Seminararbeit

Softwareentwicklung in der Wissenschaft

Steffen Göttsch

Ausarbeitung zum Vortrag vom 9.12.2010

Inhalt

Einleitung

Motivation

Im Studium haben die Zuhörer meines Vortrags bereits einen guten Überblick über Prinzipien guter Softwareentwicklung erhalten. Allerdings sind diese Prinzipien zu einem Großteil auf kommerzielle Programme ausgerichtet und damit einhergehend sind bestimmte Prämissen angenommen. Dazu gehören z.B. ein langer Softwarelebenszyklus, normalerweise von mehreren Jahren. Die Entwicklung von Programmen wird in einem großen Team vorgenommen, das Endprogramm ist auf eine kommerzielle Nutzung ausgerichtet.
Auf diesen Umständen fußen die Prinzipien, die für die meisten Programmierteams sehr sinnvoll sind und auch nötig. Auf die Softwareentwicklung in der Wissenschaft lassen sich diese Prinzipien nicht ohne weiteres anwenden und/oder sind nicht immer sinnvoll, da die Gegebenheiten hier teilweise deutlich anders sind.
In dieser Arbeit möchte ich diese Unterschiede herausarbeiten und aufzeigen, wie die Praxis in der Softwareentwicklung in der Wissenschaft aussieht und warum sie so aussieht. Diese Praxis ist teilweise sinnvoll so wie sie ist, teilweise aber nicht optimal und mehr als Relikt zu sehen, das in diese Richtung gewachsen ist und sich so gehalten hat. In dieser Arbeit untersuche ich auch, wo die Praxis sinvoll geändert werden kann, welche Ansätze es gibt, die Softwareentwicklung in der Wissenschaft effizienter zu machen, und welche Schwierigkeiten dabei auftreten.

Arbeitsweise

Mein Vortrag und diese Ausarbeitung ist das Ergebnis einer Recherchearbeit. Der größte Teil ist die Essenz dessen, was ich aus verschiedenen Quellen (siehe Quellenteil) zusammengetragen habe. Hier gab es einige Überschneidungen, ich habe versucht, das Gesamtbild treffend wiederzugeben. Zusätzlich habe ich als Einschub eine Konferenz mit Vertretern aus Softwareentwicklung und Wissenschaft wiedergegeben, die dem Gesamtbild einige konkrete Züge gibt.

Als wichtigste Quellen sehe ich die Paper „Models of Scientific Software Development“, „Scientific Software Engeneering – Basic Techniques of creating practical scientific Software“ und „Post-Workshop report for the Third International Workshop on Software Engineering for High Performance Computing Applications (SE-HPC 07)“. Konkrete Refernzen zu den einzelnen Kapiteln sind schwer zu definieren, da das meiste eine Zusammenfassung der Informationen aus diesen Papern ist. Auch eigene Überlegungen und Querverbindungen zwischen den einzelnen Quellen flossen in meine Arbeit ein.

Prinzipien guter Softwareentwicklung

Zunächst ein kleiner Überblick über Prinzipien guter Softwareentwicklung im klassischen Sinne. Diese Prinzipien sollen das Programmieren und die anschliessende Nutzung sowie Wartung und Erweiterung/Änderung effizienter machen. Effizienz hat hier im wesentlichen zwei Anteile: Zum einen Schnelligkeit, also möglichst schnell das Programm zu Erstellen bzw. zu Verändern, und Qualität, also ein möglichst gutes Programm zu schreiben. Die beiden Ziele, die zunächst gegenläufig klingen, sind teilweise einander fördernd, zumindest für das Umfeld größerer Programme. Eine gute interne Qualität des Programms bedeutet, dass mit weniger Zeitaufwand Änderungen vorgenommen werden können.

Ziele in diesem Sinne sind z.B.:
- Lesbarkeit
- Wiederverwendbarkeit
- Wartbarkeit
- Erweiterbarkeit
- Fexibilität
- Einfache Verwendbarkeit
- Änderungsstabilität

Methoden um dies zu erreichen sind z.B.:
- Klar strukturierter Code
- Eindeutige Bezeichner
- Kommentare
- Kapselung
- Eine Klasse – Eine Aufgabe
- Tests

Auf die einzelnen Punkte gehe ich hier nicht näher ein, da der Zielgruppe bereits alles bekannt sein dürfte. Zusammenfassend sieht man bei dieser Übersicht, dass die meisten Prinzipien darauf abzielen, dass der Code von anderen Programmierern schnell verstanden werden kann und dass Änderungen/Erweiterungen problemlos vorgenommen werden können. Dies ist für eine Softwareentwicklung im kommerziellen Umfeld sehr wichtig, aber wie ich im folgenden zeigen werde, auf Softwareentwicklung in der Wissenschaft nicht ohne weiteres übertragbar.

Praxis in der Softwareentwicklung in der Wissenschaft

„Die“ Praxis in der Softwareentwicklung in der Wissenschaft gibt es so nicht, da es ein ziemlich heterogenes Feld ist. Es gibt Projekte, die durchaus große Ähnlichkeiten mit klassischer kommerzieller Softwareentwicklung haben. Selbst bei diesen ist allerdings nicht alles eins zu eins übertragbar, was man für eine Softwareentwicklung normalerweise als richtig, gut, notwendig, hilfreich ansieht. Um die Unterschiede deutlich herauszuarbeiten, beschreibe ich hier, was man als extremes Gegenstück zur klassischen Softwareentwicklung ansehen kann und was durchaus einen guten Teil der Softwareentwicklung in der Wissenschaft ausmacht.

Ein solches Softwareentwicklungsprojekt wird von einem einzelnen Wissenschaftler durchgeführt. Das Ziel ist es, ein Programm zu schreiben, dass genau eine Aufgabe ausführt, nämlich eine bestimmte Berechnung durchzuführen. Das angestrebte Ergebnis ist also nicht die Software selbst, sondern am Ende die Berechnung, um z.B. eine bestimmte These zu verifizieren. Dies ist der alleinige Zweck der Software, nachdem sie diesen Zweck erfüllt hat, wird sie nicht mehr benötigt und auch nicht mehr verwendet oder verändert. Die Software wird nur von dem Wissenschaftler, der sie programmiert hat, verwendet, nicht von Dritten. Der Wissenschaftler hat keine gute Ausbildung in der Softwarentwicklung genossen, sein Ausbildungsschwerpunkt ist ganz klar die Wissenschaft. Das Ergebnis der Berechnung ist anfangs nicht bekannt, es wird bestenfalls ungefähr vermutet. Die Berechnung findet auf einer sehr großen Datenmenge statt, z.B. Klimadaten von vielen Sensoren über einen langen Zeitraum.

Da das Programm nur von dem einen Wissenschaftler verwendet wird, macht er sich keine Gedanken darüber, ob andere seinen Programmcode verstehen können. Es muss nur für ihn selbst verständlich sein, da er alles selbst programmiert und das Programm vom Umfang her sehr überschaubar ist, ist das kein Problem für ihn. Weil das Programm nur einmal verwendet wird, wird die Struktur nicht auf Wiederverwendung oder Änderung hin optimiert, nur für diese eine Verwendung soll sie das richtige Ergebnis liefern. Auch eine einfache Verwendbarkeit durch Dritte ist kein Thema, da es nur von ihm selbst verwendet wird. Da das Ergebnis der Berechnung anfangs noch nicht feststeht, kann man die Korrektheit des Programms nicht am Ergebnis ablesen. Große Teile seines Programms können nicht vernünftig getestet werden, er muss sich hier darauf verlassen, dass er es richtig gemacht hat. Wegen der großen Datenmenge sind sehr viele Berechnungen nötig, er ist also auf Hochleistungsrechner angewiesen und mietet teure Rechenzeit von einem Hochleistungsrechenzentrum. Falls andere Wissenschaftler die gleichen Untersuchungen vor ihm veröffentlichen, sind seine Berechnungen nichts mehr wert. Dadurch hat er einen gewissen Zeitdruck und möchte sein Programm möglichst schnell fertigstellen und die Berechnungen durchführen, um seine Forschungsergebnisse zu veröffentlichen.

Man sieht an dem Beschriebenen einige Unterschiede zu einer Softwareentwicklung in einer kommerziellen Softwareentwicklungsfirma. Gute Lesbarkeit, Änderbarkeit, Wartbarkeit, einfache Verwendbarkeit spielen praktisch keine Rolle. Das kann sich etwas verschieben, je größer das Team wird, die Tendenz bei wissenschaftlicher Softwareentwicklung sind aber eher kleinere Teams. Im Folgenden werde ich auf einige der Punkte näher eingehen.

Lebenszyklus

Der Lebenszyklus einer kommerziellen Software beträgt normalerweise mehrere Jahre. Im Fall eines einzeln stehenden Programms, beispielsweise einem Computerspiel, wird das Programm zumindest ein bis zwei Jahre vertrieben, genutzt und gewartet. Bei einem großen Anwendungsprogramm wie beispielsweise Microsoft Word wird das Programm über einen wesentlich längeren Zeitraum weiterentwickelt und in Form von neuen Versionen vertrieben, wobei der ursprüngliche Programmcode immer weiter verändert und erweitert wird.

In der Softwareentwicklung in der Wissenschaft gibt es häufig einen wesentlich kürzeren Lebenszyklus. In vielen Fällen wird das Programm tatsächlich nur ein mal ausgeführt, um eine einzelne Berechnung durchzuführen. Bei solch einem Programm ist es teilweise sogar so, dass der Programmcode danach nicht mehr wiederverwendet wird, obwohl er in Teilen auch für ähnliche Projekte verwendet werden könnte. Natürlich gibt es hier eine große Bandbreite, große Klimamodellen beispielsweise werden ähnlich stark wiederverwendet wie kommerzielle Programme.

Risiken in der Softwareentwicklung in der Wissenschaft

Ein wesentlicher Unterschied von wissenschaftlichen Programmen zu kommerziellen Programmen ist, dass das Ergebnis des Projekts am Anfang noch nicht feststeht, denn man schreibt das Programm ja gerade, um etwas herauszufinden, was entweder noch völlig unbekannt ist oder aber nur vermutet wird. Das birgt das Risiko in sich, dass das Projekt völlig fehlschlägt, weil man kein brauchbares Ergebnis kriegt. Zum Beispiel kann es sein, dass man eine Hypothese hat, die man verifizieren will, die Ergebnisse aber nicht mit der Vermutung übereinstimmen, oder aber in zu geringem Maß, dass man sie nicht als Beweis für die These nutzen kann. Eine andere Möglichkeit des Fehlschlagens ist, dass man versucht ein Muster bei bestimmten Vorgängen festzustellen, die Ergebnisse aber so stark variieren, dass sich daraus kein Muster bilden lässt.
Gleichzeitig sind die Kosten bei Nutzung einer wissenschaftlichen Software häufig hoch wegen der großen Datenmengen, die die Nutzung von großen High-Performance-Rechneranlagen nötig macht.
Ein weiterer Punkt, der die Verifizierung eines Wissenschaftlichen Programms erschwert, ist, dass der Black-Box-Ansatz - Richtige Eingabe + richtiges Ergebnis = korrektes Programm - nicht greifen kann, weil das Ergebnis erst nach der Berechnung feststeht.

Geldgeber

Bei einer kommerziellen Softwareentwicklung läuft die Finanzierung der Entwicklungskosten über Verkaufseinnahmen, entweder einmalig, wenn das Programm für einen einzelnen Kunden geschrieben wird, oder wiederkehrend, wenn das Programm öffentlich vertrieben wird. Bei wissenschaftlicher Softwareentwicklung sind die Geldgeber staatliche Gelder, die für die Forschung vorgesehen sind, oder Gelder von Stiftungen.

Testen

Das Testen eines wissenschaftlichen Programms ist schwerer als bei einem „normalen“ Anwendungsprogramm. Ein herkömmlicher Weg, ein Programm zu testen, ist das Soll-Ergebnis mit dem tatsächlichen Ergebnis zu vergleichen. Wenn beides übereinstimmt, kann man davon ausgehen, dass auch das Programm korrekt ist. Bei einem wissenschaftlichen Programm ist es aber so, dass das Ergebnis durch die Berechnung des Programms erst gewonnen wird. Man muss sich also in gewissem Maße darauf verlassen, dass das Programm das richtige tut. Ein Testen auf diese Art ist nur in Teilen möglich, bei einzelnen Unterprogrammen, wo man genau weiss was sie machen sollen. Zudem ist ein Anwendungstest auch aus dem Grund kaum realisierbar, dass die Rechnerzeit lange und teuer ist wegen der großen Datenmengen und vielen Rechenschritte. Man kann es sich kaum leisten, ein Programm, das evtl. das richtige tun könnte, „auf Verdacht“ durchlaufen zu lassen und zu schauen, ob die Ergebnisse sinnvoll sind.

Ein weiteres Risiko eines wissenschaftlichen Programms ist, wenn man ein Ergebnis bekommt, was in etwa dem Erwarteten entspricht, automatisch davon auszugehen, dass auch das Programm korrekt sein muss. Denn das Zusammentreffen von erwartetem und tatsächlichem Ergebnis kann auch durch Zufall entstanden sein, oder es werden kleine Unsauberheiten, die tatsächlich Programmfehler sind, fälschlich als „natürliches Rauschen“ interpretiert.

Ansätze zum Testen eines wissenschaftlichen Programms sind zum einen Testen von Teilprogrammen, wo man das Ergebnis kennt. Hier kann ein Testen quasi wie nach klassischen Programmiermethoden ablaufen. Die andere Möglichkeit, die, auch wenn sie fast trivial klingt, enorm wichtig ist, ist, den Programmcode mit besonderer Vorsicht anzuschauen. Damit meine ich, dass man sich während und nach dem Schreiben des Programms den Code noch einmal anschaut und überlegt, ob er tatsächlich das richtige berechnet. Hier kann man auch Kollegen einen Blick darauf werfen lassen, dem ohne den Tunnelblick des Schreibers selbst vielleicht noch etwas auffällt.

Programmiermethoden

In diesem Abschnitt stelle ich zunächst die klassische „saubere“ Softwareentwicklung vor, anschliessend als Gegenpol die Quick and Dirty „Methode“.

In der klassischen Softwareentwicklung gibt es einen klar festgelegten Entwicklungsprozess, der die Programmierung in einzelne Reifestadien unterteilt und Arbeitsschritte von vornherein festlegt. Der Programmcode ist klar strukturiert, ausführlich kommentiert und getestet. Er ist also entsprechend der Prinzipien guter Softwareentwicklung aufgebaut, auf Wiederverwendbarkeit, Verständerbarkeit und leichte Verwendbarkeit optimiert.
Nach so einem Schema geschriebener Code ist für die Arbeit in einem großen Team perfekt geeignet. Andere Programmierer können sich schnell in den Code einlesen, ihn mit wenig Aufwand und ohne Gefahr etwas kaputtzumachen ändern können und das Programm einfach bedienen können.
Ein Nachteil hierbei ist natürlich der größere Zeitaufwand, der für das Programmieren aufgewandt werden muss. In einem großen Projekt rentiert sich dieser anfänglich größere Aufwand häufig schnell, da die spätere Arbeit an dem Programm erheblich vereinfacht wird. Für kleine Projekte, wie sie in der Wissenschaft häufig vorkommen, kann dieser zusätzliche Aufwand unnötig sein.

Eine Quick and Dirty Programmierung wirft diese Prinzipien über Bord und lässt sich als „einfach drauf los Programmieren“ beschreiben. Es gibt keinen vorher festgelegten Entwicklungsprozess, sondern es wird einfach irgendwo angefangen und weitergemacht bis das Programm fertig ist. Auf Testen und Kommentiren wird größtenteils verzichtet, um Zeit zu sparen. Es wird kein Wert gelegt auf Wiederverwendbarkeit, leichte Benutzbarkeit und Veränderbarkeit.
Was zunächst wie schlampige, schlechte Programmierung aussieht, hat in manchen Projekten durchaus seine Berechtigung. Wenn das Programm tatsächlich sehr klein ist und nur einmal vom programmierenden Wissenschaftler selbst ausgeführt wird, ist es ein effektiver Weg, um Zeit bei der Programmierung einzusparen. Sobald das Programm und das Programmierteam allerdings größer ist, liegen in der vermeintlichen Zeitersparnis einige Fallstricke. Eine spätere Veränderung oder Erweiterung wird deutlich schwerer, ein anderer Programmierer hat Schwierigkeiten, sich in den Code einzulesen. Auch ist ein solches Programm wegen der schwereren Verständlichkeit deutlich fehleranfälliger. Wenn nicht speziell darauf geachtet wird, kann auch die Rechnerzeitausnutzung schlechter sein, so dass sich die eingesparte Programmierzeit bei der Ausführung des Programms in mehr Zeit- und Kostenaufwand für die Berechnung niederschlägt.

Jetzt stellt sich die Frage, welche Programmierungsmethode für eine Softwareentwicklung in der Wissenschaft angemessen ist. Hier kann man nach Größe des Projekts, Größe des Teams und geplanter Wiederverwendung abwägen, wie stark man sich nach der einen oder anderen Methode richtet. Bei wirklich kleinen Projekten macht die zweite Art der Programmierung durchaus Sinn, je größer das Projekt ist, desto mehr Sinn macht eine Beachtung der klassischen Programmierparadigmen. Was in der Praxis eine „gute“ Programmierung häufig erschwert, ist die Vorbildung der Programmierer. Ihr Ausbildungsschwerpunkt ist die Wissenschaft, eine Ausrichtung an guten Programmiermethoden scheitert teilweise an mangelnder Softwareentwicklungs-Fachkenntnis.

Zusammenarbeit Sofwareentwicklung und Wissenschaft

Die Entwicklung in der wissenschaftlichen Softwareentwicklung ging und geht weiterhin in die Richtung, dass die Projekte immer größer werden. Damit einhergehend werden die Entwicklungsteams größer, die Entwicklungsdauer wird länger. Die Wiederverwendung von Code nimmt immer größere Ausmaße an, was unter anderem eine Folge aus der Größe der Projekte ist. Je größer der Entwicklungsaufwand ist, desto wertvoller ist es, wenn man diesen Aufwand nicht mehrfach betreiben muss, wenn man es sich sparen kann. Open Source ist ein weiterer Grund für die Wiederverwendung, hier wird der Code nicht nur vom eigenene Team wiederverwendet, sondern auch von anderen Wissenschaftlern weltweit.

Aufgrund dieser Entwicklung hat die Wissenschaft erkannt, dass es nötig ist, eine gute Softwareentwicklung zu betreiben, die in einem größeren Kontext bestehen kann. Dies beinhaltet eine bessere Ausbildung der Wissenschaftler in der Softwareentwicklung, aber auch die Zuhilfenahme von Softwareentwicklern, als Berater oder als Programmierer.
Leider lässt sich die klassische Softwareentwicklung nicht vollständig und unverändert auf wissenschaftliche Projekte übertragen, da die Gegebenheiten andere sind. Die Akzeptanz von Softwareentwicklungsmethoden fällt vielen alteingesessenen Wissenschaftlern schwer, teils aus Gewohnheit, teils weil wissenschaftliche Software sich immer noch stark von kommerzieller unterscheidet. Als ein Punkt sei hier der Lebenszyklus in einem größeren Projekt genannt. Bei kommerzieller Software kann man relativ einfach einen Entwicklungsplan von Anfang bis Ende festlegen. Bei einer wissenschaftlichen Software verändert sich dieser zwangsläufig, wenn neue Erkenntnisse gewonnen werden.

Da die Notwendigkeit einer Zusammenarbeit erkannt wurde, wurden verschiedene Anstrengungen unternommen, die beiden Welten näher zusammen zu bringen. Als ein Beispiel dieser Anstrengungen und um einige Aspekte dieser Zusammenarbeit zu illustrieren, stelle ich im folgenden eine Konferenz zu dem Thema vor.

Third international Workshop on Software Engeneering for High Performance Computing Applications, 26.5.2007

Diese Konferenz fand als Teil der „International Conference on Software Engineering“ in Minneapolis, MN, USA statt. Es waren Teilnehmer aus der Forschung und aus der High Performance Computing (im folgenden „HPC“ abgekürzt) Community sowie Software Engeneering Forscher anwesend. Die Konferenz war in drei Teile unterteilt. Zunächst gab es einige Vorträge, dann eine gemeinsame Diskussionsrunde, anschliessend gab es zwei Breakout Groups, eine mit Teilnehmern aus der Softwareentwicklung, eine mit Teilnehmern aus dem HPC Bereich. Innerhalb dieser beiden Gruppen wurden Fragestellungen behandelt, die anschliessend allen Teilnehmern vorgestellt wurden.

Ein großer Teil der Vorträge beschäftigte sich mit der Produktivitätsmessung von HPC Anwendungen. Ein sehr wichtiges Thema für die wissenschaftliche Softwareentwicklung, da die benötigte Rechenleistung enorm groß ist. Ein Vortrag stellte ein Tool vor, das Support bei der Entwicklung von HPC-Anwendungen geben kann, indem es eine grafische Darstellung des Codes liefert. Ein weiterer Vortrag beschrieb die Vorteile einer integrierten Entwicklungsumgebung für HPC-Programmierung, anstatt eine Vielzahl einzelner Tools zu benutzen. Der Vortrag mit dem, wie ich fand, interessantesten Thema beschrieb das „Trillion Lifecycle Model“, ein Ansatz, um die Lebensdauer von HPC-Code zu verlängern. Er fußte auf der Beobachtung, dass, obwohl eine Menge Code theoretisch wiederverwendet werden könnte, dies tatsächlich sehr selten passiert. Das vorgeschlagene Lebenszyklusmodell managt verschiedene Phasen der Reife einer Software. Am Anfang des Lebenszyklus steht eine Forschungsphase, die in der klassischen Softwareentwicklung so kein Äquvalent hat.

Die anschliessende Diskussionsrunde versuchte, die Schwierigkeiten der Zusammenarbeit von Softwareentwicklern und Wissenschaftlern zu untersuchen und Lösungsansätze für eine bessere Zusammenarbet zu finden. Am Anfang der Runde waren die Unterschiede zwischen einem Forschungsplan und einem Geschäftsplan das Thema. Hier wurden als wichtige Punkte die Flexibilität und die häufigen Änderungen während des Entwicklungsprozess einer wissenschaftlichen Software genannt. Dies steht im Gegensatz zu kommerzieller Software, wo die Anforderungen an die Software von Anfang an feststehen, was ein simples, relativ starres Lebenszyklus-Modell ermöglicht. Ein weiterer Punkt ist das Risiko bei wissenschaftlicher Softwareentwicklung, dass das Projekt komplett fehlschlägt, was es so bei kommerziellen Programmen nicht gibt (d.h. Wenn man eine Software schreibt, die richtig und den Anforderungen entsprechend arbeitet, ist das Projekt bei einem kommerziellen Programm erfolgreich, bei einem wissenschaftlichen Programm nicht zwingend).
Die Diskussion ging dann in Richtung der persönlichen Unterschiede zwischen den Entwicklern beider Richtungen. Ein Punkt war hier die häufig fehlenden tiefergehenden IT-Kentnisse der Wissenschaftler, die die Zusammenarbeit erschweren. Ein anderer war der Unterschied, dass in der Wissenschaft der Programmierer auch gleichzeitig Nutzer ist, in der kommerziellen Softwareentwicklung aber normalerweise für Dritte entwickelt wird.
Dieser Punkt war auch der Übergang zu der Suche nach Gemeinsamkeiten zwischen kommerzieller und wissenschaftlicher Softwareentwicklung. Firmeninterne Projekte sind in zwei Punkten ähnlich den wissenschaftlichen, nämlich dass die Programmierer auch gleichzeitig Nutzer des Programms sind und dass das Programm häufig während der Entwicklung geändert wird.

Anschliessend setzten sich die Vertreter der beiden Richtungen getrennt zusammen und erörterten einige Fragen. Bei den Vertretern aus dem HPC Bereich war die erste Frage, welche Softwareentwicklungs-Techniken in der Vergangenheit für den HPC Bereich bereits gut funktioniert haben. Die agilen Entwicklungsmethoden wurden als am ehesten passend empfunden, allerdings nicht als Ganzes. Die Wissenschaftler neigen dazu, diese Techniken wie ein Buffet zu betrachten, wo sie das nehmen was am besten für sie passt und den Rest beiseite lassen. Pair Programming wurde z.B. als hilfreich für die Einarbeitung von neuen Leuten empfunden, aber als dauerhafte Entwicklungsmethode abgelehnt.
Die zweite Fragestellung war, welche Dinge die HPC Community nicht von Softwareentwicklern braucht. So wurde ein Entwicklungszyklus wie in der klassischen Softwareentwicklung als nicht hilfreich empfunden, das teils sehr starre Korsett das dieser liefert, sei unpassend für HPC Anwendungen.
Die letzte Frage war, was man am meisten von Softwareentwicklungs-Forschern braucht. Hier ging das Gespräch in die Richtung, dass sehr viele Projekte ohne die Hilfe von Softwareentwicklern erfolgreich waren. Gleichzeitig stellte man aber fest, dass der Entwicklungsprozess durch die Ablehnung von Softwareentwicklungs-Techniken behindert wird und eine Nutzung von Softwareentwicklungsprinzipien hilfreich wäre, nur eben nicht unverändert die Prinzipien bei kommerzieller Softwareentwicklung übernommen werden können. Eine nützliche Sache wäre der Support durch Tools.

Die Breakout Gruppe der Softwareentwickler behandelte als erste Frage, was die wichtigsten/besten Dinge sind, die die Softwareentwicklungs-Community der HPC Community anbieten kann. Es wurde als wichtig empfunden, Erfolge, die in der Vergangenheit in der Zusammenarbeit zwischen Softwareentwicklern und HPC-Leuten erzielt wurden, auch zu kommunizieren, um die Akzeptanz von Softwareentwicklern als wertvolle Hilfe für den HPC Bereich zu erhöhen. Was häufig erfolgreich verlaufen war, war die Vermittlung von effektiven, simplen Methoden, die leicht zu implementieren sind.
Die nächste Frage war, welche Probleme oder Frustrationen es bei der Zusammenarbeit mit der HPC Community oder dem Forschungsbereich gab. Ein Punkt war hier die Einstellung der Forscher zu ihrer Software. Diese sehen die Software nur als Mittel zur Berechnung, um ihr wissenschaftliches Paper zu veröffentlichen, anstatt die erstellte Software selbst auch als wertvolles Gut anzusehen. Mit dieser Einstellung geht eine geringe Wiederverwendung des Codes einher. Auch kamen Klagen auf, dass die Zusammenarbeit mit Softwareentwicklern teils eher als Bürde denn als Hilfe angesehen wurde. Hier besteht eine Menge Kommunikationsbedarf, um eine Annäherung und bessere Zusammenarbeit zu ermöglichen.
Die letzte Frage war, welche Dinge die Softwareentwickler der HPC Community gerne anbieten würde, aber momentan noch nicht kann. Ein Punkt war ein Lebenszyklusmodell, dass für wissenschaftliche Softwareentwicklung passend ist. Weiterhin mussten sich die Softwareentwickler eingestehen, dass ihr Verständnis von HPC zu gering ist, um optimal zusammenarbeiten zu können. Ausserdem müsse man noch mehr Erfahrung sammeln, welche Techniken und Herangehensweisen für den HPC Bereich funktionieren und welche nicht.

Als Zusammenfassung und Ausblick in die Zukunft wurden zwei Ziele festgehalten. Ein Ziel ist, ein Lebenszyklusmodell und Toolsupport für HPC Anwendungen zu entwickeln. Das andere ist, durch Vergleich mit Bereichen mit ähnlichen Anforderungen weitere nächste Schritte und sinnvolle Methoden zu finden, diese Bereiche sind die firmeninterne Software und High-Risk-Software. Insgesamt haben die Teilnehmer aus beiden Bereichen ein besseres Gegenseitiges Verständnis gewonnen, dass die zukünftige Zusammenarbeit verbessern wird.

IT-Modelle im Licht der wissenschaftlichen Softwareentwiccklung

Open Source

Die Verwendung von Open Source in der Wissenschaft ist eine häufig genutzte Methode, die eine höhere Wiederverwendung von Code bewirken kann. Für den wissenschaftlichen Bereich ist es ein besonders sinnvoller Ansatz, da Wissen als öffentliches Gut angesehen werden kann. Die Verbreitung von bereits bestehenden wissenschaftlichen Programmen kann dazu beitragen, einen schnelleren wissenschaftlichen Fortschritt zu erreichen, da vorhandener Code weiterverwendet werden kann. Gleichzeitig erfordert Open Source aber auch, dass ein höherer Wert auf Lesbarkeit und Wiederverwendbarkeit des Codes gelegt wird, also tendenziell ein höherer Aufwand bei der Programmierung. Viele Förderprogramme schreiben inzwischen Open Source Entwicklung sogar vor zum Nutzen des gemeinsamen Fortschritts.

Beispiele einer erfolgreichen Anwendung von Open Source in der Wissenschaft sind große Klimamodelle. Eines davon ist das vom MIT entwickelte MIT General Circulation Model (MITgcm), das zur Berechnung von Wasser- und Luftzirkulationen dient. Zwei andere sind das General Esuarine Transport Model (GETM) und das General Ocean Turbulence Model (GOTM), die zur Berechnung von Meeresströmungen dienen.

Green Computing

Green Computing bezeichnet umweltschonende Computernutzung, also die Nutzung von stromsparenden Geräten sowie umweltgerechter Produktion und Entsorgung von Geräten. Man kann den Begriff aber auch noch weiter fassen, für wissenschaftliche Softwareentwicklung mit der enormen Rechenzeitnutzung kann eine Menge gespart werden, wenn man effizient programmiert und so die benötigte Rechenzeit senkt.
Für die Wissenschaft ist Green Computing vom moralischen Aspekt her sehr sinnvoll, da Wissenschaft dem Menschen dienen sollte und damit auch der Umwelt. Ein weiterer Punkt ist, dass man mit dem Etikett Green Computing auch effektiv um öffentliche Gelder werben kann.

Ausblick

Was ich am Anfang dieses Berichts aufgezeigt habe, die Entwicklung von Programmen in einem sehr kleinen Team, mit einmaliger Nutzung der Programme und schlechter Softwareentwicklungs-Kenntnis der Wissenschaftler, ist in der Veränderung begriffen. Zwar gibt es solche Projekte immer noch und sie werden wahrscheinlich noch lange einen guten Teil der wissenschaftlichen Softwareentwicklung ausmachen, doch es gibt eine anhaltende Tendenz dazu, dass wissenschaftliche Softwareentwicklung in einem immer größeren Rahmen stattfindet.
Die Programmierteams werden größer, die Entwicklungsdauer länger, auch der Lebenszyklus von Programmcode verändert sich – unter anderem durch die Nutzung von Open Source, aber auch innerhalb eines einzigen Programmierteams – und wird immer länger, Code wird wesentlich öfter wiederverwendet. Die Projektteams werden teils auch heterogener, es werden Softwareentwicklungsexperten hinzugezogen und die Akzeptanz wie auch das Fachwissen von Prinzipien Softwareentwicklung wächst.

Diese Entwicklung ist noch lange nicht abgeschlossen, es gibt noch keine perfekten Softwareentwicklungsmethoden für wissenschaftliche Software und es gibt immer noch Bereiche, wo man mit einer Softwareentwicklung, die recht nahe am „schändlichen“ Quick and Dirty Prinzip arbeitet, am besten fährt, aber der Weg ist eingeschlagen und die Strassen werden immer breiter ausgetreten.

Quellen

Post-Workshop report for the Third International Workshop on Software Engineering for High Performance Computing Applications (SE-HPC 07)

http://www.cse.msstate.edu/~carver/Papers/SEN_32_5.pdf

Report from the Second International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE 09)

Scientific Software Engineering - Basic techniques of creating practical scientific softwarehttp://www.ita.uni-heidelberg.de/~pmelchior/talks/software_engineering_150410.pdf

Models of scientific software development

http://cs.ua.edu/~SECSE08/Papers/Segal.pdf

Klassische Fehler in der Softwareentwicklung

http://www.theoinf.tu-ilmenau.de/~riebisch/swqs/fehler.html

http://mitgcm.org/

http://www.getm.eu/

http://www.gotm.net/