Einblicke ins Testen und die testgetriebene Entwicklung in wissenschaftlicher Softwareentwicklung

— Ausarbeitung zum Vortrag —

im Seminar „Softwareentwicklung in der Wissenschaft“ SoSe 2015

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Marek Jacob
E-Mail-Adresse:	marek.jacob@studium.uni-hamburg.de
Matrikelnummer:	6212488
Studiengang:	Meteorologie
Betreuer:	Dr. Hermann Lenhart
Dozenten:	Prof. Dr. Thomas Ludwig, Dr. Hermann Lenhart und Dr. Julian Kunkel

Hamburg, den 21.09.2015

Inhaltsverzeichnis

Listings

1 Einleitung

Softwareentwicklung in der Wissenschaft zielt auf das Produzieren von Ergebnissen. Diese sollen das wissenschaftliche Verständnis erhöhen. Wissenschaftliche Software ist die Implementation einer modellhaften Vorstellung der Realität. Der Wissenschaftler nutzt den Computer, um rechenintensive Aufgaben in einem Modell, das er vorgibt, zu lösen. In dem Prozess der Wissensgewinnung erweitern neue Erkenntnisse das wissenschaftliche Modell und stellen damit neue Ansprüche an die Aufgaben der Software. Dadurch entsteht eine inkrementelle Entwicklungsweise. Die Entwicklung numerischer Klimamodelle beispielsweise ist von Software dominiert, die von Wissenschaftlern der Fachdomäne geschrieben wurde und weniger von ausgebildeten Softwareentwicklern. [1] Daher hat die Entwicklung möglichst umfassender und leicht wiederverwendbarer Software bei den Wissenschaftlern meist untergeordnete Priorität.

Die implizit inkrementelle Softwareentwicklung in der Wissenschaft weist Analogien zu dem Konzept der testgetriebenen Entwicklung, kurz *TDD* (engl. *test-driven development*) auf. Der Kern der testgetriebenen Entwicklung liegt in dem Erstellen von Tests vor der Implementation der zu testenden Komponente. Dies definiert die Idee und die erwartete Funktionalität einer Komponente. Die Komponenten sollen möglichst klein sein. Nach Erfüllung aller zuvor gestellten Anforderungen folgt der Schritt des Aufräumens (engl. *refactoring*) des Programmcodes. Dieses Verfahren erlaubt und erzwingt ein iteratives Vorgehen aus erstens Idee entwickeln und Test aufstellen, zweitens Test erfüllen und drittens Refactoring [2]. Im Rahmen dieser Ausarbeitung wird der Stand von TDD in wissenschaftlicher Softwareentwicklung untersucht. Die Grundlage bilden drei Veröffentlichungen: „Software Testing and Verification in Climate Model Development“ von Clune und Rood [1], „Building CLiiME via Test-Driven Development: A Case Study“ von Nanthaamornphong et al. [3] und „Towards Test Driven Development for Computational Science with pFUnit“ von Rilee und Clune [4]. Diese Veröffentlichungen legen den Fokus auf Softwareentwicklung in der Klimaforschung und im Hochleistungsrechnen. Daher beziehen sich die meisten Aussagen dieser Ausarbeitung ebenfalls auf diese Themenbereiche. Besonderheiten von Softwareentwicklung in anderen Fachrichtungen werden dabei außen vor gelassen.

2 Arten von Tests

Es gibt verschiedene Klassifizierungen, um Softwaretests zu unterschieden. In diesem Abschnitt werden dazu verschiedene Begriffe festgelegt.

2.1 Validation und Verifikation

Bei dem Testen wissenschaftlicher Modelle sollten zwei Begriffe unterschieden werden: Die *Validation* eines Modells ist der Vergleich von Ergebnissen eines Modells mit Beobachtungen und Messungen. Die *Verifikation* hingegen ist ein Vergleich mit theoretischen Ergebnissen. Bei numerischen Modellen können dies Lösungen von einfachen Startparametern sein, zu denen eine analytische Lösung bekannt ist. Alternativ werden auch Ergebnisse von anderen, allgemein vertrauten Modellen zum Vergleich herangezogen. Ein dritter Begriff ist die *Softwareverifikation*.

Diese beinhaltet die Überprüfung, ob ein Implementation genau dem Wissenstand und den Vorstellungen des Wissenschaftlers entspricht. Die Softwareverifikation als konkrete Problemstellung wird von wissenschaftlichen Softwareentwicklern oft verkannt. Eine korrekte Implementation der eigenen Idee wird folglich implizit angenommen. [4]

2.2 Klassifikation von Tests nach Detailgrad

Die kleinsten Details können mit Unittests getestet werden. Diese filigranen Tests überprüfen einzelne implementierte Funktionen. Eine so getestete Funktion soll möglichst eine einfache und konkrete Aufgabe lösen. Zu dieser müssen dem Entwickler zu verschiedenen Ausgangszuständen die gewünschten Ergebnisse klar sein. Die Überprüfung der Funktionalität wird typischerweise mit *assert*-Methoden (versichern) durchgeführt. Da ein einzelner Test jeweils nur einen kleinen Teil abdeckt, benötigt seine Durchführung in der Regel nur sehr wenig Zeit.

Am anderen Ende der Detailskala stehen Tests auf Systemebene. Bei einem derartigen groben Tests wird die gesamte Software mit verschiedenen Einstellungen ausgeführt. Die Einstellungen bilden häufig konkrete Fragestellungen der Fachdomäne ab. Mit Systemtests werden verschiedene Aspekte getestet: Grundlegend wird überprüft, ob eine Software ohne unerwarteten Absturz oder Abbruch durchläuft. Bei Programmen, die parallel über mehrere Rechenkerne oder -knoten ausgeführt werden, wird überprüft, ob die Ergebnisse unabhängig von der parallelen Zerlegung sind. Es wird geprüft, ob das Anlegen und Wiederaufnehmen von Fixpunkten funktioniert. Fixpunkt-Methoden werden in Modellen verwendet, die eine hohe (z.B. $> 1\,\mathrm{h}$) Rechendauer haben. Hierbei werden mehrfach Daten gespeichert, die den gesamten Zustand des Modells nach einem bestimmten Rechenschritt abbilden. Diese werden verwendet, wenn eine Modellrechnung nach einem vorzeitigen Abbruch durch äußere Fehler oder Signale fortgesetzt werden soll.

Zwischen Unittests und Systemtests lassen sich Integrationstests einordnen. Diese überprüfen das Zusammenspiel gewisser Teile einer Software. Da sie im Verhältnis zwischen wissenschaftlicher Softwareentwicklung und testgetriebener Entwicklungsweise keine besondere Rolle einnehmen, wird auf Integrationstests in dieser Arbeit nicht vertieft eingegangen.

3 Stand der Test-Methoden in der Wissenschaft und Vorteile von Unittests

In der wissenschaftlichen Softwareentwicklung werden die im vorherigen Abschnitt unterschiedlichen Aspekte von Tests oft nicht differenziert betrachtet. Am häufigsten lassen sich verschiedene Ausprägungen von Systemtests finden. Ein Wissenschaftler testet seine Software anhand ihm bekannter Testfälle. Dies sind entweder Modellkonfigurationen, die ein idealisiertes Problem simulieren, zu dem theoretische analytische Lösungen bekannt sind oder Konfigurationen, die ein realitätsnahes Problem beschreiben, zu dem es Referenzmodelläufe oder Messdatensätze gibt. Oft werden die eigenen Simulationsergebnisse graphisch dargestellt und auf Grundlage dieser Abbildungen verglichen. Konkrete Beispiele hierzu werden in Abschnitt 5 gegeben .

3.1 Motivation für testgetriebene Softwareentwicklung in der Wissenschaft

Der iterative Charakter der testgetriebenen Softwareentwicklung passt besser als plangetriebene Verfahren wie das V- oder Wasserfallmodell zur Softwareentwicklung in der Wissenschaft. Testgetriebene Softwareentwicklung lässt neue Ansprüche an die Software leichter zu, als wenn die Software nach einem zuvor in mehreren Ebenen erarbeiteten Plan entwickelt wird. Derartig neue Ansprüche sind jedoch die Natur der Wissenschaft. Neue Erkenntnisse sollen in Modelle eingebaut werden und die Modelle sollen neue Fragestellungen untersuchbar machen. Daher ist eine wissenschaftlichen Software im reinen Produktionsbetrieb eine Seltenheit.

3.2 Vorteile von Unittests

Clune und Rood beobachten, dass sich ein Großteil der Gemeinschaft der Klimamodellentwickler den allgegenwärtigen Vorteilen von Unittests nicht bewusst ist. Außerdem hätten sie falsche Vorstellungen über das Wesen und die Schwierigkeit der Implementation von Unittests. Obwohl Unittests und Systemtests nicht in direkter Konkurrenz stehen und unterschiedliche Aspekte testen, seien hier einige „Vorteile“ von Unittests zu nennen: Unittests testen nur sehr kleine Teile einer Software mit konkreten Eingabewerten. Dies führt im Vergleich zu einem Systemtest, bei dem diese Teile der Software mitunter mehrfach ausgeführt werden, zu einer wesentlich verkürzten Ausführungszeit der Tests. Dadurch lassen sich die Tests wesentlich häufiger ausführen. Zudem versuchen Unittests Fehler dort zu entdecken, wo sie entstehen können. Während ein Systemtest nur feststellen kann, dass das Ergebnis einer komplexen Berechnung der Erwartung entspricht, zeigt ein filigraner Test direkt, in welcher Funktion es Probleme gibt. Ein enges und umfassendes Netz von Unittests, die gegebenenfalls durch Integrationstests ergänzt werden, kann weitestgehend sicherstellen, dass einmal entwickelte Features einer Software durch Änderungen der Software nicht ungewollt verloren gehen. Für die einfache Erstellung von vielen und kleinen Tests ist die Nutzung eines Test-Frameworks von großem Nutzen. Diese stellt im Idealfall die Funktionalität der Vergleiche und Überprüfungen dar und sorgt für ein systematisches Durchlaufen aller Tests sowie zu einer übersichtlichen Darstellung der Ergebnisse.

3.3 Zusätzlicher Overhead während der Entwicklung durch Unittests

Dem zusätzlichen Aufwand, den Unittests in ihrer Erstellung bedeuten, steht die Zeitersparnis beim Fehlerauffinden durch schneller gefundene Fehler entgegen. Zum einen kann ein guter Test direkt auf die Fehlerquelle hinweisen, sodass ein Fehler schneller gefunden wird. Zum anderen können Fehler, die nur bei besonderen Modellkonfigurationen in Systemtests oder gar wissenschaftlichen Modelläufen auftreten, früher gefunden und behoben werden. Dadurch sinkt das Fehlerrisiko in einem Modell und das Vertrauen in die Implementation kann gesteigert werden. Wie im vorherigen Abschnitt genannt, können Unittests sicher stellen, dass bei Änderungen keine alten Features verloren gehen. Dem gegenüber steht ein doppelter Wartungsaufwand, wenn bestehende Funktionalitäten geändert werden sollen, denn dabei müssen sowohl der Funktionscode, als auch der Test überarbeitet werden.

Ein noch größerer Wartungsaufwand entsteht, wenn Clunes und Roods Vorschlag vom Refactoring für numerische Algorithmen aufgriffen wird: Nachdem ein Algorithmus fein filigran in

mehren Einheiten implementiert ist und die Tests eine erwartungsgemäße Funktionalität versichern, soll eine zweite Version des Algorithmus entwickelt werden. Diese hat im Gegensatz zur ersten Version nicht die Verständlichkeit und Lesbarkeit als oberste Entwicklungspriorität, sondern ist auf Ausführungsgeschwindigkeit optimiert. Die richtige Funktionalität dieses beschleunigten Algorithmus lässt sich mit den Ergebnissen der langsamen Version überprüfen. Änderungen müssen ab nun jedoch an drei Stellen getätigt werden. Dafür jedoch existiert überhaupt eine verständliche Version des Algorithmus im Programmcode, denn im Betracht langer Modellausführungszeiten ist jede größere Zeitersparnis ein nicht zuletzt ökonomischer Vorteil.

4 Besonderheiten von Softwaretests in wissenschaftlicher Softwareentwicklung und Hochleistungsrechnen

In der Anwendung von Test-Methoden ergeben sich in der Softwareentwicklung in der Wissenschaft Besonderheiten. Diese zeigen sich insbesondere im Bereich von numerischen Modellen. Um diese in akzeptabler Laufzeit auszuführen, werden sie auf Großrechnern parallel ausgeführt. Die Besonderheit ist die Parallelität so wie ein großer Anteil an Numerik mit Gleitkommaarithmetik. Physikalische Simulationen wie Klimamodelle bestehen zu einem großen Teil aus diesen parallelen mathematischen Operationen. Für Software aus anderen Anwendungsbereichen spielt das Testen dieser Techniken eine geringere oder gar keine Rolle.

4.1 Parallele Ausführung

Um parallele Funktionen zu testen, wird ein Testsystem mit mehreren Knoten benötigt. Eine parallele Funktion verteilt ihre Arbeit über mehrere Rechenknoten. Für die Kommunikation zwischen Rechenknoten wird im Hochleistungsrechnen in der Regel das Message Passing Interface (MPI) verwendet. Alleine das Initialisieren der MPI Strukturen und das Finden der Knoten untereinander benötigt heutzutage immer noch einige Sekunden. Dies ist länger, als ein typischer Unittest brauchen sollte. Zudem belegt ein derartiger Test mehr Ressourcen als ein einfacher serieller Test. Fehler, die aufgrund von *Race Conditions* (Wettlaufsituation) auftreten, sind mit Tests teils schwierig zu erfassen. Der Grund liegt darin, dass diese häufig von Zuständen wie der Belegung von Bussen und Netzwerken oder der Anzahl der verwendeten Knoten und Ressourcen abhängen.

Neben dem Mehraufwand durch erhöhten Ressourcenbedarf bergen Tests von parallelen Komponenten noch eine weitere Besonderheit: Die assert-Methoden zur Überprüfung richtiger Teilergebnisse müssen ebenfalls verteilt ausgeführt werden. Für den Ergebnisbericht des Tests bedeutet dies, dass dieser auch als paralleles Programm mit Kommunikation zwischen Rechenknoten implementiert sein muss. Eine Interferenz zwischen Nachrichten, die innerhalb der zu testenden Funktion ausgetauscht werden, und Nachrichten der Testprotokollierung muss ausgeschlossen sein.

4.2 Numerische Algorithmen und Gleitkommaarithmetik

Numerische Algorithmen sind eine Verknüpfung mehrerer Rechenschritte. Für eine gute Testabdeckung bietet es sich an, einen numerischen Algorithmus in viele kleine Schritte zu zerlegen. Wird ein Algorithmus zudem objektorientiert implementiert, lassen sich die zumeist trivialen Teilschritte laut Rilee und Clune [4] gut mit Mocks testen. Mocks sind Stellvertreterobjekte, die lediglich die originale Schnittstelle nachbilden, in einem Test jedoch protokollieren, ob und mit welchen Parametern gewisse Schnittstellen abgefragt werden.

Neben der Hürde des feinkörnigen Implementieren eines Algorithmus stellt die endlich genaue Abbildung von reellen Zahlen in einem Computer eine Herausforderung für Tests dar. Reelle Zahlen werden durch Gleitkommazahlen approximiert. Daher stellen Ergebnisse einer Computerrechnung auch immer lediglich eine Approximation des mathematischen Ergebnisses dar. Dies hat zur Folge, dass Tests keine exakten Ergebnisse spezifizieren können, sondern nur mit gewisser Toleranz testen können. Die Größe des Gleitkommafehlers lässt sich theoretisch unter Betrachtung jeder einzelnen Rechenoperation und den verwendeten Zahlen berechnen. Praktisch ist dies jedoch sehr aufwendig, sodass die Toleranz meist global für alle Tests vorgegeben wird. Dabei sollte eine geeignete Kombination aus zulässiger absoluter und relativer Toleranz gewählt werden. Ist eine zu hohe Toleranz gewählt, können Fehler übersehen werden. Ist die Toleranz jedoch zu klein gewählt, so schlägt ein Test fälschlich an, obwohl die Ergebnisse im Rahmen der theoretischen Genauigkeit liegt.

Beispiel: Ein Modell wird in Form von physikalischen Gleichungen mathematisch mit kontinuierlichen Funktionen aufgestellt. Für Idealfälle sind analytische Lösungen des Gleichungssystems bekannt. Zur Lösung anderer Fälle soll dieses System mit dem Computer gelöst werden. Dazu müssen die kontinuierlichen Funktionen diskretisiert werden. Die Gleichungen müssen in auf diesen diskreten Feldern gültige, äquivalente Operationen umgewandelt werden. Nach diesen Schritten sind in den Lösungen der Idealfälle Abweichungen zu den analytischen Lösungen enthalten, die allein von der Diskretisierung stammen. Bei Integrationstests eines solchen Algorithmus gegen eine analytische Lösung muss daher zusätzlich zum Gleitkommafehler der Fehler der Diskretisierung betrachtet werden. Beide Fehlerarten akkumulieren sich im Verlauf eines längeren Algorithmus. Daher sind neben Integrationstests teils triviale Unittests zur vollständigen Testabdeckung erforderlich. Der Diskretisierungsfehler kann verkleinert werden, wenn mit einem kleineren Diskretisierungsschritt gearbeitet wird. Diese feinere Auflösung bedeutet auf der anderen Seite jedoch eine erhöhte Rechendauer des Integrationstests.

4.3 Altlastencode

Besonders in der Klimawissenschaft existieren Modellimplementationen, die mehrere Dekaden alt sind. Dies ist sogenannter Altlastencode. Dieser hat häufig die Form von monolithischen Routinen mit über 1000 Programmzeilen. Es werden ebenso globale Variablen verwendet wie `goto` Sprungmarken. Jedoch beschreibt dieser Code wichtige und komplexe physikalische Vorgänge. Eine Neuentwicklung ist sehr aufwändig. Dies macht eine Verwendung dieses alten Codes oft unvermeidbar.

Um dennoch eine Weiterentwicklung testgetrieben umzusetzen, bieten sich folgende Strategien an: Mit verschiedenen Einstellungen lassen sich Modellläufe erzeugen, mit deren Ergebnis sich testen lässt, ob Umstrukturierung des alten Codes zu veränderten Ergebnissen führt. Veränderte Ergebnisse sind meist zu Beginn unerwünscht. Wenn es möglich ist, sollte der Altlastencode in kleinere, besser testbare Teile zerlegt werden. Für diese Teile gilt es, Tests zu erstellen. Damit ist in gewissen Maßen sicher gestellt, dass nun folgende Änderungen am Code, keine unerwünschten Nebenwirkungen haben. Änderungen am Code können nun in Form von Funktionen testgetrieben entwickelt werden. Im Altlastencode wird dann nur der Aufruf der neuen Funktion eingefügt. Damit kann verhindert werden, dass die Altlast durch Erweiterungen weiter anwächst.

5 Beispiele

In diesem Abschnitt werden konkrete Projektbeispiele aus den drei Veröffentlichungen [1, 3, 4] vorgestellt. Zuerst wird die testgetriebene Neuentwicklung von CLiiME [3] vorgestellt. Anschließend werden weitere Aspekte einer Neuentwicklung und einer Neufassung aus [1] ergänzt. Den Abschluss bildet die Betrachtung der Entwicklung eines Frameworks ohne strikte Teststrategien so wie die Entwicklung eines Testframeworks für Fortran [4].

5.1 Testgetriebene Neuentwicklung von CLiiME

Das „Community Laser-Induced Incandescence Modeling Environment“ (CLiiME) simuliert mittels Laser angeregtes Glühen von Partikeln. CliiME ist das erste agil entwickelte Projekt an der Combustion Research Facility (CRF) der Sandia National Laboratories. Es ist als Forschungswerkzeug für eine über den Entwicklerkreis hinaus reichende Gemeinschaft entwickelt und eine Open Source Veröffentlichung wurde angestrebt [3]. Der Entwicklerkreis besteht aus einem Fachexperten und drei Entwicklern mit jeweils über fünf Jahren Erfahrung in Softwareentwicklung. Als Forschungswerkzeug soll CLiiME anderen Wissenschaftlern dienen, indem neue Energiemodelle einfach implementierbar sein sollen. Erst während der Entwicklung hat sich der Wunsch einer graphischen Benutzerschnittstelle ergeben. Dieser Wunsch konnte dank agiler Techniken umgesetzt werden.

Die graphische Benutzerschnittstelle ist in Java programmiert. Echte Unitttests mit eigenen Testklassen wurden in Java mit dem Testframework JUnit umgesetzt [5]. Es gibt eine Testklasse pro Funktion und die Tests sind kurz, übersichtlich und präzise. Die numerischen Berechnungen sind in objektorientiertem Fortran 2003 implementiert. Das Beispiel eines Unittests (Listing 1) wird in der Veröffentlichung gegeben. Dabei wird die `Energy` Methode der `absorptionEnergy` Klasse einmal ohne und einmal mit Parametern getestet. Die Bedeutung von `properties` und `laser` im zweiten Test bleibt unbekannt, da zum Zeitpunkt der Erstellung dieser Ausarbeitung kein Quellcode von CLiiME im Internet gefunden werden konnte [6]. Als erwartetes Ergebnis wird jeweils ein Wert vorgegeben, der mit Igor berechnet wurde. Igor ist eine kommerzielle Software, die als Referenzgeber benutzt wird.

Listing 1: Beispiel eines Unittests in CLiiME. Auszug aus *Figure 2. Sample snippet of unit testing code for absorption energy* [3].

```
 subroutine absorption_energy_input(this)
    class(absorptionTest), intent(in) :: this
    type(absorptionEnergy) :: absorption
    ! Result from running the Igor code
    real(rkind) :: expect = 2.441e-10
    absorption = absorptionEnergy()
    Qabs = absorption%Energy()
    call assert(error_within_tolerance(expect, Qabs), &
        error_message("Qabs = 2.441e-10 is expected"))
end subroutine

subroutine absorption_energy_default(this)
    class(absorptionTest), intent(in) :: this
    type(absorptionEnergy) :: absorption
    real(rkind) :: Qabs
    ! Result from running the Igor code
    real(rkind) :: expect = 1.441E-10
    character(len=*), parameter :: filename='default.txt'
    absorption = absorptionEnergy(filename=filename)
    Qabs = absorption%Energy(properties,laser)
    call assert(error_within_tolerance(expect, Qabs), &
        error_message("Qabs = 1.44E-10 is expected"))
end subroutine
```

Numerische Funktionen werden wie in dem Beispiel gegen extern berechnete Beispielwerte getestet. Zudem werden sie wie in Abschnitt 4.2 in einfache Operationen geteilt. Als Fehlertoleranz für Tests in CLiiME wird 10 % angegeben. Bei iterativen Verfahren werden die ersten zehn Schritte mit präzisen Tests untersucht.

Für die testgetriebene Entwicklung von CLiiME wurden verschiedene Regeln aufgestellt. Die Wichtigsten seien hier kurz genannt:

- Vor der Implementierung Testkatalog erstellen.
- Tests müssen voneinander isoliert und unabhängig sein.
- Es darf kein manuelles Setup benötigt werden.
- Tests sollen schnell sein.
- Es soll immer nur so viel implementiert werden, bis alle aufgestellten Tests erfüllt sind.
- Nicht versuchen zukünftigen Bedürfnisse im Voraus zu erfüllen.
- Neuen Test erstellen, wenn ein Fehler unerwartet eintritt.
- Vor dem Übertragen in das Software-Repository müssen alle Tests erfüllt sein.

5.2 Neuentwicklung von Snowflake

Snowflake ist eine parallele Implementation einer Schneeflocken-Wachstumssimulation. Diese ist von einem erfahrenen TDD-Entwickler nach einer klaren mathematischen Vorlage implementiert worden. Innerhalb von 12 Stunden produzierte dieser etwa 800 Zeilen Testcode und 700 Zeilen

Funktionscode. Die Software lieferte auf Anhieb korrekte Werte. Dies wurde anhand einer Referenzabbildung aus der Originalveröffentlichung festgestellt. Dank einer guten Testabdeckung konnten noch einige Optimierungen gefunden und überprüft werden.

5.3 Neufassung vom Gtraj

Das „Goddard Trajectory Model“ (Gtraj) ist ebenfalls ein parallel rechnendes Modell. Es simuliert die Trajektorien von Milliarden von Partikeln in der Luft. Die erste Implementation wurde in IDL geschrieben. Zur besseren Portabilität und Skalierbarkeit wurde es jedoch mit C++ und MPI von einem Team erfahrener wissenschaftlicher Programmierer neu geschrieben. Diese Wissenschaftler waren unerfahren in testgetriebener Entwicklung und Unittests. Daher hatten diese während der Neufassung von Gtraj mehrfach das Verlangen, erst zu implementieren und anschließend Tests zu schreiben. Zudem sank die Bereitschaft Tests zu schreiben, wenn es um komplexe Teile des Modells ging. Diese bedürfen besonderen Nachdenkens, denn die Erwartungswerte sind teilweise schwer zu spezifizieren. Anfangs bevorzugten die Wissenschaftler, Funktionen mit realistischen Werte anstelle von im Kopf berechenbaren Werten zu testen. All diese Konflikte mit TDD legten sich jedoch schnell [1].

Zusätzlich zu einer neuen Implementation wurden im Prozess der Neufassung zwei inhaltliche Fehler gefunden. Diese seien im alten monolithischen Code schwierig zu beheben gewesen. Ein Fehler betraf eine Interpolationsmethode, die in Grenzfällen unerwünscht zu einer Extrapolation führte. Der andere Fehler war im Verhalten eines numerischen Algorithmus. Es wurde übersehen, einen Runge-Kutter Algorithmus zur Lösung von Differentialgleichung an die Erdkrümmung anzupassen. Hierdurch traten große Ungenauigkeiten in Nähe der Pole auf. Nach der Behebung dieses Fehlers wurden Trajektoren in Nähe der Pole wesentlich genauer. Zusätzlich konnte so der Zeitschritt im Modell erhöht werden, was die Ausführungszeit verringerte.

5.4 PARAMESH, entwickelt 1998 bis 2008

PARAMESH ist ein Framework zur parallelen Implementation von mehrdimensionalen partiellen Differentialgleichungen. Es nimmt einem Wissenschaftler die Arbeit einer Modellinfrastruktur ab, indem dieser direkt die Gleichungen in übersichtlicher Form, ähnlich den mathematischen Ausdrücken implementiert. PARAMESH übernimmt die Erzeugung von Modellgittern, so wie die Verteilung der Rechnung auf einen Parallelrechner. Unter anderem ist dabei eine automatische Gitterverfeinerung möglich. PARAMESH wurde 1998 bis 2008 in Fortran entwickelt. Tests spielten in der Entwicklung eine wichtige Rolle. Jedoch wurden weder Unittests noch ein Testframework verwendet. Selbst eine allgemeine Teststrategie gab es nicht.

Tests wurden individuell erstellt. Teilweise wurden dazu kleine Programme erstellt, die Teile des Frameworks verwenden. Die Richtigkeit der Ergebnisse wurde häufig per Hand beurteilt. Dazu wurden auch graphische Abbildungen erstellt. Auch temporäre `print`-Ausgaben wurden zum Debuggen genutzt. Dies hatte zur Folge, dass einige Tests nicht aufgehoben wurden. Allgemein waren die Tests schwach dokumentiert, sodass sie meist nur ihr Autor ausführen und deuten konnte. Dies führte zu „eigenen“ Codeabschnitten. Hat ein anderer Entwickler etwas

geändert, was diesen Abschnitt betrifft, so wurde der Originalautor gebeten, seine Tests erneut auszuführen. Diese Entwicklungsweise wird als mühsam beschrieben [4].

5.5 pFUnit als Testframework

Das „parallel Fortran Unit testing framework“ (pFUnit) wird seit 2005 entwickelt, um Wissenschaftlern in der Entwicklung Mühen wie bei PARAMESH zu ersparen. Grundlegend ist pFUnit ähnlich aufgebaut, wie andere so genannte xUnit Testframeworks: Es erlaubt dem Entwickler die Erstellung von Unittests, deren Sammlung und gemeinsame Ausführung. pFUnit hat Testmethoden, die den in Abschnitt 4 beschriebenen Besonderheiten Rechnung tragen. Es gibt Testmethoden für Gleitkommazahlen. Diese beinhalten Methoden für relative und absolute Fehler. Auch werden besondere Werte wie `NaN` oder `inf` besonders beachtet. `NaN` (engl. für „Not a Number“, de:7 keine Zahl) stehlt für ein undefinierten oder nicht darstellbaren Werte und tritt als Ergebnis ungültiger Operationen auf. Dies ist zum Beispiel die Berechnung der Quadratwurzel einer negativen Zahl. `Inf` steht für Unendlich (engl. *infinity*) und tritt ebenfalls als Ergebnis bei Gleitkommazahlberechnungen auf, deren Ergebnis mathematisch unendlich ist. `NaN` hat die für sonstige Gleitkommazahlen ungewöhnliche Eigenschaft, dass `NaN` ungleich jeder anderen Gleitkommazahl ist, selbst $\texttt{NaN} \neq \texttt{NaN}$. Für `inf` gilt hingegen $0 \neq \texttt{inf} - \texttt{inf} = \texttt{NaN}$ [7]. Die Erwartungswerte `NaN` und `inf` müssen daher mit speziellen Überprüfungen getestet werden. Für mehrdimensionale Felder gibt es ebenfalls Tests. Parallele Funktionen, die auf MPI oder openMP basieren, können gleichfalls getestet werden. pFUnit ist in objektorientiertem Fortran 2013 geschrieben, beinhaltet jedoch auch in Python geschriebene Teile. Letztere beinhalten unter anderen ein Script, dass die Erstellung von Mocks erleichtert.

6 Fazits

Verschiedene Herausforderungen und Anekdoten von Tests und testgetriebener Softwareentwicklung in der Wissenschaft wurden in den vorherigen Abschnitten dargelegt. Darauf aufbauend und unter Hinzunahme der Erfahrungen des Autors dieser Arbeit wird in diesem Abschnitt ein Fazit zu dem Thema gezogen. Die Abschnitte 6.1 und 6.2 geben die Fazits aus den drei Veröffentlichungen wieder. Abgeschlossen werden sie mit einem eigenen Fazit in Abschnitt 6.3.

6.1 Vorteile durch Tests

Tests werden allgemein als positiv angesehen. Ihre Existenz fördert die Wartbarkeit der Software und der Zwang, testbare Software zu schreiben führt zu flexiblerer Software. Diese lässt sich frei von Nebenwirkungen sicherer Umstrukturieren, was zu besserer Codequalität führt. Eine explizite Autorenzuordnung wie die „eigene“ Tests in PARAMESH wird verringert. Dies erleichtert Erweiterungen zusätzlich. Tests dokumentieren in Teilen den Sinn und Zweck einzelner Codeteile. Gleichzeitig definieren sie konkrete Ansprüche und dienen als Anwendungsbeispiel von Funktionen. Neben den technischen Vorteilen werden auch psychologische Mehrgewinne durch Tests genannt. Erfolgreiche Tests wirken als Erfolgsereignisse positiv auf den Entwickler. Die Angst mit einer eigenen Änderung andere Funktionalität zu stören kann abgebaut werden. Sind

die Tests als ausreichend filigrane Unittests implementiert, lassen sich diese schnell und ständig ausführen.

Bis zur flächendeckenden testgetriebenen Softwareentwicklung in der Wissenschaft müssten die Wissenschaftler in gewissen Themenbereichen weiter sensibilisiert werden. Es muss verdeutlicht werden, dass es einen Unterschied zwischen dem theoretischen Konzept und einer Implementierung dessen gibt. Zwischen den beiden Ebenen können sich durchaus Fehler einschleichen. Eine hohe Testabdeckung kann zeigen, dass das wissenschaftliche Verständnis vertrauenswürdig umgesetzt ist. Dies lässt das Vertrauen in das Modell und seine Prognosefähigkeit steigen.

Die Zusammenarbeit zwischen Wissenschaftlern und ausgebildeten Softwareentwicklern kann durch testgetriebene Entwicklung verbessert werden. Die Wissenschaftler formulieren die konkreten Anforderungen in Form von Tests und geben eine erste Implementation, die Anforderungen zu Erfüllen. Entwickler können anschließend ohne die Fachdomäne komplett zu verstehen trotzdem Optimierungen am Code vornehmen.

6.2 Herausforderungen

Altlastencode ist ein Problem. Wie in Abschnitt 4.3 dargestellt bedürfen diese aufwändiger Behandlung, bevor ein altes Modell testgetrieben weiterentwickelt werden kann. Auf Dauer sollte hier eine Zerlegung und immer dichtere Testabdeckung der Altlast angestrebt werden.

Des Weiteren lassen sich wie in Abschnitt 4.2 numerische Algorithmen nicht mit exakten Erwartungswerten testen. Einen möglichen teilweisen Ausweg stellt die vollständige Implementation eines Algorithmus mit Objektorientierung dar. Um diese zu testen werden jedoch Mocks gebraucht. Deren Generierung ist mitunter aufwändig und besonders in Fortran kaum zu automatisieren.

Rilee und Clune [4] fordern zur Lösung dieser Herausforderungen *easy-to-use* (einfach zu benutzen) und *easy-to-learn* (einfach zu erlernen) Frameworks. Sie selbst beteiligten sich dazu an der Entwicklung von pFUnit. Seitens der Entwicklungsumgebungen sollte die Unterstützung zur Entwicklung und Ausführung von Tests erhöht werden.

Schließlich stellt sich die personelle Herausforderung in der Akzeptanz von TDD. Entwickler mussten überzeugt werden, dass Tests sinnvoll sind, obwohl sie gleichviele Zeilen wie der eigentliche Funktionscode in der Entwicklung brauchen. Dazu schlagen Nanthaamornphong et al. [3] eine Schulung von Projektstart in agiler Softwareentwicklung und TDD so wie einen projektbegleitenden Mentor vor.

6.3 Fazit des Autors

Die drei Veröffentlichungen [3, 1, 4] konnten zeigen, dass testgetriebene Softwareentwicklung in Wissenschaft angewendet werden kann. Sowohl kleine, wie große Softwareprojekte können profitieren Dies ist jedoch nicht ohne Lernprozess bei den Wissenschaftlern möglich. Es besteht viel Hoffnung, dass TDD wissenschaftliche Software verbessert. Ich denke, dass eine größere Sensibilität für Softwareentwicklung grundlegend ihre Qualität verbessern kann. Je nach Wissensstand

und Erfahrung sehe ich TDD jedoch nicht für jeden programmierenden Wissenschaftler als sinnvollen nächsten Lernschritt an. Viele Testverfahren und insbesondere Unittests basieren stark auf Objektorientierung. Bevor damit die eigene Software getestet werden kann, sollte in meinen Augen zuerst die Objektorientierung gelehrt und verstanden werden. Numerische Modelle in den Klimawissenschaften werden in der Regel in Fortran implementiert. Fortran unterstützt Objektorientierung „erst“ seit Version Fortran 2003, welche 2004 spezifiziert wurde [8]. Die Adaption dieser Version ist in meinem Umfeld sowie in unserer Lehre sehr gering. Vielmehr werden Unterschiede zwischen Fortran 77 (1978) und Fortran 90 (1991) oder 95 (1997) als neu und aktuell wahrgenommen.

Viel eher kann ich mir vorstellen, dass bei einer größeren Unterstützung durch Softwareentwickler die Codequalität von wissenschaftlicher Software verbessert werden kann. In derartiger Begleitung kann der Wissenschaftler stetig lernen, wie eine aus Sicht der Informatik bessere Entwicklung gemacht wird. In einem solchen Umfeld wiederum kann, wie in dem Beispiel CLiiME, ein testgetriebener Ansatz versucht werden. Diese wurde erleichtert, da es Teammitglieder gab, deren Expertise genau in der Anwendung von Methodik liegt.

Literatur

[1] T.L. Clune und R.B. Rood. „Software Testing and Verification in Climate Model Development“. In: *Software, IEEE* 28.6 (2011), S. 49–55. ISSN: 0740-7459. DOI: 10.1109/MS.2011.117.

[2] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2003, S. 240.

[3] A. Nanthaamornphong u. a. „Building CLiiME via Test-Driven Development: A Case Study“. In: *Computing in Science Engineering* 16.3 (2014), S. 36–46. ISSN: 1521-9615. DOI: 10.1109/MCSE.2014.33.

[4] M. Rilee und T. Clune. „Towards Test Driven Development for Computational Science with pFUnit“. In: *Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), 2014 Second International Workshop on*. 2014, S. 20–27. DOI: 10.1109/SE-HPCCSE.2014.5.

[5] Aziz Nanthaamornphong u. a. „Building CLiiME via Test-Driven Development: A Case Study“. In: *Computing in Science and Engineering* 16.3 (2014), S. 36–46. ISSN: 1521-9615. DOI: http://doi.ieeecomputersociety.org/10.1109/MCSE.2014.33.

[6] Aziz Nanthaamornphong. „Is CLiiME available by now?“ persönliche Mittelung. Apr. 2015.

[7] David Goldberg. „What Every Computer Scientist Should Know About Floating-point Arithmetic“. In: *ACM Comput. Surv.* 23.1 (März 1991), S. 5–48. ISSN: 0360-0300. DOI: 10.1145/103162.103163. URL: http://doi.acm.org/10.1145/103162.103163.

[8] Wikipedia. *Fortran — Wikipedia, The Free Encyclopedia*. [Online; zuletzt abgerufen am 18. September 2015]. 2015. URL: https://en.wikipedia.org/w/index.php?title=Fortran&oldid=677925714.