Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Abschlussarbeit

im Studiengang Bachelor Informatik

Continuous Integration für die Entwicklung einer numerischen Bibliothek

von

Alexander Büchel

Erstprüfer: Prof. Dr. Simone Bürsner
Zweitprüfer: Dipl. Inform. Ottmar Krämer-Fuhrmann

Eingereicht am: 19. September 2012

Eidesstattliche Erklärung

Ich versichere an Eides statt, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäßig aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entkommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderer Prüfungsbehörde vorgelegen.

.....

(Bonn, 19. September 2012, Alexander Büchel).

Inhaltsverzeichnis

Abkürzungsverzeichnis....v
Abbildungsverzeichnis....vi
1 Einleitung....1
1.1 Motivation....1
1.2 Ziel der Abschlussarbeit....1
1.3 Vorgehensweise....2
2 Continuous Integration....3
2.1 Allgemeines....3
2.1.1 Definition Continuous Integration....3
2.1.2 Komponenten eines CI-Konzeptes....3
2.1.3 Ablauf eines Commit-Zyklus....4
2.1.4 Resultierende Vorteile durch CI....6
2.2 Spezielle Anforderungen durch einen numerischen Kontext....7
2.2.1 Eigenschaften einer numerischen Software....7
2.2.2 Anforderungen an ein numerisches Softwareprojekt....8
2.2.2.1 Korrektheit....8
2.2.2.2 Performanz....9
2.2.2.3 Portabilität....9
2.2.3 Resultierende Anforderungen an das CI-Konzept....9
2.3 Sonstige Anforderungen für Continuous Integration....10
2.3.1 Anforderungen in Bezug auf den Kunden....10
2.3.2 Anforderungen an das Entwicklerteam....10
2.3.3 Benötigte Werkzeuge....11
3 CI-Server....13
3.1 Allgemeine Vorstellung des CI-Server....13
3.2 Vorstellung der Werkzeuge....13
3.2.1 Jenkins....13
3.2.2 CruiseControl....13
3.3 Auswahl eines CI-Servers....14
4 Versionsverwaltungen....15
4.1 Aufgaben von Versionsverwaltungen....15
4.2 Werkzeuge....15
4.3 Auswahl eines Versionsverwaltungswerkzeuges....16
5 Testumgebung....17
5.1 Der Nutzes eines Test-Frameworks....17
5.2 Vorstellung der Test-Frameworks....17
5.2.1 Boost.Test....17
5.2.2 CPPUnit....18
5.2.3 GoogleTest....18
5.3 Assertions....19
5.4 Analyse....19
5.4.1 Pflichtkriterien....20
5.4.2 Weitere Kriterien....20
5.4.2.1 Komfort für Erststellung der Testhierarchie....20
5.4.2.2 Laufzeitoptionen....21
5.4.2.3 Ausgabeinformationen....21
5.4.2.4 Softwareaktualität....22
5.4.2.5 Dokumentation....23
5.4.2.6 Vergleichsmöglichkeiten von numerischen Werten....23
5.4.2.7 Zusätzliche Kriterien....24
5.5 Auswahl....25
6 Qualität des Codes – CodeCoverage....26
6.1 Codeabdeckung....26
6.2 Einsatzgebiete für eine Analyse der Codeabdeckung....27

6.3 Vorstellung von Gcov und Lcov....28
6.3.1 Vorstellung von Gcov....28
6.3.2 Benutzung von Lcov....28
7 CI-Konzeption für Anwendung in LAMA....30
7.1 Das Projekt LAMA....30
7.2 Aktueller Entwicklungsstand....30
7.3 Continuous Integration Konzept für LAMA....31
7.3.1 Aktueller Stand des CI-Konzeptes....31
7.3.2 CI Server....32
7.3.2.1 Einrichten des CI-Servers....32
7.3.2.2 Erkenntnisse der Verwendung....34
7.3.3 Vorstellung Teststrukturen....35
7.4 Analyse der Codeabdeckung....38
7.5 Portabilität....40
7.6 Anwendbarkeit von Continuous Integration....41
7.7 Evaluation des angewandten CI-Konzeptes....41
8 Zusammenfassung....44
9 Literaturverzeichnis....45
10 Anhang....48
10.1 Beispiel einer Testsuite mit Boost.Test....49
10.2 Beispiel einer Testsuite mit CPPUnit....50
10.3 Beispiel einer Testsuite mit GoogleTest....53
10.4 Übersicht einer Auswahl an Assertions der Test-Frameworks....54
10.5 Übersicht von Beispielausgaben der drei Frameworks in einem Fehlerfall....56
10.6 Skript für Durchführung der CodeCoverage Analyse von LAMA....58
10.7 Codeabdeckungsanalyse von LAMA nach Revision 1250 (April 2012)....59
10.8 Codeabdeckungsanalyse von LAMA nach Revision 1633 (August 2012)....60
10.9 Testübersicht eines spezifischen Übersetzungsvorganges in Jenkins....61

Abkürzungsverzeichnis

Abkürzung	Erklärung
CI	Continuous-Integration
CSR	Compressed Sparse Row Storage Format
CVS	Concurrent Version System
GCC	GNU Compiler Collection
HPC	High Performance Computing
LAMA	Library for Accelerated Mathematical Applications
SCAI	Fraunhofer Institute for Algorithms and Scientific Computing

Abbildungsverzeichnis

Abbildung 1: Komponenten eines CI-Konzeptes (nach Duvall 2011, S. 26).....4
Abbildung 2: Commit-Cycle nach Fowler.....5
Abbildung 3: Fehlerarten (nach Dahmen/Reusken 2006, S.11).....8
Abbildung 4: Aufbau einer Testhierarchie mit Boost.Test.....17
Abbildung 5: Registrierung von Testcases in eine TestSuite mit CPPUnit.....18
Abbildung 6: Definition eines Tests (Google 2012, "Wiki - V1_6_Primer").....18
Abbildung 7: Testlevels von Boost.Test (Rozenthal 2012, "Testing Tools").....19
Abbildung 8: Bewertung des Test-Framework-Vergleichs.....25
Abbildung 9: Formel für Anweisungsüberdeckungsgrad (Liggesmeyer 2002, S. 81)....26
Abbildung 10: Formel für Zweigüberdeckungsgrad (Liggesmeyer 2002, S. 87).....27
Abbildung 11: Benutzung von Lcov (nach LTP 2012b).....28
Abbildung 12: Logo Projekt LAMA.....30
Abbildung 13: CI-Konzept für LAMA (nach Duvall 2011, S.26).....31
Abbildung 14: Argumente für Übersetzungsvorgang.....33
Abbildung 15: Shellanweisungen für Aufrufe des seriellen Testlaufes.....33
Abbildung 16: Entwicklung der Build-Dauer in Jenkins.....34
Abbildung 17: Trend der Testergebnisse.....35
Abbildung 18: Vereinfachte Darstellung der Vererbungen der Storageklassen.....36
Abbildung 19: Ausschnitt des CSRStorageTest.....37
Abbildung 20: Vergleich der Abdeckungsgrade der LAMA-Bibliothek.....40

1 Einleitung

1.1 Motivation

Um eine Software fertigzustellen und dem Endkunden zu übergeben muss zunächst der Entwicklungsprozess durchschritten werden. Das zügige Durchlaufen dieses Entwicklungsprozesses ist besonders für den Endkunden von entscheidender Bedeutung, da die Wartezeit auf das Softwareprodukt für ihn reduziert wird. Problematisch könnte beispielsweise dabei ein modulares Vorgehen werden, wenn zunächst alle einzelnen Teilkomponenten eines Softwareproduktes entwickelt und diese daraufhin in einer anschließenden Phase, auch Integrationsphase genannt, zusammengefügt würden. Die Länge dieser Integrationsphase kann nur schwer vorausgesagt werden, so dass weder das Entwicklerteam noch der Endkunde wissen, wie lang die Fertigstellung des Produktes dauern wird. Dabei entsteht ein weiterer Nachteil. Da die Komponenten separat voneinander entwickelt werden, könnte es passieren, dass diese beim finalen Zusammenfügen nicht kompatibel sein und müssten, falls notwendig, angepasst werden. Die Folge wäre eine Verschwendung von personellen und somit auch finanziellen Ressourcen seitens des entwickelnden Unternehmens.

Dem Entwicklerteam ist daher anzuraten, den Entwicklungsprozess zu optimieren, beispielsweise mit der Anwendung von Continuous Integration. Dies ist ein Konzept, welches Risiken für Fehlerquellen während einer Softwareentwicklung reduziert. Mittels CI kann durch das Einrichten von automatisierten Testläufen das Funktionieren von entwickelten Methoden sichergestellt werden. Durch die Anwendung des Konzepts kann auch die erwähnte nachteilhafte Vorgehensweise verbessert werden, da die Integrationsphase auf ein Minimum reduziert wird. Die Integration wird zum Standardelement der täglichen Codeentwicklung indem das Zusammenspiel der Softwarekomponenten regelmäßig überprüft wird.

Als praktisches Beispiel für einen konkreten Einsatz von Continuous Integration wurde in dieser Arbeit die numerische und quelloffene Bibliothek LAMA ausgewählt. Diese wird im Fraunhofer Institut für Algorithmen und wissenschaftliches Rechnen SCAI im Schloss Birlinghoven, in Sankt Augustin, entwickelt. Der numerische Kontext der Bibliothek LAMA bietet ein ideales Einsatzszenario, da hierbei unter anderem die Sicherstellung der Korrektheit von numerischen Berechnungen, die Performanz der Algorithmen sowie die Portabilität der Software dauerhaft gewährleistet sein muss. Diese Sicherstellung wird durch die Einrichtung und Konfiguration der verschiedenen Komponenten, wie beispielhaft eine Teststruktur oder eines CI-Servers, umgesetzt.

1.2 Ziel der Abschlussarbeit

Das Ziel dieser Abschlussarbeit ist der Aufbau eines Continuous-Integration-Konzeptes, welches anschließend in die aktuelle Softwareentwicklung der numerischen Bibliothek LAMA integriert wird. In der Umsetzung dieses Konzeptes sind besonders die Auswahl eines Test-Frameworks für einen numerischen Kontext, einer Teststruktur, die dauerhaft die Korrektheit der numerischen Berechnungen sicherstellt und dem Einrichten und Verwenden eines CI-Servers hervorzuheben. Die daraus entstehenden Ergebnisse sollen auf ihren Nutzen für die LAMA Softwareentwicklung untersucht werden.

1.3 Vorgehensweise

Um ein CI-Konzept in der Praxis einrichten und anwenden zu können, muss dies zunächst auf theoretischer Ebene erarbeitet werden (Kapitel 2). Vor allem durch den numerischen Kontext der Entwicklung der Bibliothek LAMA ergeben sich Anforderungen, die das Konzept und die dabei benutzten Werkzeuge unterstützen müssen. Für die allgemeine Vorstellung des Konzeptes werden die Funktionen der einzelnen Komponenten, deren Aufgaben innerhalb des CI-Konzeptes sowie deren Zusammenhänge untereinander erläutert. Unter anderem zählen dazu die Versionsverwaltung (Kapitel 3), der CI-Server (Kapitel 4) und das Test-Framework (Kapitel 5). Für das CI-Konzept der Entwicklung der numerischen Bibliothek LAMA muss ein CI-Server und ein Test-Framework gefunden werden. Durch einen Vergleich zwischen CI-Servern bzw. Test-Frameworks soll jeweils das Softwarewerkzeug ausgewählt werden, welches die Anforderungen am Besten erfüllt. Um die Qualität der für LAMA entwickelten Teststruktur zu optimieren, wird das Vorgehen für eine Analyse der Codeabdeckung (Kapitel 6) erläutert.

Durch die Anwendung des Konzeptes und die Benutzung der einzelnen Werkzeuge ergeben sich Erkenntnisse, die für die Entwicklung der Bibliothek LAMA qualitative Vorteile ergeben. Die Vorstellung von LAMA, der konkreten Umsetzung des Konzeptes und dem daraus resultierenden Nutzen finden sich im siebten Kapitel wieder. Letztendlich werden die wesentlichen Punkte dieser Abschlussarbeit zusammengefasst (Kapitel 8).

2 Continuous Integration

Damit die Entwicklung einer Software möglichst effektiv verläuft, sollte ein gut strukturiertes Konzept als Grundlage für diesen Entwicklungsprozess dienen. Continuous Integration (CI) ist ein solches Konzept, welches dem Entwicklerteam viele Vorteile bringt und insbesondere die Integrationsphase, laut Martin Fowler, mit einem „non-event“ vergleichbar macht (Fowler 2006, „Continuous Integration“).

In diesem Kapitel der Arbeit wird das abstrakte Grundkonzept von Continuous Integration vorgestellt. Die Basis dieser Vorstellung ist der veröffentliche Artikel „Continuous Integration“ von Martin Fowler (Fowler 2006). Dieses Konzept soll die Grundlage für die konkrete Anwendung in der Entwicklung der numerischen Bibliothek LAMA sein. Sowohl die aus der Anwendung von CI resultierenden Vorteile, als auch Anforderungen, die durch das CI-Konzept aufgestellt werden, werden beschrieben. Anforderungen beziehen sich dabei nicht nur auf Ansprüche gegenüber den im Entwicklungsprozess beteiligten Personengruppen, wie zum Beispiel die Softwareentwickler oder der potentielle Endbenutzer, sondern sollen auch Anforderungen widerspiegeln, die speziell durch den numerischen Kontext im wissenschaftlich-technischen Sektor von einer Softwareentwicklung verlangt werden.

2.1 Allgemeines

2.1.1 Definition Continuous Integration

Fowler beschreibt Continuous Integration (zu deutsch: Kontinuierliche Integration) als eine weitgehend automatisierte Anwendungstechnik im Bereich der Softwareentwicklung. Diese sei geprägt durch kontinuierliche, sprich regelmäßige, Übergaben von Codeänderungen der Entwickler, sogenannte Commits, an eine zentralverwaltete Codebasis. Für die Sicherstellung der Softwarequalität werden alle Codeänderungen durch Testläufe und automatisierte Builds verifiziert.

2.1.2 Komponenten eines CI-Konzeptes

Um die Gesamtsituation zwischen allen involvierten Personen und Maschinen, sowie deren Kommunikationswegen zu verdeutlichen ist es zunächst ratsam, diese näher zu erläutern.

Fowler setzt für ein erfolgreiches CI-Konzept eine gemeinsam verwendete Codebasis, ein sogenanntes Repository, voraus. Dies ist ein zentraler Speicherort, an dem alle Daten liegen sollen, die für das Übersetzen der Software benötigt werden. Ein sogenanntes Source Code Management Tool verwaltet die Änderungen, die von Seiten der Entwickler an diesem Repository durchgeführt werden. Möglich ist es zwar, verschiedene Entwicklungszweige parallel zu einander verwalten zu können, allerdings ist es empfehlenswert, die Anzahl dieser Zweige für die Entwicklung der Software auf einen Hauptzweig (engl. mainline) zu minimieren. Entwickler sollen primär ihre entwickelten Codestücke in diesen Hauptzweig integrieren, um so gemeinsam an einem Entwicklungsstrang ziehen zu können.

Der Vorteil dieser Verwaltungsmöglichkeit ist die Zentralisierung des Projektes. Jeder Entwickler kennt diesen Speicherplatz, an dem alle zum Projekt gehörigen Daten enthalten sind und abgelegt werden können. Änderungen am Projekt geschehen über eine vorher auf einem lokalen Rechner gespeicherte Kopie des Repositories – eine

sogenannte „working copy“. Der Vorgang des Kopierens des gesamten Datenbestandes aus dem Repository auf eine lokale Maschine wird auch als Checkout bezeichnet. Nachdem Änderungen des Codes durch Entwickler an das Repository übergeben worden sind (engl. commit), sind andere Entwickler fähig, diese Änderungen nachträglich für ihre lokale Arbeitskopie zu erhalten. Sie führen eine Aktualisierung (engl. update) auf ihrem lokalen Datenbestand aus.

Laut Fowler gibt es zwei Möglichkeiten eine Codeänderung als erfolgreich abzuschließen und somit sicherzustellen, dass keine Fehler während des Übersetzungsvorganges existieren. Zum einen wird die Möglichkeit genannt, die Software manuell auf einer Integration Machine zu bauen, zum anderen die Möglichkeit, dies automatisiert durch einen CI-Server vornehmen zu lassen. Eine Einrichtung eines CI-Servers ist optional, allerdings wird das Entwicklerteam durch die Einrichtung eines eingerichteten CI-Servers unterstützt, indem sie möglichst automatisch informiert werden, wenn Probleme aufgetreten sind und der Übersetzungsvorgang oder die anschließend durchgeführten Testläufe fehlschlugen.

Die verschiedenen Kommunikationswege zwischen den einzelnen lokalen Maschinen der Entwicklern, der eingerichteten gemeinsamen Codebasis, dem CI-Server und der Möglichkeit des für den Entwickler sinnvollen Feedbacks, zum Beispiel in Form von E-Mails, dem Senden von SMS oder eines RSS-System (vgl. Duvall 2011, S. 10) wird in Abbildung 1 dargestellt.

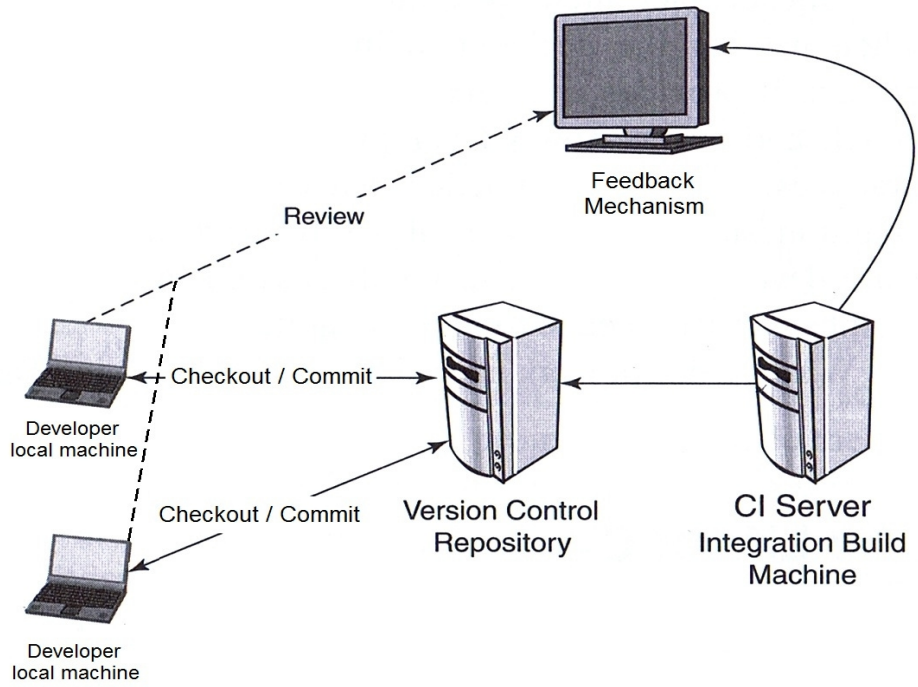


Abbildung 1: Komponenten eines CI-Konzeptes (nach Duvall 2011, S. 26)

2.1.3 Ablauf eines Commit-Zyklus

Nachdem ein Checkout durchgeführt wurde und somit eine Kopie des aktuellen Hauptzweiges auf dem lokalen Rechner existiert, ist es möglich, Änderungen an diesem Code vorzunehmen und diese daraufhin an das Repository zu überführen.

Um während der gesamten Entwicklungszeit die Fehlerfreiheit und Lauffähigkeit des Hauptzweiges des Projektes garantieren zu können, ist der Entwickler gezwungen, sich an einzelne Schritte zu halten. Diese Schritte - von der Änderung des Codes auf

der lokalen Maschine bis hin zur Feststellung der Lauffähigkeit auf der Integration Machine - werden auch als Commit-Zyklus bezeichnet und werden im Folgenden dargestellt.

Der Ausgangspunkt für diesen Zyklus ist ein Checkout des Hauptzweiges aus dem Repository, an dem der Entwickler seine Änderungen vollzieht. Der Entwickler ist nun in der Lage, Änderungen am Code vornehmen zu können. Danach wird der Übersetzungsvorgang auf Grundlage des geänderten Codes auf seiner lokalen Maschine gestartet. Laut Fowler sollen nach dem Übersetzen ebenfalls die Testläufe angestoßen werden, um so die Funktionalität des Codes sicherzustellen und gleichzeitig potentielle Laufzeitfehler feststellen zu können. Die zwingende Konsequenz ist, dass ein Entwickler Fehler, die nun entdeckt wurden, zunächst durch Korrekturen am Code beheben muss. Die Lauffähigkeit ist somit für den Entwickler genau dann gegeben, wenn durch den lokalen Übersetzungsvorgang keine Fehler beim Kompilieren, beim Linken oder anschließend beim Durchlaufen der Tests auftreten.

Unter der Voraussetzung, dass eine fehlerfreie lokale Übersetzung existiert, sollte zunächst ein Update des Repositories ausgeführt werden. Dies ist nötig, da in der Zwischenzeit andere Entwickler Änderungen an das Repository übergeben haben können. Ein weiteres Übersetzen auf der lokalen Maschine stellt somit sicher, dass auch mit den nun erhaltenden Updates ein fehlerfreies Funktionieren möglich ist. Falls dies nicht der Fall ist, müssen Korrekturen am Quellcode vorgenommen werden. Anzumerken bleibt, dass alle bis hierhin durchgeführten Schritte auf der lokalen Maschine des Entwicklers stattfanden. Falls allein dadurch schon Fehler aufgetreten sind, so sind diese auf der lokalen Maschine zu beseitigen. Dies bedeutet gleichzeitig, dass der zentral gelagerte und lauffähig funktionierende Code im Hauptzweig des Repository bis dato nicht durch fehlerbehaftete Commits gefährdet wird.

Jetzt kann die Codeänderung an das Repository übergeben werden. Durch die Tatsache, dass eine Versionsänderung des Repositories geschehen ist, kann nun entweder automatisiert oder auch manuell das Übersetzen auf der „Integration Build Machine“ gestartet werden. Falls auch dieser Übersetzungsvorgang einwandfrei funktioniert, kann nun der Commit-Zyklus als erfolgreich abgeschlossen werden. Fehler, die zu diesem Zeitpunkt auf der Integration Machine entstehen, müssen durch erneute Korrekturen behoben werden. Ein erneuter Checkout auf der lokalen Maschine ist nicht notwendig, da der Quellcode in der aktuellsten Revision durch den gerade durchgeführten Commit vorliegt. Dieser Zyklus, welcher in Abbildung 2 dargestellt ist, beginnt an dieser Stelle von vorne.

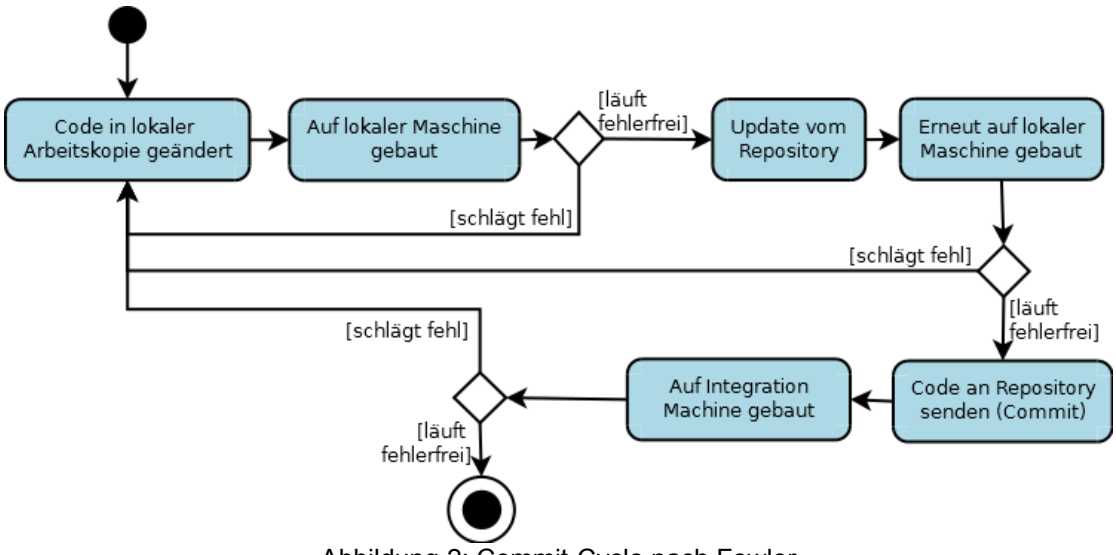


Abbildung 2: Commit-Cycle nach Fowler

2.1.4 Resultierende Vorteile durch CI

Durch das Anwenden des CI-Konzeptes ergeben sich deutliche Vorteile für die Entwicklung eines Softwareprojektes. Im Artikel von Fowler werden diese in vier Kategorien eingeteilt.

Fowler legt Wert auf die Einführung und Benutzung eines Source Code Management Tools. Dieses ist für die Verwaltung des Repository zuständig und stellt den einzelnen Entwicklern einen zentralen Ort zur Verfügung, an dem alle notwendigen Daten für das Softwareprojekt liegen. Diese Zentralisierung von projektrelevanten Daten wirkt sich vorteilhaft auf das gesamte Team aus, da nun keine Zeit verschwendet wird für das Suchen und Finden von Dateien, die möglicherweise auf verschiedensten Speichermedien liegen können.

Ein weiterer Vorteil wird durch die Automatisierung von Vorgängen gebildet. Ziel ist es, dass der Entwickler durch einen einzigen Anstoß – von Fowler als „single command“ (Fowler 2006, „Introducing Continuous Integration“) bezeichnet - eines Skriptes oder eines Softwarewerkzeuges eine Reihe von Schritten durchlaufen kann. Vorteile ergeben sich dadurch, dass zum einen Fehler bei immer wiederkehrenden manuellen Eingaben von Befehlen vermieden werden, da diese beispielsweise in einem Skript gebündelt und festgehalten werden werden und zum anderen dies automatisch eine Zeitersparnis für den Entwickler bedeutet. Eine konkrete Anwendung eines solchen Automatisierungstools ist der Übersetzungsvorgang einer Software – beispielsweise existieren Tools wie make oder Ant, welche das Kompilieren, Linken und das endgültige Erstellen eines ausführbaren Programmes übernehmen. Ein weiterer Einsatzort wäre ebenfalls das Verwenden eines CI-Server. Der CI-Servers reagiert auf Versionsänderungen des Repositories, führt ein Checkout des aktuellen Datenbestandes aus, startet den Übersetzungsvorgang sowie die Testläufe und gibt eine Rückgabe in Form eines Feedbacks über den aktuellen Status des Projektes an den Entwickler aus.

Der dritte Vorteil von CI ist die frühzeitige Fehlerbeseitigung. In der Softwareentwicklung gibt es unterschiedlichste Arten von Fehlern, die gefunden und beseitigt werden müssen. Diese können sowohl beim Übersetzungsvorgang auftreten sowie aber auch während der Laufzeit. Inhaltliche Fehler können möglicherweise durch eine geeignete Teststruktur erkannt werden, die parallel zum eigentlichen Code mitentwickelt werden sollte. Die Kombination aus Test-Driven-Development und der Verwendung eines Test-Frameworks sind für Fowler Grundlagen für eine gut strukturierte Teststruktur, die Bestandteil eines CI Konzeptes sein sollte. Auch für diese Teststrukturen sollte das eben erwähnte Prinzip der Automatisierung gelten. Durch einen einzigen Befehl sollte die gesamte Teststruktur durchlaufen werden und letztendlich ein Ergebnis über die Korrektheit der getesteten Funktionen liefern. Durch diese immer wieder – d.h. somit auch täglich mehrfach - ausgeführten Testläufe wird festgestellt, ob zwischenzeitlich Fehler entstanden sind. Daraus resultiert ein schnelles Finden und Eingrenzen der Fehler, welches Anreiz für den Entwickler ist, diese Fehler auch schnell zu beseitigen.

Des Weiteren erläutert Fowler einige Aspekte im Bereich der Kommunikation und den Informationsaustausch während einer Softwareentwicklung. Dadurch, dass alle Teammitglieder Zugriff auf eine zentrale Stelle, dem Repository, haben und somit Tag für Tag Änderungen am Hauptzweig geschehen, sollte so auch jeder über aktuelle Änderungen informiert werden. Durch das Source Code Management Tool geschieht dies meistens durch das Versenden von Mails an alle Entwickler, in denen die neuen Änderungen zusammengefasst werden. Ebenfalls geschieht dies durch den CI-Server und dem Erstellen eines Feedbacks über den aktuellen Buildstatus in Form einer Webseite. Durch diese Informationen erkennt ein Entwickler, ob der Buildvorgang auf

der Integrationsmaschine geglückt ist, die Testläufe erfolgreich waren und ob das Zusammenspiel zwischen verschiedenen Softwaretools, wie zum Beispiel Compilern und den verwendeten Bibliotheken funktioniert. Duvall ergänzt dies durch die Tatsache, dass diese gesammelten und dargestellten Informationen als Grundlage für Entwicklungsentscheidungen dienen können (Duvall 2011, S.31).

Neben diesen Vorteilen lässt sich der Nutzen von CI noch weiter ausdehnen, denn die klassische Integrationsphase einer Softwareentwicklung entfällt. Sie hat sowohl für den Entwickler, als auch für den Kunden Ungewissheit gebracht hat, da die Länge dieser Phase oft nicht vorhersehbar ist. Daher ist es möglich, in fast jederzeit lauffähige Versionen zu übersetzen, die zu Testzwecken verwendet oder an Endkunden weitergegeben werden können (Duvall 2011, S.31).

Summa summarum lässt sich sagen, dass der Faktor der Risikoreduzierung als zentrale Vorteil von CI genannt werden kann. Risiko der Fehleranfälligkeit von Methoden der Bibliothek lässt sich verringern durch eine Teststruktur, die mit Hilfe der Verwendung eines Test-Frameworks erstellt wird. Durch die Automatisierungsvorgänge des Übersetzungswerkzeug, dem CI-Servers sowie der Testläufe lassen sich Fehler beim manuellen Eingeben und Ausführen von vielen einzelnen Befehlen vermeiden. Der nötige Informationsfluss für den Entwickler wird von Seiten des Source-Code-Management-Tools und des CI-Server in Form von Benachrichtigungen geleistet.

2.2 Spezielle Anforderungen durch einen numerischen Kontext

2.2.1 Eigenschaften einer numerischen Software

Ein numerisches Modell ist eine Abbildung eines mathematischen Modells in ein für eine Rechenmaschine verständliches Format. Dieser Abbildungsvorgang wird auch als Diskretisierung bezeichnet und bietet die Möglichkeit anhand dieser diskreten Modelle numerische Lösungen zu mathematischen Problemstellungen zu finden. Eine Lösungsfindung auf Basis der mathematischen Modelle scheint in vielen Fällen entweder nicht berechenbar oder zu aufwendig (Sonar 2001, S. 22f.).

Der Bereich des „Wissenschaftlichen Rechnens“ (engl. Scientific Computing) umfasst diese möglichst realitätsgetreuen Modellierungen von Prozessen aus dem technischen bzw. naturwissenschaftlichen Sektor, die Berechnung der dazugehörigen Lösungen und die Simulation mit verschiedenen Testdaten. Darüber hinaus werden ebenfalls die Anwendung von numerischer Software, die für solche Berechnungen mathematischer Probleme geschaffen sind, hinzu gezählt. (Hanke-Bourgeois 2009, S. 12 und S. 465).

Aus der Tatsache, dass mathematische Modelle diskretisiert werden, ergeben sich charakteristische Eigenschaften für numerische Berechnungsprogramme.

Aus verschiedenen Gründen entstehen bei Berechnungen, auf Grundlage eines diskretisierten Modells, auf Rechenmaschinen Fehler. Im Allgemeinen können diese an drei Stellen auftreten: Eingabedaten können schon anfänglich, sprich vor dem Durchlaufen eines Algorithmus, fehlerhaft sein, zum anderen können während des Durchlaufes Fehler entstehen und abschließend wirken sich logischerweise diese beiden Fehlerquellen auf die Endergebnisse einer Berechnung aus (Deuflhard/Hohmann 2002, S. 26).

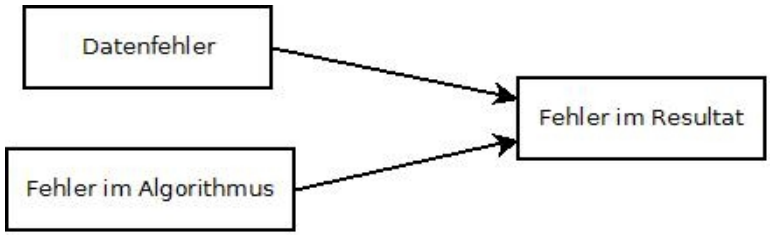


Abbildung 3: Fehlerarten (nach Dahmen/Reusken 2006, S.11)

Eingabedaten sind Werte, die einem Algorithmus als Ausgangssituation dienen. Solche Werte können schon vor einer einzigen Rechenoperation fehlerbehaftet sein, da deren Ursache unter anderem ungenaue Messwerte von durchgeführten Experimenten sein kann (Hermann 2001, S. 5). Neben diesen beiden Möglichkeiten sind Rundungsfehler eine weitere Fehlerquelle. Diese treten insbesondere bei der Übertragung von Zahlen in eine Maschine auf. Durch die Tatsache, dass Ressourcen einer Maschine begrenzt sind, müssen somit auch Daten auf Zahlen mit einer endlichen Anzahl an Stellen abgebildet werden, sogenannte Maschinenzahlen. Die Schlussfolgerung daraus ist, dass somit auch nur eine endliche Anzahl an Zahlen auf einer Maschine repräsentierbar sind, welches wiederum bedeutet, dass grundsätzlich eine mathematische Zahl nur gerundet durch die Darstellung einer Maschinenzahl ausgedrückt werden kann (Dahmen/Reusken 2006, S. 34ff.).

Der zweite genannte Typ von Fehlern sind die, die während der Ausführung eines Algorithmus entstehen. Das Ausführen jeder einzelnen Rechenoperation eines Algorithmus bewirkt die Verrechnung zweier Maschinenzahlen. Das Resultat dieser Berechnung muss, falls es keine Maschinenzahl ist, erneut auf eine solche abgebildet werden. Hierbei entsteht erneut das Phänomen der eben schon erwähnten Rundungsfehler. Allerdings fällt auch die Anzahl der durchgeführten Operationen ins Gewicht. Da die Anzahl dieser Operationen gewaltig sein kann, entstehen dementsprechend häufig Rundungsfehler, die sich durch den mathematischen Berechnungsprozess gesehen, addieren können (Dahmen/Reusken 2006, S.41).

Approximationsfehler sind ebenfalls Fehler, die in einem Algorithmus entstehen können. Dabei wird ein unendlich fortlaufender Algorithmus durch einen endlichen Ausdruck ausgetauscht. Durch diese Ersetzung ist das Ergebnis keine exakte Lösung zu einem gestellten mathematischen Problem, sondern nur eine Annäherung. (Hermann 2001, S. 5 und Deuflhard/Hohmann 2002, S. 41).

2.2.2 Anforderungen an ein numerisches Softwareprojekt

2.2.2.1 Korrektheit

Die im vorherigen Kapitel beschriebenen Fehlerarten sind die Grundlage für die Frage, wie groß die Differenz zwischen einem mathematisch exakten Ergebnis und der numerischen Annäherung sein darf, damit das Ergebnis der Approximation dennoch als korrekt angesehen werden kann. Der Faktor „Korrektheit“ kann daher als erste Anforderung einer numerischen Software festgehalten werden. Ein Ergebnis kann dann als korrekt angesehen werden, wenn die Differenz in einem möglichst kleinen, definierten Zahlenbereich liegt. Logisch wäre es natürlich, wenn diese Differenz gegen die Zahl 0 streben würde. Somit stellt man sicher, dass auch die Approximation gegen das mathematisch exakte Ergebnis strebt. Je näher die Annäherung an der exakten

Lösung liegt, desto „korrekter“ kann diese angesehen werden.

2.2.2.2 Performanz

Eine zweite Anforderung, die man an eine numerische Software stellt, ist die Ausführungsgeschwindigkeit, d.h. die Leistung oder Leistungsfähigkeit, die eine solche Software erbringen kann. Häufig werden für hoch komplexe Berechnungen Hardwarearchitekturen von IT-Diensten in Anspruch genommen, deren Benutzung anhand der verwendeten CPU-Zeit in Rechnung gestellt werden (vgl. Michels 2008, S. 50f.). Je weniger Rechenoperationen auf einer CPU für ein und dasselbe Ergebnis von Nöten sind, desto schneller und somit performanter ist diese Software. Eine Kostenersparnis wäre die Konsequenz, da die Rechenzeit optimiert, sprich verringert worden ist. Dies wirkt sich sowohl zeitlich, als auch finanziell gesehen auf Unternehmen aus, die eine solche Software für Berechnungen verwenden oder eine eben erwähnte Dienstleistung in Anspruch nehmen. Speziell für das Entwicklerteam ergibt sich daraus auch der Vorteil, dass die Wartezeit während Test- oder Benchmarkphasen verringert werden und anschließend schneller zurück zur Entwicklung übergegangen werden kann. Häufig werden an Computerprogramme Anforderungen an ihre Laufzeit gestellt, zum Beispiel die maximale Wartezeit einer Berechnung für den Benutzer.

2.2.2.3 Portabilität

Eine dritte Anforderung, welche nicht nur einer numerischer Software zugeordnet werden kann, ist die Portabilität, das heißt die Lauffähigkeit auf unterschiedlichen Zielsystemen. Einerseits ergeben sich wirtschaftliche Vorteile, wenn mit einer portabel entwickelten Software eine größere Zielgruppe angesprochen werden kann und somit sich die Menge der potentiellen Kunden erhöht. Ein größerer Umsatz und somit eine gesamtwirtschaftlich gesehen bessere Position des Unternehmens kann die Folge sein. Andererseits finden sich auch auf der technischen Ebene Aspekte, die vorteilhaft sind. Durch die Entwicklung einer Software und die dabei entstehenden Tests und Installationen auf unterschiedlichen Plattformen können potentielle Inkompabilitäten zwischen der entwickelten Software und diesen Plattformen in Erscheinung treten (Hook 2006, S.21ff. und Dalheimer 1999, S.254). Eine Kostenersparnis bei frühzeitig gefundenen Fehlern kommt erneut dem Softwareunternehmen zu Gute, da nachträgliche Installationen von Updates beim Endkunden entfallen. Ein Nutzen aus einer portablen Software zieht nicht nur das Unternehmen, welche diese Software entwickelt, sondern auch der Käufer der Software. Eine portable Software kann so gleichzeitig auf unterschiedlich verwendeten Plattformen verwendet werden, ohne für jedes System auf ein anderen Softwareprodukt zurückgreifen zu müssen (Hook 2006, S. 22). Bei einem unternehmensweiten Umstieg auf ein anderes Betriebssystem würde dies vorteilhaft sein, so dass eine portable Software somit direkt auch auf dem neuen System benutzbar sein kann. Eine Lernphase für die neue Software der Mitarbeiter entfällt somit.

2.2.3 Resultierende Anforderungen an das CI-Konzept

Die Aufgabe ist es nun, diese speziellen Anforderungen in das CI-Konzept zu übertragen, welches die Durchsetzung aller drei Anforderungen unterstützt. Die Korrektheit soll durch eine Teststruktur überprüft werden, die besonders für Vergleiche von mathematischen Werten geeignet ist. Die dabei verwendete Technik des Test-Driven-Development verhilft zu einem systematischen Vorgehen, um Tests optimal zu

entwickeln. Erwähnenswert ist dabei, dass bei den in Tests durchgeführten Berechnungen grundsätzlich mit den in Kapitel 2.2.1 erwähnten Fehlertypen zu rechnen ist. Die Grundlage für eine solche Teststruktur soll deshalb ein Test-Framework sein, welches auf einfache Weise Vergleiche zwischen berechneten und erwarteten Resultaten durchführen kann. Dabei muss die Möglichkeit gegeben sein, ein möglichst kleines Zahlenintervall festlegen zu können, in denen die Differenz des berechneten Wertes und des Erwartungswertes liegen müssen.

Es muss dabei nicht nur sichergestellt sein, dass die Ergebnisse von numerischen Berechnungen korrekt sind, sondern ebenfalls, dass alle Methoden überhaupt getestet, sprich durch die geschriebenen Tests mindestens einmal aufgerufen wurden. Je umfangreicher der Quellcode eines Projektes ist, desto komplexer wird auch der Nachweis, dass die Ausführung aller Methoden gegeben ist.

Die Performanz kann im Kontext einer numerischen Software als Laufzeit einer numerischen Berechnung interpretiert werden. Das Entwicklerteam verfolgt das Ziel, die Schnelligkeit einer Berechnung effektiver zu gestalten, so dass die Zeit, die für eine Berechnung nötig ist, abnimmt. Bei Optimierungen, die die Performanz betreffen, muss darauf geachtet werden, dass trotz den durchgeführten Veränderungen am Code die Korrektheit noch gewährleistet ist.

Ein weiteres Ziel des CI-Konzeptes soll die Portabilität einer Software sein. Daraus folgt, dass das korrekte Übersetzen eines Projektes mit unterschiedlichen Bibliotheken oder unterschiedlich verwendeten Compilern überprüfbar sein soll. Letztendlich gilt für eine portable Software, dass sie auch auf unterschiedlichen Betriebssystemen lauffähig sein soll.

2.3 Sonstige Anforderungen für Continuous Integration

Neben den spezifisch für LAMA geltenden, numerischen Anforderungen können auch allgemeine Anforderungen aufgestellt werden, die sich in jedem Projekt, welches durch ein Continuous-Integration-Konzept geprägt ist, ergeben.

2.3.1 Anforderungen in Bezug auf den Kunden

Durch ein angewandtes CI-Konzept soll das Entwicklerteam in der Lage sein, die Integrationsphase des Projekts so zu reduzieren, dass diese, wie von Fowler bezeichnet, zu dem „non-event“ (Fowler 2006, „Continuous Integration“) wird. Ein Kunde erfährt durch diese Anwendung der Softwareentwicklungstechnik den Vorteil, dass die Wartezeit auf das individuell zu entwickelnde Produkt reduziert wird. Anstatt wochen- bzw. monatelang auf ein Ende der aktuellen Integrationsphase zu warten, soll es nun jederzeit möglich sein, dem Kunden eine lauffähige Version zu übergeben, trotz dabei entstehenden funktionalen Einschränkungen. Die aus älteren Softwarepraktiken, wie z.B. dem Wasserfallmodell bekannte Anforderung an den Kunden, sich so lange in Geduld zu üben bis die laufende Integrationsphase und somit die aktuelle Produktversion fertiggestellt ist, soll somit auf eine minimale Zeitspanne reduziert werden.

2.3.2 Anforderungen an das Entwicklerteam

Neben der Anforderung an Softwarewerkzeuge, dass alle Komponenten von CI selbst funktionieren und miteinander agieren, so müssen auch Anforderungen an das

Entwicklerteam gestellt werden. Die besten Softwarewerkzeuge bringen für eine Softwareentwicklung keinen Nutzen, wenn diese nicht ordnungsgemäß durch eine vorher fest definierte Reihe von Arbeitsschritten benutzt werden, um so die optimalen Vorteile der Werkzeuge und letztendlich des CI-Konzeptes herauszuholen.

Die Basis für die gesamte Entwicklung einer Software soll ein stabiles, das heißt somit ein lauffähiges und auch fehlerfreies Repository sein, welches durch die an der Entwicklung beteiligten Personen bewahrt muss. Unter anderem sollen Entwickler darauf achten, dass alle relevanten Daten für das Projekt tatsächlich in das Repository übergeben werden und somit zentralisiert für alle bereitstehen.

Falls während des Übersetzungsvorgangs oder den Durchläufe von Tests Fehler auftreten, führt dies zu einer nächsten Anforderung an die Entwickler. Solche entstandenen Fehler sollen schnell gefunden und korrigiert werden. Werkzeuge helfen dem Entwickler, diese möglichst effektiv zu lokalisieren und in Bezug zum gesamten Quellcode einzugrenzen. Allerdings ist es immer noch Aufgabe vom Entwickler selbst, diese zu beseitigen.

2.3.3 Benötigte Werkzeuge

Um ein erfolgreiches Continuous Integration Konzept in ein Softwareprojekt zu integrieren, werden unterschiedliche Werkzeuge benötigt. Obwohl nicht jedes Werkzeug unbedingt notwendig ist, sollen in diesem Kapitel alle potentiell verwendbaren Softwarewerkzeuge Erwähnung finden, die für ein CI-Konzept wichtig sind.

Die Zentralisierung lässt sich am Besten durch die Benutzung eines Source Code Management Tools umsetzen. Dieses verwaltet die zentrale Lagerstätte des Quellcodes eines Projektes. Fowler empfiehlt, in jeder Softwareentwicklung ein solches Werkzeug zu benutzen.

Neben diesem hilft ein Automatisierungswerkzeug, wie zum Beispiel make oder Ant, den Übersetzungsvorgang von Fehlern, die durch die manuelle Eingabe von Befehlen eines Entwicklers entstehen, zu befreien und gleichzeitig zu beschleunigen, da nur modifizierte Quelldateien neu kompiliert und gelinkt werden müssen, wenn dies durch das Übersetzungswerkzeug unterstützt wird.

Die Fehlerbeseitigung ist ein weiterer Vorteil des CI-Konzeptes. Die Grundlage dafür ist eine ausgereifte Teststruktur. Um eine solche Teststruktur zu schaffen, ist ein geeignetes Test-Framework von Nöten. Die Feststellung der Codequalität kann durch zusätzliche Werkzeuge unterstützt werden, wie zum Beispiel einem CodeCoverage-Tool (vgl. Duvall 2011, S.54f.), welches die aktuelle Codeabdeckung darlegt oder Debugger-Tools, die Fehler in der Programmausführung finden können. Letztendlich ist es nützlich einen CI-Server einzurichten, der Aufschluss darüber gibt, ob das aktuelle Projekt in einer konfigurierten Umgebung übersetzt werden kann und ob bzw. wie viele Fehler bei den dazu aufzurufenden Testläufen entstehen.

Die konkrete Auswahl von Tools für ein Projekt, welches durch CI geprägt ist, hängt natürlich von dem gegebenen Kontext, dem Verlangen von Kunden oder den Vorlieben der Entwickler ab. Für den numerischen Entwicklungskontext der Bibliothek LAMA, in dem vor allem Berechnungen von Fließkommazahlen im Vordergrund stehen, müssen Werkzeuge ausgewählt werden, die eine gute Unterstützung bieten, Werte von Berechnungen und die passenden Erwartungswerte vergleichen können. Dies gilt besonders für das Test-Framework. Die verschiedenen Testläufe, welche mit diesem erstellt werden, müssen durch regelmäßige Ausführungen, die Korrektheit dieser

Berechnungen bestätigen. Das Einrichten eines CI-Server hilft, um den Übersetzungsvorgang von LAMA zu überprüfen und die regelmäßige Durchführung der Testläufe zu automatisieren.

3 CI-Server

3.1 Allgemeine Vorstellung des CI-Server

Eines der zentralen Softwaretools in einem CI-Konzept ist der CI-Server. Dieser dient als Verbindungsmodul zwischen allen anderen Bauteilen dieses Konzeptes und kommuniziert mit diesen. Der CI-Server hält eine Verbindung zu dem benutzten Source Code Management Tool, welches den Quellcode des Projektes verwaltet. Das Ausführen von betriebssysteminternen Befehlen, beispielsweise auf der Shell in Unixsystemen, sollte der CI-Server unterstützen, um das Starten von Testläufen, die durch das Verwenden eines Test-Framework entstanden sind, zu automatisieren. Weiterhin können vom CI-Server andere Werkzeuge, wie Debugger- oder Profilertools, aufgerufen werden. Die Ergebnisse der Ausführungen, wie zum Beispiel dem Übersetzungsvorgang, werden durch den CI-Server für den Entwickler protokolliert und für einen Webbrowser dargestellbar bereitgestellt. Vorteilhaft für Entwickler ist es, wenn ihnen Verschlechterungen des Buildstatus gemeldet werden, beispielsweise per E-Mail. Die sorgfältige Auswahl eines CI-Servers ist von großer Bedeutung für die gesamte Entwicklung eines Softwareprojektes.

Für diese Arbeit sollen sowohl Jenkins, als auch CruiseControl für die potentielle Verwendung im Projekt LAMA betrachtet werden. Beide sind als kostenlose Softwareprodukte verfügbar. Ebenfalls sind beide javabasiert, so dass sie auf alle gängigen Betriebssystemen eingesetzt werden können.

CI-Server, wie Draco.NET oder CruiseControl.NET werden nicht betrachtet, da diese speziell für .NET Projekte geeignet sind. Ebenso entfallen für diese Auswahl kommerzielle Produkte, wie Anthill Pro oder Bamboo.

3.2 Vorstellung der Werkzeuge

3.2.1 Jenkins

Jenkins ist ein javabasierter CI-Server, der als Open Source Projekt verfügbar ist. Jenkins entstand auf der Basis des Quellcodes von Hudson, welcher durch die Übernahme von Sun durch Oracle eine andere Entwicklungsstrategie erfuhr, mit der viele beteiligte Entwickler nicht zufrieden waren (vgl. Smart 2011, S. 4). Die Informationen auf der Internetpräsenz von Jenkins (Jenkins CI 2012) deuten darauf hin, dass die Entwicklung von Jenkins fortlaufend ist. Die aktuelle Version 1.479 ist Ende August 2012 veröffentlicht worden. Durch das Installieren von Plugins kann der Server auf die persönlichen Belange angepasst werden. Die Nutzung verschiedener Build- oder Source-Code-Management-Tools ist durch Jenkins ebenfalls mittels Plugins möglich.

3.2.2 CruiseControl

CruiseControl ist ebenfalls ein javabasierter CI-Server, dessen neueste Version 2.8.4 aus dem September 2010 stammt. Die Basis dieser Software besteht aus dem „Build Loop“, welches zu vergleichen ist mit einer Schleife, die für die regelmäßige Ausführung der Übersetzungsvorgänge zuständig ist, dem Generieren von JSP-Seiten basierend auf den Resultaten, die durch den „Build Loop“ entstanden sind, und einen

„Dashboard“, welches in einem Webbrowser dargestellt werden kann, um einen Überblick über die aktuellen konfigurierten Projekte zu erhalten. Das Informieren in Form von Emails, RSS-News oder Instant Messagern ist ebenfalls möglich (CruiseControl 2012a, „Overview“).

config.xml ist eine im XML-Format geschriebene Konfigurationsdatei, welche als Grundlage für den „Build Loop“ dient und die auszuführenden Anweisungen mit den nötigen Parametern liefert. Da die Anzahl der verwendbaren XML-Elemente gewaltig ist (vgl. CruiseControl 2012a, „config.xml“), ist dementsprechend auch der Aufwand, um einen funktionierenden CI-Server mit auszuführenden Builds einzurichten, deutlich höher und komplexer einzuschätzen. Auch bei diesem Server existieren viele Plugins, die das Erweitern des CI-Server durch zusätzliche Funktionalitäten ermöglichen.

3.3 Auswahl eines CI-Servers

Vom Funktionsumfang gleichen sich beide CI-Server, da fehlende Unterstützung der Basisversionen durch das Installieren und Verwenden von Plugins ausgeglichen werden kann. Beide unterstützen das Source Code Management Tool Subversion, welches schon fester Teil der Entwicklung von LAMA ist. Jenkins bietet desweiteren ein Plugin namens "cmakebuilder" an, um das ebenfalls eingerichtete Buildtool Cmake benutzerfreundlich verwenden zu können. Bei CruiseControl müsste dafür auf Befehle, die zum Beispiel an eine Shell weitergeleitet werden, zurückgegriffen werden. Jenkins verwendet eine intuitiv gestaltete Benutzeroberfläche, die auch für Personen ohne weiterführendes Wissen geeignet ist. Bei CruiseControl ist dieser Punkt ein klarer Nachteil, da das Benutzen der XML-Syntax vorausgesetzt werden muss, um die benötigte Datei config.xml zu erstellen und so Jobs einrichten zu können.

Ein weiterer Punkt, der für die Verwendung von Jenkins spricht, ist die aktive Entwicklung des Servers, die anhand der regelmäßig veröffentlichten Aktualisierungen ersichtlich wird. Daher kann davon ausgegangen werden, dass auch bei auftretenden Fehlern oder plötzlichen Inkompabilitäten, beispielsweise zu einer aktualisierten Java-Version, diese möglichst zeitnah beseitigt werden. Über die Entwicklung von CruiseControl kann keine genau Angabe gemacht werden, da seit dem Jahr 2010 keine weitere Version veröffentlicht worden ist.

Aufgrund der intuitiven Bedienung, sowie der erkennbaren aktiven Weiterentwicklung ist die Wahl für den zu wählenden CI-Server für die Softwareentwicklung des Projektes LAMA auf Jenkins gefallen.

4 Versionsverwaltungen

4.1 Aufgaben von Versionsverwaltungen

Um ein Projekt, besonders bei einer größeren Anzahl an Entwicklern optimal organisieren zu können, ist es ratsam ein Versionskontrollsystem zu verwenden. Durch die Aufzeichnung der einzelnen Änderungen am Quellcode ergeben sich diverse Vorteile für die Softwareentwicklung und dementsprechend für das Entwicklerteam. Der Quellcode befindet sich an einem zentralen Ort, auf den die Entwickler jederzeit Zugriff haben. Die Aufgabe in regelmäßigen Zeitabständen ein Backup des Datenbestandes auf der lokalen Maschine zu erstellen entfällt somit, da der Server auf dem das Repository liegt, nun als Backup für den Entwickler dient. Falls ein lokaler Datenbestand nicht mehr funktionsfähig ist, kann so schnell eine Kopie des Repositories geholt werden. Änderungen werden grundsätzlich nur auf der Arbeitskopie auf der lokalen Maschine vorgenommen und am Ende dem Repository übergeben.

Jederzeit kann mittels einer Versionsverwaltung die Möglichkeit geboten werden, Zugriff auf eine ältere Version des Projektes zu erhalten, um so nachträgliche Fehlerkorrekturen durchzuführen. Ebenfalls können die Unterschiede zwischen zwei Entwicklungszuständen (Revisionen) veranschaulicht werden.

Wenn mehrere Entwickler gleichzeitig an einer Stelle im Quellcode entwickeln, so können sich Konflikte ergeben, wenn diese Änderungen an das Repository übergeben werden. Eine Versionsverwaltung sollte in der Lage sein, diese Konflikte zu erkennen und dem Entwickler dazulegen, so dass der Entwickler selbst entscheiden muss, ob seine Änderung nun beibehalten oder die Änderung eines anderen Entwicklers angenommen wird (Vesperman 2004, S. 3ff.).

Die allgemeinen Vorteile einer Versionsverwaltung spiegeln sich auch im Konzept von Continuous Integration wieder. Ein CI-Server wird im Optimalfall einmalig konfiguriert und führt seine Aufgaben ab diesem Zeitpunkt selbstständig aus. Bei der Konfiguration muss die Adresse des Speicherorts angegeben werden, durch die der Quellcode des Softwareprojektes zu erreichen ist. Durch den von der Versionsverwaltung zentral bereitgestellten Speicherort kann vermieden werden, dass jeder Entwickler mit seiner persönlichen und „manuellen Versionshistorie“ (vgl. Budszuhn 2005, S. 21f.) arbeitet. Ein CI-Server agiert beim Start eines Übersetzungsvorganges genau wie ein Entwickler auf seiner lokalen Maschine, indem er auf der Integrationsmaschine ein Checkout durchführt.

4.2 Werkzeuge

Auf dem Markt befinden sich einige Softwarewerkzeuge, die als Versionsverwaltungen dienen können. Revision Control System (RVS) war der Ursprung für eine Versionverwaltung, die die Zusammenarbeit von mehreren Entwicklern in einem Softwareprojekt unterstützte. Darauf aufbauend galt Concurrent Version System (CVS) als eines der ältesten Werkzeuge, deren Anfang schon im Jahr 1986 lag. Die ursprüngliche Version dieses Tools ist für Unix-Systeme bestimmt gewesen. Mangels Wahlmöglichkeiten beherrschte dieses Open Source-Werkzeug für einen langen Zeitraum den Markt (Budszuhn 2005, S. 27f.).

Subversion (SVN) wurde auf der konzeptionellen Grundlage von CVS entwickelt und wurde im Jahr 2004 in der ersten Version, sowohl für den freien, als auch für den

kommerziellen Markt veröffentlicht. Der portable Einsatz auf verschiedenen Betriebssystemen ist dabei durch die Verwendung der Apache Portable Runtime Library sichergestellt. SVN ist für die reine Verwaltung von Dateibeständen zuständig und besitzt daher keine Funktionalität, um die vorhandenen Dateien inhaltlich zu interpretieren (Budszuhn 2005, S.28ff.).

Ein weiteres aktuell entwickeltes Open Source Versionsverwaltungwerkzeug ist GIT. Vorteile liegen bei diesem Tool bei der Erstellung von unabhängigen lokalen Entwicklungszweigen. Diese können bei einer erfolgreichen Entwicklung in den Hauptzweig integriert werden oder wieder herausgelöst werden, ohne dass dieser Eingriff andere Entwicklungszweige gefährdet. Es unterscheidet sich unter anderem von Subversion, in dem das Repository verteilt auf verschiedenen lokalen Maschinen der Entwicklern liegen kann (Git 2012).

4.3 Auswahl eines Versionsverwaltungswerkzeuges

In LAMA wird Subversion als Versionsverwaltungswerkzeug eingesetzt, da es durch die zentrale Infrastruktur des Instituts bereitgestellt wird. Ein Vergleich und eine Auswahl bezüglich dieses Tools ist daher nich notwendig.

5 Testumgebung

5.1 Der Nutzes eines Test-Frameworks

Unittests sind eine Möglichkeit, die Korrektheit von Programmeinheiten sicherzustellen. Je größer ein Projekt ist, desto höher kann auch die Anzahl der Funktionen und somit die Anzahl der Testmethoden sein. Testmethoden werden in einer Hierarchie, bestehend aus Testsuites und Testcases, strukturiert. Dabei dienen Testsuites dazu Testcases zu gruppieren. Ein Test-Framework bietet den Rahmen für die Erstellung einer solchen Unitteststruktur, die letztendlich in eine ausführbare Datei gebündelt wird. Daher sind nicht nur Testmethoden an sich für die Sicherstellung der Qualität eines Softwareprojektes notwendig, sondern auch die Festlegung der Art und Weise, wie diese Testmethoden organisiert werden. Ohne eine solche Struktur wird es für den Programmierer schwer, den Überblick über die zum Projekt gehörigen Testmethoden zu behalten und eine korrekte Ausführung sicherzustellen.

5.2 Vorstellung der Test-Frameworks

Um ein Test-Framework für den Einsatz im Projekt LAMA zu finden, werden in den folgenden Unterkapiteln drei potentielle Frameworks vorgestellt und auf ihre praktische Eignung untersucht. Zur praktischen Veranschaulichung der Anwendung dieser Frameworks befinden sich im Anhang dieser Arbeit verschiedene erstellte Beispiele der Klasse ScalarTest, die die Klasse "Scalar" des Projektes LAMA testen. Durch einen Vergleich soll das Vorteilhafteste dieser Frameworks für das Softwareprojekt LAMA herausgearbeitet werden.

5.2.1 Boost.Test

Das erste potentielle Test-Framework für den Einsatz im Softwareprojekt LAMA ist Boost.Test. Dieses Framework ist Teil der Boost-Library, welche im Juni 2012 in der Version 1.50 erschienen ist. Durch die regelmäßigen Aktualisierungen dieser Open Source Library kann davon ausgegangen werden, dass auch eine zukünftige Weiterentwicklung gegeben ist (Dawes/Abrahams/Rivera 2012).

Sowohl die Erstellung von Testsuites und Testcases, als auch die angebotenen Assertions werden durch Makros bewerkstelligt, wobei jede Assertions in drei Abstufungen CHECK, WARN und REQUIRE angeboten wird. Die Verwendung von Makros wirkt sich benutzerfreundlich für den Entwickler aus und bewirkt eine schnellere und effektivere Programmierung (Rozenthal 2012, "Test suite – Automated Registration", "Testing tools"). Ein einfaches Beispiel der Verwendung der Makros für die Erstellung einer Testhierarchie wird in Abbildung 4 dargestellt.

```
BOOST_AUTO_TEST_SUITE( test_suite_name )
    BOOST_AUTO_TEST_CASE( test_case_name )
    { ... }
BOOST_AUTO_TEST_SUITE_END();
```

Abbildung 4: Aufbau einer Testhierarchie mit Boost.Test

5.2.2 CPPUnit

Das zweite Test Framework, welches vorgestellt werden soll, ist CPPUnit. Dieses ist ein Open Source Test-Framework, welches als Pendant zu JUnit für die C++-Welt portiert wurde. Plattformübergreifend liegt es zur Zeit in der Version 1.12.1 vor, welche Anfang 2008 veröffentlicht wurde. Die Nachfolgerversion "CPPUnit 2" ist zwar in der Entwicklung, allerdings stagniert diese seit dem Jahr 2009 (N.N. 2012a, Lepilleur 2012a und 2012b).

Im Gegensatz zu Boost.Test gibt es bei CPPUnit eine strikte Trennung zwischen der Testklasse und der dazugehörigen Headerfile. Eine Testklasse erbt von der Klasse TestFixture, welche im Namespace CPPUNIT_NS zu finden ist (Anhang 10.2). Durch angebotene Makros muss eine Testsuite für die Testklasse in der Headerdatei angelegt werden, in der daraufhin manuell die einzelnen Testmethoden registriert werden (siehe Abbildung 5), die hier gleichbedeutend sind mit Testcases. Durch die Vererbung der TestFixture-Klasse können die tearDown()-, sowie die setUp()-Methode überschrieben werden, die für das Erstellen und Löschen von Objekten gedacht sind, die in mehreren Testcases benötigt werden. Eine Verringerung von Codeduplizierung kann hierdurch erreicht werden.

```
CPPUNIT_TEST_SUITE( testsuite_name )
    CPPUNIT_TEST( test_name_1 )
    CPPUNIT_TEST( test_name_2 )
    CPPUNIT_TEST( test_name_3 )
CPPUNIT_TEST_SUITE_END();
```

Abbildung 5: Registrierung von Testcases in eine TestSuite mit CPPUnit

5.2.3 GoogleTest

Das dritte und letzte vorzustellende Framework ist GoogleTest. Es ist ein von Google entwickeltes und herausgegebenes Framework, welches zur Zeit in der Version 1.6 aus dem April 2011 angeboten wird (Google 2012).

GoogleTest bietet ebenfalls Gruppierungen von Einzeltests an. Das Makro TEST() verlangt dafür grundsätzlich zwei Parameter (Abbildung 6): Einerseits einen Namen für den Testcase – hier gleichbedeutend mit einer Testsuite in Boost.Test - andererseits den Testnamen, der einem Testcase im Framework von Boost entsprechen würde. Die Benutzung dieses Makros und die Erstellung eines Testcase mit drei registrierten Tests ist durch das Beispiel dargestellt, welches im Anhang zu finden ist (Anhang 10.3). Assertions von GoogleTest werden, wie die von Boost.Test, in Abstufungen – hier die beiden Abstufungen EXPECT und ASSERT – angeboten.

```
TEST(test_case_name, test_name) {
 ... test body ...
}
```

Abbildung 6: Definition eines Tests (Google 2012, "Wiki - V1_6_Primer")

5.3 Assertions

Jedes Test-Framework unterstützt Annahmen (engl. assertions), die einen übergegebenen Ausdruck auf Korrektheit prüfen. Boost.Test, CPPUnit und GoogleTest bieten einen unterschiedlichen Umfang an Assertions an, welche im Anhang dieser Arbeit (Anhang 10.4) dargestellt sind.

Ein wesentlicher Unterschied zwischen den angebotenen Assertions der verschiedenen Frameworks ist derjenige, dass die Fehlerbehandlung jeweils unterschiedlich abgestuft ist. Boost.Test bietet für jede Assertion drei Abstufungen an: Warn, Check und Require (Abbildung 7). So lässt sich durch die Verwendung dieser Abstufungen kontrollieren, ob ein Testlauf eine nicht korrekte Behauptung als Fehler zählen, den Testlauf weiterlaufen lassen oder ihn gar abbrechen soll. Eine Optimierung eines Tests ist die Folge, da unnötige Testabschnitte somit übergangen werden können, wenn bestimmte Assertions nicht erfüllt sind.

Level	Test log content	Errors counter	Test execution
WARN	warning in *<test case name>*: condition *<assertion description>* is not satisfied	not affected	continues
CHECK	error in *<test case name>*: test *<assertion description>* failed	increased	continues
REQUIRE	fatal error in *<test case name>*: critical test *<assertion description>* failed	increased	aborts

Abbildung 7: Testlevels von Boost.Test (Rozenthal 2012, "Testing Tools")

Beispielsweise sollte ein Test abgebrochen werden, wenn das Erstellen, Zurückgeben und Zuweisen eines in einer Factory-Klasse erstellten Objektes nicht erfolreich war und dieses allerdings für den weiteren Verlauf des Tests benötigt wird. Ohne die Existenz dieses Objektes ist es nicht sinnvoll, die dann folgenden Testanweisungen aufzurufen.

GoogleTest bietet dem Entwickler zwei Abstufungen (Assert und Expect). Die Abstufung "Assert" überprüft einen gegebenen Ausdruck und lässt die aktuelle Testmethode abbrechen, falls die Behauptung nicht stimmt; "Expect" gibt bei einem fehlgeschlagenen Test nur eine Fehlernachricht aus und führt den Testlauf fort.

CPPUnit hingegen bietet keine Abstufungen an.

5.4 Analyse

Aus den vorgestellten Test-Frameworks soll nun das Vorteilhafteste für den Anwendungsfall LAMA ausgewählt werden. Um einen Vergleich durchführen zu können, müssen zunächst Kategorien identifiziert werden, anhand denen die Eignung der Test-Frameworks untersucht wird. Dabei werden pro Kategorie Punkte verteilt, die daraufhin anhand der Summe der Punkte das Framework zeigen, welches im Gesamtvergleich als Sieger hervorzuheben ist. Je höher die Punktezahl, desto besser ist die vom Framework angebotene Leistung. Die Kategorie für die numerischen Vergleiche erhält im Vergleich zu den anderen Kategorien die meisten Punkte, nämlich 10, um so den Schwerpunkt des Vergleichs auf den numerischen Kontext legen zu können. Die Kategorie "Dokumentation", sowie die Kategorie "Zusätzliche Kriterien" erhalten 3 Punkte, da diese im Vergleich zu den anderen Kategorien nur indirekt für die

konkrete Anwendung des Framework von Interesse sind. Die restlichen Kategorien erhalten 5 Punkte.

5.4.1 Pflichtkriterien

Die drei Frameworks sind ausgewählt worden, da sie die folgenden beiden Pflichtvoraussetzungen erfüllen:

Zum einen muss das Test-Framework für die Sprachen C bzw. C++ einsetzbar sein, da das Projekt LAMA auf diesen Sprachen beruht.

Die Bibliothek LAMA wird plattformunabhängig entwickelt. Diese Bedingung muss auch für das Test-Framework gelten, um so die Teststruktur sehr einfach auf verschiedenen Systemen übersetzten und testen zu können.

Andere Test-Frameworks, wie JUnit oder cfix, welche für die Sprache Java bzw. nur für Windows-basierte Systeme geschaffen ist, scheitern somit an diesen Hürden.

5.4.2 Weitere Kriterien

Im Folgenden werden weitere Kriterien vorgestellt, die anhand der Frameworks für den zukünftigen potentiellen Einsatz bewertet werden.

5.4.2.1 Komfort für Erststellung der Testhierarchie

Die Bedienung und Benutzung eines Test-Frameworks sollte einfach und intuitiv von Statten gehen; dies gilt vor allem für einen Entwickler, der mit diesem Framework noch nicht gearbeitet hat. Die Hauptaufgabe bei dem Schreiben von Testklasen sind die Erstellung von Testcases bzw. Testsuites. Daher sollte bei der Auswahl eines Test-Frameworks Wert gelegt werden, wie hoch der Aufwand der ist, um solche Testkonstrukte zu erstellen.

Das Konzept des Aufbauens einer Testhierarchie, bestehend aus Testsuites und Testcases implementieren alle drei Frameworks. Allerdings ist der Aufwand Testsuites und Testcases in der Praxis zu realisieren unterschiedlich. CPPUnit unterscheidet zwischen einer Headerfile, in der vom Entwickler selbst die Erstellung der Testklasse sowie die Registrierung der Testcases in die Testsuites vorgenommen werden müssen und einer Quelltextdatei, in der separat die einzelnen Testmethoden implementiert werden. Daher ist der Aufwand einen Test neu hinzuzufügen im Vergleich zu Boost.Test und GoogleTest relativ groß. Bei den anderen beiden Frameworks werden Makros angeboten, die die Deklaration der Klasse, der Testmethoden sowie die Registrierung automatisch übernehmen. Lediglich der Name der Testsuite und die Benennung der Testcases sind hier als Parameter den Makros zu übergeben. Die Implementierung der Testmethode folgt in einer dem Makro nachfolgenden Blockanweisung. Der Programmierer wird somit um einige Arbeitsschritte entlastet und kann sich direkt um den wichtigsten Arbeitsschritt, nämlich der Implementierung der Testmethode, widmen. Daher erhalten Boost.Test und GoogleTest die identische und im Vergleich zu CPPunit höhere Bewertung.

Kategorie:	Boost.Test	5/5
"Komfort für Erststellung der Testhierarchie"	CPPUnit	2/5
	GoogleTest	5/5

5.4.2.2 Laufzeitoptionen

Neben dem Aufwand der Erstellung von Teststrukturen, welches somit auch der Bedienung zur Entwicklungszeit gleichkommt, ist ebenfalls die Bedienung zur Laufzeit von Relevanz. Ein Test-Framework sollte dem Benutzer Optionen anbieten, die die Handhabung des Testlaufes vereinfacht, um so schneller und effizienter an ein Testergebnis zu gelangen.

Boost.Test bietet eine Reihe von Parametern an, die zur Laufzeit benutzt werden können, unter anderem die Option "--run_test", die es ermöglicht, einen bestimmten Testcase innerhalb der Testhierarchie aufzurufen. Durch die Benutzung des Symbols * ist es sogar möglich die vorhandenen Testsuites bzw. Testcases nach bestimmten Ausdrücken zu filtern und nur Gruppen von Tests laufen zu lassen, die dieses Kriterium erfüllen. Ein Vorteil ergibt sich logischerweise dadurch, dass die Laufzeit der Tests drastisch reduziert wird, wenn gezielt nur einzelne Tests ausgeführt werden. Das Ausgabeformat kann durch die Option "--report_format" wechselweise auf ein für den Menschen leserfreundliches Format (HRF - Human Readable Format) eingestellt werden oder auch auf XML eingestellt werden. Ein Zufallsgenerator, der eine zufällig Reihenfolge der auszuführenden Tests erzeugt ist durch die Option "--random" ebenfalls integriert, sowie die Möglichkeit Abstufungen der Ausgabenachrichten zu ändern. Dies geschieht durch den Parameter "--log_level", welcher es unter anderem ermöglicht keine Ausgabe der Informationen, eine Ausgabe der gerade ausgeführten Testsuite- bzw. Testcasenamen oder eine Ausgabe aller gesammelten Informationen zu erzeugen.

GoogleTest implementiert ebenfalls die Funktionen einer Testfilterung durch die Option ("--gtest_filter=suite.case"), einen Zufall der Testausführung zu erzeugen ("--gtest_shuffle") oder die Testinformationen als XML-Format auszugeben ("--gtest_output=xml:path"). Die Ausgabe der gesammelten Daten auf der Standardkonsole ist starr gegeben und kann nicht durch Abstufungen geändert werden. Zusätzlich bietet GoogleTest durch den Parameter "--gtest_list_tests" an eine Übersicht der vorhandenen Teststruktur auszugeben. Diese Option erleichtert es für den Entwickler besonders in einer Teststruktur mit vielen implementierten Tests einen bestimmten Testsuite- oder Testcasenamen zu finden bzw. darzulegen mit welchem Namen ein konkreter Test durch die Benutzung der Option „gtest_filter“ überhaupt anzusprechen ist.

Bei CPPUnit sind keine Parameter zu finden (vgl. N.N. 2012b). Boost.Test und GoogleTest bieten in vielen Bereichen Parameter mit identischen Funktionen. Zusätzlich existieren Parameter, die das jeweils andere Framework nicht besitzt. In der Summe liegen diese beiden Frameworks gleich auf.

Kategorie:	Boost.Test	4/5
"Laufzeitoptionen"	CPPUnit	0/5
	GoogleTest	4/5

5.4.2.3 Ausgabeinformationen

Eine ebenfalls wichtige Rolle spielt die Reaktion eines Test-Frameworks auf einen eintretenden Fehlerfall. Ein Test-Framework muss nicht nur feststellen, ob eine Annahme unkorrekt ist, sondern muss auch Informationen ausgeben, die Aufschluss darüber geben, an welcher Stelle im Quellcode dieser Fehlerfall entstanden ist. Nur mit aufschlussreichen Informationen kann der Entwickler den entdeckten Fehler schnell und effektiv finden und beseitigen.

Anhang 10.5 demonstriert je eine Standardausgabe für jedes der Test-Frameworks, die ohne die Benutzung weiterer Laufzeitoptionen, ausgegeben wird. Boost.Test besitzt eine standardmäßig minimal gestaltete Ausgabe, die nur die Basisinformationen eines Testlaufes ausgibt. Darunter fällt die Anzahl der durchlaufenden Tests, Ausgabeinformationen über fehlgeschlagene Annahmen und letztendlich eine kurze Zusammenfassung des Testlaufes. Die Informationen, die dabei für einen Fehlerfall ausgegeben werden umfassen alle wichtigen Angaben, um die Position im Quellcode zu finden, an der die Annahme fehlschlug. Dies sind der Name der Testklasse und des Testcases sowie die Zeilennummer im Quellcode.

CPPUnit gibt ebenfalls noch Informationen über funktionierende Tests mit der Angabe "OK" aus. Daher entsteht zunächst eine Auflistung aller Testmethoden, die aufgerufen worden sind. Bei großen Projekten kann eine solche Ausgabe keine Wirkung entfalten, da fehlgeschlagene Tests einer solchen Auflistung nicht direkt ins Auge fallen. Es erscheint aber eine nummerierte Liste am Ende des Testlaufes, in der diese entstandenen Misserfolge näher erläutert werden. Alle genannten Angaben, die auch von Boost.Test ausgegeben werden, finden sich bei der Ausgabe von CPPUnit wieder. Zusätzlich dazu wird der falsch berechnete Wert ausgegeben, welcher besonders bei mathematischen Berechnungen ein Indiz sein kann, ob eine Berechnung sich nah an dem korrekten und erwarteten Ergebnis befindet, so dass eventuell nur ein Rundungsfehler Ursache für den fehlgeschlagenen Test ist oder ein willkürlich berechneter Wert ausgegeben wird.

GoogleTest hingegen besitzt eine unflexible Ausgabe. Durch jeden Test entstehen grundsätzlich zwei Ausgabezeilen. Die erste Zeile gibt die Information aus, dass ein Test aktuell läuft (run), die zweite Zeile, ob dieser auch korrekt beendet wurde (passed) oder fehlgeschlagen ist (failed). Ein Nachteil ergibt sich durch die Tatsache, dass die Ausgabeinformationen für einen Fehlerfall zwischen diesen beiden zum Test gehörigen Zeilen ausgegeben werden. Bei hunderten von Tests muss der Aufwand betrieben werden, diese Auflistung komplett zu durchschauen, um diese Angaben zu finden, da nach dem Testdurchlauf nur nur die Namen der fehlgeschlagenen Tests ausgegeben werden. Standardmäßig werden die Zeilen eines fehlgeschlagenen Test rot und eines korrekten Test in grün eingefärbt. Am Ende dieser Auflistung, werden, ähnlich wie bei CPPUnit die fehlgeschlagenen Test noch einmal aufgelistet ausgegeben. Die Möglichkeit die Ausgabe durch die Verwendung von Laufzeitparametern zu modifizieren, existiert nicht.

Kategorie:		
"Ausgabeinformationen"	Boost.Test	5/5
	CPPUnit	4/5
	GoogleTest	3/5

5.4.2.4 Softwareaktualität

Wartung und zukünftige Aktualisierungen eines Test-Framework können ebenfalls ausschlaggebend sein, ob ein Framework in der Entwicklung eines Projektes eingesetzt wird, oder nicht. In der Regel wird Software fortlaufend entwickelt, dies gilt somit auch für Komponenten des benutzten Betriebssystems, oder für Werkzeuge, die für ein Softwareprojekt von Nöten sind, wie zum Beispiel dem Compiler. Eine regelmäßige Aktualisierung des Test-Framework stellt somit sicher, dass somit die Kompatibilität zwischen dem Test-Framework und den verwendeten Softwareprodukten sichergestellt ist.

Boost ist eine Sammlung von verschiedenen Bibliotheken, die mehrfach im Jahr aktualisiert wird. Die letzte Aktualisierung wurde im Juni 2012 veröffentlicht. Dies bedeutet zwar nicht, dass unbedingt auch Boost.Test, als Teil dieser Sammlung, in der jeder Veröffentlichung einer Aktualisierung unterzogen wird, aber eine generelle

Bereitschaft des Entwicklerteams von Boost bei entdeckten Fehlern, diese auch zu finden und zu entfernen ist gegeben. Die letzte relevante Versionsänderung von CPPUnit hingegeben liegt schon über vier Jahre zurück, nämlich Anfang 2008 (vgl. Lepilleur 2012a, "Files - cppunit"). Für eine Entwicklung eines Softwareproduktes ist dies, relativ gesehen, eine lange Zeit. Daher ist eine Aussage über die zukünftige Weiterentwicklung dieser Version des Frameworks schwer zu treffen. Die Entwicklung von CPPUnit 2 hat zwar begonnen, allerdings stagniert diese und ein mögliches Release ist nicht abzusehen. Bei GoogleTest sieht dies zwar leicht besser aus, aber auch hier erscheint seit fast anderthalb Jahren keine neue Version; die letzte wurde im April 2011 veröffentlicht, obwohl die Entwicklung geprägt durch regelmäßige Veröffentlichungen neuer Versionen in den Jahren 2008 bis 2010 blühte (vgl. Google 2012, "Downloads").

Kategorie:	Boost.Test	4/5
"Softwareaktualität"	CPPUnit	1/5
	GoogleTest	2/5

5.4.2.5 Dokumentation

Eine kleinere Rolle für den Einsatz eines Test-Frameworks spielt ein Vorhandensein einer Dokumentation. Eine solche Dokumentation hängt nur indirekt mit der Benutzung des Frameworks zusammen, ist aber nützlich, um ein fehlendes oder vergessenes Wissen aufzufrischen. Alle drei Test-Frameworks bieten hierbei ausführliche Dokumentationen in Form von einem "Cookbook" (vgl. N.N. 2012b), einer Online-Dokumentationen (vgl. Rozenthal 2012) oder einem Wiki (vgl. Google 2012, "Wiki") an. Der Punkteunterschied entsteht durch die Ausführlichkeit der vorhandenen Dokumentationen.

Kategorie:	Boost.Test	3/3
"Dokumentation"	CPPUnit	2/3
	GoogleTest	3/3

5.4.2.6 Vergleichsmöglichkeiten von numerischen Werten

Das Ziel des Vergleiches ist das Finden eines Test-Framework, welches in einem numerischen Kontext eingesetzt wird. Dies bedeutet, dass die nötigen Möglichkeiten von Seiten des Frameworks existieren müssen, numerische Werte vergleichen zu können, um so die Korrektheit von Berechnungen feststellen zu können. Eine Übersicht über eine Auswahl der von den Frameworks angebotenen Assertion befindet sich im Anhang dieser Arbeit (Anhang 10.4).

Gemeinsam unterstützen alle drei Frameworks die Überprüfung ob eine Bedingung den boolschen Wert true zurückliefert, den Vergleich von zwei numerischen Werten auf Gleichheit und die Möglichkeit, ob die Differenz zweier Fließkommazahlen innerhalb eines Zahlenbereiches (delta) liegt. Des Weiteren ist die Unterstützung gegeben, einen aufgerufenen Ausdruck auf eine geworfene Exception zu überprüfen. Dies ist wertvoll für Situationen während nicht erlaubten mathematischen Berechnungen, die durch das Werfen einer Exception abgebrochen werden. Auch in der vorhandenen Teststruktur von LAMA können solche Ausnahmesituationen auftreten, deren Exceptions abgefangen werden müssen, ohne dass der gesamte Testlauf durch diese eine Exception abbricht.

Mathematische Berechnungen funktionieren für viele Kombinationen von numerischen Werten. Allerdings kann es passieren, dass genau in einem von vielen Fällen eine Ausnahme geschieht, wie beispielsweise eine Entstehung einer Division durch den

Wert Null. Einen Test für alle erdenklichen Werte durchzuführen ist unmöglich. Abhilfe schafft aber das Durchlaufen des Tests mit ein paar definierten Werten, die kritisch sein können. Von GoogleTest wird dies unterstützt. Es existiert die Parametrisierung eines Tests. Dabei wird ein Test mit definierten Werten instanziiert und die Möglichkeit gegeben, durch den Aufruf der Methode GetParam() diese festgelegten Werte im Testlauf zu erhalten und mit ihnen die Berechnung durchzuführen. Der Test wird solange automatisch wiederholt, bis alle festgelegten Werte durchlaufen worden sind.

Ein Vorteil, der von Boost.Test unterstützt wird, ist das Templatisieren von Testcases. So lässt sich mit wenig Aufwand ein Test schreiben, der mit verschiedenen Datentypen, wie z.B. float und double, durchlaufen werden kann. Anhang 10.1 demonstiert dies anhand des Testcases "AdditionTest" der Klasse ScalarTest.

Häufig werden auch mathematische Objekte, wie Vektoren oder Matrizen berechnet, die nichts anderes sind, als Felder mit Werten. Boost.Test bietet durch ein Makro an, genau diese Felder mit Erwartungswerten, die ebenfalls in einem Feld gespeichert sind, zu vergleichen. Dies geschieht durch die Assertion BOOST_<level>_EQUAL_COLLECTION. Der Entwickler ist somit nicht gezwungen Schleifen für diese Felder schreiben zu müssen, die besonders bei mehrdimensionalen Feldern komplex werden können.

Sowohl Boost.Test als auch GooleTest bieten darüber hinaus Makros für die mathematischen Vergleiche "kleiner als", "echt kleiner", "größer als" und "echt größer" an. CPPUnit unterstützt diese Vergleiche nicht direkt, sondern nur über den Umweg durch einen boolschen Ausdruck und der Benutzung der Assertion CPPUNIT_ASSERT.

Vergleichend kann gesagt werden, dass CPPUnit erneut nicht mit den beiden anderen Frameworks mithalten kann. Die Basisfunktionen, die von einem Test-Framework erwartet werden, sind zwar vorhanden, aber für den komplexen numerischen Kontext im Projekt LAMA ist dies nur bedingt ausreichend. Tests fürs verschiedene Datentypen müssen redundant für jeden Datentyp geschrieben und für Vektoren oder Matrizen müssten komplexe Schleifen geschrieben werden, die diese mathematischen Felder durchlaufen lassen. Boost.Test und GoogleTest bieten einen großen Funktionsumfang an, der auch ergänzende Möglichkeiten, wie das Templatisieren oder Parametrisieren besitzt. Besonders bei einer großen Anzahl an Tests reduziert sich dadurch die Zeit, die für das Schreiben von Tests aufgewendet wird.

Kategorie:	Boost.Test	8/10
"Vergleichsmöglichkeiten von numerischen Werten"	CPPUnit	4/10
	GoogleTest	8/10

5.4.2.7 Zusätzliche Kriterien

Als Vorraussetzung für die Verwendung von LAMA müssen einige externe Bibliotheken auf dem Zielsystem installiert sein. Darunter fällt bereits Boost. LAMA verwendet, neben einem potentiellen Einsatz von Boost.Test als Test-Framework, bereits andere Teilbibliotheken von Boost, wie zum Beispiel Boost.Program_Options oder Boost.Thread. Ein Vorteil resultiert dadurch, dass Boost somit schon lauffähig gegeben sein muss und ein Linken gegen eine weitere externe Bibliothek entfällt.

Des Weiteren sind alle drei Framework kostenfrei erhältlich.

Kategorie:	Boost.Test	2/3
"Zusätzliche Kriterien"	CPPUnit	1/3
	GoogleTest	1/3

5.5 Auswahl

Kriterium	Boost.Test	CPPUnit	GoogleTest	Maximalpunkte
Komfort für Erststellung der Testhierarchie	5	2	5	5
Laufzeitoptionen	4	0	4	5
Ausgabeinformationen	5	4	3	5
Softwareaktualität	4	1	2	5
Dokumentation	3	2	3	3
Vergleichsmöglichkeiten von numerischen Werten	8	4	8	10
Zusätzliche Kriterien	2	1	1	3
Summe	**31**	**14**	**26**	**36**

Abbildung 8: Bewertung des Test-Framework-Vergleichs

Die in dem vorherigen Kapitel vorgestellten Vergleichskriterien sind in Abbildung 8 übersichtlich zusammengefasst. CPPUnit belegt den letzten Platz mit 14 Punkten. Dies liegt unter anderem an dem Stagnieren der Entwicklung der aktuellen, als auch der angekündigten zweiten Version. Boost.Test liegt mit 31 Punkten knapp vor GoogleTest, welches 26 Punkte erzielte. Beide Frameworks überzeugen mit ihrem mitgelieferten Funktionsumfang. Allerdings bieten beide Frameworks Funktionen, die das jeweils andere nicht besitzt. Boost.Test bietet das einfach umzusetzende Templatisieren von Testcases an; GoogleTest hingegeben das Parametrisieren. Die Auswahl des Test-Framework Vergleichs fällt daher auf Boost.Test. Eine Installation von Boost.Test bleibt im Projekt LAMA erspart, da Boost als Bibliothekssammlung schon in der Standard Linuxkonfiguration von SCAI vorhanden ist.

6 Qualität des Codes – CodeCoverage

Eine Möglichkeit, um die Qualität von Teststrukturen nachzuweisen, ist die Durchführung einer Codeabdeckungsanalyse. Die Grundlagen und nötigen Anweisungen für die Benutzung von einem benötigten Werkzeug werden in diesem Kapitel dargelegt. Die konkrete Durchführung der Analyse von LAMA und Interpretation der daraus entstandenen Ergebnisse werden im siebten Kapitel präsentiert.

6.1 Codeabdeckung

Mit Testklassen kann sichergestellt werden, dass die Funktionalität einzelner Methoden einer Klasse überprüft werden. Allerdings ist dies kein Indiz dafür, dass auch tatsächlich alle Methoden einer Klasse durch einen dazugehörigen Test aufgerufen worden sind. Insbesondere ist es für einen Entwickler schwierig, bei Klassen mit einer hohen Anzahl an Methoden, die Vollständigkeit der dazugehörigen Testklasse sicherzustellen.

Die Codeabdeckung (engl. code coverage) ist ein Faktor, der die Testvollständigkeit, basierend auf den Kontrollstrukturen und Kontrollflüssen eines Programms, beschreibt und anschließend Programmteile, die nicht durch Tests aufgerufen worden sind, ausweist. Abdeckungstests lassen sich in die kontrollflussorientierten und somit strukturorientierten Testtechniken einordnen (Liggesmeyer 2002, 80f.).

Die Basis für die Codeabdeckung ist der Kontrollflussgraph, welcher die Ablaufstruktur eines Programms darstellt. Dieser ist ein gerichteter Graph, deren Knoten einzelnen Basisblöcken eines Programms und deren Kanten - oder auch Zweige genannt - den einzelnen Übergängen von einem Block zum anderen entsprechen. Ein Weg vom Anfangsknoten bis zum Endknoten wird auch als Pfad zu bezeichnet (Liggesmeyer 2002, S. 257f. und Sneed/Winter 2002, S. 141f.).

Überdeckungstests lassen sich in verschiedene Arten unterteilen: Anweisungsüberdeckungstests, Zweigüberdeckungstests und Bedingungsüberdeckungstests. Diese sollen im Folgenden kurz erläutert werden, um so die theoretische Grundlage für die durchgeführte Abdeckungsanalyse zu bilden.

Anweisungsüberdeckungstests (engl. statement coverage test) dienen zur Sicherstellung der Abdeckung der Knoten eines Kontrollflussgraphen. Das bedeutet, dass das optimale Ziel erreicht ist, wenn durch den Aufruf einer Teststruktur alle Anweisungen eines Programms mindestens einmal ausgeführt worden sind. Nicht ausführbare Befehle können identifiziert werden. Der Anweisungsüberdeckungsgrad wird wie folgt berechnet (siehe Abbildung 9).

$$C_{Anweisung} = \frac{Anzahl\ der\ ausgeführten\ Anweisungen}{Anzahl\ der\ Anweisungen}$$

Abbildung 9: Formel für Anweisungsüberdeckungsgrad (Liggesmeyer 2002, S. 81)

Das Ziel der Zweigüberdeckungstests (engl. branch coverage test) ist es, dass alle Zweige mindestens einmal ausgeführt werden. Auf den Kontrollflussgraphen bezogen müssen alle Kanten durchlaufen sein. Dies heißt implizit, dass der

Zweigüberdeckungstest bewirkt, dass automatisch alle Knoten durchlaufen werden müssen und somit den Anweisungsüberdeckungstest vollständig abdeckt. Der Nachteil dieses Testtyps ist es jedoch, dass es problematisch sein kann, für alle auszuführenden Zweige Testfälle zu erstellen. Dies kann auch auf der Existenz eines Zweiges beruhen, der bspw. durch einen Entwurfsfehler gar nicht ausführbar sein kann. Auch wenn eine maximale Abdeckung des Zweigüberdeckungstests existiert, heißt dies jedoch nicht, dass das Programm auch fehlerfrei funktioniert. Dies liegt unter anderem daran, dass beispielsweise Schleifen und komplexe Ausdrücke von Bedingungen nicht in allen Kombinationen getestet werden. Der Zweigüberdeckungsgrad wird wie folgt berechnet (siehe Abbildung 10), wobei ein nicht primitiver Zweig in einem Kontrollflussgraphen genau dann gegeben ist, wenn er von anderen Zweigen abhängig ist:

$$C_{primitiv} = \frac{Anzahl\ ausgeführter\ primitiver\ Zweige}{Anzahl\ primitiver\ Zweige}$$

Abbildung 10: Formel für Zweigüberdeckungsgrad (Liggesmeyer 2002, S. 87)

Bedingungsüberdeckungstests (engl. condition coverage tests) testen Fälle von zusammengesetzten Entscheidungen, wie dies bei bedingten Anweisungen beispielsweise der Fall ist. Die einfachste Möglichkeit ist, es alle atomaren Bedingungen – das sind solche Bedinungen, die nicht aus Unterentscheidungen bestehen - jeweils gegen wahr und falsch zu testen. Laut Liggesmeyer existieren weitere Kategorien von Bedingungsüberdeckungstests, wie zum Beispiel der minimale oder der modifizierte Mehrfachüberdeckungstest. Bedingungstests sollen für diese Arbeit nur zur Vollständigkeit erwähnt sein und werden aus zeitlichen Gründen nicht weiter betrachtet.

6.2 Einsatzgebiete für eine Analyse der Codeabdeckung

Für die Qualitätssicherung des Codes eines Softwareprojektes finden sich somit zwei wesentliche Einsatzgebiete, in denen Codeabdeckung von Relevanz ist.

Der erste Bereich, indem eine Analyse der Codeabdeckung Vorteile erbringt, ist die Abdeckung der Testklassen. Der Entwickler erhält ein Feedback, dass tatsächlich auch jede Testmethode aufgerufen worden ist. Erst die Tatsache, dass jede Testmethode aufgerufen worden ist, ist somit die Grundlage dafür, dass alle Methoden der Bibliothek durch die Testausführung überhaupt aufgerufen werden können. Falls dennoch Methoden oder Anweisungen der Testklassen nicht ausgeführt wurden, so lässt sich rückschließen, dass entweder der implementierte Test nicht korrekt arbeitet und ausgeführt wird oder in diesem Test beispielsweise Stellen von "totem Code" entstanden sind.

Unter dieser Voraussetzung kann man sich dem zweiten Gebiet widmen, nämlich der Codeabdeckung der Bibliotheksklassen. Das Ziel hierbei ist es nachzuweisen, dass alle entwickelten Methoden bzw. um genauer zu sein Anweisungen aufgerufen und somit getestet werden. Obwohl sichergestellt ist dass eine Methode aufgerufen wurde, kann man dennoch nicht immer sicher sein, dass diese Methode auch fehlerfrei funktioniert; dies liegt daran, dass es in großen Projekten zu aufwendig, wenn nicht gar unmöglich ist, alle erdenklichen Testszenarien mit allen unterschiedlichsten Testparametern durchzuführen. Allerdings kann die Fehlerrate einer Methode durch die

wenigen tatsächlich durchgeführten Tests deutlich reduziert werden. Es ist immer besser eine Methode wenigtens einmal aufgerufen zu haben und somit getestet zu haben, da nämlich keine Aussage über die Funktionsfähigkeit einer Methode getroffen werden kann, wenn diese nie aufgerufen wurde.

6.3 Vorstellung von Gcov und Lcov

6.3.1 Vorstellung von Gcov

Gcov ist ein Softwarewerkzeug, welches als Profiler-Tool kategorisiert werden kann und Teil der kostenlosen GNU Compiler Collection (GCC) ist. Als Teil von GCC ist Gcov in jeder Linuxdistribution zu finden und ist dementsprechend weit verbreitet. Dies erleichtert die Benutzung auf unterschiedlichen Rechenmaschinen, da die Anwendung identisch ist und ein Zusammenspiel mit allen weiteren GCC-Werkzeugen gegeben ist. Allgemein dient Gcov dazu, ein Programm zu analysieren, um Stellen im Quellcode zu entdecken, die optimiert werden können. Dabei wird unter anderem aufgezeigt, ob eine Zeile überhaupt durchlaufen worden ist bzw. wie hoch die Anzahl dieser Durchläufe ist (GCC 2012a).

Als Voraussetzung zum Ausführen von Gcov muss das Programm, welches zu analysieren ist mit den Flags "-fprofile-arcs" und "-ftest-coverage" kompiliert werden. Beim Übersetzen des Programms werden dadurch *.gcno-Dateien kreiert, die Daten enthalten, um Zeilennummern des Quellcode einzelnen Basisblöcken zuzuordnen. Durch die anschließende Ausführung des Programms werden *.gcda Dateien erstellt, die Informationen über die Anzahl der ausgeführten Zweige eines Kontrollflussgraphen beinhalten. Der anschließende Aufruf von Gcov mit den Quelldateien als Parameter erzeugt pro Quelldatei je eine *.gcov-Datei, die die aktuelle Codeabdeckung enthält (GCC 2012b).

Diese Ausgabe der gesammelten Informationen wird jedoch, besonders bei einer großen Anzahl von Quelldateien, für einen Menschen unübersichtlich und damit schwer zu analysieren.

6.3.2 Benutzung von Lcov

Abhilfe schafft dabei eine Erweiterung für Gcov, namens Lcov. Aus den erstellten *.gcno und *.gcda-Dateien sammelt Lcov die Daten der Abdeckung. Das mit Lcov mitgelieferte Tool genhtml erzeugt daraus eine HTML-Ausgabe, welche durch Hyperlinks einen schnellen Ansichtswechsel zwischen verschiedenen analysierten Ordnern und Quelldateien bietet. Des Weiteren werden diese Ergebnisse eingefärbt, so dass ein schlechtes Analyseergebnis und somit eine geringe Abdeckung eines Ordners oder einer Klasse dem Entwickler schneller ins Auge fällt (LTP 2012a und LTP 2012b).

```
lcov --directory <dir> --zerocounters
lcov --directory <dir> --capture --output-file output.info
genhtml output.info
```

Abbildung 11: Benutzung von Lcov (nach LTP 2012b)

Abbildung 11 zeigt die Reihenfolge und Anwendung der Befehle, um eine simple Analyse mit Hilfe von Lcov durchzuführen.

Der Ausdruck der ersten Zeile bewirkt, dass die möglicherweise durch ältere Analysen vorhandenen Zählerstände in der angegebenen Ordnerstruktur durch Lcov rekursiv zurückgesetzt werden. Anschließend wird Lcov für die Codeabdeckungsanalyse ausgeführt und die Ergebnisse in eine Ausgabedatei geleitet. Diese Ausgabedatei ist die Quelle für die HTML-Struktur, welche mit dem Werkzeug genhtml erstellt wird. In Kapitel 7.4 wird die Codeabdeckung von LAMA mit Hilfe von Lcov dargelegt und erläutert, wie durch diese Erkenntnisse die Teststruktur verbessert werden kann.

7 CI-Konzeption für Anwendung in LAMA

Dieses Kapitel beschreibt die Integration und Verwendung der auserwählten Werkzeuge in die Entwicklung von LAMA. Dabei wird zunächst die Bibliothek LAMA vorgestellt, der aktuelle Stand der laufenden Entwicklung beschrieben und anschließend das entwickelte CI-Konzept erläutert. Die Schritte, um dieses Konzept nun zu vollendet einzurichten, wie beispielsweise der Aufbau der Teststruktur, dem Einrichten des CI-Servers und der Möglichkeit von diesem Server aus ein Feedback zu versenden, werden dargestellt. Anschließend wird erläutert, ob dieses Konzept tatsächlich die aus dem zweiten Kapitel dieser Arbeit angeforderten Eigenschaften erfüllt.

7.1 Das Projekt LAMA

Im Fraunhofer Institut für Algorithmen und Wissenschaftliches Rechnen SCAI in Sankt Augustin wird in der Arbeitsgruppe High Performance Computing (HPC) seit circa zwei Jahren die Library for Accelerated Mathematical Applications (LAMA), basierend auf den Sprachen C und C++, entwickelt. Diese quelloffene und hoch performante Bibliothek bietet dem Benutzer Funktionen aus dem Bereich der linearen Algebra an, um möglichst effektiv numerische Problemstellungen sowohl auf einfachen Rechenmaschinen, als auch auf großen Computerclustern lösen zu können.

Abbildung 12: Logo Projekt LAMA

Viele Studenten waren und sind Teil des Entwicklerteams und bereichern das Projekt unter anderem durch ihre Abschlussarbeiten, die im mathematischen und naturwissenschaftlichen Sektor geschrieben werden. Voraussichtlich im Oktober 2012 wird die Entwicklung so weit fortgeschritten sein, dass das erste Release von dieser Bibliothek veröffentlicht wird. Es existieren zwei Quellen, die wichtige Informationen über LAMA darlegen: Dies ist zum einen die offizielle Internetpräsenz (Fraunhofer SCAI 2012a) und ein Wiki, in dem die wichtigsten Funktionalitäten von LAMA erläutert werden (Fraunhofer SCAI 2012b).

7.2 Aktueller Entwicklungsstand

Das Konzept von LAMA sieht die Unterstützung von numerischen Berechnungen auf verschiedenen Hardwarearchitekturen vor. So werden zur Zeit, neben der Durchführung von Berechnungen auf der CPU auch Anbindungen an verschiedene Schnittstellen entwickelt, wie zum Beispiel CUDA (vgl. Nvidia 2012), dass Berechnungen auf der GPU ermöglicht.

LAMA unterstützt sowohl sequentielle Berechnungen, als auch eine Möglichkeit, diese parallel auszuführen. Durch eine Verwendung der MPI-Schnittstelle (vgl. ANL 2012) können einzelne Schritte einer Berechnung auf verschiedene CPUs verteilt werden und so ein schnelleres Ergebnis erzielt werden. Zur weiteren Optimierung werden durch die Verwendung von Direktiven im Quellcode Schleifen parallelisiert, welches durch die Nutzung von OpenMP (vgl. OpenMP ARB 2012) realisiert wird.

Besonders effizient werden Berechnungen in LAMA für dünnbesetzte Matrizen durchgeführt. Dabei werden nur die Stellen einer Matrix in einem Speicherobjekt (engl. storage) gespeichert, die nicht den Wert 0 besitzen. Bei einer großen Dimension einer Matrix bleibt auf diese Weise die unnötige Belegung von Ressourcen in Form von

Speicherplatz erspart. LAMA unterstützt verschiedene Speicherformate: Compressed Sparse Row Storage Format (CSR), ELLPACK Storage Format, Jagged Diagonal Storage Format, Diagonal Storage Format und Coordinate Storage Format. Die Möglichkeit eine vollbesetzte Matrix zu erstellen, die somit alle Stellen – inklusive aller Nullen – speichert, ist ebenfalls durch die Klasse DenseMatrix gegeben. Das letzt genannte Prinzip wird auch bei Vektoren angewandt, so dass aktuell nur eine Klasse von Vektor, nämlich vom Typ DenseVector existiert, die instanziiert werden kann.

Der Schwerpunkt der praktischen Verwendung von LAMA ist das Lösen von linearen Gleichungssystemen. Verschiedene Verfahren, um ein solches System näherungsweise zu bestimmen, sind in sogenannten Solver-Klassen implementiert. Bekannte mathematische Verfahren, wie das Jacobi, das SOR oder CG-Verfahren existieren und können je nach Problemstellung verwendet werden.

7.3 Continuous Integration Konzept für LAMA

Das Continuous Integration Konzept kann nun in die Entwicklung von LAMA integriert werden. Die nun festgelegten Softwarewerkzeuge sind in Abbildung 13 dargestellt. Als Versionsverwaltung wird das Werkzeug Subversion eingesetzt; als CI-Server dient Jenkins. Boost.Test und Cmake müssen auf den lokalen Maschinen der Entwickler sowie auf der Maschine, auf die der CI-Server läuft, installiert sein. Die Unterstützung durch den CI-Server den Status des aktuellen Builds an die Entwickler weiterzugeben, soll durch Versendung von E-Mails eingerichtet werden.

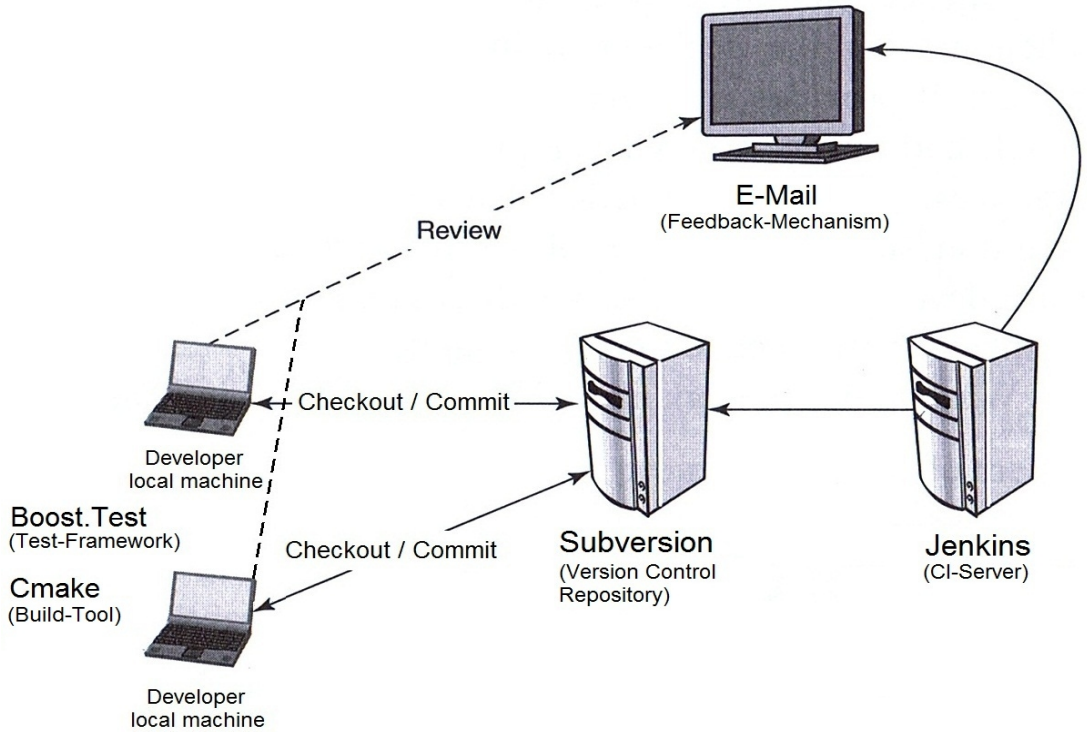


Abbildung 13: CI-Konzept für LAMA (nach Duvall 2011, S.26)

7.3.1 Aktueller Stand des CI-Konzeptes

Es ist zu beachten, dass die Entwicklung der Bibliothek LAMA schon fortgeschritten ist und das CI-Konzept nicht zu Beginn der Entwicklung aufgesetzt worden ist, sondern erst jetzt im Nachhinein Einzug erhält. Die Auswahl und Einrichtung einer

Versionsverwaltung ist nicht notwendig, da Subversion im Projekt LAMA schon seit den Anfängen der Entwicklung verwendet wird. Für das automatisierte Bauen der Bibliothek LAMA ist das Softwarewerkzeug Cmake eingerichtet.

Die noch umzusetzenden Schritte sind das Einrichten von Boost.Test als Test-Framework und Jenkins als CI-Server. Zur Optimierung der Teststrukturen wird eine Codeabdeckungsanalyse mit Hilfe des Werkzeugs Lcov durchgeführt. Der Feedback-Mechanismus wird durch das Versenden von E-Mails durch den CI-Server realisiert.

7.3.2 CI Server

7.3.2.1 Einrichten des CI-Servers

Jenkins ist als CI-Server auf ein Unix basiertem System (Ubuntu 12.04) installiert worden. Dies geschieht effektiv und benutzerfreundlich, da eine passende Installation von Jenkins in der Version 1.479 durch den Paketmanager angeboten wird.

Für eine benutzerfreundliche Handhabung des CI-Servers steht eine webbasierte Benutzeroberfläche bereit, die durch die Adresse „localhost:8080“ in einem beliebigen Webbrowser angesprochen werden kann. Die Aufgaben, die von Jenkins auf einem Projekt angewandt werden sollen, werden in einem Job gebündelt. Hier werden alle Einstellungen und Konfigurationen, die dieses Projekt betreffen, vorgenommen.

Damit Jenkins in der Entwicklung der LAMA-Bibliothek optimal genutzt werden kann, müssen zwei Plugins installiert werden. Zum einem ist dies das Plugin „cmakebuilder“, welches die Möglichkeit bietet, den Übersetzungsprozess auf einem Cmake-basierten Projekt zu starten. Zum anderen wird, unter anderem auch von Fowler, geraten die vorhandenen Testläufe einer Softwareentwicklung nach einem Übersetzungsvorgang zu durchlaufen. Dies kann durch Shellaufrufe nach dem Übersetzungsvorgang realisiert werden. Dafür wird verlangt, dass das Plugin „postbuild-task“ vorhanden ist.

In der Konfiguration des für LAMA erstellten CI-Jobs, wird das Aktivieren des aktuell benutzten Source Code Management Tool verlangt. Subversion ist für diesen Job die richtige Auswahl und fordert daraufhin die Angabe der Adresse des Repositories, mit der ein Checkout der aktuellen Revision durchgeführt werden kann. Das Repository kann unter der Adresse https://libama.svn.sourceforge.net/svnroot/libama angefordert werden.

Der nächste wichtige Block der Einstellungen ist die Festlegung eines Auslösers für den Übersetzungsvorgang. Da täglich mehrere Commits an das Repository von LAMA übertragen werden, ist es sinnvoll, die Option "Source Code Management abfragen" zu wählen. Diese überprüft nach durch ein anzugebenes Zeitintervall, ob eine Versionsänderung im Repository vorliegt. Ist dies der Fall, so wird die Bibliothek LAMA ausgecheckt und der Übersetzungsvorgang gestartet. Weiterhin wird die Möglichkeit geboten den Bauvorgang manuell jederzeit starten zu können.

Jenkins integriert die Verwendung von diversen Softwaretools, die den Übersetzungsvorgang automatisieren. Das Plugin "cmakebuilder" bietet die Möglichkeit für ein Softwareprojekt, in dem Cmake verwendet wird, den Bauvorgang durchzuführen ohne dass hierzu ein selbstgeschriebenes Skript notwendig ist. Es sind lediglich die Angaben für den Source- und Buildordner von Nöten und gegebenenfalls auch die Angaben von Argumenten, die für den Aufruf des Übersetzungsvorganges übergeben werden müssen. Für LAMA bedeutet dies, dass die Angaben der Argumente (Abbildung 14) auf dieser Maschine notwendig sind, da einerseits Schnittstellen, wie

etwa OpenCL oder CUDA hier nicht vorhanden sind und andererseits das Buildverzeichnis der Boost-Library durch Cmake nicht automatisch erkannt wird.

```
-DBOOST_ROOT=/home/user/boost_1_49_0/build
-DLAMA_USE_CUDA=OFF
-DLAMA_USE_OPENCL=OFF
```

Abbildung 14: Argumente für Übersetzungsvorgang

Das Plugin „postbuild-task“ erweitert den gesamten Übersetzungsvorgang, in dem Shellaufrufe nach dem Bauen möglich sind. Das Durchlaufen der vorhandenen Teststruktur kann so automatisch nach dem Bauen gestartet werden. Um nach dem Durchlaufen die Testergebnisse im CI-Server anzeigen zu können, empfiehlt sich die Benutzung eines weiteren Plugins namens „xUnit“. Dieses unterstützt die Darstellung von Testausgaben verschiedenster Test-Frameworks, wie zum Beispiel CPPUnit oder Boost.Test. Im Falle von Boost.Test müssen diese Ausgaben im XML-Format in eine Datei geleitet werden. Die Laufzeitparameter von Boost.Test „log_sink=file“ und „log_format=XML“ bieten die Unterstützung, um die Loginformationen im XML-Format direkt in eine Datei zu leiten, anstatt diese auf der Standardausgabe auszugeben. Diese XML-Datei wird vom Plugin „xUnit“ eingelesen, interpretiert und die Ergebnisse der Testläufe im CI-Server als tabellarische Auflistung angezeigt. Es empfiehlt sich die existierenden XML-Dateien, vor einem erneuten Starten des Buildvorgangs des CI-Servers zu löschen, um keinen veralteten Stand der Testergebnisse vor dem erneuten Einlesen der Dateien vorliegen zu haben. Das Löschen der XML-Dateien, sowie das Starten mit den notwendigen Laufzeitparametern des Testlaufes für die seriellen Unittests wird durch Abbildung 15 dargestellt. xUnit summiert die in jedem Testlauf die fehlgeschlagenen Testscases und kann auf Grundlage dieser Angabe den aktuellen Build als „fehlgeschlagen“, „instabil“ oder „erfolgreich“ einstufen. Zur ersten Einrichtung des CI-Servers wurden eine Grenze von maximal 10 fehlgeschlagenen Testcases für einen erfolgreichen Build eingestellt und eine Grenze von 20 für einen instabilen Build. Eine noch höhere Anzahl lässt den Build fehlschlagen. Für die aktuelle Entwicklungsphase sollen diese Werte ausreichen. Je näher das erste Release rückt, desto eher werden diese Werte reduziert, um so das Entwicklerteam anzuhalten die Testläufe möglichst fehlerfrei zu halten.

```
cd libama/trunk/build/test

if [ -f output_lama_test.xml ]; then
rm output_lama_test.xml
fi

./lama_test --log_sink=output_lama_test.xml --log_format=XML
```

Abbildung 15: Shellanweisungen für Aufrufe des seriellen Testlaufes

Abschließend wird der Feedback-Mechanismus eingebaut. Dieser soll in Form von E-Mails über die Ergebnisse der Buildvorgänge berichten. In der Konfiguration von Jenkins wird die Angabe des SMTP-Servers sowie die Zugangsdaten des E-Mailfachs angegeben. In den jeweiligen Jobs werden die E-Mailadressen eingetragen, an die die Ergebnisse der Builds verschickt werden.

7.3.2.2 Erkenntnisse der Verwendung

Der CI-Server deckt viele Schritte für eine Fertigstellung einer Software ab. Darunter fällt unter anderem das Erstellen einer Arbeitskopie vom Server und den automatisierten Übersetzungsvorgang sowie der Unittests. Die Ergebnisse der Testläufe können qualitative Aussagen über die aktuelle Version des übersetzten Projektes liefern.

Jenkins stellt auf der Übersichtsseite eines Jobs eine Zeitleiste dar, die die Zeitpunkte der jeweiligen Builds darstellt. Die Bezeichnungen der Builds sind farblich gekennzeichnet, um einen fehlgeschlagenen (rot), instabilen (gelb) oder erfolgreichen (blau) Build zu charakterisieren. Neben dieser Zeitleiste wird ein Diagramm erstellt, welches die Dauer der verschiedenen Builds aufzeichnet (Abbildung 16). Dieses zeigt die Entwicklung der Build-Dauer für die aufgezeichneten Übersetzungsvorgänge von LAMA. Auf der X-Achse des Diagramms ist Buildnummer zu finden, auf der Y-Achse ist die benötigte Übersetzungszeit aufgetragen. Die Farben lassen wieder auf einen fehlgeschlagenen, instabilen oder erfolgreichen Build rückschließen. Zu erkennen ist, dass die Dauer von 15 Minuten beim ersten Build deutlich höher ausfällt. Dies liegt daran, dass bei diesem Build ein vollständiger Checkout durchgeführt wurde und daher alle Quelldateien neu kompiliert und gelinkt werden müssen. Bei allen kommenden Builds wurde auf Veränderungen von Dateien geprüft und dabei nur die modifizierten Dateien übertragen. Cmake erkennt diese modifizierten Quelldateien und kompiliert nur diese neu. Dadurch, dass auf der Integrationsmaschine nicht entwickelt wird, sollten keine Konflikte zwischen Dateien auf dem Server und der Maschine entstehen, die dieses Update der Arbeitskopie unterbrechen sollte.

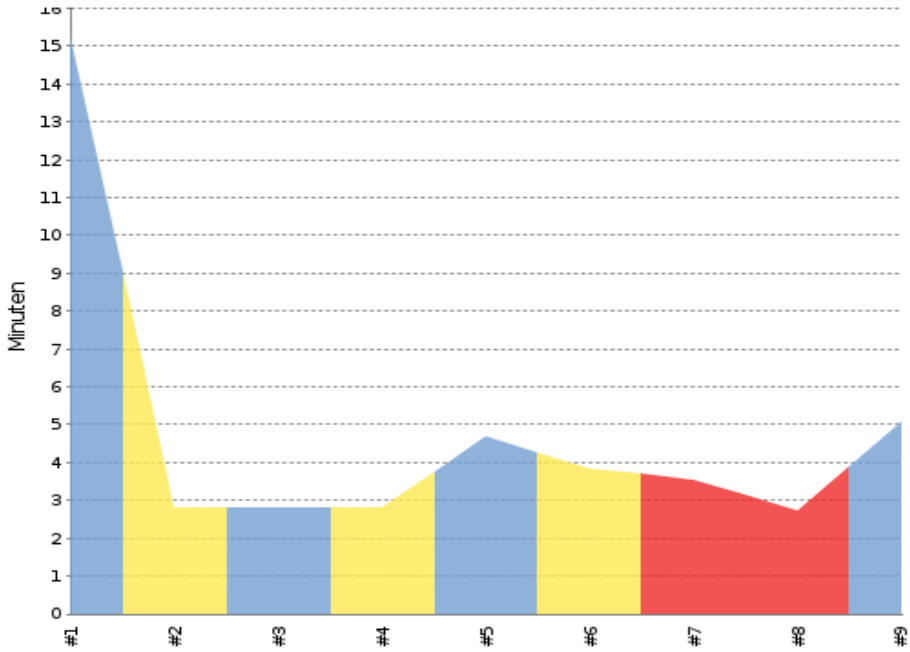


Abbildung 16: Entwicklung der Build-Dauer in Jenkins

Durch das Plugin „xUnit“ wird ein weiteres Diagramm erstellt, durch das die Ergebnisse der Testläufe verglichen werden können (Abbildung 17). Aufgetragen sind hierbei die Anzahl der durchgeführten Tests (blau), welche bei LAMA fast die Marke von 350 erreichen, sowie die Anzahl der, von der Gesamtzahl gesehen, fehlgeschlagenen Tests (rot). Die deutliche Steigerung des ersten zum zweiten Build erklärt sich dadurch, das erst nachträglich der Testlauf für die parallelen Tests in die Konfiguration des CI-Servers integriert worden ist. Für die Zukunft wird die Anzahl der Tests weiter steigen,

da Unittests für die Verwendung von CUDA hinzukommen werden. Einerseits ist durch das Diagramm das Verhältnis der fehlgeschlagenen Tests zu den jeweils durchgeführten Tests eines Builds möglich und andererseits lässt sich die Erfolgsquote über die zeitliche Entwicklung der Testergebnisse darlegen. Für die Entwicklung von LAMA zeigt sich, dass der Anteil der fehlgeschlagenen Tests im Verhältnis zum Gesamtumfang sehr gering ist, aber nicht fehlerfrei ist, so wie es im Optimalfall sein sollte.

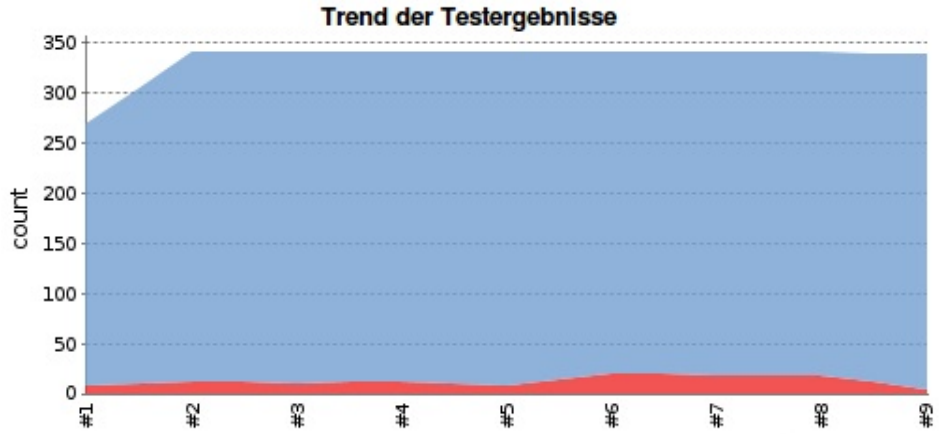


Abbildung 17: Trend der Testergebnisse

Neben diesen Schaubildern, die einen übergreifenden Vergleich von verschiedenen Builds zulassen, gibt es weiterhin für jeden einzelnen Build die Möglichkeit, die exakten Namen der fehlgeschlagenen Tests zu erfahren. Anhang 10.9 legt dies anhand des sechsten Buildvorgang des LAMA-Jobs exemplarisch dar. Das Testergebnis besteht aus der Auflistung der Testnamen, sowie der Angabe, wie sich diese Anzahl im Vergleich zu vorherigen Build verändert hat. Im sechsten Build liegt diese Änderung bei +12. Es schlugen demnach 12 weitere Tests fehl, als im Build zuvor. Es lässt die Vermutung zu, dass diese 12 Tests durch die letzte Modifizierung des Repositories (Revision 1656) entstanden sind.

E-Mails werden genau dann versendet, wenn eine Verschlechterung des Buildstatus aufgetreten ist. Die E-Mail zeigt den Status des Builds, den Namen den Jobs sowie einen Link, mit dem die Seite des verschlechterten Builds im CI-Server aufgerufen werden kann. Weitere Informationen, wie zum Beispiel der Grund, warum ein Übersetzungsvorgang nicht funktionierte oder die Namen von fehlschlagenden Unittests sind nicht innerhalb der E-Mail vorhanden, können aber durch den Aufruf dieses Links nachgeschlagen werden.

7.3.3 Vorstellung Teststrukturen

Eine gute Teststruktur ist eine wichtige Grundlage, um die Fehlerfreiheit der Programmeinheiten, die die mathematischen Berechnungen implementieren, zu erreichen. Da die Entwicklung von LAMA noch nicht abgeschlossen ist, unterliegen die Testklassen ständiger Veränderung.

Es existieren zur Zeit drei verschiedene ausführbare Testläufe, die Gruppen von Unittests zusammenfassen. Im ersten Testlauf (lama_test) werden Tests gebündelt, die für die serielle Ausführung gedacht sind. Hierbei werden vor allem die Grundfunktionen der gegebenen mathematischen Basisobjekte getestet; dazu zählen unter anderem

Matrizen, Vektoren, Storages und die Klasse Scalar.

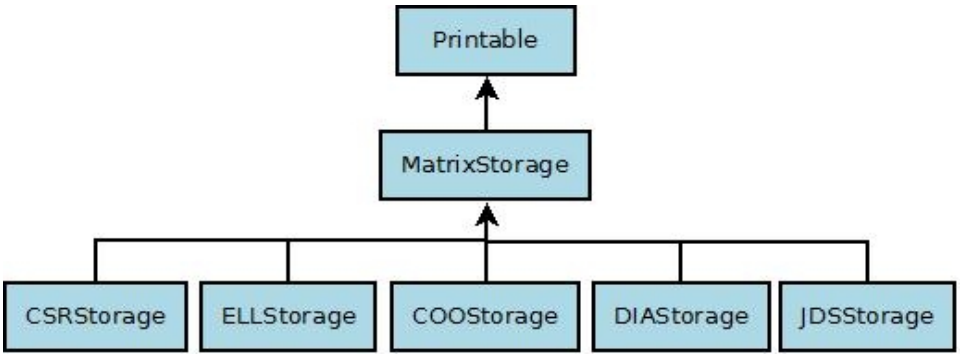


Abbildung 18: Vereinfachte Darstellung der Vererbungen der Storageklassen

Damit die Handhabung der Teststrukturen in LAMA optimiert wird, besitzen alle Testklassen von LAMA Gemeinsamkeiten. Für jede Projektklasse existiert grundsätzlich eine dazugehörige Testklasse. Dies ist vorteilhaft, da so neue Testcases direkt an einen passenden Platz geschrieben werden können, der auch schnell wiedergefunden wird. Innerhalb einer jeden Testklasse existiert wiederum exakt eine Testsuite, die denselben Namen trägt wie ihre Testklasse. So wird sichergestellt, dass das Ausführen von einzelnen Tests durch die Laufzeitoption „--run_test“ intuitiv zu bedienen ist und ein Suchen nach dem exakten Namen einer Testsuite entfällt. Innerhalb dieser Testsuite werden die jeweiligen Testcases geschrieben, die die Methoden aus der dazugehörigen Projektklasse testet. Testcases, die keiner Testsuite zuzuordnen sind, existieren somit nicht. Für das im Anhang 10.1 vorhandene Beispiel der Klasse ScalarTest genügt so der Aufruf des Testlaufes mit dem Argument „--run_test=ScalarTest“, um alle Testcases, die sich in der Testsuite und somit in der Testklasse ScalarTest befinden, aufzurufen.

Der zweite Testlauf (lama_dist_test) fasst alle Tests zusammen, deren Absicht es ist, parallele Berechnungen durchzuführen. Die numerischen Werte und deren Verrechnungen werden hierbei auf mehrere CPUs verteilt, um insgesamt die Berechnungszeit zu reduzieren. Um alle Anweisungen des Quellcodes von LAMA ausführen zu können, ist es nötig, diesen Testlauf mehrfach, jeweils mit einer unterschiedlichen Anzahl an Prozessoren aufzurufen. Nur so lässt sich die Korrektheit in möglichst vielen, in der Praxis angewandten Fällen, abdecken. Auch in Bezug auf eine Codeabdeckungsanalyse muss dies durchgeführt werden, um überhaupt alle Anweisungen von LAMA aufzurufen und so den Optimalwert des Anweisungsüberdeckungsgrades erreichen zu können.

Bei allen parallel auszuführenden Anweisungen ist es nötig ein Objekt vom Typ Communicator zu erstellen, der für die Kommunikation der verschiedenen Prozesse während den Berechnungen zuständig ist. So gilt dies auch für alle Testklassen in dem Testlauf lama_dist_test. Die Instanz vom Typ Communicator wird für fast alle Testcases in diesem Testlauf benötigt und kann in einem TestFixture erstellt werden. Daraus ergibt sich der Vorteil der Codereduzierung, da die Erstellung des Objektes für den Programmierer nun einmalig ist.

Der Fokus des dritten Testlaufes liegt auf Berechnungen, die durch die CUDA-Anbindung auf der GPU ausgeführt werden. Da allerdings diese Anbindung noch nicht final abgeschlossen ist und somit die Testklassen auch nicht vollständig sind, werden diese geschriebenen Tests in dieser Arbeit nicht weiter betrachtet.

Es existieren viele Vererbungshierarchien zwischen verschiedenen Klassen von LAMA. Um dies effektiv in die Teststruktur einzuarbeiten sind Testklassen erstellt worden, die ein instanziiertes Objekt übergeben bekommen und mit diesem die vorhandenen

Testmethoden durchlaufen. Exemplarisch wird ein solches Vorgehen für die Erstellung von Testklassen nun anhand des Beispiels der Storageklassen dargelegt. Die Vererbungshierarchie der Storageklassen ist in Abbildung 18 dargestellt.

Die instanziierbaren Storageklassen, wie zum Beispiel der CSR-Storage leiten von der Klasse MatrixStorage ab; diese wiederum letztendlich von der Klasse Printable. Jeder der fünf vorhandenen Storages besitzt somit alle Methoden, die durch die Klasse MatrixStorage gegeben sind. Des Weiteren besitzt jede konkrete Storageklasse spezifische Methoden, die nur in dieser Storageklasse implementiert sind. Für die Teststruktur bedeutet dies, dass eine Instanz der templatisierten Klasse MatrixStorageTest erstellt wird und ihr ein erstelltes Objekt eines konkreten Storage übergeben wird.

Abbildung 19 zeigt dies beispielhaft für den CSRStorageTest. Mit diesem Storageobjekt werden nun die einzelnen Testmethoden in der Klasse MatrixStorageTest durchlaufen, deren Aufrufe in der ebenfalls in dieser Klasse gegebenen Methode runTests() zusammengefasst sind. Daher genügt der Aufruf dieser Methode auf dem instanziierten Objekt der Klasse MatrixStorageTest.

```
typedef boost::mpl::list<float, double>  test_types;

[...]

BOOST_AUTO_TEST_CASE_TEMPLATE( MatrixStorageTestWithCSRStorage, T,
                                                              test_types )
{
       CSRStorage<T> csrStorage;
       MatrixStorageTest<T> storageTest( csrStorage );
       storageTest.runTests();
}
```

Abbildung 19: Ausschnitt des CSRStorageTest

Dieses Vorgehen geschieht äquivalent in jeder der fünf StorageTest-Klassen, die für dünnbesetzte Matrizen vorhanden sind. Anstatt für jeden Storage einen eigenen Test zu schreiben, reduziert sich somit der Quellcode insgesamt und zentralisiert sich in der Klasse MatrixStorageTest. Die spezifisch gegebenen Methoden eines jeweiligen Storages müssen allerdings separat als Testcases in die konkreten StorageTest-Klassen, wie zum Beispiel in der Klasse CSRStorageTest.cpp, implementiert werden.

Das Konzept der Vererbungen der Klassen und der vorgestellten Testklassenstruktur findet sich nicht nur bei den Storages wieder, sondern unter anderem bei den in LAMA implementierten mathematischen Normen, den umgesetzten Verteilungen (Distributions) oder den verschiedenen Matrizenarten.

Um den in dem Test-Framework-Vergleich erläuterten Vorteil der Templatisierung eines Testcases darzulegen, soll das eben aufgeführte Beispiel ebenfalls dafür genutzt werden. Boost bietet durch das Makro BOOST_AUTO_TEST_CASE_TEMPLATE eine Möglichkeit an, effektiv einen Test mit verschiedenen Datentypen zu durchlaufen. Dies ist besonders in dem gegebenen numerischen Kontext interessant. Der gesamte Test kann durch die Verwendung dieses Makros mit den gängigen Datentypen float und double instanziiert werden. Weitere Datentypen, wie beispielsweise „long double“ sind für die zukünftige Entwicklung durch das einfache Hinzufügen in die Variable test_types möglich. Anstatt der konkreten Angabe eines Datentyps, verwendet der Entwickler lediglich eine Variable, hier mit „T“ bezeichnet, in der Implementierung des Testcases. Die Mühe mehrere inhaltsgleiche Tests mit unterschiedlichen Datentypen schreiben zu müssen, bleibt somit erspart. Die mehrfache Ausführung des Testcases

mit den unterschiedlichen Datentypen wird von Boost.Test gesteuert und durchgeführt und ist somit nicht Aufgabe des Entwicklers.

Durch die regelmäßige Ausführung der Testläufe konnten einige Fehler gefunden und anschließend auch behoben werden. Belegt werden kann dies beispielsweise durch die Revisionsänderung 1141. In dieser wurde die Übergabe einer falschen Vektorgröße korrigiert. Ein anderes Beispiel ist die Änderung der Revision 1120, in der ein Vorzeichen für eine Operation der Klasse DenseVector angepasst wurde.

7.4 Analyse der Codeabdeckung

Um nachzuweisen, welche Funktionen von LAMA tatsächlich durch die Testläufe getestet und somit mindestens einmal aufgerufen wurden, bietet sich eine Durchführung einer Codeabdeckungsanalyse an. Die Revision 1250 von LAMA aus dem April dieses Jahres ist die Grundlage für eine erste Analyse, aus der Schwachstellen herausgearbeitet werden können. Die nötigen Anweisungen, um möglichst automatisiert diese Analyse durchführen zu können, sind in einem Skript zusammengefasst (Anhang 10.6). Ein einmaliges Ausführen dieses Skriptes bewirkt, dass alte Rückstände von früheren Analysen gelöscht, die vorhandenen Testläufe durchgeführt, die Ergebnisse bereinigt und letztendlich eine HTML-Struktur mit den Abdeckungsergebnissen erstellt wird. Unter anderem wird die Option „--extract“ bei dem Aufruf von Lcov benutzt, um die Abdeckungen externer Libraries, wie zum Beispiel der Standard-C++-Library oder den Bibliotheken von MPI herauszurechnen. Solche Abdeckungen sind nicht von Belang und stellen keine Anhaltspunkte für eine Verbesserung für das Projekt LAMA dar. Anhang 10.7 zeigt die oberste Ordnerebene der erstellten HTML-Struktur. Der Aufwand alle Verzweigungen (Branches) durch Tests abzudecken ist zu komplex, um dies weiter zu verfolgen. Daher wird der Schwerpunkt auf die Kategorie der Anweisungsüberdeckung gelegt. Durch den Parameter „-DCODE_COVERAGE=1“ beim Aufruf von Cmake, werden die nötigen Flags für eine Codeabdeckungsanalyse für das Übersetzen von LAMA gesetzt und das Skript automatisch in den Ordner „test“ kopiert, aus dem es auch aufzurufen ist.

Das Ergebnis der Analyse lässt sich in zwei Abschnitt unterteilen: Die Abdeckung der Teststruktur – in der Revision 1250 noch eine einzelne ausführbare Datei namens btests - und die Abdeckung der Bibliotheksklassen von LAMA. In der damaligen Entwicklungsphase unterteilte sich die Bibliothek in eine C-Schicht (lama) und eine C++-Schicht (lama++). Dadurch, dass sowohl die seriellen als auch parallelen Tests in einer ausführbaren Datei gebündelt waren, musste diese mehrfach mit einer unterschiedlichen Anzahl an Prozessen aufgerufen werden, um so auch Anweisungen auszuführen, die für parallele Berechnungen zuständig sind.

Die Abdeckungen der Testklassen liegen deutlich über 90% und reichen somit fast an den zu erwartenden Optimalwert heran. Die fehlenden Prozentpunkte entstehen durch eine nicht vollständige Abdeckung von Hilfsdateien, wie beispielsweise btests/Context.cpp oder btests/Configuration.cpp, die selbst keine Tests beinhalten, sondern nur Konfigurationsdaten, die bei dem Ausführen von verschiedensten Tests gebraucht werden.

Der, vom Anteil an LAMA gesehen, größte Ordner ist „lama++“. Mit 54.9% liegt die Abdeckung der Klassen, die sich in diesem Ordner befinden, im „roten“ Bereich. Fast alle in lama++ befindlichen Unterordner schließen sich diesem Ergebnis an. Die Aufgabe ist es nun, die Abdeckung der Bibliotheksklassen durch Verbesserungen an den einzelnen Testmethoden bzw. durch das Hinzufügen von fehlenden Tests zu erhöhen. Die Ergebnisse der ersten Analyse zeigen Codeteile auf, die bislang vernachlässigt worden sind.

Ein erster Anhaltspunkt, um die Codeabdeckung zu optimieren, ist das Werfen von Exceptions, wie beispielsweise in der Klasse DenseVector.cpp (Zeile 550). In der Funktion swap wird geprüft, ob die beiden zu tauschenden Vektoren dieselben Datentypen besitzen. Um einen Test vollständig zu entwickeln, muss auch der Zweig überprüft werden, in dem unterschiedliche Datentypen miteinander vertauscht werden sollen, was als fehlerhafter Aufruf dieser Funktion gewertet werden muss und somit eine Exception geworfen wird. Daher muss in diesem Test – auch Failure Test genannt, absichtlich diese Methode mit einem nicht passenden Vektor aufgerufen und durch das Makro BOOST_CHECK_THROW die geworfene Exception abgefangen werden.

Fast jede Klasse im lama++-Ordner leitet von der Klasse Printable ab und überschreibt daher die Methode writeAt(). Diese Methode dient dazu, Informationen eines Objekts auf der Standardkonsole auszugeben. In der Umsetzung ist es aufwendig für jedes Objekt, welches diese Methode überschreibt, einen Test zu schreiben, der die in einem Stringstream ausgegebene Zeichenkette mit einer zu erwartenden Zeichenkette abgleicht. Die Schwierigkeit liegt hierbei in der Länge und der Komplexität der Ausgaben. Dennoch muss, um die Vollständigkeit der Abdeckung und zumindest einen korrekten Aufruf dieser Methode zu erreichen, ein Test geschrieben werden, der für den Aufruf zuständig ist. Die Lösung für dieses Problem ist das Makro LAMA_WRITEAT_TEST, welches als Argument ein von der Klasse Printable abgeleitetes Objekt erhält. Die Methode writeAt() kann somit durch einen simplen Aufruf des Makros zumindest indirekt getestet werden. Überprüft wird in der Implementierung des Makros lediglich, ob überhaupt eine auszugebende Zeichenkette erstellt wird, denn dies müssen alle implementierten writeAt-Methoden erfüllen. Eine Prüfung, ob der Inhalt dieser Zeichenkette korrekt ist, wird nicht vorgenommen.

Ebenfalls ist aufgefallen, dass Methoden, wie zum Beispiel die Methode „scale“, die von allen vorhandenen Storageklassen implementiert ist, existieren, die nicht durch die Testläufe aufgerufen und somit getestet wurden. Hierfür ist die einzige Lösung einen neuen Testcase zu schreiben, der für die Ausführung dieser Methode und einem Vergleich zwischen dem berechneten Ergebnis und dem zu erwartenden Wert sorgt.

Durch eine gründliche Analyse der Testabdeckung kann nicht nur festgestellt werden, dass neue Test erstellt bzw. vorhandene Tests aktualisiert und angepasst werden müssen, sondern es können auch unerreichbare Stellen gefunden werden. Dies war in LAMA zum Beispiel der Fall bei einigen Konstruktoren der Klasse DenseVector.cpp. Verschiedene mathematische Ausdrücke, wie zum Beispiel „Matrix*Vektor“ werden in LAMA durch inline-Funktionen auf einen komplexen mathematischen Ausdruck, in diesem Fall „Scalar*Matrix*Vektor+Scalar*Vektor“ abgebildet, so dass nur ein Konstruktor für den letzt genannten Ausdruck existieren muss. Zu dem Zeitpunkt dieser Revision wurden allerdings Konstruktoren erschaffen, die z.B. als Argument den Ausdruck „Matrix*Vektor“ erhalten. Durch die vorher durchgeführte Abbildung des mathematischen Ausdrucks können diese nie aufgerufen werden. Daher kann der Quellcode von diesen Konstruktoren befreit werden.

Eine erneut durchgeführte Analyse aus dem August 2012 (Anhang 10.8) gibt zu erkennen ist, dass Verbesserungen hinsichtlich der Codeabdeckung zu erkennen sind, da ein Großteil der Zellen nun durch Lcov teilweise gelb und grün eingefärbt wurden. Der Vergleich der Gesamtabdeckung der LAMA-Bibliothek (Abbildung 20) belegt dies durch eine Steigerung der Anweisungsüberdeckungsgrades von 44% auf 62%. Zur Vollständigkeit sei erwähnt, dass auch der Anteile der aufgerufenen Funktionen, sowie die abgedeckten Verzweigungen gestiegen sind. Ein direkter Vergleich zwischen den einzelnen Ordnern der beiden Analysen ist jedoch kaum möglich, da sich die Struktur der Bibliothek LAMA in diesem Zeitraum sehr stark durch Refactoring verändert hat. So wurde zum Beispiel die C-Schicht in die C++-Schicht integriert, so dass nur noch ein Ordner namens lama existiert, der alle Bibliotheksklassen enthält.

Die Durchführung einer Codeabdeckungsanalyse kann ebenfalls in das CI-Konzept eingefügt werden. Der Aufruf, um das Skript ausführen zu können, kann als Shellbefehl integriert und daher in einem Postbuild-Task aufgerufen werden. Allerdings ist es nicht sinnvoll, diese Analyse bei jedem Anstoß eines Jobs auszuführen. Die Dauer des Jobs würde zum einen um ein Vielfaches länger dauern, als der bloße Übersetzungsvorgang und zum anderen werden sich die Änderungen durch einen einzigen Commit nicht wesentlich auf die Abdeckungsrate auswirken. Daher ist anzuraten, parallel einen zweiten Job zu erstellen, der nur wöchentlich oder gar monatlich alle Tests ausführt, im Anschluss die Analyse der Codeabdeckung durchführt und diese Ergebnisse dem Entwicklerteam bereitstellt.

	Hit	Total	Coverage
Lines:	8634	19353	44.6 %
Functions:	2231	4694	47.5 %
Branches:	4549	19772	23.0 %

Codeabdeckung nach Revision 1250 (April 2012)

	Hit	Total	Coverage
Lines:	11102	17808	62.3 %
Functions:	2917	4111	71.0 %
Branches:	4971	12211	40.7 %

Codeabdeckung nach Revision 1633 (August 2012)

Abbildung 20: Vergleich der Abdeckungsgrade der LAMA-Bibliothek

7.5 Portabilität

LAMA ist konzipiert worden, um sowohl auf Unixsystemen, als auch auf Windowssystemen einsatzfähig zu sein und dadurch eine große Zielgruppe an potentiellen Kunden zu erreichen. Dies ist zum Beispiel wichtig, wenn in einem Unternehmen ein Wechsel des Betriebssystem stattfindet.

Die Bibliothek LAMA ist dafür ausgelegt, dass sie mit verschiedenen Compilern übersetzt werden kann. Der Standard ist die Verwendung des GCC (vgl. GCC 2012a), welcher standardmäßig in allen Unixsystem zu finden ist und auch unter Windows mittels MinGW (vgl. MinGW 2012) funktioniert. Als Alternativprodukt kann zum Beispiel der Intel-Compiler verwendet werden. Diese Eigenschaft ist vorteilhaft, da im Falle einer Einstellung der Entwicklung eines Compilers LAMA immer noch mit einem alternativen Compiler verwendet werden kann. Ferner fordern die Kunden die Übersetzbarkeit des Code mit den von Ihnen präferierten Produkten.

Der CI-Server hat nun die Aufgabe, das Produkt in Kombination mit möglichst unterschiedlichen Softwarewerkzeugen und auf verschiedenen Zielsystemen zu übersetzen. Verschiedene Bibliotheken können auf der unterschiedlichen Zielsystemen durch die Angabe ihres Pfades an das Übersetzungstool gekoppelt werden. In der Praxis heißt dies, dass der Übersetzungsvorgang mehrfach für jede Zielplattform gestartet werden muss. Im anschließenden Bauen der Software können so Inkompabilitäten zwischen Betriebssystemen, Compilern und Bibliotheken erkannt werden. Tools, wie Compiler können auf diese Weise auch ausgetauscht werden, um so die Übersetzbarkeit auch mit diesen zu gewährleisten.

LAMA verwendet einige Komponenten der Bibliothekssammlung Boost, unter anderem auch Boost.Test. Boost ist ebenfalls plattformunabhängig und bietet Unterstützung für die Entwicklung einer portablen Bibliothek, da keine spezifischen Anpassungen für die Verwendung an ein bestimmtes Betriebssystem vorgenommen werden müssen.
Eine Plattformunabhängigkeit ist ebenfalls durch alle Softwarewerkzeuge, die für die Entwicklung von LAMA vorhanden waren bzw. durch das hier geschaffene CI-Konzept integriert wurden, gegeben. Darunter zählt Cmake, welches den Übersetzungsvorgang automatisiert, der verwendete CI-Server Jenkins, der javabasierend auf jedem Betriebssystem agiert, als auch das zur Versionsverwaltung verwendete Tool Subversion.

7.6 Anwendbarkeit von Continuous Integration

Kontinuierliche Integration bedeutet, dass ein CI-Server das Projekt nach jeder Revisionänderung übersetzt und testet (vgl. Duvall 2011, S.8) und ein Feedback über den Status des aktuellen Build zurückliefert.

Anstatt nach diesem Schema vorzugehen, ist es ebenfalls möglich, den CI-Server so zu konfigurieren, dass zu einem bestimmten Zeitpunkt das Projekt neu gebaut wird, beispielsweise durch eine stündliche Ausführung der Übersetzungsvorganges. Dieses Vorgehen ist vor allem bei sogenannten „Nightly Builds“ üblich. Diese sind automatisierte Übersetzungsvorgänge, die nachts durchgeführt werden. Nachts erwartet das Entwicklerteam keine Revisionsänderungen, so dass die Möglichkeit gegeben ist, zusätzlich zum Übersetzen auch weiterführende Schritte durchzuführen, die Aufschluss über die aktuelle Qualität des Projektes geben. Im Projekt LAMA wären solche „Nightly Builds“ nützlich, um auch Benchmarks und Performancetests zu durchlaufen und zu protokollieren, so dass das Entwicklerteam am nächsten Morgen aus den gesammelten Informationen Erkenntnisse ziehen kann, inwiefern die Änderungen am Quellcode einen Vorteil für die Bibliothek LAMA ergeben. Ebenfalls können Testläufe für parallele Ausführungen nun einen größeren Stellenwert zugesprochen bekommen, da die Anzahl an festgelegten Prozessen durch umfangreichere Tests untersucht werden kann.

Damit die Lauffähigkeit noch weiter gesichert und garantiert werden kann ist es ebenfalls möglich, einen Commit durch einen CI-Server zurückzuweisen, wenn die Übersetzungs- und Testvorgänge auf Grundlage dieses Commits fehlschlagen. Dieser Schritt würde die Stufe für eine qualitativ hochwertige Software weiter erhöhen, da fehlerbehaftete Commits als Fehlerquelle ausgeschlossen werden. Jedoch erhöht sich der Testaufwand vor einem Commit beträchtlich, was Wartezeiten für die Entwicklerteams bedeuten.

7.7 Evaluation des angewandten CI-Konzeptes

Die abschließende Frage ist nun, ob die Verwendung des CI-Konzeptes tatsächlich einen Nutzen für die Softwareentwicklung der Bibliothek LAMA erbracht hat. Daher soll abschließend überprüft werden, ob die an das CI-Konzept aufgestellten Anforderungen, der Kapitel 2.2.3 und 2.3 auch tatsächlich erfüllt wurden.

Die Anforderungen wurden in zwei Bereiche gegliedert: Allgemeine Anforderungen, die alle Projekte betreffen, welche durch CI unterstützt sind und den Anforderungen, die sich speziell durch den numerischen Kontext für LAMA ergeben.

Die Anforderung von älteren Vorgehensmodellen gegenüber einem Kunden sich

solange auf die Software zu gedulden bis die Integrationsphase der aktuellen Entwicklung abgeschlossen ist, kann durch die Verwendung eines Continuous Integration Konzeptes fast verworfen werden. Die nun durch CI entstandene Integrationsphase kann mit dem von Fowler geprägten Begriff als „non-event“ (Fowler 2006, „Continuous Integration“) bezeichnet werden. Anstatt einer wochen- und monatelang dauernden Phase ist diese auf wenige Minuten geschrumpft. Dies wird belegt durch die aufgezeichnete Builddauer des CI-Servers Jenkins (vgl. Abbildung 16). Nach jedem erfolgreichen Übersetzungsvorgang ist eine lauffähige Version entstanden, auch wenn diese noch nicht alle zu implementierenden Funktionen enthält.

Durch die Verwendung eines Source Code Management Tools ist erreicht worden, dass alle relevanten Daten zentral verwaltet werden. Dieses Repository gilt als Basis aller Entwicklungsarbeiten und muss daher lauffähig und fehlerfrei gehalten werden. Daher ist der Entwickler zum einen gezwungen, alle zum Projekt gehörigen Daten dort abzulegen, so dass fehlgeschlagene Übersetzungs- oder Testvorgänge auf Grundlage von fehlenden oder nicht erreichbaren Daten unterbunden werden. Zum anderen müssen die beteiligten Entwickler bei entstandenen Fehlern diese möglichst schnell beseitigen. Sowohl das Übertragen aller relevanten Daten an das Repository, als auch der Drang der schnellen Fehlerbeseitigung können nicht durch die Benutzung von Softwaretools behoben werden, sondern nur durch Disziplin seitens der Entwickler. Durch Boost.Test konnte eine Teststruktur erschaffen werden, die, in regelmäßigen Abständen ausgeführt, die Existenz von Fehlern nachweist (vgl. Abbildung 17).

Auch die Anforderung der Korrektheit einer numerischen Berechnung konnte durch die Benutzung des Test-Frameworks Boost.Test erfüllt werden. Da durch die implementierten Lösealgorithmen meist eine approximative Lösung entsteht, muss vor allem bei Fließkommaberechnungen ein passender Zahlenbereich festgelegt werden, in dem sich die Differenz zwischen Erwartungswert und berechnetem Wert befinden muss, um die Lösung der Berechnung dennoch als korrekt einzustufen. Dieser Zahlenbereich hängt auch stark von dem im Test verwendeten Datentyp ab. Bei der Verwendung von Datentypen, beispielsweise der einfachen oder doppelten Genauigkeit von Fließkommazahlen, ist der zur Darstellung der Zahl notwendige Speicher unterschiedlich groß. Daher entsteht bei doppelter Genauigkeit der Nachteil, dass mehr Speicherplatz verwendet wird, allerdings erhöht sich durch die größere Anzahl an Speicherstellen die Genauigkeit, eine Zahl darstellen zu können. Boost.Test bietet durch das Makro BOOST_CHECK_CLOSE die Möglichkeit, eine solche Situation zu handhaben. Die Differenz des berechneten Wertes sowie des Erwartungswertes kann mit einem angegebenen Zahlenbereich verglichen werden und so entschieden werden, ob das berechnete Ergebnis nah genug am Erwartungswert liegt.

Die Teststrukturen zu optimieren verhalf eine Analyse der Codeabdeckung. In dieser Arbeit wurde dies anhand von zwei Messungen (Revisionen 1250 und 1633) exemplarisch durchgeführt. Durch die Verwendung von Lcov konnte sehr effektiv in einem HTML-Report dargelegt werden, welche Methoden von LAMA nie oder nur teilweise durch die Testläufe aufgerufen und somit getestet wurden. So konnten Fehler gefunden und behoben sowie Stellen im Quellcode, die unerreichbar waren, aufgedeckt und anschließend gelöscht werden. Beide Analysen konnten Verbesserungen der Qualität des Codes durch die positiven Entwicklung der Abdeckungsgrade nachweisen.

Die Anforderung an die Performanz von LAMA konnte mittels Benchmarks nachgewiesen werden. Durch diese können festgelegte Szenarien immer wieder durchlaufen und die benötigte Zeit gemessen werden. Somit kann über eine Projektlaufzeit gesehen eine mögliche Optimierung der Performanz festgehalten werden. Parallel dazu können die Testläufe verwendet werden, um auch die Korrektheit trotz Optimierungen der Berechnungszeit des Algorithmus weiterhin zu prüfen.

Der Nachweis, dass eine Software auch kompatibel mit verschiedenen angebundenen Bibliotheken geschrieben wurde, lässt sich auf zwei Wegen realisieren. Zum einen können die Ressourcen der Maschine zugegriffen werden, auf der auch der CI-Server installiert ist, zum anderen kann das Übersetzen durch den Zugriff auf einem externen Server eingerichtet werden.

Das hier entwickelte CI-Konzept wurde auf einem Linuxsystem eingerichtet, welches, jedoch Einschränkungen hinsichtlich der potentiell verwendbaren Softwareschnittstellen aufweist, da er über keine Acceleratorkarten verfügt. Nicht alle durch LAMA unterstützten Schnittstellen, wie zum Beispiel CUDA oder OpenCL, können auf diesem Server übersetzt, verwendet und getestet werden. Das Übersetzen mit unterschiedlichen Versionen von vorhandenen Bibliotheken, wie zum Beispiel Boost, könnte allerdings leicht realisiert werden, indem der Installationspfad zu verschiedenen Boostversionen durch den Parameter -DBOOST_ROOT an den Aufruf von Cmake übergeben wird. Im CI-Server Jenkins kann dies durch die Einrichtung verschiedener Jobs mit unterschiedlichen Angaben der Installationspfade umgesetzt werden. Äquivalent könnte dies bei den Installationspfaden der angebundenen MPI-Bibliothek angewendet werden, um so die Kompatibilität zwischen LAMA und der MPI-Anbindung sicherzustellen. Auch wenn das CI-System dies bereits vorsieht, lag bei dem umgesetzten Konzept noch keine Anbindung an ein externes Rechnercluster vor, um so das Übersetzen an Maschinen mit unterschiedlichen Betriebssystemen und verschiedenen Hardwarekonstellationen zu senden und zu testen.

Für den zukünftigen Einsatz des CI-Konzeptes sind die Bereiche der Performanz sowie der Portabilität weiter auszubauen. LAMA kann zwar manuell auf verschiedene Betriebssysteme eingesetzt und getestet werden, da alle zur Verwendung von LAMA nötigen Softwarewerkzeuge, wie Cmake, Boost und andere Bibliotheken ebenfalls einen portablen Einsatz unterstützten. Allerdings kann dies automatisiert nur dann geschehen, wenn der eingesetzte CI-Server Jenkins Zugriff auf die von SCAI zur Verfügung stehenden Rechernetze hat, die die Möglichkeit besitzen alle von LAMA angebundenen Schnittstellen zu verwenden und so durchzuführende Jobs an Maschinen mit verschiedenen Plattformen schicken kann.

8 Zusammenfassung

Das Ziel dieser Abschlussarbeit war es, ein Continuous Integration Konzept für die numerische Bibliothek LAMA zu entwickeln und es anschließend in die aktuelle Entwicklung zu integrieren. Eine tragende Rolle im Laufe der Konzeptentwicklung hatten die Softwarewerkzeuge, die für diese Anwendungstechnik von Nöten sind. Subversion als Werkzeug für die Versionsverwaltung und das Übersetzungswerkzeug Cmake waren schon durch die Entwicklung gegeben, so dass der Schwerpunkt auf dem Finden von Boost.Test als Test-Framework für den numerischen Sektor und Jenkins als einzurichtenden CI-Server lag. Die für LAMA entwickelte Teststruktur und eine zur Optimierung der Teststruktur verwendete Codeabdeckungsanalyse wurde, ebenso wie die gesammelten und bereitgestellten Informationen der Übersetzungsvorgänge und Testläufe, von Jenkins vorgestellt. Anschließend wurden die Ergebnisse, die durch die Anwendung dieser Werkzeuge erzielt wurden, auf ihren Nutzen untersucht.

Das Continuous Integration Konzept dieser Arbeit konnte alle Vorteile, die von Fowler und Duvall beschrieben wurden, belegen und konnte so zeigen, dass die Anwendung eines CI-Konzepts für eine Softwareentwicklung eindeutige Vorteile bringt. Dazu gehört unter anderem die Reduzierung der Fehler sowie eine Verkürzung der Integrationsphase. Ein Endkunde kann durch die Anwendung dieses CI-Konzeptes jederzeit eine lauffähige Version erhalten. Ebenfalls wurden Zeitersparnisse durch automatisierte Vorgänge, wie beispielsweise durch die Benutzung eines Buildtools, des erstellten Skripts für eine Abdeckungsanalyse oder die Verwendung eines CI-Servers, und die Sicherstellung des korrekten Funktionieren von Methoden, die in den geschriebenen Unittests geprüft wurden, gezeigt.

Im Laufe dieser Arbeit ist dafür eine Integrationsumgebung aufgebaut und angewendet worden, welches allerdings, besonders für die Erfüllung der Anforderung der Portabilität, erweitert werden muss. Dies war auf Grundlage der bislang verfügbaren Hardware nicht vollständig möglich, da Schnittstellen, wie CUDA und OpenCL, und das Ausführen von automatisierten Übersetzungsvorgängen auf verschiedenen Betriebssystemen nicht unterstützt wurden. Auch im Hinblick auf die Anforderung der Performance einer numerischen Bibliothek wäre es hilfreich, diese durch den CI-Server gezielt auszuführen und die Messergebnisse zu protokollieren, so dass auch bei diesen Messungen ein Trend abzulesen ist, ob die Gesamtentwicklung des Projektes sich in eine positive Richtung bewegt.

Insbesondere lässt sich sagen, dass die Erweiterungsmöglichkeiten des CI-Konzeptes sehr hoch sind. Durch die Fähigkeit im CI-Server, auf eine Bandbreite an Plugins zurückzugreifen sowie Zugriff auf systeminterne Befehle zu haben, kann sehr leicht die Verwendung von weiteren Werkzeugen in das Konzept integriert werden. Auch der Feedback-Mechanismus kann variabel geändert werden, je nach Anforderung der Entwickler.

Letztendlich kann festgestellt werden, dass Continuous Integration eine Bereicherung für die Softwareentwicklung eines Entwicklungsteams ist und als „non-event“ (Fowler 2006, „Continuous Integration“) wertvolle Dienste zur Verbesserung der Codequalität leistet.

9 Literaturverzeichnis

ANL (Hrsg.): "The Message Passing Interface (MPI) standard". http://www.mcs.anl.gov/research/projects/mpi/. Stand: 09.11.2012.

Budszuhn, F.: "Subversion". Galileo Press, Bonn 2005.

CruiseControl (Hrsg.): "CruiseControl". http://cruisecontrol.sourceforge.net/index.html. Stand: 16.08.2012a.

CruiseControl (Hrsg.): "Plugins". http://cruisecontrol.sourceforge.net/main/plugins.html. Stand: 16.08.2012b.

Dahmen, W.; Reusken, A.: "Numerik für Ingenieure und Naturwissenschaftler". Springer-Verlag, Berlin 2006.

Dalheimer, M.: "Programming with Qt". O'Reilly, Köln 1999.

Dawes, B.; Abrahams, D.; Rivera, R.: "Welcome to Boost.org". www.boost.org. Stand: 23.07.2012.

Deuflhard, P.; Hohmann, A.: "Numerische Mathematik I". 3. Auflage, Walter de Gruyter, Berlin 2002.

Duvall P.: "Continuous Integration". 6. Auflage, Addison-Wesley, Upper Saddle River 2011.

Fowler, M.: "Continuous Integration". martinfowler.com/articles/continuousIntegration.html. Stand: 01.05.2006.

Fraunhofer SCAI (Hrsg.): „LAMA – Library for Accelerated Math Applications“. http://www.libama.org/overview.html. Stand: 12.08.2012a.

Fraunhofer SCAI (Hrsg.): „LAMA documentation“. http://sourceforge.net/apps/trac/libama/wiki. Stand: 12.08.2012b.

GCC (Hrsg.): „GCC, the GNU Compiler Collection“. http://gcc.gnu.org/. Stand 22.07.2012a.

GCC (Hrsg.): „gcov – a Test Coverage Program“. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html. Stand: 22.07.2012b.

Git (Hrsg.): „About“. http://git-scm.com/about/. Stand: 22.08.2012.

Google (Hrsg.): "Googletest – Google C++ Testing Framework". code.google.com/p/googletest. Stand: 13.06.2012.

Hermann, M.: „Numerische Mathematik“. Oldenburg Verlag, München 2001.

Hook, B.: „Portabler Code“. Open Source Press, München 2006.

Jenkins CI (Hrsg): "Jenkins". jenkins-ci.org. Stand: 16.08.2012.

Lepilleur, B.: "CppUnit – C++ port of JUnit". sourceforge.net/projects/cppunit/. Stand: 22.07.2012a.

Lepilleur, B.: „CppUnit 2 Testing Framework“. https://launchpad.net/cppunit2. Stand: 22.07.2012b.

Liggesmeyer, P.: „Software-Qualität“. Spektrum Akademischer Verlag, Heidelberg 2002.

LTP(Hrsg.): „LCOV – the LTP GCOV extension“. http://ltp.sourceforge.net/coverage/lcov.php. Stand: 22.07.2012a.

LTP(Hrsg.): „Readme file for the LTP GCOV extension (LCOV)“. http://ltp.sourceforge.net/coverage/lcov/readme.php. Stand: 22.07.2012b.

Michels, J.: „IT-Benchmarking“. 5. Auflage, Selbstverlag, Neuss 2008.

MinGW (Hrsg.): „Welcome to MinGW.org“. http://www.mingw.org/. Stand: 11.09.2012.

N.N.: „Main Page. Welcome! to CppUnit Wiki“. http://sourceforge.net/apps/mediawiki/cppunit/index.php. Stand: 22.07.2012a.

N.N.: „CppUnit Cookbook“. http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html. Stand: 22.07.2012b.

Nvidia (Hrsg.): „Introducing CUDA 5“. http://www.nvidia.com/object/cuda_home_new.html. Stand: 05.09.2012.

OpenMP ARB (Hrsg.): „OpenMP News“. http://openmp.org/wp/. Stand: 11.09.2012.

Rozenthal, G.: "Boost Test Library". http://www.boost.org/doc/libs/1_50_0/libs/test/doc/html/index.html. Stand: 25.07.2012.

Smart, J.: „Jenkins“. O'Reilly, Sebastopol, 2011.

Sneed, H.; Winter, M.: „Testen objektorientierter Software“. Carl Hanser Verlag, München 2002.

Sonar, T.: „Angewandte Mathematik, Modellbildung und Informatik“. Vieweg Verlag, Braunschweig 2001.

Vesperman, J.: „CVS“. O'Reilly, Köln 2004.

Zhanyong, W: „V1_6_ Primer – Getting starten with Google C++ Testing Framework“. http://code.google.com/p/googletest/wiki/V1_6_Primer. Stand: 24.07.2012.

10 Anhang

10.1 Beispiel einer Testsuite mit Boost.Test

```
/*
 *  Implementation of class ScalarTest.cpp with Boost.Test.
 *  Class ScalarTest is part of project LAMA.
 */

#include <boost/test/unit_test.hpp>
#include <boost/mpl/list.hpp>
#include <lama++/Scalar.hpp>

using namespace lama;
typedef boost::mpl::list< float, double, long double >  test_types;

BOOST_AUTO_TEST_SUITE( ScalarTest );
      BOOST_AUTO_TEST_CASE_TEMPLATE( InlineCtorTest, T, test_types )
      {
            typedef T ValueType;
            ValueType value = 1.0;
            Scalar s( value );
            BOOST_CHECK_EQUAL( s.getValue<ValueType>(), 1.0 );

            std::complex<double> cvalue( 1.0, 2.0 );
            Scalar t( cvalue );
            CPPUNIT_ASSERT_EQUAL( t.getValue<std::complex<double> >(),
                                                              cvalue );
      }
/* ------------------------------------------------------------------- */
      BOOST_AUTO_TEST_CASE( ScalarGetTypeTest )
      {
            float value_s = 2.0;
            Scalar s( value_s );
            BOOST_CHECK_EQUAL( s.getType<float>() , Scalar::FLOAT );

            double value_t = 2.0;
            Scalar t( value_t );
            BOOST_CHECK_EQUAL( t.getType<double>() , Scalar::DOUBLE );
      }
/* ------------------------------------------------------------------- */
      BOOST_AUTO_TEST_CASE_TEMPLATE( AdditionTest, T, test_types )
      {
            typedef T ValueType;
            Scalar s ( 2.0 );
            Scalar t ( 3.0 );
            Scalar u = s + t;

            BOOST_CHECK_EQUAL( u.getValue<ValueType>(), 5.0 );
      }
BOOST_AUTO_TEST_SUITE_END();

/* =================================================================== */
/*
 *  Implementation of a simple testrunner with Testing-Framework
 *  Boost.Test. The Headerfile unit_test.hpp includes method main().
 */

#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE my_mastertestsuite

#include <boost/test/unit_test.hpp>
```

10.2 Beispiel einer Testsuite mit CPPUnit

```
/*
 *  Implementation of ScalarTest.hpp with CPPUnit.
 */

#include <cppunit/extensions/HelperMacros.h>
#include <cppunit/TestFixture.h>

#include <lama++/Scalar.hpp>

using namespace lama;

class ScalarTest : public CPPUNIT_NS::TestFixture {

CPPUNIT_TEST_SUITE( ScalarTest );

      CPPUNIT_TEST( InlineCtorTest );
      CPPUNIT_TEST( ScalarGetTypeTest );
      CPPUNIT_TEST( AdditionTest );

CPPUNIT_TEST_SUITE_END();

protected:

      void InlineCtorTest();
      void ScalarGetTypeTest();
      void AdditionTest();

};
```

```
/*
 *  Implementation of ScalarTest.cpp with CPPUnit.
 */

#include <tests/ScalarTest.hpp>
#include <cppunit/config/SourcePrefix.h>
#include <lama++/Scalar.hpp>

using namespace lama;

void ScalarTest::InlineCtorTest()
{
    double value = 1.0;
    Scalar s( value );
    CPPUNIT_ASSERT_EQUAL( s.getValue<double>(), 1.0 );

    std::complex<double> cvalue( 1.0, 2.0 );
    Scalar t( cvalue );
    CPPUNIT_ASSERT_EQUAL( t.getValue<std::complex<double> >(),
                                                        cvalue );
}

/* ------------------------------------------------------------------- */

void ScalarTest::ScalarGetTypeTest()
{
    float value_s = 2.0;
    Scalar s( value_s );
    CPPUNIT_ASSERT_EQUAL( s.getType<float>(), Scalar::FLOAT );

    double value_t = 2.0;
    Scalar t( value_t );
    CPPUNIT_ASSERT_EQUAL( t.getType<double>(), Scalar::DOUBLE );
}

/* ------------------------------------------------------------------- */

void ScalarTest::AdditionTest()
{
    Scalar s ( 2.0 );
    Scalar t ( 3.0 );
    Scalar u = s + t;

    CPPUNIT_ASSERT_EQUAL( u.getValue<double>(), 5.0 );
}
```

```
/*
 * Implementation  of a testrunner with CPPUnit. This testrunner is a
 * example given by the Cookbook of CppUnit (N.N. 2012b).
 *
 */

#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/ui/text/TestRunner.h>

int main( int argc, char **argv)
{
      CppUnit::TextUi::TestRunner runner;
      CppUnit::TestFactoryRegistry &registry =
                        CppUnit::TestFactoryRegistry::getRegistry();

      runner.addTest( registry.makeTest() );
      bool wasSuccessful = runner.run( "", false );

      return wasSuccessful;
}
```

10.3 Beispiel einer Testsuite mit GoogleTest

```
/*
 *  Implementation of ScalarTest.cpp with GoogleTest
 */

#include <gtest/gtest.h>
#include <lama++/Scalar.hpp>

using namespace lama;

TEST( ScalarTest, InlineCtorTest )
{
    double value = 1.0;
    Scalar s( value );
    ASSERT_EQ( s.getValue<double>(), 1.0 );

    std::complex<double> cvalue( 1.0, 2.0 );
    Scalar t( cvalue );
    ASSERT_EQ( t.getValue<std::complex<double> >() , cvalue );
}

/* ------------------------------------------------------------------ */

TEST( ScalarTest, ScalarGetTypeTest )
{
    float value_s = 2.0;
    Scalar s( value_s );
    ASSERT_EQ( s.getType<float>() , Scalar::FLOAT );

    double value_t = 2.0;
    Scalar t( value_t );
    ASSERT_EQ( t.getType<double>() , Scalar::DOUBLE );
}

/* ------------------------------------------------------------------ */

TEST( ScalarTest, AdditionTest )
{
    Scalar s ( 2.0 );
    Scalar t ( 3.0 );

    Scalar u = s + t;

    ASSERT_EQ( u.getValue<double>(), 5.0 );
}

/* ================================================================== */

/*
 *  Implementation of a simple testrunner with GoogleTest given by the
 *  documentation of GoogleTest ( Google 2012, „Wiki - Writing the
 *  main() Function“).
 */

#include <gtest/gtest.h>

int main ( int argc, char **argv )
{
      ::testing::InitGoogleTest( &argc, argv );
      return RUN_ALL_TESTS();
}
```

10.4 Übersicht einer Auswahl an Assertions der Test-Frameworks

Die folgende Tabelle bietet eine Auswahl an Assertions, die von den drei vorgestellten Frameworks angeboten werden. Diese Tabelle basiert auf den Daten der Webseiten dieser Frameworks (N.N. 2012b, "Modules – Making Assertions", Rozenthal 2012 – "Reference" und Zhanyong 2012).

Boost.Test	**CPPUnit**	**GoogleTest**	**Funktion**
<level> = WARN, CHECK, REQUIRE	-	<level> = ASSERT, EXPECT	Abstufungen (<level>) für Assertions, die im jeweiligen Framework angeboten werden.
BOOST_<level>	CPPUNIT_ASSERT (condition)	<level>_TRUE (condition)	Vergleich, ob eine Bedingung wahr ergibt.
*	CPPUNIT_ASSERT_MESSAGE(msg, condition)	** z.B. ASSERT_TRUE(cond)<<"message";	Vergleich, ob eine Bedingung wahr ergibt; Ausgabe einer Nachricht.
BOOST_FAIL(msg)	CPPUNIT_FAIL (message)	**	Ausgabe einer Nachricht bei einem Test der als fehlgeschlagen notiert wird
BOOST_<level>_EQUAL(left, right)	CPPUNIT_ASSERT_EQUAL(expected, actual)	<level>_EQ(expected, actual) <level>_NE(expected, actual)	Vergleich, ob zwei Werte gleich sind. Vergleich, ob zwei Werte ungleich sind.
*	CPPUNIT_ASSERT_EQUAL_MESSAGE (msg, expected, actual)	**	Vergleich, ob zwei Werte identisch sind mit anschließender Ausgabe einer Nachricht.
BOOST_<level>_EQUAL_COLLECTION (l_begin, l_end, r_begin, r_end)			Vergleich, ob zwei Sammlungen (z.B. Felder) auf ihren Positionen identisch sind.
BOOST_<level>_CLOSE(left, right, delta)	CPPUNIT_ASSERT_DOUBLES_EQUAL (expected, actual, delta)	<level>_NEAR(val1, val2, abs_error)	Vergleich von zwei Fließkommazahlen; Differenz darf in Delta liegen.
		<level>_FLOAT_EQ(expected, actual) <level>_DOUBLE_EQ(expected, actual)	Vergleich zweier Fließkommazahlen für unterschiedliche Datentypen: float und double.

BOOST_<level>_SMALL(value, tolerance)			Überprüfung, ob eine Zahl unterhalb eines Toleranzwerts liegt.
BOOST_<level>_THROW(expression, exception)	CPPUNIT_ASSERT_THROW (expression, ExceptionType)	<level>_THROW(expression, ExceptionType) <level>_ANY_THROW(statement)	Überprüfung, ob ein Ausdruck einen bestimmten Fehlertyp wirft.
BOOST_<level>_NO_THROW(expression)	CPPUNIT_ASSERT_NO_THROW (expression)	<level>_NO_THROW(statement)	Überprüfung, ob ein Ausdruck keinen Fehler wirft.
BOOST_<level>_GE(left, right) BOOST_<level>_GT(left, right) BOOST_<level>_LE(left, right) BOOST_<level>_LT(left, right)	-	<level>_GE(val1,val2) <level>_GT(val1,val2) <level>_LE(val1,val2) <level>_LT(val1,val2)	Vergleich ob zwei Werte größer gleich, echt größer, kleiner gleich oder echt kleiner sind.
		<level>_STREQ(expected_str, actual_str)	Vergleicher zweier C-Strings

* = Boost.Test bietet ein Makro BOOST_TEST_MESSAGE(message) an, um jederzeit Nachrichten während Testläufen auszugeben.

** = GoogleTest bietet die Möglichkeit benutzerdefinierte Nachrichten durch den Shift-Operator direkt an das Makro zu übergeben.

10.5 Übersicht von Beispielausgaben der drei Frameworks in einem Fehlerfall

Standardausgabe von Boost.Test bei einem fehlgeschlagenen Testlauf:

```
~/workspace/BoostTest$ ./exampleboosttest

Running 3 test cases...
ScalarTest.cpp(43): error in "AdditionTest": check
                        u.getValue<ValueType>() == 5.0 failed

*** 1 failures detected in test suite "testing"
```

Standardausgabe von CPPUnit bei einem fehlgeschlagenen Testlauf:

```
~/workspace/CppUnitTest$ ./examplecppunittest

ScalarTest::InlineCtorTest : OK
ScalarTest::ScalarGetTypeTest : OK
ScalarTest::AdditionTest : assertion

!!!FAILURES!!!
Test Results:
Run: 3      Failures: 1       Errors: 0

1) test: ScalarTest::AdditionTest (F) line: 43 ScalarTest.cpp
equality assertion failed
- Expected: 5.0
- Actual: 4.0
```

Standardausgabe von GoogleTest nach einem fehlgeschlagenen Testlauf:

```
~/workspace/GoogleTest$ ./examplegoogletest

[==========] Running 3 tests from 1 test case.
[----------] Global test enviroment set-up
[----------] 3 tests from ScalarTest
[ RUN      ] ScalarTest.InlineCtorTest
[       OK ] ScalarTest.InlineCtorTest (0 ms)
[ RUN      ] ScalarTest.ScalarGetTypeTest
[       OK ] ScalarTest.ScalarGetTypeTest (0 ms)
[ RUN      ] ScalarTest.AdditionTest
ScalarTest.cpp:43: Failure
Value of: u.getValue<double>()
  Actual: 4.0
Expected: 5.0
[  FAILED  ] ScalarTest.AdditionTest (0 ms)
[----------] 3 tests from ScalarTest (1 ms total)

[----------] Global test enviroment tear-down
[==========] 3 tests from 1 test case ran. (1 ms total)
[  PASSED  ] 2 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] ScalarTest.AdditionTest

1 FAILED TEST
```

10.6 Skript für Durchführung der CodeCoverage Analyse von LAMA

```
# Creating dir named by YEARS_MONTHS_DAYS-HOURSMINUTES
directory=$(date +%y_%m_%d-%k%M)
mkdir ${directory}

cd ${directory}

# Clearing up environment
/home/lama/bin/lcov --base-directory ../.. --directory ../..
                                                --zerocounters

# Setting environment variable
export LAMA_UNSUPPORTED=IGNORE

cd ..

# Running serial tests
./lama_test

# Running parallel tests serial and with two processes
cd distributed
./lama_dist_test
mpirun -np 2 ./lama_dist_test
cd ..

cd ${directory}

#Running lcov and creating data
/home/lama/bin/lcov --base-directory ../.. --directory ../..
                                --capture --output-file=data.info

#Extracting just Sourcefiles from LAMA/src/*
/home/lama/bin/lcov --extract data.info "*/LAMA/src/*" --output-
                                                file=data.info

# Generating html-structure
/home/lama/bin/genhtml data.info
```

10.7 Codeabdeckungsanalyse von LAMA nach Revision 1250 (April 2012)

Directory	Line Coverage		Functions		Branches	
btests	98.9 %	3051 / 3084	81.0 %	927 / 1144	37.6 %	3332 / 8852
btests/distributed	93.2 %	1256 / 1347	91.7 %	410 / 447	56.7 %	558 / 984
lama	4.4 %	100 / 2292	15.6 %	15 / 96	2.3 %	30 / 1313
lama++	54.9 %	5587 / 10168	47.5 %	1472 / 3096	23.1 %	3290 / 14263
lama++/OpenMP	65.9 %	708 / 1075	71.9 %	105 / 146	57.1 %	377 / 660
lama++/distribution	53.0 %	237 / 447	66.3 %	57 / 86	27.2 %	49 / 180
lama++/distribution/loadBalancing	8.9 %	81 / 914	34.1 %	45 / 132	8.1 %	38 / 467
lama++/exceptions	72.9 %	43 / 59	69.2 %	18 / 26	67.6 %	23 / 34
lama++/expressions	100.0 %	108 / 108	70.0 %	84 / 120	22.2 %	8 / 36
lama++/matutils	85.9 %	195 / 227	81.0 %	17 / 21	84.4 %	92 / 109
lama++/mpi	55.9 %	274 / 490	59.7 %	83 / 139	24.0 %	100 / 417
lama++/norm	97.6 %	40 / 41	88.6 %	31 / 35	50.0 %	12 / 24
lama++/pgas	6.5 %	48 / 737	13.3 %	32 / 240	8.8 %	27 / 306
lama++/pgas/functor	0.0 %	0 / 86	0.0 %	0 / 73	0.0 %	0 / 76
lama++/solver	51.0 %	992 / 1947	62.0 %	207 / 334	29.6 %	430 / 1451
lama++/solver/logger	71.9 %	128 / 178	80.4 %	41 / 51	56.8 %	42 / 74
lama++/tracing	3.2 %	9 / 278	11.1 %	6 / 54	3.7 %	6 / 162
lama/MatrixMarket	23.3 %	54 / 232	40.0 %	6 / 15	13.5 %	27 / 200
lama/cpp	40.5 %	30 / 74	40.0 %	12 / 30	-	0 / 0
logging	40.1 %	114 / 284	54.0 %	34 / 63	25.3 %	48 / 190

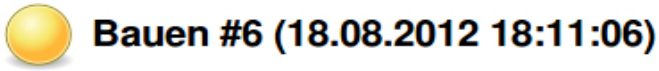
Diese Übersicht legt die Abdeckungen über die beiden Schichten der LAMA-Bibliothek dar. Darunter fällt auch die Abdeckungen der Teststrukturen.

10.8 Codeabdeckungsanalyse von LAMA nach Revision 1633 (August 2012)

Directory	Line Coverage		Functions		Branches	
lama	76.8 %	2817 / 3668	80.9 %	894 / 1105	45.9 %	2723 / 5933
lama/cuda	24.1 %	479 / 1991	41.5 %	76 / 183	14.8 %	316 / 2129
lama/distribution	85.4 %	831 / 973	86.5 %	167 / 193	51.2 %	578 / 1129
lama/exception	34.1 %	62 / 182	82.4 %	14 / 17	18.6 %	29 / 156
lama/expression	95.5 %	85 / 89	87.1 %	74 / 85	40.0 %	12 / 30
lama/io	45.0 %	208 / 462	73.3 %	63 / 86	29.4 %	149 / 506
lama/matrix	63.9 %	1224 / 1914	69.3 %	388 / 560	35.6 %	1215 / 3417
lama/matutils	87.1 %	195 / 224	70.8 %	17 / 24	64.6 %	175 / 271
lama/mpi	80.9 %	398 / 492	77.7 %	108 / 139	45.2 %	328 / 725
lama/norm	100.0 %	40 / 40	94.1 %	32 / 34	50.0 %	12 / 24
lama/openmp	36.2 %	1281 / 3535	61.0 %	169 / 277	28.4 %	1253 / 4405
lama/solver	65.9 %	1104 / 1674	65.4 %	248 / 379	40.8 %	630 / 1544
lama/solver/criteria	88.3 %	189 / 214	95.8 %	69 / 72	38.6 %	44 / 114
lama/solver/logger	91.6 %	186 / 203	89.7 %	52 / 58	66.0 %	62 / 94
lama/storage	59.8 %	2378 / 3978	57.2 %	588 / 1028	35.8 %	2357 / 6578
lama/task	91.7 %	155 / 169	88.6 %	31 / 35	55.0 %	121 / 220
lama/tracing	3.2 %	9 / 280	10.9 %	6 / 55	2.3 %	6 / 258
logging	42.3 %	118 / 279	56.2 %	36 / 64	25.3 %	48 / 190
test	97.2 %	3604 / 3706	84.6 %	1402 / 1657	50.3 %	9117 / 18116
test/cuda	100.0 %	360 / 360	71.0 %	88 / 124	61.8 %	141 / 228
test/cuda/distributed	99.1 %	113 / 114	85.0 %	17 / 20	50.7 %	72 / 142
test/distributed	97.8 %	1846 / 1888	80.7 %	519 / 643	50.6 %	2345 / 4630

Diese Übersicht legt die Abdeckungen über der LAMA-Bibliothek dar, inklusive der Abdeckung der Teststrukturen.

10.9 Testübersicht eines spezifischen Übersetzungsvorganges in Jenkins



Bauen #6 (18.08.2012 18:11:06)

Revision: 1656
Änderungen

1. Added inverse test for matrix storage. (Details)

Build wurde durch eine SCM-Änderung ausgelöst.

Testergebnis (19 fehlgeschlagene Tests / +12)

- lama_test.COOStorageTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MatrixStorageTest.MatrixStorageTestWithCOOStorage<f>
- lama_test.COOStorageTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MatrixStorageTest.MatrixStorageTestWithCOOStorage<d>
- lama_test.CSRStorageTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MatrixStorageTest.MatrixStorageTestWithCSRStorage<f>
- lama_test.CSRStorageTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MatrixStorageTest.MatrixStorageTestWithCSRStorage<d>
- lama_test.DenseStorageTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MatrixStorageTest.MatrixStorageTestWithDenseStorage<f>
- lama_test.DenseStorageTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MatrixStorageTest.MatrixStorageTestWithDenseStorage<d>
- lama_test.DIAStorageTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MatrixStorageTest.MatrixStorageTestWithDIAStorage<f>
- lama_test.DIAStorageTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MatrixStorageTest.MatrixStorageTestWithDIAStorage<d>
- lama_test.ELLStorageTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MatrixStorageTest.ELLStorageWithMatrixStorageMethodsTest<f>
- lama_test.ELLStorageTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MatrixStorageTest.ELLStorageWithMatrixStorageMethodsTest<d>
- lama_test.JDSStorageTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MatrixStorageTest.MatrixStorageTestWithJDSStorage<f>
- lama_test.JDSStorageTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MatrixStorageTest.MatrixStorageTestWithJDSStorage<d>
- lama_test.MetaSolverTest./var/lib/jenkins/jobs/Job_Lama/workspace/libama/trunk/src/test/MetaSolverTest.testSolve<d>
- lama_dist_test.P_InverseSolverTest.testSolve<f>
- lama_dist_test.P_InverseSolverTest.testSolve<d>
- lama_test.InverseSolverTest.InverseTest<f>
- lama_test.InverseSolverTest.InverseTest<d>
- lama_test.InverseSolverTest.InverseTest2<f>
- lama_test.InverseSolverTest.InverseTest2<d>