Entwickler an den gleichen Versionen und Dateien arbeiten oder sich um die Lösung desselben Problems kümmern. Änderungen an der Software werden immer durch so genannte Initial Modification Requests (IMR) und davon ausgehend Modification Requests (MR) eingeleitet. Solche IMR oder MR können aber auch wieder eine oder mehrere Dateien betreffen, genauso wie umgekehrt eine Datei von mehreren IMR/MR betroffen sein kann. Des weiteren ist es leicht möglich, dass sich ein Entwickler mit verschiedenen Dateien und verschiedenen IMR/MR beschäftigt.

Zu allen diesen Parallelitätsebenen werden von der Studie Grafiken vorgestellt, die auch interpretiert werden. Hierbei wird ein Maß der Parallelität definiert und anhand diesem versucht zu erforschen, wie sich parallele Phänomene - insbesondere parallele Änderungen - auf die Qualität der Software auswirken. Man hat geschlossen, dass viele parallele Änderungen eine signifikant höhere Fehlerwahrscheinlichkeit nach sich ziehen und versucht zu ergründen, warum das so ist.

Anschließend stellt die Studie verschiedene Ideen vor, wie die gewonnenen Erkenntnisse in der Softwareentwicklung besser berücksichtigt werden können. Diese Vorschläge gehen allerdings größtenteils nicht direkt aus den Studienergebnissen hervor, sondern stellen aufbauend auf ihnen neue Ideen in den Raum.

[1] Zu Grunde liegendes Papier: "Parallel Changes in Large-Scale Software Development: An Observational Case Study", ACM Transactions on Software Engineering and Methodology, 10(3), 2001

1.2 Ziele dieser Ausarbeitung

Diese Ausarbeitung soll nicht nur einen Überblick über die vorliegende Studie geben, sondern diese auch kritisch betrachten, Schwachstellen und Alternativen aufzeigen, und Möglichkeiten, ähnliche Studien zu erstellen, erörtern. Für letzteres will ich vor allem auf OpenSource-Projekte zu sprechen kommen, da das die einzigen Projekte sind, wo die benötigten Daten öffentlich zugänglich sind. Hierbei werde ich mich auf die Internetsoftware Mozilla[2] (ein Browser, HTML-Editor, eMail- und Newsclient) und die Office-Suite OpenOffice.org[3] beschränken, weil das sehr große Projekte sind, die über einen längeren Zeitraum entwickelt wurden und weiterhin werden, und von denen auch auch kommerzielle Versionen (Netscape 6.x bzw. StarOffice 6) existieren. Aber auch hier ist natürlich zu überlegen, inwieweit die veröffentlichten Daten ausreichen, um bestimmte Schlüsse ziehen zu können.

2 Messverfahren und Durchführung der Studie

2.1 Datensammlung

Damit solch eine solche Studie überhaupt einen wissenschaftlichen Wert hat, muss natürlich klar zu erkennen sein, wie die zugrunde liegenden Daten erfasst wurden. Hier wurden die Daten über die History des Change Management Systems und des Versionskontrollsystems ermittelt.

2.1.1 Kontrolle der Änderungen

Zur Kontrolle der Änderungen ist beim hier betrachteten System das Change Management Layer System, kurz ECMS, zum Einsatz gekommen. Mit ECMS können Änderungen am System initiiert und protokolliert werden. Jedes zu lösende Problem, z.B. Verbesserungen (*Improvements*), Fehlerbeseitigungen (*Bugfix*) und Systemerweiterungen (*Feature*), wird durch ein Initial Modification Request (*IMR*) dargestellt. Ein solcher IMR wird dann durch einen oder mehrere Modification Requests (*MR*) von den Entwicklern implementiert. Modification Requests sind also die kleinste Einheit im ECMS-System.

Natürlich ist ECMS nicht das einzige System, das zur Änderungsprotokollierung eingesetzt werden kann. Das Mozilla-Projekt verwendet zu eben diesem Zweck eine eigens entwickelte, öffentlich zugängliche Bug-Datenbank namens Bugzilla[4], wo jeder Probleme oder Änderungswünsche eintragen kann und Abhängigkeiten dieser "Bugs" untereinander und andere Meta-Daten verwaltet werden. Ähnlich geht OpenOffice.org vor. Dieses Projekt hat Bugzilla als Ausgangspunkt genommen und an seine Bedürfnisse angepasst. Das Ergebnis nennt sich IssueZilla[5] und ist ebenfalls über das Internet frei zugänglich.

2.1.2 Code-Management

Bei der Lucent 5ESS wird der Code in einem SCCS-System verwaltet. Dieses verwaltet nur den Quelltext, ohne jede Meta-Information. Ähnlich wie bei CVS

[2] http://mozilla.org
[3] http://openoffice.org
[4] http://bugzilla.mozilla.org
[5] http://www.openoffice.org/project_issues.html

wird hier auch mit Ein- und Auschecken gearbeitet, allerdings werden Dateien, die gerade in Bearbeitung sind, gesperrt (*locking-Mechanismus*). Wie die verschiedenen Versionskontrollsysteme (CVS, SCCS und andere) genau funktionieren, kam in einem anderen Beitrag dieses Seminars zur Sprache. Welche Code-Änderungen im SCCS zu welcher IMR oder MR gehören kann aus dem ECMS herausgelesen werden.

Bei Mozilla dagegen wird der Code in einem CVS verwaltet, auf das nur bestimmte Mozilla-Mitarbeiter Schreibzugriff haben. Außerdem kommt ein weiteres Verwaltungssystem namens Bonsai[6] zum Einsatz, indem Meta-Informationen zum CVS, etwa welcher *Commit* zu welchem Bug gehört, verwaltet werden.

2.1.3 Offene Fragen / Kritik

Leider geht aus dem Papier zur Studie nicht hervor, in welcher Programmiersprache das behandelte 5ESS-System programmiert wurde. Das macht ein Vergleich mit anderen Softwaresystemen insofern schwieriger, als dass man vermuten kann, dass zumindest unterschiedliche Programmierparadigmen (objektorientierte Programmierung - z.B. in C++ oder Java - gegenüber klassischer imperativer Programmierung, etwa in C) durchaus einen Einfluss auf die Ergebnisse haben kann. Ein Grund für diese Annahme ist, dass in objektorientierten Sprachen Änderungen oftmals mehrere Dateien betreffen als bei anderen Sprachen, da ja zumeist jedes Objekt in einer einzelnen Datei gekapselt ist.

Aufgrund des Alters des Systems ist übrigens zu vermuten, dass es nicht objektorientiert programmiert ist. Möglicherweise kam also C zum Einsatz, aber auch andere Sprachen sind natürlich denkbar.

2.2 Interpretation der gesammelten Daten

2.2.1 Fakten

Als erstes wird festgestellt, dass auf eine IMR oftmals nur wenige MR fallen, es aber signifikante Ausreißer gibt. Außerdem wird festgestellt, dass an einem Tag im Mittel 22 IMR offen sind, wobei das Maximum sogar bei 62 offenen IMRs liegt. Daraus schließt man, dass die Parallelität auf dieser Ebene sehr groß ist.

Auch bei der Dauer der Features, IMRs und MRs - betrachtet hat man, wie lange diese im ECMS geöffnet waren - hat man große Unterschiede festgestellt. Einige Features waren weniger als 10 Tagen offen, andere über 1000 Tage. Die Hälfte aller MRs wurde an nur einem Tag gelöst, aber auch hier gab es signifikante Ausreißer. Dies ist deshalb wichtig, weil man vermutet, dass eben gerade Ausreißer in der Statistik immer einen recht großen Einfluss auf das Ergebnis der Auswertung haben.

Des weiteren hat man festgestellt, dass ein Drittel der Features sich auf mehr als 20 Dateien erstrecken, IMRs oftmals auch noch viele Dateien betreffen, aber MRs durchgehend nur sehr wenige (nie mehr als 3). Somit ist zu vermuten, dass bezüglich der MRs eine sehr große Granularität vorliegt.

Während die meisten Dateien nur genau eine Funktion erfüllen, gibt es andererseits auch einige Dateien, die von bis zu 20 Features gebraucht werden. Bei der Version I6 (sechste Version für den internationalen Markt), die man beispiel-

[6] http://bonsai.mozilla.org

haft betrachtet hat, waren 60% der Dateien von mehr als einer MR betroffen und man hat auch hier wieder einen hohen Randbereich (*Tail*) ermittelt.

Bei der Beziehung zwischen Dateien und Entwicklern hat man erkannt, dass an vielen Dateien nur wenige Entwickler gearbeitet haben, es aber auch hier einen großen Tail gibt, denn zu einigen wenigen Dateien haben über 20 Entwickler Änderungen beigetragen. Es ist zu vermuten, dass das solche Dateien sind, die entweder oftmals Fehler enthielten oder viele Funktionen auf sich vereinigen.

Für sehr interessant hat man die gleichzeitige Bearbeitung von Dateien gehalten. Darunter versteht man, dass die Datei in mehrere parallel offene MRs verwickelt ist. Daher hat man die maximale Anzahl der gleichzeitig offenen MRs über den Betrachtungszeitraum als das Maß der Parallelität einer Datei definiert: $PCmax = max_{Tage}(offene\ MRs)$.

Nachdem man geschlossen hat, dass Parallelität auf vielen Ebenen stattfindet und sich PCmax definiert hat, wollte man nun überprüfen, inwiefern die Anzahl der Defekte in der Software mit PCmax zusammenhängt. Als Defekt hat man dabei jeden MR definiert, der ein Problem in einer Datei anspricht. Die Analyse hat gezeigt, dass Parallelität und Zahl der Fehler stark zusammen hängen. So ist ab PCmax=3 immer mindestens ein Fehler in der betreffenden Datei aufgetreten. Außerdem steigt die Kurve der Erwartungswerte für die Anzahl der Fehler mit steigendem PCmax sehr stark an. Um die Glaubwürdigkeit der Studie zu untermauern, hat man noch zwei alternative, allerdings scheinbar sehr ähnliche Messverfahren bemüht, die zum selben Ergebnis geführt haben.

2.2.2 Sich überlappende Änderungen

Da man ja nun festgestellt hat, dass parallele Änderungen leicht Fehler nach sich ziehen, wollte man nun feststellen, warum dies so ist. Hierbei ging man so vor, dass man schon im Vorhinein vermutete, dass sich überlappende Änderungen der Grund für die hohe Fehlerwahrscheinlichkeit sind, und dies nur noch überprüfte und so untermauerte. Unter sich überlappenden Änderungen versteht man Änderungen, die an der selben Stelle des Codes, innerhalb kurzer Zeit, gemacht wurden - oftmals von verschiedenen Entwicklern - und sich möglicherweise gegenseitig stören könnten. Man ermittelte, dass 12,5 % aller Änderungen in derselben Datei innerhalb von 24 Stunden von verschiedenen Entwicklern gemacht wurden und sich 3 % dieser Änderungen sogar physisch überlappen, also genau die gleiche Stelle der Datei betreffen. Man vermutet, dass es außer diesem, im Papier "prima facie" genannten, einfachen Fall, auch noch semantische Überlappungen geben kann, die aber nicht weiter erforscht wurden und wahrscheinlich aus den gegebenen Daten auch nicht so leicht herauslesbar gewesen wären.

2.2.3 Kritik / Unstimmigkeiten

Kritikwürdig ist die Wahl von PCmax. Einerseits wird ausführlich darauf eingegangen, dass sich die Parallelität auf sehr viele Ebenen bezieht, doch dann wird eben nur eine einzige Ebene ausgewählt, um die Auswirkungen der Parallelität auf die Fehlerhäufigkeit zu zeigen. Dies wird zwar etwas ausgeglichen durch die zwei alternativen Messverfahren mit PCdays und PCwdays, wo die Anzahl der Tage mit mehr als einer MR zum Tragen kommen (einmal ungewichtet und

einmal gewichtet); jedoch muss man sagen, dass beide Messverfahren doch sehr ähnlich zum ersten Messverfahren sind und auch hier Faktoren wie verschiedene Entwickler, gleichzeitiges Arbeiten an mehreren Versionen usw. nicht direkt in die Berechnung einfließen.

Einerseits ist verständlich, dass man sich ein Maß der Parallelität definieren muss, um zu Ergebnissen kommen zu können, aber warum eben diese eine Ebene so wichtig ist, dass die nachfolgenden Forschungen nur darauf bezogen werden, hätte jedoch entweder besser begründet oder überdacht werden müssen.

Auch die Vermutung, dass es gerade die sich überlappenden Änderungen sind, die Probleme machen, ist zwar plausibel, aber die Begründung ist etwas dünne. Man hätte zum Beispiel nochmal nachforschen müssen, ob solche Änderungen wirklich mehr Probleme machen als andere. Dies wäre anhand der Daten sicherlich möglich gewesen.

Man muss anerkennen, dass Lucent die Studie veröffentlicht hat, denn das ist eine der wenigen öffentlich zugänglichen Studien dieser Art. Man erwähnt ja, dass man diese Arbeit auch als Wegbereiter für weitere Studien versteht und gibt praktisch ein Modell zur Hand um solche Studien durchzuführen. Aufgrund der oben genannten Schwächen wäre es nötig, nun anhand anderer Software zu überprüfen, ob man zu ähnlichen Ergebnissen kommt. Einige OpenSource-Projekte, zum Beispiel Mozilla oder OpenOffice.org sollten bis zu einem gewissen Grad für solche Forschungen geeignet sein, denn sie werden schon recht lange entwickelt, haben einen großen Umfang und stellen viele zur Auswertung benötigten Daten schon öffentlich zur Verfügung. Aufgrund der Informationen in Bugzilla und Bonsai sollte zum Beispiel bei Mozilla eine ähnliche Studie, wie die hier vorgestellte, möglich sein.

3 Gezogene Schlüsse

3.1 Ideen und deren Kritik

Aufbauend auf den Erkenntnissen, aber losgelöst von den eigentlichen Ergebnissen der Studie, werden am Schluss der Papiers Ideen vorgestellt, um parallele Phänomene so zu unterstützen, dass sie zu mehr Produktivität und weniger Problemen führen. So schlägt man vor, gar nicht erst zu viele Parallelversionen entstehen zu lassen, also möglichst oft die gemachten Änderungen in's Code-Verwaltungssystem zu übernehmen (*commit*). Außerdem empfiehlt man, ein Modell der "*Feature Ownership*", wo bestimmte Entwickler für ein bestimmtes, klar definiertes Feature zuständig sind, gegenüber der "*Code Ownership*", wo bestimmte Entwickler für ein bestimmtes Stück Code verantwortlich sind, vorzuziehen. Wenn man sich Bugzilla ansieht wird man feststellen, dass bei Mozilla bereits nach dieser Devise gearbeitet wird, hier gibt es sowas wie eine "*Bug-Ownership*".

Außerdem wird darauf hingewiesen, dass bei den Merging-Tools, die parallel entwickelte Versionen wieder zu einer Version zusammenschmelzen, noch Entwicklungsbedarf ist. Diese sollten in Zukunft besser dazu beitragen, dass ein Entwickler bei sich überlappenden Änderungen nicht nur weiß, sondern auch versteht, was der jeweils andere Programmierer getan hat. Hier muss man einen Kompromiss finden, dass soviel automatisch gemergt wird, wie möglich ist, aber auch überall manuell gemergt werden muss, wo dies nötig ist. Dazu muss man,

denk ich, anmerken, dass in Systemen wie CVS das Merging schon recht gut funktioniert, so weit es sich physisch überlappende Änderungen betrifft. In dieser Beziehung sind die heutigen Systeme schon recht weit fortgeschritten. Leider ist es noch nicht möglich, sich semantisch überlappende Änderungen zu erkennen. Das zu ermöglichen dürfte eine interessante, aber auch sehr schwierige Aufgabe für die Zukunft sein. Hilfreich hierbei könnten Systeme sein, die Änderungshäufigkeiten grafisch darstellen, so wie sie in der Projektarbeit zum Thema "Visualisierung" vorgestellt wurden. Zusätzlich zur Verbesserung dieser Merging-Tools könnte eine gezielte - möglicherweise automatische - Anwendung von Refactoring Patterns, die auch in einem Seminarbeitrag vorgestellt wurden, den Entwicklern helfen, fremden Code besser zu verstehen.

Schließlich schlägt die Studie noch vor, die Koordination an parallele Phänomene anzupassen, so dass erstens die Parallelität kostengünstig unterstützt werden und zweitens die einzelnen Entwickler wissen, an wen sie sich wenden müssen, wenn sie ein ganz bestimmtes Problem feststellen. Bezüglich des ersten Punkten könnte man sich fragen, inwieweit von zu Hause und wann von Büros aus gearbeitet werden kann, und wie oft man sich zu Koordinationsgesprächen treffen sollte. Außerdem stellt sich hier auch die Frage nach der zu verwendenden Software für die Projektverwaltung. Zum zweiten Punkt kann man sagen, dass in heutigen Projekten eine solche Koordination schon in gewisser Weise existiert. Sowohl bei OpenOffice.org als auch bei Mozilla ist die Software in so genannte Komponenten unterteilt, für die jeweils bestimmte Entwickler zuständig sind und deren Entwicklung von einer bestimmten Person geleitet wird. Hier besteht also zusätzlich zu den oben genannten Modellen noch eine "*Component Ownership*". Außerdem kann die Koordination natürlich durch Mailinglisten und Newsforen verbessert werden. Aber trotz allem gibt es hier sicherlich noch Optimierungsbedarf.

3.2 Fazit und Schlusswort

Die hier besprochene Studie stellt sehr interessante Ergebnisse und einige sinnvolle Ideen vor. Sie ist vor allem dadurch interessant, dass sie die Auswirkungen paralleler Entwicklungen an einem tatsächlich existierenden System zeigt und damit gute Rückschlüsse für die Praxis liefert. Trotz einiger kleiner Schwächen in der Art der Datensammlung und Argumentation, wie sie zu den Ergebnissen kommt, ist sie ein guter Ausgangspunkt für Nachfolgestudien und liefert gute Denkanstöße, die man bei der Arbeit an großen Projekten mit hohem Ausmaß an parallelen Aktivitäten beachten sollte.