Reengineering of a legacy complex FORTRAN 77 simulation system

Markus Hobisch

Reengineering of a legacy complex FORTRAN 77 simulation system

Master's Thesis
at
Graz University of Technology

submitted by

Markus Hobisch, BSc

Institute for Software Technology,
Graz University of Technology,
A-8010 Graz, Austria

5th August 2014

This thesis is written in German.

Advisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Reengineering eines komplexen FORTRAN 77 Altsystems

Masterarbeit
an der
Technischen Universität Graz

vorgelegt von

Markus Hobisch, BSc

Institute für Softwaretechnologie,
Technische Universität Graz,
A-8010 Graz, Austria

5. August 2014

Betreuer: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …… ……..

(Unterschrift)

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…… ……..

date (signature)

Danksagung

An dieser Stelle möchte ich mich bei all jenen Bedanken die mich im Laufe meiner Diplomarbeit unterstützt haben.

Insbesondere widme ich diese Arbeit meinen Eltern als Dankeschön und Anerkennung für ihre jahrelange Unterstützung und Geduld während meines Studiums.

Mein Dank geht zudem an das Unternehmen JOANNEUM RESEARCH, insbesondere an Univ. Doz. Dr. Johann Fank, der dieses Projekt ermöglicht hat. Des Weiteren möchte ich mich bei Mag. Dr. Gernot Klammler sowie Ing. Gerhard Rock für die tatkräftige und fachkundige Unterstützung vor Ort bedanken.

Abschließend bedanke ich mich bei Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany für die Betreuung der Diplomarbeit.

Zusammenfassung

Die Diplomarbeit entstand infolge einer Zusammenarbeit zwischen dem Institut für Softwaretechnologie, des Unternehmens JOANNEUM RESEARCH in Graz und dem Institut für Kultur und Bodenwasserhaushalt am Bundesamt für Wasserwirtschaft in Petzenkirchen.

Ausgangspunkt war eine in FORTRAN 77 entwickelte Software namens STOTRASIM. Das Programm wird in den Bereichen Hydrologie und Bodenphysik zu der Berechnung von Wasser- und Stofftransport in ungesättigten Zonen eingesetzt. Ziel der Diplomarbeit war das Reengineering dieses sogenannten Altsystems als objektorientierte Anwendung. Dabei sollte die vorhandene Funktionalität der FORTRAN 77 Simulation erhalten bleiben, aber zugleich der Programmcode für geplante Erweiterungen vorbereitet werden.

Diese Arbeit befasst sich mit dem theoretischen Hintergrund und der praktischen Umsetzung eines Reengineering Prozesses.

Abstract

This thesis is the result of a collaboration between the Institute for Software Technology, the company JOANNEUM RESEARCH in Graz and the Institute for Land & Water Management Research at the Federal Agency for Water Management in Petzenkirchen.

The starting point was a software developed in FORTRAN 77 called STOTRASIM. The program is used in the fields of hydrology and soil physics for the calculation of water and solute transport in unsaturated zones. The aim of the thesis was the reengineering of legacy system as a so-called object-oriented application. On the one hand the existing functionality of the FORTRAN 77 application should remain, but on the other side the program should have been prepared for future expansion.

This work deals with the theoretical background and the practical implementation of a re-engineering process.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Diese Arbeit befasst sich mit der Umsetzung eines Reengineering Prozesses eines Altsystems (engl. Legacy System). Die Arbeit ist einen theoretischen sowie praktischen Teil gegliedert.

Zu Beginn werden im theoretischen Teil die Grundlagen des Reengineering Prozesses dargestellt. Dieser umfasst Definitionen sowie Techniken bzw. Abläufe, wie sie in einem typischen Reengineering Prozess vorkommen können.

Der praktische Teil der Arbeit wurde in Kooperation mit dem Unternehmen JOANNEUM RESEARCH durchgeführt. Inhalt dieser Kooperation war die Neuentwicklung einer in die Jahre gekommener FORTRAN 77 Entwicklung. Ziel ist es, das in die Jahre gekommene Altsystem STOTRASIM durch eine objektorientiert entwickelte C# Neuentwicklung zu ersetzen.

Bei STOTRASIM handelt es sich um ein Simulationsprogramm, welches den Wasser- und Stickstoffkreislauf in landwirtschaftlich genutzten Böden berechnet.

2 Theorie

In diesem Kapitel wird der theoretische Hintergrund der Arbeit behandelt. Es soll die Basis für die praktische Arbeit darstellen und das notwendige Handwerkszeug für die Umsetzung des Reengineering Prozesses liefern.

Zu Beginn des Theorieteils wird erklärt was ein Legacy System eigentlich ist und welche Symptome bzw. Merkmale es kennzeichnen. Des Weiteren werden Vorschläge geliefert wie man mit einem Altsystem (engl. Legacy System) weiter verfahren kann. Die folgenden Kapitel erklären wie ein Reengineering Prozess beispielsweise aussehen kann und welche Schritte dieser umfasst. Zum Abschluss des Theorieteils wird noch kurz die Thematik des Testens behandelt.

2.1 Legacy System

Ein Legacy System, auf Deutsch Altsystem, ist für ein Unternehmen eine Ressource von besonderer Bedeutung. Oft sind Legacy Systeme über Jahre in einem Unternehmen gewachsen und ein Ausfall des Altsystems ist für das Unternehmen für gewöhnlich kritisch. Deswegen sind Unternehmen bedacht darauf Altsysteme am Leben zu halten, da ein Austausch des Systems entweder zu kostspielig bzw. ökonomisch unrentabel oder auch einfach technisch nicht machbar ist.

Ein Legacy System wird nicht unbedingt durch sein Alter definiert. Wenn man die folgenden Punkte betrachtet, werden viele dieser Punkte mit zunehmendem Alter wohl eher eintreten, aber trotzdem ist das Alter kein Definitionspunkt an sich.

Kernmerkmale eines Legacy Systems:

- Unstrukturierte Kontrollflüsse
- Fehlende Datenstrukturierung
- Veraltete Programmiersprachen
- Veraltete Methoden der Softwareentwicklung
- Software läuft nur auf alten Plattformen
- Compiler sind nicht mehr verfügbar bzw. nicht lauffähig
- Fehlende, fehlerhafte bzw. kaum vorhandene Dokumentation
- Hoher Wartungsaufwand und somit hohe Kosten
- Hoher Aufwand bei Erweiterungen
- Personalfluktuation durch beispielsweise Jobwechsel oder Pensionierungen
- Hohe Abhängigkeit von einzelnen Personen, die über das gesamte Wissen verfügen

Wie man auf den ersten Blick erkennen kann, gibt es verschiedenste Gründe warum eine Software ein Legacy System darstellen kann. Vor allem die ersten Punkte gehen meist mit dem Alter eine Software einher. Im Laufe der Jahre werden Programmiersprachen von moderneren praktikableren Programmiersprachen abgelöst. Die prozedurale Programmierung wurde weitestgehend durch die objektorientierte Programmierung abgelöst.

Ein anderes Problem kann die Verfügbarkeit der Compiler bzw. der ursprünglichen Entwicklungsumgebung darstellen. Problematisch wird es auch wenn zwar ein Compiler verfügbar, dieser aber auf aktuellen Betriebssystemen nicht mehr lauffähig ist.

Ein weiterer wichtiger Punkt ist die Dokumentation. Hier gibt es mehrere Szenarien. Einerseits kommt es vor, dass es für eine Software, die vor Jahrzehnten entstand, überhaupt keine Dokumentation gibt, weil sie entweder nie angefertigt wurde oder weil sie im Laufe der Jahre verloren ging. Ein anderes Szenario wäre, dass zwar eine Dokumentation vorhanden ist, diese aber aufgrund von Änderungen bzw. Erweiterungen der Software nicht mehr gültig ist.

Oder ursprünglich gab es zwar eine Dokumentation aber bei Änderungen im Sourcecode wurde darauf vergessen dementsprechende Änderungen in der Dokumentation durchzuführen.

Bei komplexen Softwaresystemen führt insbesondere die fehlende Dokumentation bei Wartungsarbeiten oder Änderungen zu immensen Zeitaufwänden und in weiterer Folge zu unnötigen Kosten. Es kommt auch vor, dass manche Codepassagen gar nicht mehr entziffert werden können. Beispielsweise waren in FORTRAN 77 Variablennamen auf sechs Zeichen beschränkt. Dies zwang viele Entwickler teils unlesbare Abkürzungen zu wählen, die für den Moment zwar eindeutig waren aber von einer anderen Person nicht entziffert werden können. Woher soll nun ein Entwickler, der nach Jahren Änderungen durchführen soll wissen, was diese Abkürzungen bedeuten, es sei denn es gibt eine entsprechende Dokumentation.

Dieser enorme Zeitaufwand der für Wartung oder Änderungen aufgewendet werden muss, bindet natürlich wichtige Ressourcen wie Arbeitskräfte und Geld, die in anderen Bereichen, wie etwa für Neuentwicklungen, fehlen. Laut Bernado Wagner binden im Bereich großer Informationssysteme Wartung und Weiterentwicklung 80% aller verfügbaren Softwareentwickler.[1]

2.1.1 Alterungssymptome

Guiseppe Visaggio[2] hat anhand eines Reengineering Projekts eines komplexen Softwaresystems für Legacy Systeme bestimmte Alterungssymptome festgestellt:

- Verunreinigung - Pollution
- Eingebettetes Wissen - Embedded knowledge
- Mangelhafte Benennungen - Poor lexicon
- Kopplung - Coupling

[1] Vgl. [WAGNER1, S.42].

[2] Vgl. [VISAGGIO1].

- Schichtenarchitekturen - Layered architectures

2.1.1.1 Verunreinigung - Pollution

Eine Software verfügt über zahlreiche Komponenten die der Benutzer nicht braucht und auch nicht benützt. Hintergrund ist der, dass diese Funktionen vielleicht einmal benötigt wurden, in der Zwischenzeit aber überflüssig sind. Möglicherweise wurde die entsprechende Funktion bei Wartungsarbeiten zwar deaktiviert, die entsprechenden Codeteile im Quellcode aber nicht entfernt. Solche Verunreinigungen machen eine Software unnötig komplex und erschweren dadurch die Systemanalyse.

2.1.1.2 Eingebettetes Wissen - Embedded knowledge

Hiermit beschreibt der Autor das Problem, dass alle Erkenntnisse einer Software und deren Weiterentwicklung nicht mehr aus der Dokumentation ableitbar sind, sondern nur durch das Lesen des Programmcodes. Verantwortlich dafür sind Programmänderungen, die in der Dokumentation nicht mehr vermerkt wurden. Die Wartung der Software wird durch fehlende Dokumentation ungemein schwerer und Folgen von Änderungen sind für die Entwickler teilweise nicht mehr abschätzbar.

2.1.1.3 Mangelhafte Benennungen - Poor lexicon

Mit mangelhafter Benennung sind schlecht gewählte Variablen-, Funktions- oder Klassennamen gemeint. Anhand dieser schlecht gewählten Namen ist die wahre Bedeutung der Benannten oft nur mehr schwer nachvollziehbar. Vor allem Beschränkungen bei der Länge der Namen können für schwer zu entziffernde Konstruktionen führen. Als Beispiel sei hier FORTRAN 77 erwähnt, in der man mit sechs Zeichen auskommen musste und auch die Groß- und Kleinschreibung ignoriert wurde. In Fortran90 waren es dann schon immerhin 31 Zeichen.

2.1.1.4 Kopplung – Coupling

Miteinander verbundene Anwendungen und Komponenten einer Software weisen eine sehr hohe Abhängigkeit auf. Eine Kopplung tritt auf, wenn zu wenig auf die Entropie der Software

geachtet wird. Durch die hohe Abhängigkeit werden Änderungen des Systems sehr aufwendig und kostenintensiv.

2.1.1.5 Schichtenarchitekturen - Layered architectures

Unter diesem Punkt beschreibt der Autor das Problem, das im Laufe der Zeit eine ursprünglich qualitativ hochwertige Grundstruktur in Folge von Wartungsarbeiten und Erweiterungen, immens an Qualität einbüßt. Er führt das darauf zurück, dass die Entwickler im Laufe des Wartungsprozesses mit Problemen zu kämpfen haben, die in der Planungsphase noch nicht vorhersehbar waren und diese beim Lösen selbst in den Planungsprozess miteingreifen. In weiterer Folge wird die Wartung des Systems immer komplizierter und es erhöht sich die Gefahr von Redundanzen.

2.1.2 Was tun mit einem Legacy System

Hat man ein Altsystem identifiziert, stellt sich die Frage was man damit anfängt.

a) Neuentwicklung

b) Systemersetzung

c) Systemerhaltung

d) Systementfernung

Ad a) Die erste Option stellt eine völlige Neuentwicklung des Systems dar. Hier stellt sich immer die Frage, ob es wirtschaftlich für ein Unternehmen zu stemmen ist und ob der Kosten-Nutzen-Faktor stimmt.

Ad b) Eine andere Möglichkeit ist es, das System durch eine anderes zu ersetzen. Dabei gilt es vorab abzuklären ob es überhaupt Systeme gibt, die die gewünschte Funktionalität bieten.

Ad c) Bieten keine der vorher genannten Möglichkeiten eine akzeptable Lösung, kann man immer noch versuchen das System so lange wie nur irgendwie möglich am Leben zu erhalten.

Ad d) Die drastischste Lösung stellt das Aussortieren eines Legacy Systems dar. Wenn man darauf jedoch verzichten kann, stellt dies die einfachste Möglichkeit dar.

2.2 Reengineering

Zu Beginn stellt sich natürlich die Frage, was ist ein Reengineering eigentlich. Hier trifft man in der Literatur immer wieder auf die Aussage der Herren Hammer & Champy, die sich mit dem Reengineering von Geschäftsprozessen beschäftigen:

"(..) the fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical, contemporary measures of performance, such as cost, quality, service and speed." [3]

Reengineering ist demnach also das grundlegende Umdenken bei Geschäftsprozessen und damit geht eine radikale Neugestaltung dieser einher. Dadurch sollen erhebliche Verbesserungen bei messbaren Kriterien wie Kosten, Qualität, Service und Geschwindigkeit erzeugt werden.

Reengineering ist somit der Prozess der Neugestaltung bestehender Systeme. Dieses System kann die Organisationsstruktur eines Unternehmens sein, der Produktionsablauf eines Betriebes oder in der Softwareentwicklung ein so genanntes Legacy System. Die Ziele sind in allen Bereichen gleich: qualitativ bessere, effizientere und damit kostengünstigere Systeme.

Auf die Softwareentwicklung umgelegt bedeutet Reengineering also die Bearbeitung oder Modifizierung von einer bereits existierenden Software, einem so genannten Legacy System, welches meist in der Wartung unrentabel geworden ist.

Der Prozess des Reengineering wird in der Literatur immer wieder unterschiedlich aufgefasst. Die gängigste Variante ist das Reengineering als Oberbegriff für den kompletten Prozess der Sanierung einer Software heranzuziehen. Dies umfasst Schritte wie die Überprüfung, Analyse und Modifikation oder Änderung eines bestehenden Programms.

[3] [HAMMER1, S. 32].

Nach Helmut Balzert[4] ist das Reengineering ein Teil der Sanierung eines Altsystems. Dieser Prozess der Sanierung besteht aus den Teilen Reverse-Engineering, Reengineering und Forward-Engineering. Hier stellt das Reverse-Engineering eine Voraussetzung für die Aktivitäten des Reengineering dar. Laut Balzert umfasst Reengineering „alle Aktivitäten zur Änderung von Software-Altsystemen, um sie in einer neuen Form wieder implementieren zu können“[5]. Wie in Abb. 2.1 zu sehen ist, können diese Änderungen auf verschiedenen Ebenen stattfinden. Wird Reengineering demnach auf der Implementierungsebene angewandt, entfallen in weiterer Folge mit der Entwurfs- bzw. die Definitionsüberarbeitung die darüber liegenden Ebenen. Je höher man in der Grafik das Reengineering ansetzt desto grundlegender werden die Änderungen am Altsystem.

Nach Balzert[6] lässt sich das Reengineering, oder wie er es nennt die Sanierung, in drei Kategorien teilen:

- Verpacken von Altsystemen
- Verstehen von Altsystemen
- Verbessern von Altsystemen

In den folgenden drei Unterkapiteln wird auf die Details der einzelnen Arten der Sanierung nach Bazert genauer eingegangen.

[4] Vgl. [BALZERT1, S.665ff].

[5] [BALZERT1, S.667].

[6] Vgl. [BALZERT1, S. 669].

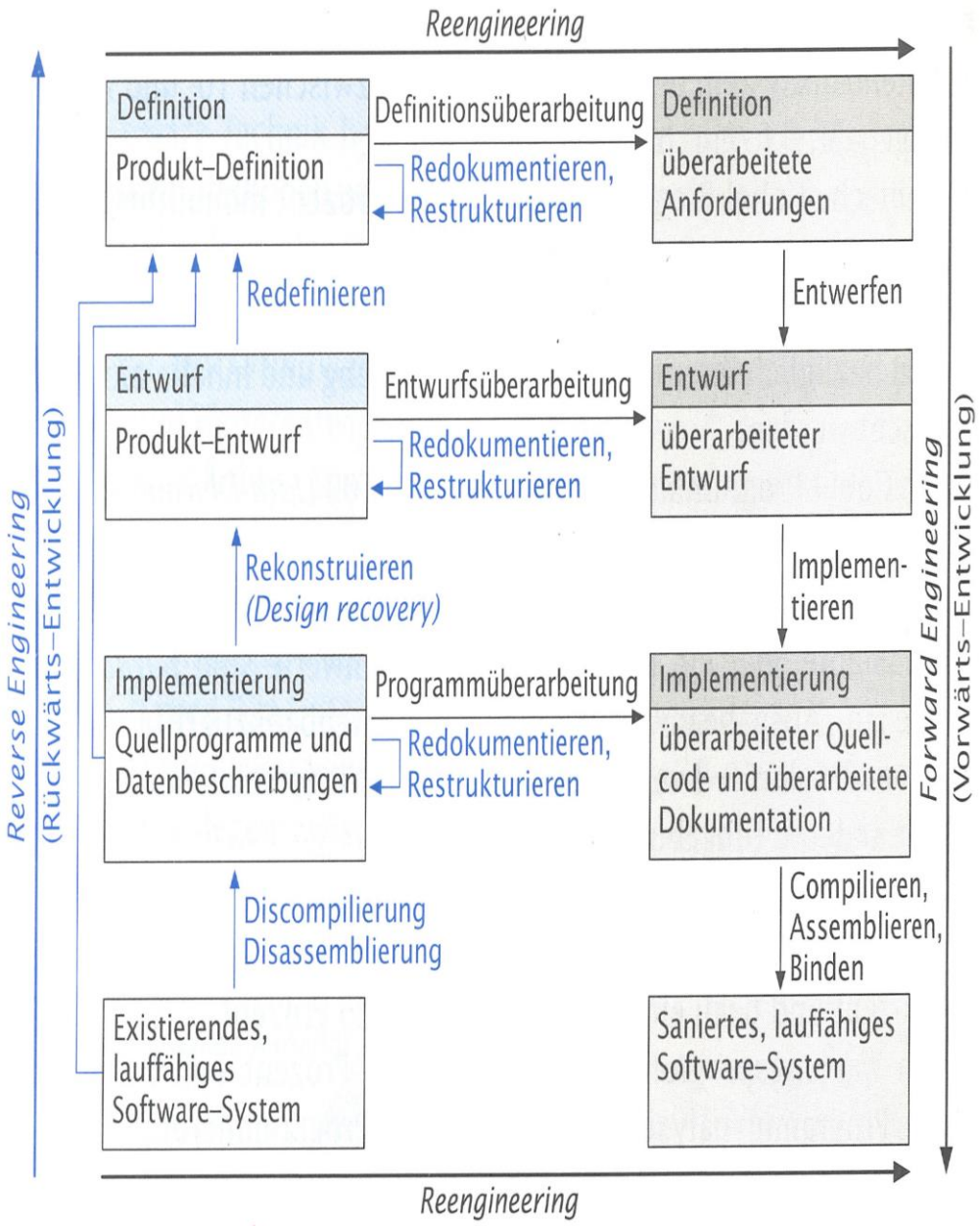

Abb. 2.1 Terminologie der Softwaresanierung[7]

2.2.1 Verpacken

Unter Verpacken eines Altsystems (engl. wrapping) kann man nicht wirklich eine Sanierung eines Altsystems verstehen. Vielmehr geht es darum, entweder das ganze System oder auch

[7] [BALZERT1, S.666].

nur einen Teil so zu verpacken, dass eine Benutzung in einer objektorientierten (Neu-) Entwicklung möglich ist. Der Benutzer der objektorientierten Entwicklung soll dabei nichts mehr vom Altsystem mitbekommen. Ein großer Vorteil des Verpackens besteht darin, dass man das Altsystem relativ schnell im Zuge einer Neuimplementierung benutzen kann und dieses bei Bedarf aber auch später noch durch eine Neuentwicklung ersetzen kann, ohne dass der Benutzer dies mitbekommt.

Balzert beschreibt zwei Arten von Verpackern:

a) Objektorientierter Verpacker

b) Prozeduraler Verpacker

Ad a) Beim objektorientierten Verpacker wird von einem objektorientierten Programm mithilfe der Verpacker-Klasse auf ein Altsystem zugegriffen.

Ad b) Der prozedurale Verpacker läuft genau in die andere Richtung. Hier wird von einem Altsystem auf objektorientierte Programme zugegriffen.

In beiden Fällen bleiben die Schnittstellen und Datenstrukturen dem jeweils anderem System verborgen.

2.2.2 Verstehen

Um ein Altsystem zu sanieren, muss man logischerweise wissen was eigentlich in diesem vorgeht. Deswegen ist das Verstehen des Altsystems ein essentieller Teil des Reengineering. Bei diesem Verstehen handelt es sich um das Wiedererlangen von verlorenem Wissen, sei es weil zur Zeit der Entwicklung des Altsystems auf Dokumentation verzichtet wurde oder weil durch Änderungen am System die Dokumentation fehlerhaft und damit nicht mehr am aktuellsten Stand ist. Auch hier führt Balzert verschiedene Methoden des Verstehens an:

- Verstehen als Black Box
- Verstehen als White Box
- Mischformen beider Methoden

Das Verstehen als Black Box gliedert Balzert in drei Schritte:

a) Reverse Engineering

b) Reengineering

c) Forward Engineering

Ad a) Hier soll anhand des Benutzerhandbuches, und falls vorhanden den Dialogen im laufenden System ein objektorientiertes Analyse-Modell(OOA) erstellt werden.

Ad b) In diesem Schritt wird das OOA-Modell verbessert und umstrukturiert.

Ad c) Das OOA-Modell wird als neues System umgesetzt.

Verstehen als White Box wiederum besteht für Balzert aus:

- Redokumentation
- Restrukturierung

Der Unterschied dieser zwei Vorgehensweisen besteht darin, dass bei ersterer Methode nur auf Ein- und Ausgaben des Systems bzw. etwaige vorhandene Dokumentationen zurückgegriffen wird. Bei der zweiten Methode wird hingegen das innere Verhalten des Systems genauer betrachtet.

Balzert weist darauf hin, dass aber in den meisten Fällen die Methoden der Black Box ausreichen um mit geringem Aufwand ein OOA-Modell zu erstellen und dieses in ein neues System umzusetzen.

2.2.3 Verbessern

Der Vorgang des Verbesserns setzt nach einem erfolgreichen Reverse Engineering Prozess ein. Hat man genug Wissen und Verständnis über das Altsystem erlangt, kann man notwendige Verbesserungen durchführen. Dieser Schritt ist mit dem Prozess des Forward Engineerings gleichzusetzten.

2.2.4 Unterschiede Forward Engineering und Reengineering

In der folgenden Tabelle sind ein paar typische Unterschiede zwischen dem klassischen Softwareentwickungsprozess, welches als Forward Engineering bezeichnet wird um es eindeutig vom Reengineering bzw. Reverse Engineering zu unterscheiden, und dem Reengineering Prozess angeführt.

Forward Engineering	**Reengineering**
Etwas Neues entwickeln	Vorhandenes verbessern
Leeres Blatt	Fertiges System
Oft unklares Ziel	Klares Ziel

Tab. 2.1 Unterschiede zwischen Forward Engineering und Reengineering

Der wesentlichste Unterschied ist, dass man in der klassischen Softwareentwicklung von Null anfängt. Der Entwickler sitzt quasi vor einem leeren Blatt und es gilt für ihn in der Anforderungsanalyse die Wünsche und Bedürfnisse des Kunden an die zu entwickelnde Software herauszufinden. Die Ziele werden oft erst im Laufe der Analyse klar oder ändern sich später während der Entwicklung.

Im Reengineering haben wir bereits eine fertige Software, die es bei gleichbleibender Funktionalität unter gewissen Voraussetzungen neu zu entwickeln gilt. Das heißt, das Ziel ist hier von vorhinein vorgegeben.

2.2.5 Beispiele für Reengineering

In der folgenden Aufzählung sind ein paar typische Beispiele für Reengineeringtätigkeiten aufgeführt:

- Spaghetti-Code in strukturiertem Code
- Den kompletten Source Code in einer neuen Programmiersprache schreiben

- Umstellung von einer Stand-Alone Applikation, wie etwa früher bei Großrechnern, zur Client-Server-Architektur

- Die Einführung einer grafischen Benutzeroberfläche bei einer Konsolenapplikation, während die Funktionalität des Programms gleich bleibt

2.3 Reverse Engineering

Am Anfang eines Reengineering Prozesses steht so gut wie immer der Reverse Engineering Prozess. „So gut wie immer“ deshalb, weil die beauftragten Entwickler in den seltensten Fällen mit ausreichend Verständnis über das neu zu entwickelnde System ausgestattet sind. Ähnlich verhält es sich beim Wartungsprozess. Auch hier sind für gewöhnlich zumindest einfache Analysen des Codes unumgänglich, um die betroffenen Systeme im Code auch zu verstehen und somit die Wartungsarbeiten durchführen zu können.

Reverse Engineering befasst sich mit der Wiedergewinnung von Programmspezifikationen und -dokumentationen. Im Idealfall stellen die wiedergewonnen Dokumente eine vollständige und konsistente Repräsentation des Altsystems dar.

Chu et al.[8] fassen für das Reverse Engineering drei Ziele bzw. Gründe zusammen:

- Systemverstehen

- Grundlage für Wartung bzw. Neuentwicklung

- Wiederverwendung von Komponenten

Das Verstehen eines Systems gilt als eine der Grundaufgaben des Reverse Engineerings. Es dient in weiterer Folge als Grundlage für jeden Wartungsprozess, aber auch für eine etwaige Weiterentwicklung.

[8] Vgl. [CHU1].

Reverse Engineering kann sowohl manuell als auch automatisch durchgeführt werden. Dies kann von verschiedenen Faktoren abhängen, wie zum Beispiel der Komplexität des vorliegenden Altsystems, aber auch vom vorhandenen Budget.

Es gibt verschiedene Techniken um eine abstraktere Repräsentation des Altsystems zu erhalten. Man kann diese Techniken in die *statische* und *dynamische* Analyse unterteilen.

Die *statische* Analyse untersucht die Struktur und den Aufbau des Altsystems und wird durchgeführt ohne dass das System läuft. So können etwa Schwachstellen des Codes aufgedeckt und Ansätze zur Verbesserung geschaffen werden.

Nach Bennicke und Rust[9] zeichnet sich die *statische* Programmanalyse vor allem dadurch aus, dass sie keine Programmeingaben benötigt und zugleich für alle möglichen Programmeingaben eine allgemeingültige Aussage treffen kann. Des Weiteren führen sie an, dass Aussagen über die tatsächliche Struktur des Programms gemacht werden können, weil bei den Analysen der Quelltext als exakte Beschreibung verwendet wird. Statische Analysen verlangen keinen lauffähigen Code. Allerdings können die Berechnungen wegen der allgemein gültigen Aussage zeitlich und speichermäßig sehr aufwändig werden.

Balzert[10] führt verschiedene *statische* Analysemethoden an:

- Lexikalische Analyse
- Syntaktische Analyse
- Semantische Analyse
- Datenflussanalyse
- Strukturanalyse
- Wirkungsanalyse

[9] Vgl. [BENNICKE1].

[10] Vgl. [BALZERT1] .

- Querverweis-Analyse

Die *dynamische* Analyse geschieht zur Laufzeit und versucht das Programmverhalten des Altsystems zu untersuchen. Hier identifizieren Bennicke und Rust[11] den Vorteil, dass man spezielle Anwendungsszenarien durchspielen kann, welche nur bestimmte Teile des Programms nutzen. Die gewonnenen Informationen sind in ihrer Quantität und Qualität über die der *statischen* Programmanalyse zu stellen, weil diese das Programmverhalten nur abschätzen. Da *dynamische* Analysen während der Laufzeit durchgeführt werden, setzt das natürlich voraus, dass für das Altprogramm ein lauffähiger Quellcode vorhanden ist.

2.3.1 Reverse Engineering Tools

Reverse Engineering Tools unterstützen den Entwickler bei der Re-Spezifikation von Programmen. Am Markt ist eine Vielzahl von Programmen verfügbar. Diese reichen von einfachem Verstehen, Evaluieren und Re-Dokumentieren des Programmcodes bis hin zu grafischen Darstellungen von Klassen- und Funktionsabhängigkeiten.

Schon in den 1990er Jahren gab es verschiedene Programme (CASE) die den Entwickler bei seiner Arbeit unterstützten. Beispielsweise soll das „Entwicklungs- und Projektmanagement Orientierte Spezifikationssystem“ (kurz EPOS) dem Softwareentwickler bei der Wartung und bei Erweiterungsmaßnahmen unter die Arme greifen. So gibt es für EPOS Erweiterungen, die dem Entwickler die Arbeit beim Reverse-Engineering erleichtern. Mit Hilfe des Werkzeuges RE-SPEC kann der EPOS-Anwender nachträglich für Programmiersprachen wie Fortran oder Pascal automatisch eine Re-Spezifikation durchführen und damit eine Systemspezifikation erzeugen. Ein anderes EPOS-Werkzeug, nämlich RE-DOC, generiert aus einem Quellcode auch nachträglich, also lange nach der Fertigstellung des Programms, automatisch eine Programmdokumentation.

Ein Vergleich verschiedener Reverse Engineering Werkzeuge zeigt, dass es kein ultimatives Werkzeug gibt. Jedes Tool hat seine eigenen Stärken und Schwächen und liefert für seinen speziellen Bereich, für den es entwickelt wurde, gute Ergebnisse.

[11] Vgl. [BENNICKE1].

Bellay und Gall identifizieren Schwächen, die die meisten Reverse Engineering Tools gemeinsam haben. Die untersuchten Werkzeuge liefern zwar einen sehr detaillierten Überblick der untersuchten Altsoftware aber insgesamt sind sehr häufig noch manuelle Adaptionen nötig, um ein korrektes Abbild zu erhalten. Zudem ist für viele Schritte beim Ausführen der Tools ein hohes Maß an programmspezifischem Wissen notwendig. Des Weiteren sind die vielen Programme, auch wenn sie prinzipiell mehrere Programmiersprachen unterstützen, selten in der Lage, Projekte korrekt zu parsen, welche mehrere Programmiersprachen auf einmal beinhalten. [12]

2.4 Refactoring

Das Refactoring, auch Restructuring genannt, befasst sich mit der Neuorganisation des Codes oder auch Neustrukturierung. Ein wichtiger Punkt bei dieser Tätigkeit ist, dass das Verhalten des Programms in keiner Weise verändert wird. Es geht also darum, einen bestehenden Code zu verbessern und ihn leichter verständlich zu machen, um in weiterer Folge Wartung und Weiterentwicklungen zu erleichtern.

Besonders bei altem Code – funktionaler Code wird in objektorientierten überführt – kommt man nicht ohne Refactoring aus. Da hier die Struktur vom funktionalen Code nicht der vom objektorientierten Code entspricht.

Nach Balzert[13] umfasst das Restrukturieren folgende Möglichkeiten:

a) Reformatierung

b) Code-Restrukturierung

c) Modularisierung

d) Datenstandardisierung

[12] Vgl. [BELLAY1].

[13] Vgl. [BALZERT1, S.674].

Ad a) Hierbei wird der Quellcode optisch aufbereitet um ihn leichter lesbar zu machen.

Ad b) Hier gilt vor allem für einen unstrukturierten Code die Einführung von linearen Kontrollstrukturen (zB. Eliminierung von GOTOs).

Ad c) Eine monolithische Codestruktur wird in Module zerlegt.

Ad d) Hierzu gehört die Einführung von einem einheitlichen Schema für Benenner wie Variablennamen, Klassennamen etc.

Refactoring kann man in verschiedenen Größenordnungen durchführen, seien es simple Umbenennungen von Variablennamen oder größere strukturelle Veränderungen. Sebastian Kübeck[14] spricht hier von elementaren Refactorings bzw. Mikro-Refaktorings für einfache Änderungen und umfangreiches Refactoring für größere strukturelle Veränderungen, welche aber auch wieder aus elementaren Refactorings bestehen. Kübeck führt auch einige elementare Refactorings auf:

a) Umbenennung von Bezeichnern

b) Methode extrahieren (Extract Method)

c) Methode auflösen (Inline Method)

d) Methode verschieben (Move Method)

e) Methode hochziehen (Pull-up Method)

f) Interface extrahieren (Extract Interface)

g) Klasse extrahieren (Extract Class)

Ad a) Bei Umbenennung von Bezeichnern handelt es sich um die Umbenennung von Variablen, Methoden, Klassen etc. Diese Tätigkeit scheint im ersten Augenblick eher nebensächlich. Aber ganz im Gegenteil, die sorgfältige Wahl dieser Bezeichner kann viel Zeit und somit Kosten sparen. Durch aussagekräftige und gut gewählte Namen wird jeder

[14] Vgl. [KÜBECK1, S.144ff].

Quellcode viel leichter zu lesen. Spätere Wartungsarbeiten oder Erweiterungen werden nicht mehr durch lästiges Durchsuchen von Dokumentationen verzögert nur weil man aufgrund der Namen nicht auf die Bedeutung der Variablen oder gar ganzer Funktionen schließen kann. Früher mag man aufgrund gewisser Restriktionen, wie maximale Länge für Bezeichner, noch gezwungen gewesen sein für Bezeichner oft unleserliche Namen zu wählen. Heute bestehen allerdings solche Einschränkungen nicht mehr und gerade beim Reengineering oder der Sanierung gilt es nun, sich die nötige Zeit zu nehmen und geeignete und vor allem aussagekräftige Namen zu finden. In Tab. 2.2 sieht man gut wie ein Quellcode aussehen kann, wenn man an bestimmte Restriktionen gebunden ist. Für den Entwickler, der diese Zeilen geschrieben hat, sind die Abkürzungen bei späterer Betrachtung vielleicht noch klar, für jemanden, der den Code zum ersten Mal sieht, ist es sehr schwer nachzuvollziehen was hier Sache ist.

```
    DO 200 I=1,NCURVE
     IF(IWS(I).GT.IPV) GO TO 220
200   CONTINUE
220   I1=I-1
     I2=I
     if(i1.eq.0) then
     i1=1
     xs=1
     goto 230
     end if
     X1=(IPV-IWS(I1))
     X2=(IWS(I2)-IWS(I1))
     xx=X1/X2
```

```
230   DO 300 I=1,20

      READ(3,1030)                                    P1(I),VAL
      AP(I)=DLOG10(P1(I))

      wa(I)=VAL(1,I1)+(VAL(1,I2)-VAL(1,I1))*xx

      if(p1(i).eq.0.4) wss=wa(i)

      A=VAL(2,I1)+(VAL(2,I2)-VAL(2,I1))*xx
      B(I)=VAL(3,I1)+(VAL(3,I2)-VAL(3,I1))*xx

300   AK(I)=DLOG10(A)
```

Tab. 2.2 Beispiel für einen schwer zu lesenden Code aus STOTRASIM

Ad b) Lange Subroutinen bzw. Methoden, die sich über mehrere hundert Zeilen ziehen, auf Dauer zu übernehmen, macht keinen Sinn und erschwert zukünftige Wartungsaufgaben aber auch die Testbarkeit ungemein. Methoden zu extrahieren stellt daher eine wichtige Tätigkeit des Refactoringprozesses dar.

Ad c) Der Arbeitsschritt „Methode auflösen“ stellt das Gegenstück zu „Methode extrahieren“ dar. Der Grund für diesen Schritt mag sein, dass eine zuvor extrahierte Methode sich im Nachhinein doch nicht als sinnvoll herausstellt oder der betroffene Code in einer anderen Methode besser aufgehoben ist.

Ad d) Unter Methode verschieben versteht man das Verschieben einer ganzen Methode von einer Klasse in eine andere.

Ad e) Unter Methode hochziehen versteht man das Verschieben einer Methode von der Kindklasse in die Mutterklasse. Dies ist dann sinnvoll wenn in zwei oder mehreren abgeleiteten Klassen dieselbe Methode implementiert ist.

Ad f) Beim Extrahieren eines Interfaces wird aus einer Klasse ein Interface abgeleitet. Die ursprüngliche Klasse stellt dann die abgeleitete Klasse des Interfaces dar. Der Vorteil liegt darin, dass die Kindklasse durch andere vom Interface abgeleitete Klassen jederzeit ersetzt werden kann.

Ad g) Das Extrahieren einer Klasse läuft ähnlich ab wie das Extrahieren eines Interfaces. Der Unterschied liegt darin, dass in der Elternklasse Methoden auch direkt implementiert bzw. von der Kindklasse übernommen werden können, während im Interface nur abstrakte Klassen zur Verfügung stehen und somit die Implementierung in der ehemaligen Kindklasse zu finden ist.

An dieser Stelle sei auch die Wichtigkeit von Testfällen erwähnt. Auch wenn man aufgrund des Refactorings die Testfälle möglicherweise adaptieren muss, kann man mit ihrer Hilfe sicherstellen, dass nach allen strukturellen Veränderungen das Verhalten des Programms unangetastet bleibt.

2.4.1 Tägliche Hygiene

Bei der täglichen Hygiene handelt es sich, um vielleicht auf den ersten Blick, lästige Arbeiten. Doch gerade diese Tätigkeiten stellen einfache, aber sinnvolle und effektive Refactoringarbeiten dar um einen besseren Code zu produzieren. Wenn man sich regelmäßig um gewisse Dinge kümmert, spart man sich in Zukunft viel Zeit und Arbeit. Kübeck[15] identifiziert drei besonders wichtige Punkte:

a) Aussagekräftige Bezeichner

b) Sichtbarkeit von Feldern

c) Auskommentierter Code

Ad a) Die Wahl von geeigneten aussagekräftigen Bezeichnern ist zwar oft eine aufwendige Tätigkeit, sorgt aber dafür, dass man bei zukünftigen Arbeiten sofort weiß um was es sich bei der Variable handelt. Ist hingegen eine Variable, Klasse oder Methode mit xy bezeichnet, herrscht bei späterem Betrachten des Codes Unklarheit über deren Bedeutung.

[15] Vgl. [KÜBECK1, S.235].

Ad b) Hierbei geht es um das regelmäßige Überprüfen ob eine Variable wirklich public sein muss. Dies stellt eine einfache Tätigkeit dar, die entscheidend zur Kapselung des Programms beiträgt.

Ad c) Auskommentierter Code hat meistens einen Grund und zwar dass er nicht mehr gebraucht wird. Somit kann er auch sofort gelöscht werden. Sollte er doch noch einmal gebraucht werden, ist er über die Quellcodeverwaltung wieder aufzufinden.

2.5 Forward Engineering

Forward Engineering bezeichnet in diesem Fall den klassischen Softwareentwicklungsprozess wie er für eine neue Software verwendet wird. Das Forward wird in diesem Fall bewusst verwendet, um den Unterschied zum Reverse Engineering hervorzuheben.

Angelehnt an das Wasserfallmodell stellen hierfür folgende Punkte typische Schritte für das Forward Engineering dar:

- Anforderungsanalyse
- Designentwurf
- Implementierung
- Testen
- Einsatz
- Wartung

Natürlich sind auch andere Vorgehensmodelle der Softwareentwicklung für diesen Schritt zulässig. Mögliche Beispiele wären das V-Modell, das Spiralmodell aber auch agile Methoden wie SCRUM oder Extreme Programming.

2.6 Testen

Softwaretests sollen sicherstellen, dass das Programm auch das tut, was gewünscht wird. Auch wenn in vielen Programmen nie zu 100% gesichert werden kann, dass ein Programm fehlerfrei ist, so kann man durch sorgfältiges Testen die Fehleranzahl minimieren. Edsger Wybe Dijkstra formulierte es einst sehr treffend:

„Program testing can be used to show the presence of bugs,
but never to show their absence!“[16]

Siedersleben definiert einige Punkte warum Softwaretests wichtig sind:

- „Testen misst die Qualität eines Systems.
- Testen zeigt, ob die Anforderungen erfüllt sind.
- Testen betrachtet die ganze Bandbreite der Anforderungen und Qualitätsmerkmale: Funktionsumfang, Zuverlässigkeit, Benutzbarkeit, Effizienz.
- Testen hilft bei der Abschätzung des Risikos des Einsatzes in Produktion.
- Testen findet Fehler, bevor sie in der Produktion Schaden anrichten.
- Testen unterstützt kurze Integrations- und Releasezyklen durch schnelle automatisierte Regressionstests.“[17]

Der frühzeitige Einsatz etwa von Tests kann im Laufe des Projekts helfen eine Vielzahl an Ressourcen einzusparen. Auch wenn das Erstellen von zum Beispiel Unit Tests zeitintensiv ist, kostet die Fehlersuche in einer späteren Phase des Prozessmodelles vergleichsweise mehr Zeit und damit Geld. Dazu kommt, dass das Auffinden von Fehlern in kleinen Einheiten, wie es bei Unit Tests üblich ist, um vielfaches einfacher ist, als wenn man den kompletten Quellcode eines Programms vor sich hat, wie es bei Black Box Tests der Fall ist.

[16] [DIJKSTRA1].

Aber auch in der Wartungsphase oder bei Erweiterungen bzw. späteren Änderungen können vorhandene Testfälle eine große Hilfe sein. Wenn etwa ein Entwickler mit einem fremden Quellcode konfrontiert wird, können ihm Testfälle ungemein helfen, den fremden Code zu verstehen. Bei Änderungen, vor allem beim Refactoring, kann man mit Hilfe von Testfällen davon ausgehen, dass man auch wirklich nur das ändert, was man auch ändern will.

Die Durchführung von geeigneten Softwaretests dient also der Qualitätssicherung und nicht nur dem „Fehler finden“. Es soll gewährleisten, dass das Produkt die gewünschten Anforderungen erfüllt.

2.6.1 Black Box Tests

Für den Black Box Test ist kein genaues Wissen über die Programminterna notwendig. Diese Art des Testens ist besonders geeignet, wenn man keinen Zugriff auf den Quellcode eines Programms hat. Das Programm wird nur anhand von Eingabe- und Ausgabedaten getestet.[18]

Der Vorteil dieser Tests liegt darin, dass man ohne viel Wissen über das Programm testen kann und dies somit auch von externen Teams durchgeführt werden kann. Ein großer Nachteil hingegen liegt in der Tatsache, dass man nur sehr schwer alle Teile des Programms testen kann. Vor allem für jemanden, der den Quellcode nicht kennt, ist es sehr schwer, geeignete Testfälle zu finden und damit eine möglichst hohe Testabdeckung zu erreichen.

2.6.2 White Box Tests

Der White Box Test, auch Glas Box Test oder Strukturtest genannt, testet die innere Struktur des Programms auf Korrektheit und Vollständigkeit. Dabei wird überprüft ob auch alle Anweisungen bzw. Pfade des Programms ausgeführt werden. [19]

---

[17] [SIEDERSLEBEN1, S.289].

[18] Vgl. [FRANZ1, S.28].

[19] Vgl. [FRANZ1, S.28].

Da für diese Art des Testens genaue Kenntnisse über den Quellcode und die programminternen Abläufe notwendig sind, sollten diese Tests von den Entwicklern selbst geschrieben werden.

2.6.3 Unit Tests

Der Unit Test oder Komponententest stellt eine Kombination aus White Box und Black Box Tests dar. Deswegen wird er auch vielfach als Grey Box Test bezeichnet. Wie auch White Box Tests werden Unit Tests in der Regel von den Entwicklern selbst geschrieben, da man zum Generieren der Tests Kenntnisse über die internen Abläufe des Programms haben sollte. Mit Black Box Tests haben Unit Tests gemeinsam, dass man keinen Einblick in die Interna der Einheit, welche getestet wird, hat bzw. braucht. Das liegt daran, dass Unit Tests vor der eigentlich zu testenden Klasse oder Methode geschrieben werden. Siehe dazu auch 2.6.4 „Warum Tests vorher schreiben?“

Besonders in der testgetriebenen Entwicklung stellen Unit Tests einen nicht wegzudenkenden Bestandteil dar.

2.6.4 Warum Tests vorher schreiben?

Doch wieso sollten Tests vorher geschrieben werden? Der Vorteil liegt im Mehr an Design und Analyse, denn man macht sich schon vor der eigentlichen Implementation einer Methode darüber Gedanken welche Parameter übergeben, welche Werte zurückgegeben, welche Fehlermeldungen ausgegeben werden oder wie die Methode überhaupt benannt wird.

Ein weiteres gewichtiges Argument stellt eine ganz einfache Tatsache dar: Wenn man sie vorher nicht schreibt, schreibt man sie gar nicht mehr. Viel zu oft passiert es, dass man im Nachhinein keine Zeit mehr hat bzw. einfach darauf vergisst, weil so viel anderes zu tun ist. So werden dann Tests auch oft als lästige Tätigkeit empfunden, vor allem weil das Programm ja sowieso schon läuft und man sich der nächsten Problemstellung widmen will.

Wenn Tests zuvor geschrieben werden, ist immer garantiert, dass es keinen ungetesteten Code gibt.

Auch beim Schreiben der Tests ist Vorsicht geboten. Es ist ganz wichtig folgende Reihenfolge zu beachten:

1) Test schreiben
2) Test ausführen und Test MUSS fehlschlagen
3) Zu testenden Code schreiben
4) Test ausführen und „hoffen“

Besonders Punkt 2 ist essentiell. Denn was hilft ein Test, der etwas testet, das auch ohne den zu implementierenden Code durchgeht. Das würde das Vorhaben, etwas zu programmieren schon im Vorhinein überflüssig machen.

2.6.5 Test-Frameworks

Um Modultests oder Unit Tests durchzuführen, gibt es verschiedene Frameworks, die automatisierte Softwaretests zulassen.

Einer der bekanntesten Test-Frameworks ist wohl JUnit für Java. Mittlerweile gibt es für die meisten gängigen Programmiersprachen Portierungen von JUnit, sodass eine eigene xUnit-Reihe entstanden ist. Der .Net-Ableger nennt sich nUnit.

Für .Net Sprachen bietet Microsoft über sein Visual Studio selbst zwei Frameworks an, MSTest und PEX.

3 Case Study

In diesem Kapitel geht es um die praktische Umsetzung eines Reengineering Prozesses. Im Auftrag von JOANNEUM RESEARCH wurde das Simulationsprogramm STOTRASIM von einer FORTRAN 77 in eine C# Entwicklung überführt.

3.1 Einleitung

In regelmäßigen Abständen ist in den Medien von Grundwasserverschmutzungen aufgrund erhöhter Nitratkonzentrationen infolge von Ackerbau zu lesen. Um einschätzen zu können wie hoch die Belastungen auf das Grundwasser, durch landwirtschaftlich genutzte Flächen sind, werden Rechenmodelle herangezogen. Ein für solche Aufgaben bestens geeignetes Werkzeug ist STOTRASIM[20]. STOTRASIM ist ein Simulationsprogramm, welches den Wasser- und Stickstoffkreislauf in pflanzenbaulich genutzten Böden berechnet.

Die Anfänge des Programms liegen in den achtziger Jahren des letzten Jahrhunderts als Dr. Elmar Sternitzer, am Institut für Kulturtechnik und Bodenwasserhaushalt des Bundesamtes für Wasserwirtschaft in Petzenkirchen, mit der Entwicklung des Moduls SIMWASER[21] begann. DI Franz Feichtinger setzte seine Arbeit fort und integrierte das Modul in seine Simulation STOTRASIM.

[20] Vgl. [FEICHTINGER1].

[21] Vgl. [STENITZER1].

In den letzten Jahrzehnten wurde das Programm stetig weiterentwickelt, sodass es zu einem umfangreichen Simulationsprogramm für Wasser- und Stofftransporte in Böden wurde.

Seit einigen Jahren wird mit dem Programm in Zusammenarbeit mit JOANNEUM RESEARCH und dem Institut für Kultur und Bodenwasserhaushalt am Bundesamt für Wasserwirtschaft an der Umsetzung eines gemeinsamen Bodernwasserhaushalts- und Stickstofftransportmodells gearbeitet.

Schon seit geraumer Zeit gibt es Bestrebungen, die Simulation in einer modernen, zeitgemäßen Programmiersprache neu umzusetzen. Da sich Dr. Stenitzer schon vor längerem in den Ruhestand zurückzog und DI Feichtinger ihm in Kürze ebenfalls folgen wird, war es nun an der Zeit dieses Vorhaben endgültig umzusetzen. Insbesondere da nach dem Ausscheiden von DI Feichtinger der Mastermind hinter der Simulation nicht mehr greifbar ist und somit auch für etwaige Fragen nicht mehr zur Verfügung steht. Dadurch ergibt sich ein guter Zeitpunkt für eine Neuprogrammierung der FORTRAN 77-Simulation STOTRASIM als moderne C# Applikation.

Die objektorientierte Neuentwicklung soll gewährleisten, dass zukünftige Weiterentwicklungen ohne großen Aufwand realisiert werden können. Das bedeutet zum Beispiel, dass man einzelne Klassen einfach austauschen kann. Sei es einfach nur, um die Ein- und Ausgaben an unterschiedliche Anwendungen anzupassen.

Einen gewichtigen Punkt für die Neuentwicklung stellt auch die Entwicklungsumgebung dar. Die FORTRAN 77 Version ist nur mehr mit Hilfe des Watcom 10.6 Compilers kompilierbar. Dieser läuft in Windows 7 etwa nur mehr im Windows XP-Kompatibilitätsmodus. Unter dem Nachfolger Watcom 11.0 bzw. unter den folgenden Open Watcom Produkten ist der Code nicht mehr kompilierbar, da es mit der 11.0 Version drastische Änderungen gab und eine Abwärtskompatibilität nicht gegeben ist. Mit der Neuentwicklung in C# bzw. mit Microsofts Visual Studio Reihe kann man davon ausgehen in den nächsten Jahren eine zuverlässige Entwicklungsumgebung zur Verfügung zu haben, bei der man auch bei den nächsten Produktgenerationen von einer Abwärtskompatibilität ausgehen kann.

3.2 Zielsetzung

In dem Projekt „STOTRASIM neu“ geht es um die Neuentwicklung der Simulation STOTRASIM. Dabei soll eine Software zur Berechnung des Wasser- und Stickstofftransportes in der ungesättigten Zone geplant und neu umgesetzt werden.

Zur Verfügung stand eine FORTRAN 77 Entwicklung, deren Funktionalität durch die neue Software abgedeckt werden sollte. Der Schwerpunkt liegt auf der Umsetzung der vorhandenen Funktionalität. Zusätzlich soll aber auch ein Augenmerk auf zukünftige Weiterentwicklungen gelegt werden. Hier gibt es für die Zukunft von Seiten der JOANNEUM RESEARCH vor allem den Wunsch die Simulation nach jedem Rechentag zu unterbrechen und verschiedene Parameter mittels eines externen Simulationsprogramms zu aktualisieren. Weitere Wünsche stellen die Anbindung an das Web bzw. die Entwicklung eines grafischen Userinterfaces dar.

Bei der Umsetzung ist dem Auftraggeber eine objektorientierte Struktur mit klarer Unterteilung und Strukturierung in thematische Module wichtig.

Es ist besonders darauf zu achten, dass die Schnittstellen zur Peripherie (Input bzw. Output) aufrechterhalten bleiben, damit weiterhin ein korrektes Zusammenspiel mit Pre- und Postprocessing gewährleistet bleibt.

Es wird auch großer Wert auf geeignete Klassen-, Methoden-, Konstanten- und Variablennamen gelegt, um das Lesen des neu entstandenen Codes ohne Dokumentation möglich zu machen. Somit ist das Entziffern der genannten Namen aus der FORTRAN 77 Version, in welcher die Namenlänge noch auf sechs Zeichen beschränkt war, ein wichtiger Bestandteil der Arbeit.[22]

Die Testung der Neuentwicklung erfolgt über sechs, von JOANNEUM RESEARCH und dem Institut für Kultur und Bodenwasserhaushalt am Bundesamt für Wasserwirtschaft ausgewählte, Testfälle (je drei). Die Testfälle decken dabei ein breites Spektrum der

[22] FORTRAN 77, Verfügbar unter: http://www.computing.me.uk/tutorials/FTN77%20Variables%20and%20Identifiers.pdf Seite 2, [Datum des Zugriffs: 30.04.2014].

Funktionalität von STOTRASIM ab. Abweichungen müssen nachweislich eine Verbesserung des Ergebnisses darstellen.

3.3 Entwicklungsumgebungen

STOTRASIM wurde in FORTRAN 77 mit Hilfe der Entwicklungsumgebung Watcom entwickelt. Bei der Neuentwicklung kam Microsofts Visual Studio 2010 zum Einsatz. Die Wahl der Programmiersprache fiel auf C#.

3.3.1 FORTRAN 77

FORTRAN steht für FORmula TRANslation. Wie der Name schon sagt ist FORTRAN für wissenschaftliche bzw. numerische Berechnungen bestens geeignet.

FORTRAN 77 hat einige Eigenheiten, die in heute gängigen Programmiersprachen nicht mehr anzutreffen sind. So wird noch ein festes Spaltenformat verwendet, welches ein Erbe aus der Lochkartenzeit darstellt. Dies bedeutet, dass die ersten sechs Zeichen einer jeden Zeile für spezielle Anweisungen reserviert sind. In Abb. 3.1 kann man die Aufteilung einer Zeile gut erkennen. Steht an erster Stelle einer Zeile ein „c“ so ist die ganze Zeile als Kommentar markiert. Ab dem siebentem Zeichen kann man einen Kommentar auch mit einem Rufzeichen einleiten. Die ersten fünf Zeichen sind für Anweisungsnummern (engl. statement label) reserviert. Steht an sechster Stelle ein Zeichen, bedeutet das, dass die vorherige Zeile fortgesetzt wird.

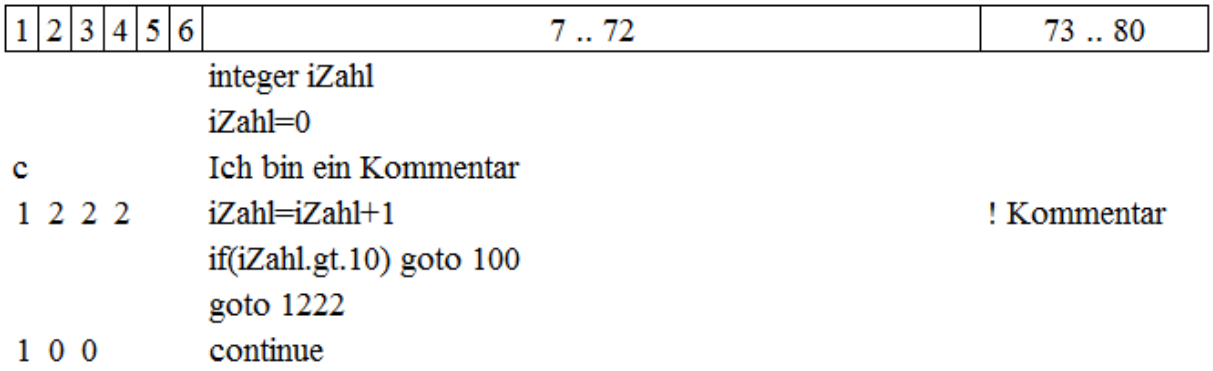

Abb. 3.1: FORTRAN 77 Spaltenformat

Wie Abb. 3.1 zeigt, ist es in FORTRAN 77 möglich das GOTO Statement, mit dessen Hilfe man zu definierten Anweisungsnummern springen kann, zu verwenden. Teilweise macht dieses Statement den Code sehr schwer nachvollziehbar, da von einer Stelle an in eine beliebig andere gesprungen werden kann. Trotz der Tatsache, dass es eigentlich möglich ist diese Statements mittels Schleifen zu ersetzen, wurden sie in der FORTRAN 77 Version noch sehr häufig eingesetzt.

Zu beachten ist auch, dass man in FORTRAN 77 noch mit sechs Zeichen auskommen musste. Daraus ergeben sich leider bei Benennern häufig sehr interessante Abkürzungen.

Des Weiteren ist FORTRAN 77 eine „case insensitive“ Programmiersprache, was bedeutet dass zwischen Groß- und Kleinschreibung kein Unterschied gemacht wird. Groß- und Kleinschreibung dient in diesem Fall einfach nur der Lesbarkeit.

3.3.2 Watcom 10.6

Als Compiler für den FORTRAN 77 Quellcode dient Watcom 10.6 aus dem Jahre 1995 der Universität Waterloo in Kanada. Heute bekannter unter dem Open Source Projekt „Open Watcom“.[23]

Von Seiten früherer Entwickler wurde mitgeteilt, dass der Compiler etwa unter Windows 7 nicht mehr zum Laufen gebracht wurde. Da ein paralleles Betreiben mehrerer Betriebssysteme nicht infrage kam, musste eine Lösung gefunden werden. Über den Windows XP Kompatibilitätsmodus konnte der Watcom Compiler schließlich zum Laufen gebracht werden.

Auch beim Einrichten des Projekts in Watcom sind einige Tricks nötig. So dürfen nicht alle FORTRAN-Dateien in das Watcom Projekt importiert werden. Hierbei handelt es sich um die Dateien bbsstruk.for, dsprostr.for und stotrinc.for. Zudem muss beim ersten Mal kompilieren für die Datei stobeginfrost.for ein einzelnes make gemacht werden, da der Compiler sonst fehlschlägt.

Hat man alle Hürden gemeistert, liefert Watcom ein Windows 32 Executable.

[23] Open Watcom, Verfügbar unter: http://www.openwatcom.org, [Datum des Zugriffs: 21.05.2014].

3.3.3 C#

Als gewünschte Programmiersprache der Neuprogrammierung wurde, von Seiten des Auftraggebers JOANNEUM RESEARCH, C# gewählt. Wichtig ist dabei der Übergang von einer prozeduralen Programmiersprache, wie FORTRAN 77 eine ist, zu einer objektorientierten.

C# bietet dem Entwickler viele Annehmlichkeiten. Dazu gehören etwa zahlreiche vorgefertigte Klassen und Bibliotheken, einen Garbage Collector und eine große Community, die bei Fragen und Problemen schnell zur Hilfe bereit steht.

Zusätzlich kann man dank Microsoft davon ausgehen, dass die Sprache auf lange Sicht zur Verfügung steht und weiterentwickelt wird. Somit stellt C# eine zukunftssichere Lösung dar.

Natürlich begibt man sich bei der Wahl von C# auch in eine gewisse Abhängigkeit von Microsoft. Die Plattformabhängigkeit stellt für JOANNEUM RESEARCH jedoch kein Problem dar, da die Software ohnehin auf Windowsrechnern zum Einsatz kommt.

3.3.4 Microsoft Visual Studio C# 2010

Bei der Entwicklungsumgebung wurde auf das von Microsoft zur Verfügung gestellte Visual Studio gesetzt. Zum einen bietet es dem Entwickler viele praktische Funktionen, die ihm das Entwickeln erleichtern, auf der anderen Seite wird es auch vom Auftraggeber JOANNEUM RESEARCH verwendet. Und mit der Express Version ist es mit eingeschränktem Funktionsumfang für jedermann frei erhältlich.[24] Eben diese kam auch bei der Neuentwicklung von STOTRASIM zum Einsatz.

[24] Microsoft Visual Studio C#, Verfügbar unter: http://www.visualstudio.com/downloads/download-visual-studio-vs, [Datum des Zugriffs: 21.05.2014].

3.4 Umfang

In Tab. 3.1 und Tab. 3.2 sind einige Kennzahlen der FORTRAN 77 bzw. C# Version nachzulesen, die den Umfang des Projekts kurz beschreiben sollen.

Zeilen	12864
Dateien	47
Subroutinen	43
GOTOs	531

Tab. 3.1 Statistik FORTRAN 77

Zeilen	13591
Dateien	38
Klassen	84

Tab. 3.2 Statistik C#

Des Weiteren ist in Abb. 3.2 eine Verteilung der Zeilen auf die einzelnen Quelldateien zu finden. Man sieht dabei sehr gut, dass fast ein Viertel der Zeilen in der Datei *stotrsub.for* anfallen. Die Datei stellt das Herzstück der Simulation dar, in welcher sich die so genannte Fruchtfolgeschleife sowie die Tagesschrittschleife, von welcher die meisten Subroutinen aufgerufen werden, befinden.

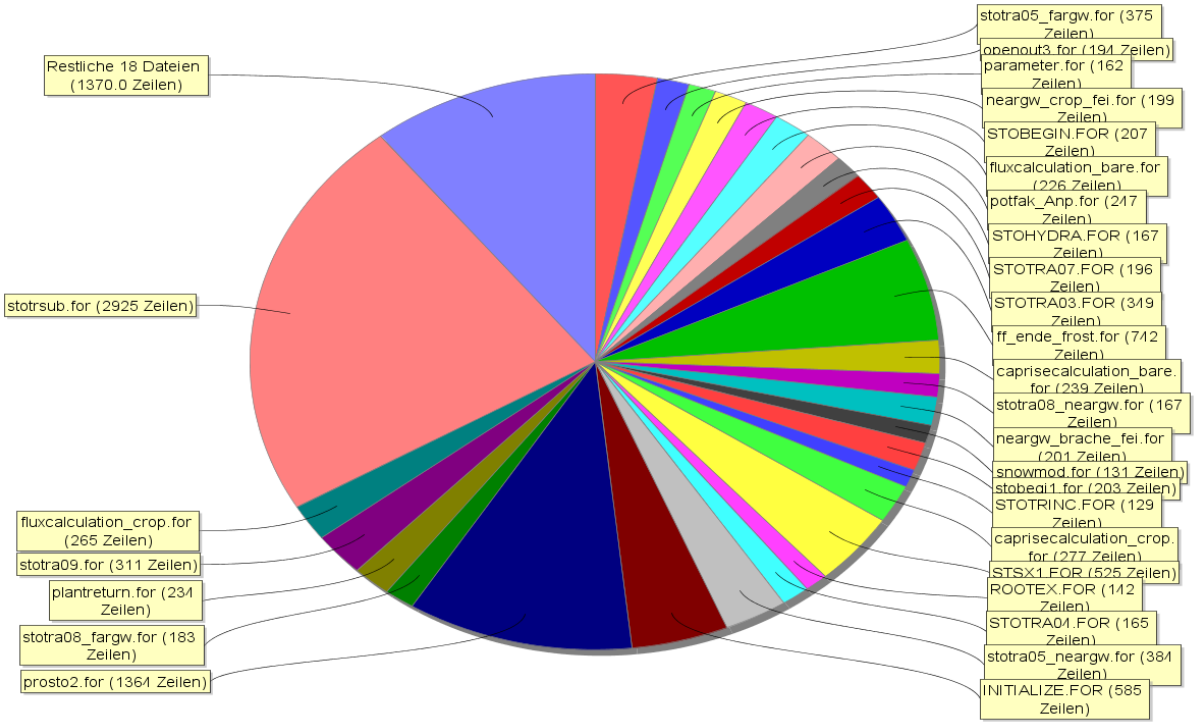

Abb. 3.2: Verteilung der Zeilen auf Dateien

3.5 Vorgehensweise

Zu Beginn der Arbeit standen zwei Punkte im Vordergrund. Das war auf der einen Seite das Bekanntmachen mit dem Programmiercode selbst und auf der anderen Seite das zeitgleiche Kennenlernen einer neuen Programmiersprache. FORTRAN war zwar vom Namen her bekannt, aber gearbeitet hatte ich damit noch nie.

Nachdem ein erster Überblick über die Programmabläufe geschaffen war, wurde ein grobes Design erstellt. Vor allem auch um mit dem Auftraggeber abzuklären auf was dieser bei der Umsetzung besonders Wert legt.

Bei der Implementierung wurden die Methoden des Extreme Programming als Vorbild genommen. Schritt für Schritt wurden kleine User Storys gewählt und umgesetzt. Dabei wurde die Aufmerksamkeit darauf gelegt, dass die einzelnen User Storys zu einer möglichst einfachen abgespeckten Version von STOTRASIM führen. Zuerst wurde also eine STOTRASIM Version geschaffen, die den einfachsten Funktionsumfang abdeckte.

Der neu eingebaute Code wurde sorgfältig mit dem entsprechenden Pendant des FORTRAN Codes getestet. Regelmäßige Refactoring Schritte, um die ursprüngliche prozedurale Vorgehensweise in ein objektorientiertes Schema zu überführen, standen auf der Tagesordnung.

Wiederholte Treffen mit Vertretern von JOANNEUM RESEARCH dienten dazu die geleistete Arbeiten bzw. auftretende Problem zu besprechen. Schlussendlich folgten Treffen um die auftretenden Abweichungen zu besprechen und ihnen auf den Grund zu gehen. Bis schließlich ein für alle Seiten zufriedenstellendes Programm das Ergebnis war.

3.6 Legacy System

In diesem Kapitel werden für ein Legacy System typische Punkte, wie sie in Kapitel 2.1 „Legacy System“ beschrieben sind und so wie sie in STOTRASIM vorkommen, erläutert.

3.6.1 Verunreinigung

Im Laufe der Jahre wurde in STOTRASIM immer wieder eine neue Funktionalität implementiert. Dadurch kam es vor, dass alte Codepassagen überflüssig, aber oftmals leider vergessen wurden.

An der einen oder anderen Stelle wurden eben solche Passagen einfach mit GOTO Statements übersprungen. In anderen Beispielen wurden if-Abfragen auf Variablen überprüft, die schon gar nicht mehr existierten.

3.6.2 Eingebettetes Wissen

Das Problem des eingebetteten Wissens trifft sehr gut auf STOTRASIM zu. Es sind zwar einige Dokumente zur Dokumentation vorhanden, diese sind aber durchgehend älter als 15 Jahre. Da aber auch in den letzten Jahren Änderungen durchgeführt wurden, sind sie keinesfalls vollständig. Teilweise befinden sich in den Dokumenten Beschreibungen von Subroutinen, die in der letzten FORTRAN Version gar nicht mehr existieren.

3.6.3 Mangelhafte Benennungen

Wie schon im Kapitel 3.2 „Zielsetzung“ erwähnt, stellte die Findung geeigneter Namen für Klassen, Methoden, Konstanten und Variablen einen sehr wichtigen Teil der Arbeit dar. Bei der Entwicklung der FORTRAN 77 Version von STOTRASIM waren die Entwickler mit der Tatsache konfrontiert, dass bei der Wahl von Namen nur Wörter mit maximal sechs Zeichen erlaubt waren. Des Weiteren wurde bei FORTRAN 77 auch kein Unterschied zwischen Groß- und Kleinschreibung gemacht, was die Möglichkeiten der Entwickler ebenfalls einschränkten. Daraus ergaben sich oft abenteuerliche Abkürzungen, die für einen anderen Entwickler oft nur mehr schwer bis gar nicht mehr nachvollziehbar waren. Bei der Identifikation so mancher Variable waren zeitintensive Recherchen notwendig, um für die Neuentwicklung einen geeigneten Namen zu finden. In Tab. 3.3 ist der Unterschied bei der Benennung von Variablen in den verschiedenen Versionen schön zu sehen. Von Namen aus einem Buchstaben, die alles Mögliche bedeuten können, wurden eindeutige Variablennamen, die sofort auf die Bedeutung der Variable schließen lassen.

real*8 h	double schichthoehe;
real*8 z	double geometrie;
real*8 p	double saugspannung;
real*8 ck	double leitfaehigkeit;
real*8 v0	double filtergeschwindigkeit;
real*8 ambda	double dispersivitaet;
real*8 Strssf	double stressfaktorWassermangel;
real*8 SSTRSD	double summeStresstage;

Tab. 3.3 Variablennamen in FORTRAN und C#

Wie in Abb. 3.3 dargestellt, waren auch die Benennungen der FORTRAN Dateien nicht immer ideal. Vor allem bei Namen wie STOTRA02 etc. ist es unmöglich vom Namen auf den tatsächlichen Inhalt der Datei zu schließen. Im Vergleich dazu wurde versucht bei der Wahl der Klassenbezeichnungen, wie in Abb. 3.4 zu sehen, eindeutige Namen zu wählen.

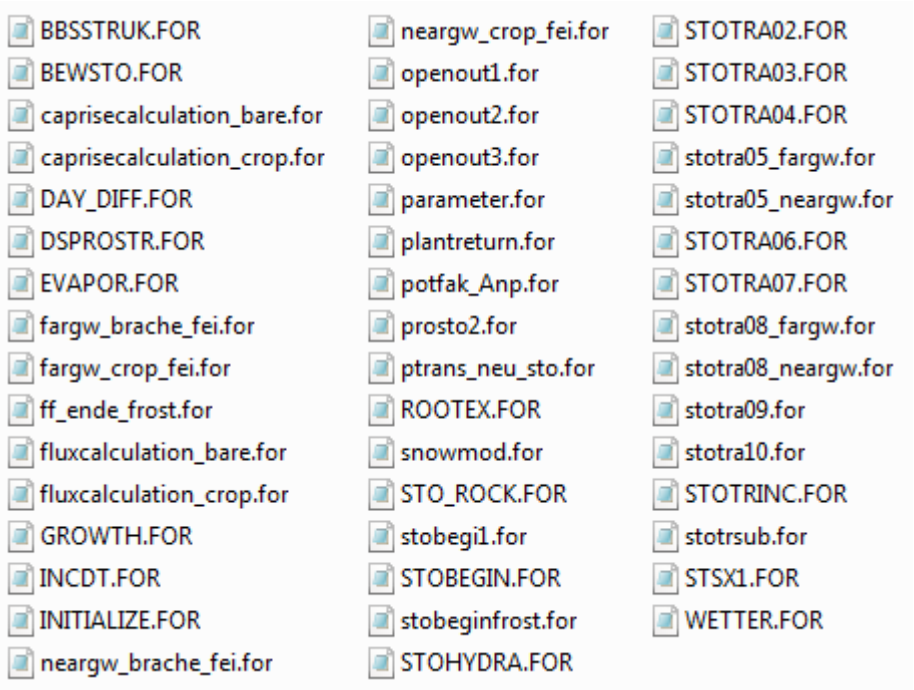

Abb. 3.3 Benennung der Dateien bzw. Subroutinen

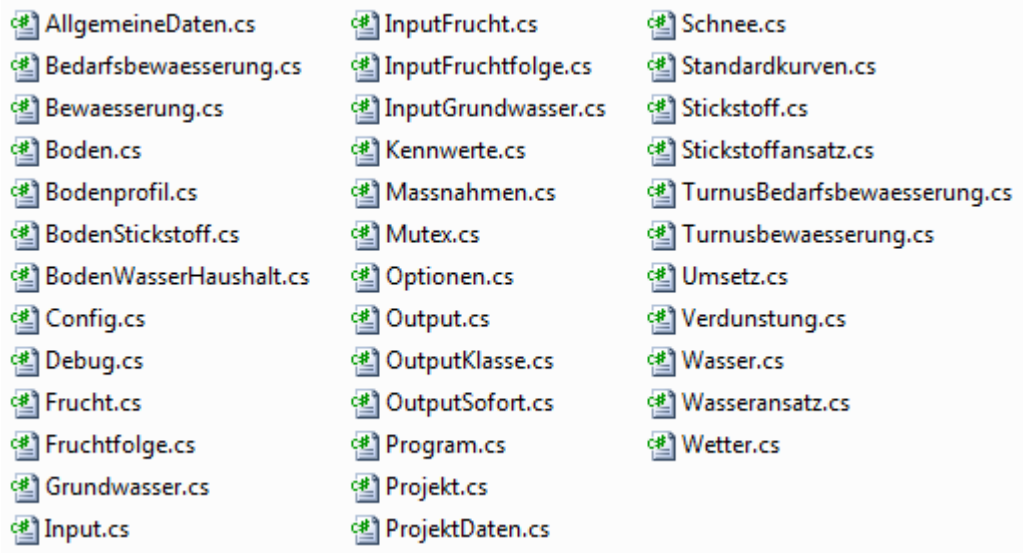

Abb. 3.4 Klassennamen C#

3.7 STOTRASIM – Programmablauf

Im Großen und Ganzen finden in der C# Version alle Abläufe in der gleichen Reihenfolge statt wie in der FORTRAN Version. Eine Änderung ist kaum möglich, weil die Daten zu sehr von den davor berechneten abhängig sind. Einzig die Datenausgabe hat sich insofern verändert, dass Daten nicht mehr an Ort und Stelle wo sie anfallen in Dateien geschrieben werden. Sie werden gesammelt und am Ende geschrieben. Dabei handelte es sich um einen Wunsch des Auftraggebers, da dieser plant das Postprocessing der Daten in Zukunft anders zu gestalten.

Die Simulation beginnt mit dem Einlesen der Projektdatei, welche alle möglichen Konfigurationsparameter zur Programm- und Ausgabensteuerung sowie eine Abfolge der Rechenvarianten, auch Lauf, Runs oder Projekte genannt, enthält.

Nach dem Lesen der Projektdatei startet mit der Schleife über alle Rechenvarianten auch die erste große Schleife. In Abb. 3.5 ist ein grober Überblick des Programmablaufs dargestellt. Dort sieht man als erstes die erwähnte Schleife, aber auch die anderen drei den Ablauf entscheidend mitbestimmenden Schleifen.

Im Initialisierungsteil der ersten Schleife werden für jeden Lauf verschiedenste Inputdateien gelesen sowie die Outputdateien für die Ergebnisse vorbereitet. In Tab. 3.4 sind die wichtigsten Inputdateien aufgelistet. Inputdaten werden auf ihre Konsistenz überprüft und in weiterer Folge findet der Aufbau der Böden für den Wasser- und Stickstoffteil getrennt statt, da diese mit unterschiedlich großen Kompartimenten gerechnet werden. Zuletzt findet die Variableninitialisierung für die erste Frucht statt.

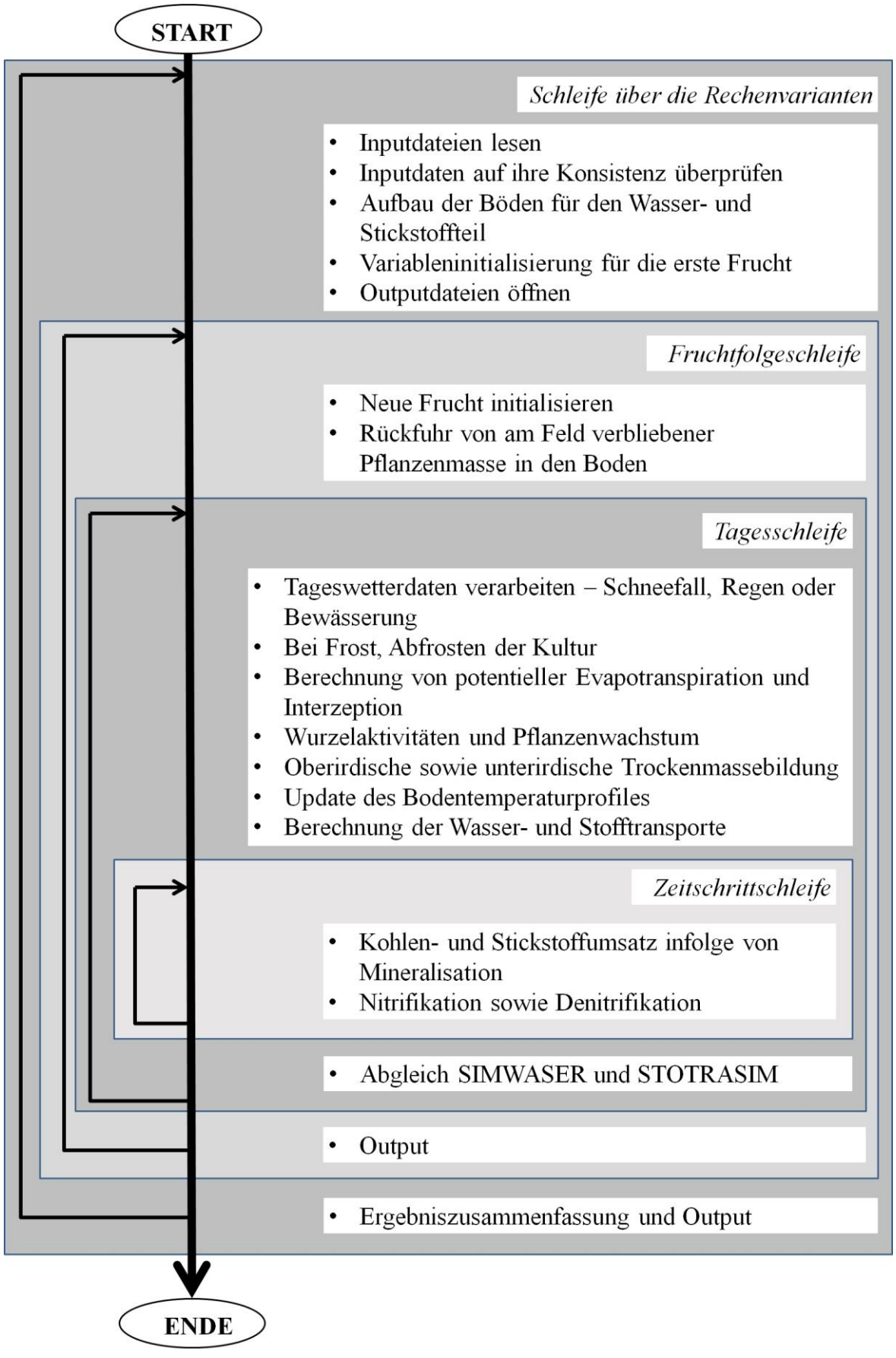

Abb. 3.5 Überblick über den Programmablauf

Projektdatei	Projekt_erw.prj
Bodentextur, Diffusionsparameter	BOD_DIFF.dat
Standortinformationen und Aufbau der Bodenschichten	BODPROF2.bin
Pflanzenkennwerte	kennwerte.dat
Parameter zum Sticktoffumsatz	umsetz.dat
Definitionsdatei für die Bedarfsberegnung	bedarfsberegnung_definition.dat
Fruchtfolgedatei. Genauer Dateiname in Projektdatei.	„fruchtfolge“.dat
Wetterdatei. Genauer Dateiname in Projektdatei.	„wetter“.wtr

Tab. 3.4 STOTRASIM Input Dateien

Auf die Initialisierung folgt die nächste große Schleife, die so genannte Fruchtfolgeschleife. Sie läuft über alle in der Fruchtfolgedatei eingetragenen Früchte. Zu Beginn wird jede Frucht neu initialisiert und eine Rückfuhr von möglicherweise am Feld verbliebener Pflanzenmasse in den Boden durchgeführt.

Die nächste Schleife ist die Tagesschleife. In ihr werden tagesaktuelle Wetter und bei Bedarf Grundwasserdaten eingelesen, sowie in der Folge auch gleich verarbeitet. Einzelne Punkte sind in Abb. 3.5 genauer zu sehen.

In die Tagesschrittschleife ist die Zeitschrittschleife eingebettet. In ihr finden vorwiegend Stofftransporte bzw. Flüsse zwischen den Bodenschichten statt.

Am Ende der Schleife über die Rechenvarianten sowie der Tagesschritt- und Fruchtfolgeschleife werden noch etwaige notwendige Ausgabedaten geschrieben.

Zuletzt erfolgen Berechnungen bzw. Zusammenfassungen von Daten, die sich über die komplette Simulation ziehen und die letzten Programmoutputs.

3.8 Die Rundungsfehlerproblematik

Schon in einer frühen Phase des Projekts kam die Frage auf, welcher Datentyp für Gleitkommazahlen verwendet werden sollte. In der FORTRAN-Entwicklung wurde für Gleitkommazahlen durchgehend der Typ REAL*4 verwendet. Bei diesem Datentyp wird nur mit einfacher Genauigkeit gerechnet. In FORTRAN 77 gibt es allerdings mit REAL*8 auch einen Datentyp, der doppelte Genauigkeit aufweist und welcher für mathematische Berechnungen vorzuziehen ist. Auf alten Systemen mag der höhere Speicherverbrauch und die längere Rechenzeit bei Rechnungen mit doppelter Genauigkeit noch ins Gewicht gefallen sein. In der heutigen Zeit sind diese Folgen zu Gunsten einer höheren Genauigkeit der Werte vernachlässsigbar.

Mit dem Auftraggeber JOANNEUM RESEARCH wurde schließlich vereinbart für alle Gleitkommazahlen in C# double einen Datentyp mit doppelter Genauigkeit zu verwenden. Höhere Kosten für Laufzeit oder Speicher sollten aufgrund der höheren Genauigkeit in Kauf genommen werden.

Wie sich später herausstellte wurde es, um die FORTRAN 77 und C# Version exakter vergleichen zu können, nötig in einer der beiden Entwicklungen die Datentypen an die andere anzupassen. Schließlich wurden in der FORTRAN 77 Simulation die REAL*4 Werte bis auf wenige kleine Ausnahmen auf REAL*8 abgeändert. Die Ausnahmen betreffen das Einlesen aus Binärdateien in Structures welche REAL*4 Typen verlangten. Wie Tab. 3.5 und Tab. 3.6 zeigen sind aber selbst die Wertebereiche von Datentypen mit gleicher Genauigkeit nicht ident.

	Ungefährer Bereich	**Genauigkeit**
Float	± 1.5e-45 zu ± 3.4e38	7 Stellen
double	± 5.0e-324 zu ± 1.7e308	15-16 Stellen

Tab. 3.5 C# einfache und doppelte Genauigkeit[25]

	Ungefährer Bereich	**Genauigkeit**
REAL*4	1.2 x 10e38 bis 3.4 x 10e38	7
REAL*8	2.2 × 10−308 bis 1.8 × 10−308	15

Tab. 3.6 FORTRAN einfache und doppelte Genauigkeit[26]

In Tab. 3.7 ist eine für das Projekt besonders problematische Quellcodepassage zu sehen. Der Ausschnitt stammt aus der FOTRAN 77 Version. Bei *r* handelt es sich um die relative Transpiration, bei *atrans* um die aktuelle Transpiration und bei *ptrans* um die potentielle Transpiration. Die aktuelle Transpiration wird durch Summenwerte gebildet, während die potentielle Transpiration mit Hilfe einer Formel ermittelt wird. Auch wenn die Eingangswerte für die Berechnung der aktuellen Transpiration ident sind, treten bei der C# und FORTRAN 77 Version minimale Unterschiede auf. Die Unterschiede sind hier meist bei der 15. Nachkommastelle zu finden. Diese minimalen Abweichungen wirken sich in der Folge auf die if-Abfrage auf und die Bedingung tritt bei beiden Programmversionen nicht mehr ident ein. Da es sich hier um einen minimalen Rundungsfehler handelt, wurde beschlossen in der if-Abfrage anstatt auf 1 auf 0.9999 zu überprüfen. Diese Änderung hatte zur Folge, dass die Anweisungen synchron bei beiden Versionen ausgeführt werden.

[25] Verfügbar unter: http://msdn.microsoft.com/de-AT/library/9ahet949%28v=vs.80%29.aspx, [Datum des Zugriffs: 30.04.2014].

[26] Verfügbar unter: http://www2.informatik.uni-halle.de/lehre/fortran/sprache/ftn623.html, [Datum des Zugriffs: 30.04.2014].

```
r=atrans/ptrans

if(r.lt.1.0) then

(..)

end if
```

Tab. 3.7 Ein problematisches Beispiel aus STOTRASIM

Aufgrund einiger kleiner Adaptionen konnten die Unterschiede zwischen den Ergebnissen der FORTRAN und der C# Version schließlich auf ein akzeptables Maß reduziert werden. Komplett ident wurden sie aufgrund der geringen Unterschiede zwischen den Datentypen beider Versionen aber nicht.

3.9 Sichtbarkeit von Feldern

„Muss das Feld oder die Methode XY public sein oder sollte es bzw. sie nicht doch besser private sein?“ [27]

Die Kapselung der Daten war bei der Entwicklung der FORTRAN 77 Version von keinerlei Bedeutung. Somit ist es kaum verwunderlich, dass bis auf wenige Ausnahmen die Variablen in der FORTRAN Version global definiert sind. Das bedeutet natürlich somit, dass jede Variable an jeder Stelle im Programm les- und schreibbar, also jederzeit manipulierbar ist.

Bei dem Reengineeringprozess von STOTRASIM galt es das „Prinzip der Datenkapselung“ im Auge zu behalten. Unerwünschte Zugriffe und Manipulationen, die zuvor durchaus möglich waren, sollen in Zukunft der Geschichte angehören. Durch die Kapselung der Daten wird die Konsistenz der Daten sichergestellt.

[27] [KÜBECK1, S. 235].

3.10 Design Patterns

Nach Möglichkeit kamen bei der Neuumsetzung auch Design Patterns zum Einsatz. Folgende Patterns sind in STOTRASIM umgesetzt:

- Singleton Pattern
- Strategie Pattern
- Fabrikmethode
- Schablonenmethode

In Abb. 3.6 ist etwa die Umsetzung eines Strategie Patterns in STOTRASIM als UML Diagramm zu sehen.

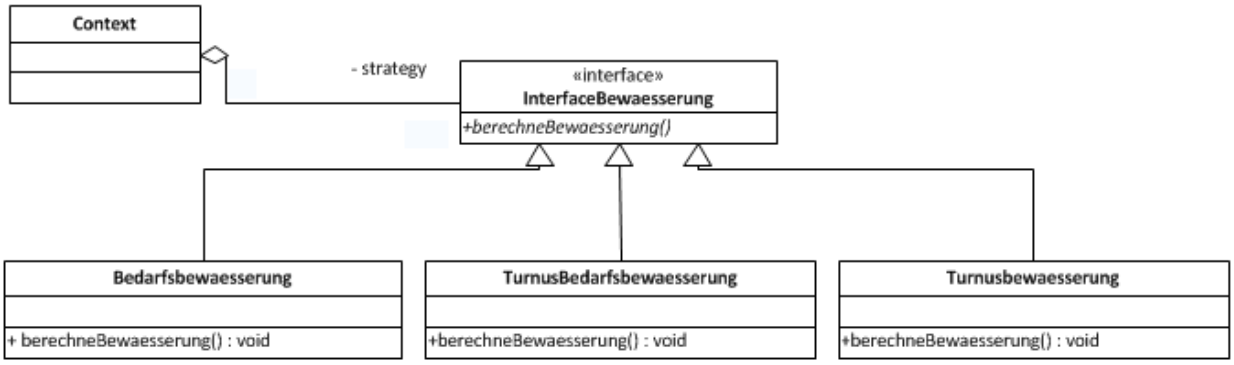

Abb. 3.6 Beispiel für Strategie Pattern in C#

3.11 Ein- und Ausgaben

In der FORTRAN 77 Version wird das Ein- bzw. Auslesen aller Daten so gehandhabt, dass sie an Ort und Stelle eingelesen bzw. geschrieben werden.

Vor allem beim Lesen von Daten führt dieser Umstand dazu, dass Daten, die bereits an anderer Stelle eingelesen wurden, wiederholt gelesen werden. Besonders bei der Berechnung

von mehreren Biotopen, die mit identen Daten aber abweichenden Einstellungen gerechnet werden, ergeben sich daraus unnötige Zugriffe auf den Speicher.

Wird beispielsweise ein Biotop über 30 Jahre gerechnet, ergeben sich alleine für die Wetterdaten daraus knapp 11.000 Zeilen mit je 20 Einträgen, die gelesen werden müssen. Wird das gleiche Biotop allerdings fünf Mal mit unterschiedlicher Konfiguration gerechnet sind wir bei 55.000 gelesenen Zeilen. Da die Wetterdaten allerdings zweimal pro Biotop gelesen werden, verdoppeln sich die Zugriffe. Somit sind wir bei 22.000 (für ein Biotop) bzw. 110.000 (für fünf Biotope) Leseoperationen. Daraus ergeben sich bei diesem Beispiel 88.000 read() Aufrufe, die eigentlich unnötig sind. Kommt ein eventueller Grundwasseranschluss bei den Berechnungen hinzu, kann man noch einmal 11.000 bzw. 55.000 gelesene Zeilen addieren.

Bei der neuentwickelten STOTRASIM Simulation werden alle Einleseoperationen über die Klasse Input abgewickelt. Sollten in Zukunft die Daten aus anderen Quellen bezogen werden, muss man entweder nur die ganze Klasse tauschen oder die entsprechenden Methoden der Klasse abändern. Denkbar wäre an dieser Stelle, dass die Daten nicht mehr aus (Text- bzw. Binär-) Dateien sondern aus Datenbanken gelesen werden. Außerdem wurde ein Augenmerk darauf gelegt, wiederholte Einleseoperationen zu vermeiden. Dafür ist eine eigene Klasse für die Bereitstellung bereits gelesener Datensätze zuständig.

Ausgaben werden in der FORTRAN 77 Version an Ort und Stelle geschrieben. Dort wo die Ausgabedaten im Code erzeugt werden, sind auch alle write()-Anweisungen zu finden. Werden idente Schreiboperationen an verschiedenen Stellen durchgeführt, so sind diese auch jedes Mal neu ausformuliert. Alleine bei kleinen Änderungen an einer Ausgabedatei müssen alle diese write()-Anweisungen durchgegangen und entsprechend der gewünschten Änderungen adaptiert werden.

In der Neuentwicklung ist alleine die Output-Klasse für das Generieren jeglicher Ausgaben zuständig. So ist es in Zukunft schneller möglich die Ausgaben auf andere Kanäle umzuleiten.

3.11.1 Konsolenausgaben

Über die Konsole wird der Benutzer über den Fortschritt der Simulation informiert. Um dabei dem Nutzer mehr Informationen zur Verfügung zu stellen, wurde die Ausgabe ein klein wenig überarbeitet.

Abb. 3.7 Ausgabe der FORTRAN 77 Version

Bei der FORTRAN 77 Version, zu sehen in Abb. 3.7, erhält man nicht sonderlich viel Informationen zum aktuellen Status. Lediglich welche Frucht aktuell berechnet wird. Da bei der C# Version mehrere Läufe auf einmal berechnet werden und somit mit dem alten Ausgabeformat die Früchte nicht mehr dem richtigen Lauf zuordbar wären, wird nun bei jeder Frucht die dazugehörige Laufnummer davor gestellt. Zudem erhält man zur Laufzeit ebenso Informationen darüber, welche Dateien gelesen werden, welcher Lauf gerade initialisiert wird und wenn Ausgabedaten in Dateien geschrieben werden. Zu sehen ist das Beispielhaft in Abb. 3.8 für einen gerechneten Lauf.

Abb. 3.8 Ausgabe der C# Version

3.11.2 Ordnerstruktur

Standardmäßig werden alle Ergebnisse der Simulation in den Ordner Erg geschrieben. Im Unterschied zur FORTRAN 77 Version kann man nun mit Hilfe der Befehlszeilenoption „–o“ die Ausgabe für die einzelnen Runs, welche in der Projektdatei angeführt sind, in eigene Ordner umleiten. In Abb. 3.9 ist auf der rechten Seite ein Beispiel für die Ausgabe mit getrennten Runs zu sehen. Dateien auf die alle Runs zugreifen, in diesem Beispiel MITODSUM.ERG und ffg_senf.dat, bleiben im Erg Ordner.

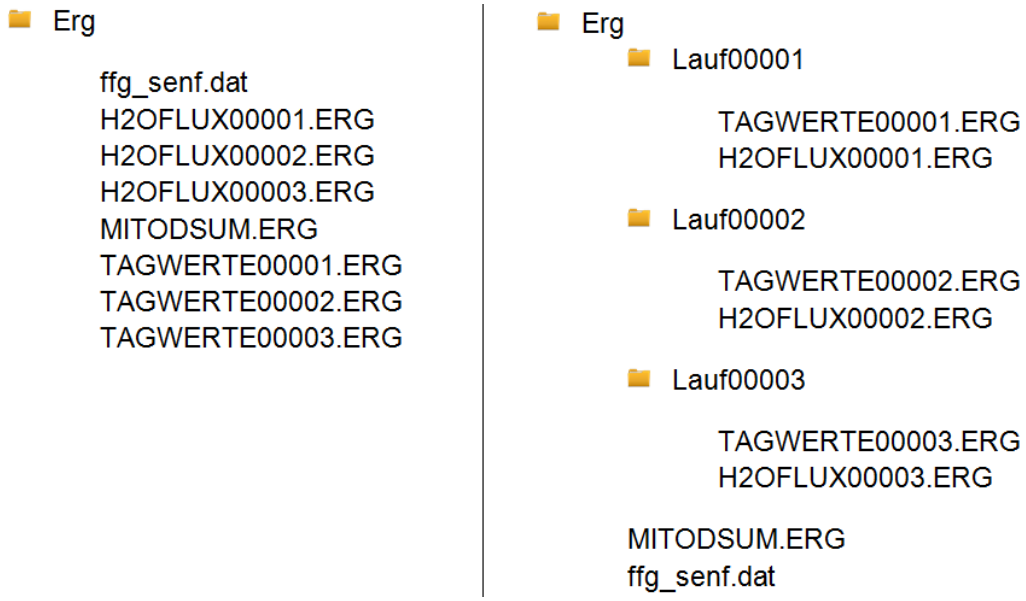

Abb. 3.9 Ausgabe im Standardformat (links) und mit getrennten Läufen(rechts)

3.11.3 Fehlermeldungen

In der FORTRAN 77 Version waren Fehlermeldungen nicht immer so leicht zu interpretieren. Vor allem bei den Eingabedaten wurde versucht mit zusätzlichen Meldungen eine schnelle Fehlerbehebung zu erleichtern. Aber auch zahlreiche andere Meldungen wurden bearbeitet bzw., wenn sie überflüssig waren, entfernt.

3.12 Programmbeschleunigung

Da mit Hilfe von STOTRASIM sehr rechenaufwendige Simulationen, Berechnungen von über zwölf Stunden sind keine Seltenheit, durchgeführt werden, war es natürlich bei der Neuentwicklung von besonderem Interesse, das Programm möglichst effizient und somit schnell zu gestalten. Hierfür wurden, wie in den nächsten Kapiteln beschrieben, verschiedenste Tätigkeiten durchgeführt, die, wie in Kapitel 3.12.4 „Geschwindigkeitsentwicklung“ zu sehen ist, durchaus fruchteten.

3.12.1 Beseitigung von unnötigem Code

Im Laufe der Zeit sammeln sich im Quellcode Abschnitte an, die bei der Programmausführung nicht von Bedeutung sind. In diesem Fall ist von totem oder unerreichbarem Code die Rede.

3.12.1.1 Toter Code

Ein toter Code stellt eine Codepassage dar, die zwar ausgeführt wird, deren Ergebnisse aber für die Anwendung irrelevant sind bzw. nie verwendet werden.

Solche Codepassagen sind in der FORTRAN-Version öfters vorhanden. Vor allem da im Laufe der Jahre immer wieder verschiedene Berechnungen ergänzt und entfernt wurden. Oft wurden dann Abschnitte, die nicht mehr in Verwendung waren, vergessen zu entfernen. Auf die korrekte Ausführung der Simulation hatten diese Abschnitte keine Auswirkung. Aber sie verbrauchten natürlich unnötige Rechenzeit.

```
if((jdayac-iprimda).gt.0.and.(jdayac-iprimda).le.110) then
iki=jdayac-iprimda
primfunk=1.8871e-06*iki**3-4.3583e-04*iki**2+2.5137e-02*iki
prim=1+primfunk*primhoch
end if

if((jdayac-iprimda1).gt.0.and.(jdayac-iprimda1).le.110) then
iki=jdayac-iprimda1
primfunk1=1.8871e-06*iki**3-4.3583e-04*iki**2+2.5137e-02*iki
prim=prim+primfunk1*primhoch1
end if

prim=1.0
```

Tab. 3.8 Beispiel für einen toten Code

In Tab. 3.8 sieht man wie in der FORTRAN 77 Version *prim* berechnet wird. Die Berechnungen in den zwei if-Zweigen sind aber völlig überflüssig, da am Ende prim sowieso gleich eins gesetzt wird.

3.12.1.2 Unerreichbarer Code

Bei einem unerreichbaren Code handelt es sich um einen Quellcode der aufgrund unterschiedlicher Faktoren nicht mehr ausgeführt wird. Vor allem bei Verwendung von GOTO Statements kann es schnell einmal passieren, dass ganze Codeblöcke übersprungen werden. Aber auch Schleifen, die an Bedingungen, die nie eintreten, geknüpft sind, kann ein Grund für einen unerreichbaren Code darstellen.

Bei der FORTRAN-Entwicklung waren es in erster Linie GOTO Statements, die für einen ebensolchen Quellcode gesorgt haben. Aber auch if-Abfragen, die auf Variablen, welche gar nie gesetzt wurden, überprüft wurden, sorgten für einen unerreichbaren Code.

```
    goto 324

    quotient=0.0
    bedarf_n=0.0
    angebot_n=0.0
    if(rootspeicher.eq.0.0)bedarf_n=(drymat+rootdrymat)*weniger15/100.
    if(rootspeicher.ne.0.0)bedarf_n=(drymat)*weniger15/100.
    if(depthc(1).gt.crootl*10.) goto 322
    if(c(1).gt.2.) angebot_n=(c(1)-2.)*smoic(1)*depthc(1)/10.
    Do j=2,nlayec
    if(depthc(j).gt.crootl*10.) goto 322
    if(c(j).gt.2.)  angebot_n=(c(j)-2.)*smoic(j)*(depthc(j)-depthc(j-1))/10.
    end do

322  continue

    (..)

324  continue
```

Tab. 3.9 Beispiel für einen unerreichbaren Code

In Tab. 3.9 ist ein typisches Beispiel mit einem GOTO zu sehen. Zu Beginn des Codeausschnittes steht ein GOTO Statement, welches ohne Bedingung zu der Anweisungsnummer 324 springt. Somit werden an dieser Stelle immer die folgenden Zeilen übersprungen. Zwischen der GOTO Anweisung und der Anweisungsnummer sind noch weitere GOTO Statements bzw. die Anweisungsnummer 322 zu finden, da diese aber nur innerhalb dieses Blockes verwendet werden, haben sie keine Auswirkungen auf den Programmablauf.

3.12.2 Zusammenfassen von Schleifen

Bei dem Originalprogramm kommt es immer wieder vor, dass mehrere (for-)Schleifen hintereinander über die gleichen Indizes geführt werden. Wenn die zweite bzw. nachfolgenden Schleifen nicht abhängig von den Ergebnissen der vorherigen sind, kann man diese

problemlos zusammenfassen. Wenn man bedenkt, dass an einem Simulationstag mehrere hunderte bis tausende solcher Schleifendurchläufe eingespart werden können, ergibt sich auch hier ein klares Einsparungspotential.

3.12.3 Multithreading

Einer der größten Geschwindigkeitsgewinne konnte dank Multithreading erzielt werden. Siehe dazu auch Kapitel Geschwindigkeitsentwicklung. Multithreading wurde so umgesetzt, dass jede Rechenvariante, welche in der Projektdatei angeführt ist, als eigener Thread erzeugt wird. Dies führt dazu, dass Multithreading erst zur Geltung kommt, wenn mehr als eine Rechenvariante gerechnet wird. Da für gewöhnlich die Simulation sowieso mit zahlreichen Rechenvarianten ausgeführt ist, stellt dieser Umstand aber kein Problem dar.

Damit sich die einzelnen Threads beim Zugriff auf gemeinsam genützte Ressourcen nicht in die Quere kommen, sind diese natürlich durch Mutexe geschützt.

3.12.4 Geschwindigkeitsentwicklung

Schon am Anfang des Projekts war klar, dass FORTRAN bei numerischen Berechnungen einen gewissen Geschwindigkeitsvorteil gegenüber C# hat. Das zeigen auch Versuche wie sie bei JOANNEUM RESEARCH durchgeführt wurden.

FORTRAN (Original)	C# (Singlethread)	C# (Multithread)
47,5 h	59,5 h	12,0 h
1,36 min/Lauf	1,71 min/Lauf	0,35 min/Lauf

Tab. 3.10 Geschwindigkeitsentwicklung der STOTRASIM Versionen

Wie Tab. 3.10 zu sehen ist, ist die originale STOTRASIM Version um etwa 25% schneller als die C# Version in der Mitte. Die C# Version wurde dabei mit einem einzelnen Thread ausgeführt. Bei der Ausführung mit mehreren Threads, standardmäßig sind zehn eingestellt,

ist dann klarerweise die C# Version im Vorteil. Die Rechendauer wurde dabei auf knapp ein Viertel der Zeit der FORTRAN Version reduziert.

Getestet wurden die STOTRASIM Versionen mit jeweils 2080 Läufen. Testgebiet war das westliche Leibnitzer Feld.

3.13 Adaptionen im FORTRAN 77-Code

Im Laufe der C#-Umsetzung stellte sich heraus, dass auch im Originalquellcode Änderungen notwendig sein werden. Einerseits mussten Fehler korrigiert werden, andererseits waren die Änderungen wichtig um die beiden Versionen überhaupt miteinander vergleichbar zu machen. Dies bezieht sich vor allem auf die Angleichung der Datentypen wie sie in Kapitel 3.8 erklärt werden.

Kleinere Fehler, wie etwa Schlampigkeitsfehler, werden hier natürlich nicht aufgelistet, wurden aber selbstverständlich auch in der FORTRAN 77 Version ausgebessert. Ein Beispiel für eine Adaption ist in dem Kapitel 3.8 bei der Rundungsfehlerproblematik bereits zu finden. Im folgenden Unterpunkt „Berechnung der Regendauer“ ist ein weiteres Beispiel zu finden, welches die ursprünglichen Ergebnisse doch auch beträchtlich verfälscht hat.

3.13.1 Berechnung der Regendauer

In der FORTRAN 77 Version sorgte die Berechnung der Regendauer in der Subroutine *WETTER_STO* für ein wenig Kopfzerbrechen. Die LVAL-Werte, welche für die Berechnung der Regendauer (durationrain), wie sie in Tab. 3.11 zu sehen sind, verwendet werden, sind als Integer*2 definiert. Der Wertebereich für Integer*2 liegt zwischen -32768 und 32767.[28] Die Daten werden zuvor als Record aus der binären Wetterdatei eingelesen und verlangen daher den Typ Integer*2. Bei der Berechnung der Regendauer werden die Werte teilweise potenziert und so lange die potenzierten LVAL-Werte nicht größer als 181 sind, tritt auch kein Problem

auf. Sind sie allerdings größer, sprengen sie den Wertebereich und die Ergebnisse ergeben keinen Sinn. Das Problem bei der Sache ist, dass die eingelesenen LVAL-Werte sehr selten größer als der erwähnte Wert sind und dadurch der Fehler vermutlich auch nie aufgefallen ist und wäre. Erst bei den Vergleichen der C# mit den FORTRAN 77 Ergebnissen sind sie schließlich zum Vorschein gekommen.

```
      durationrain=6.498367*LVAL(4)-0.023965*LVAL(4)**2-8.52780*LVAL(11)
     1                    +0.072303*LVAL(11)**2+1.632261*LVAL(19)-0.001268*
     2                  LVAL(19)**2-0.179148*LVAL(20)+0.000036*LVAL(20)**2+
     3                                                  59.79254*alog(LVAL(19)*1.)
```

Tab. 3.11 Fehlerhafte Berechnung der Regendauer

Das Beheben des Fehlers war um ein Vielfaches leichter als das Auffinden. Wie in Tab. 3.12 zu sehen ist, wurde der Fehler einfach mittels Hilfsvariablen behoben.

```
      integer*4 iLVAL20,iLVAL19,iLVAL4,iLVAL11
      iLVAL20 = LVAL(20)
      iLVAL19 = LVAL(19)
      iLVAL4 = LVAL(4)
      iLVAL11 = LVAL(11)

      durationrain=6.498367d0*iLVAL4-0.023965d0*iLVAL4**2-8.52780d0
     1    *iLVAL11+0.072303d0*iLVAL11**2+1.632261d0*iLVAL19-0.001268d0*
     2          iLVAL19**2-0.179148d0*iLVAL20+0.000036d0*iLVAL20**2+
     3          59.79254d0*alog(iLVAL19*1.)
```

Tab. 3.12 Korrigierte Berechnung der Regendauer

[28] Datentyp INTEGER, Verfügbar unter: http://www2.informatik.uni-halle.de/lehre/fortran/sprache/ftn622.html, [Datum des Zugriffs: 10.06.2014].

3.13.2 Vergleich der originalen FORTRAN Versionen

Die Änderungen im FORTRAN Quellcode haben natürlich auch auf die Ergebnisse der Simulation Auswirkungen. Hier soll nun anhand eines Beispiels die Auswirkungen der Änderungen aufgezeigt werden.

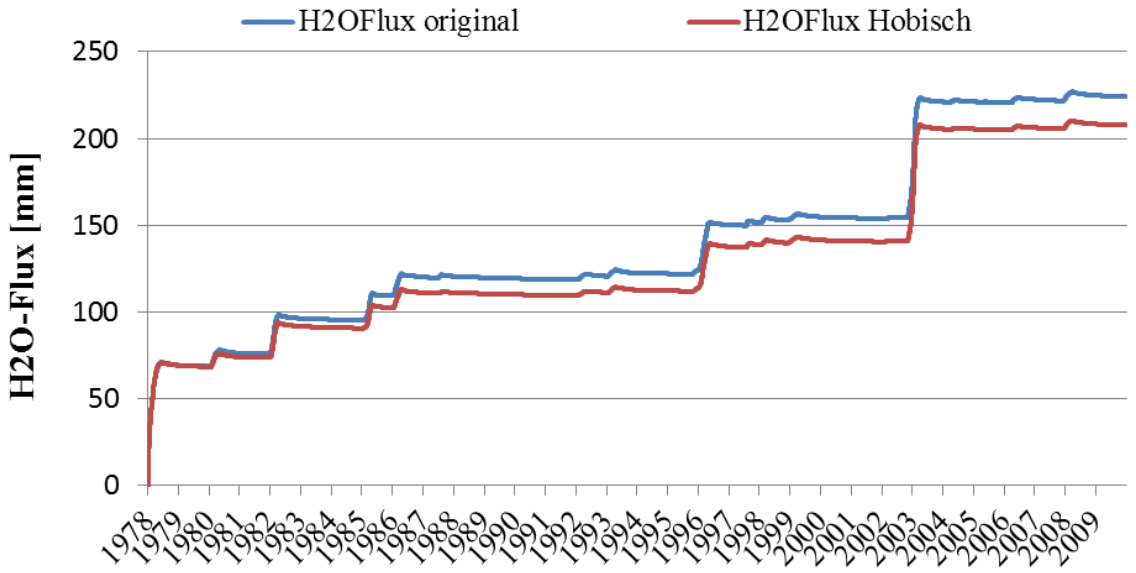

Abb. 3.10 Kumulierter Wasserfluss

In Abb. 3.10 sind die summierten Wasserflüsse in 130 cm Tiefe für ein Testgebiet in Niederösterreich dargestellt. Die blaue Linie stellt den originalen FORTRAN Code ohne Änderungen dar, die rote Linie wiederum ist die FORTRAN Version mit Adaptionen im Quellcode. Anhand der Grafik ist gut zu erkennen, dass sich im Laufe der Zeit die zwei Kurven auseinander bewegen. An Stellen in der Grafik wo größere Sprünge zu sehen sind, findet ein größerer Wasserfluss als gewöhnlich statt und auch zu diesem Zeitpunkt sieht man, dass sich Unterschiede zwischen den beiden Kurven ausbilden. Hier ist es naheliegend, dass sich die fehlerhafte Berechnung der Regendauer, wie in Kapitel 3.13.1 beschrieben, auf die Werte auswirkt. Denn gerade wenn die Werte erhöht sind, wie etwa bei größeren Regenmengen, führt dies zu Abweichungen zwischen den beiden Versionen.

3.14 Testen

Von Seiten des Auftraggebers waren grundsätzlich nur Black Box Tests, also funktionale Tests, gewünscht. Dafür wurden sechs verschiedene Testfälle erstellt, die ein möglichst breites Spektrum des Funktionsumfangs abdecken sollten. Diese sechs Tests sollten gewährleisten, dass die FORTRAN 77 und die C# Versionen dieselben Ergebnisse liefern.

Während der Entwicklungsphase kamen aber auch Unit Tests zum Einsatz.

3.14.1 Funktionale Tests

Um die Ergebnisse des Legacy Systems und der Neuentwicklung vergleichen zu können, wurde mit dem Auftraggeber vereinbart, dass dieser zu diesem Zweck fünf Testfälle erstellt. Schließlich wurden daraus sechs Testfälle, die, soweit von den Testerstellern einschätzbar, möglichst den kompletten Funktionsumfang der Simulation abdecken sollten.

Das Institut für Kulturtechnik und Bodenwasserhaushalt stellte drei Böden aus Niederösterreich, Oberösterreich und dem Burgenland zur Verfügung. Von JOANNEUM RESEARCH gab es drei Böden aus ihrem Versuchsgebiet um Wagna bei Leibnitz. In der folgenden Aufzählung sind die sechs Testfälle mit ihren projektinternen Bezeichnungen angeführt. Zuerst sind die drei Beispiele des Institutes für Kulturtechnik und Bodenwasserhaushalt angeführt.

- BGLD
- NOE
- OOE
- JR_1_LysimeterWagnaKON
- JR_2_Leguminosen
- JR_3_rechenaufwaendigerBoden

Während der Entwicklung des Projekts sind mithilfe des vorhandenen Testmaterials auch zahlreiche weitere kreiert worden. Dies war nötig, da mit den vorhandenen Testfällen nicht immer alle Programmzweige komplett abgedeckt werden konnten. Hier sei als einfaches Beispiel der Aufruf von nicht implementierten Programmteilen, wie etwa bei der Bewässerung die Kombination aus Turnus und Bedarfsbewässerung, erwähnt. Dabei handelt es sich um eine Art der Bewässerung, die es schon in der FORTRAN 77 Version gibt, welche aber nie ausformuliert wurde.

Da bei den gelieferten Tests jedes Mal sowohl der Wasserhaushalt als auch der Stickstofftransport berechnet wurde, wurden auch Beispiele simuliert, welche nur den Wasserhaushalt berechnen. Diese war insofern nötig da bei dieser Art der Simulation nicht einfach nur das komplette Stickstoffmodul wegfällt, sondern das Stickstoffmodul durch ein einfacheres ersetzt wird. Denn auch wenn nicht der komplette Stickstofftransport berechnet wird, benötigt die Simulation einfache Stickstoffberechnungen.

Bei der Erstellung der funktionalen Tests, die vom Auftraggeber zur Verfügung gestellt wurden, war Multithreading noch kein Thema. Um eine korrekte Arbeitsweise und ein geordnetes Zusammenspiel der Threads zu gewährleisten, wurden auch dafür eigene Testfälle erzeugt, in denen verschiedene Szenarien durchgespielt wurden. Insbesondere war es wichtig sicherzugehen, dass gemeinsam genutzte Ressourcen auch ausreichend abgesichert sind.

3.14.1.1 Sonderfall Grundwasseranschluss

Im Testfall *JR_3_rechenaufwaendigerBoden* wird mit einer Bodenstruktur, die sehr viele einzelne Schichten beinhaltet, gerechnet und die, wie der Name schon sagt, dadurch besonders rechenintensiv ist. Ebenfalls wird bei diesem Test auch die Berechnung mittels Grundwasseranschluss durchgeführt. Das heißt, dass Grundwasserdaten aus einer externen Datei gelesen werden und in die Simulation einfließen. Bei der Durchsicht bzw. dem Vergleichen der Ergebnisse der C# und der FORTRAN Version bin ich aber auf Ergebnisse gestoßen, bei der gewisse Werte extreme Schwankungen durchmachen. An einem Tag waren sie beispielsweise sehr hoch, am nächsten wiederum vergleichsweise nieder. Nach längerer Recherche stellte ich fest, dass das Problem an der Umsetzung der Zeitschrittschleife für den Grundwasseranschluss festzumachen ist. Das Team von JOANNEUM RESEARCH war von der Entdeckung dieses Phänomens nicht besonders überrascht. Bei der Verwendung der

Simulation mit Grundwasseranschluss waren ihnen selbst schon Werte ausgefallen, die ihrer Meinung nach nicht der Realität entsprechen können.

3.14.1.2 Fazit

Die funktionalen Tests liefern zwar den Beweis, dass die Neuentwicklung im Rahmen der getesteten Funktionalitäten die gleichen Ergebnisse abliefern wie die FORTRAN 77 Version. Allerdings sollte man bei einer so komplexen Entwicklung, wie sie diese FORTRAN 77 Simulation darstellt, nicht davonausgehen, dass wirklich alle Programmzeige mit den sechs gelieferten Testfällen komplett abgedeckt werden können.

Auch bei der Fehlersuche sind diese Arten von Tests keine besonders große Hilfe. Sie zeigen zwar auf, dass es zwischen den Version Abweichungen gibt, aber wo genau diese Unterschiede liegen, kann man mit Hilfe der funktionalen Tests nicht auf der ersten Blick feststellen. Um die Ursachen für diese Abweichungen zu finden, sind in der Regel zeitintensive und oft auch nervenaufreibende Recherchen notwendig. Als Entwickler bekommt man zwar mit der Zeit ein Gefühl dafür wo der Ursprung des Problems liegen könnte, aber als jemand der sich das erste Mal mit dem Projekt beschäftigt, ist es ungemein schwerer die Ursache für dieses zu finden.

3.14.2 Unit Tests

Unit Tests oder dergleichen waren nicht gewünscht, zumindest nicht vom Auftraggeber. Während der Umsetzungsphase kamen sie trotzdem zum Einsatz. Für den Einsatz von Unit Test gab es in erster Linie zwei gute Gründe:

- Einerseits konnte dadurch sichergestellt werden, dass die neu geschaffenen Methoden und Klassen in C# die gleichen Ergebnisse liefern wie die entsprechenden Codepassagen der Subroutinen der FORTRAN 77 Version. Der Vorteil an der bereits bestehenden FORTRAN 77 Version lag darin, dass durch sie einfach Testfälle kreiert werden konnten und dadurch keine neuen erfunden werden mussten.

- Andererseits konnte während den verschiedenen Refactoring-Arbeiten gewährleistet werden, dass sich keine Fehler einschleichen.

Vor allem in Hinsicht auf Refactoring sind Unit Tests zu empfehlen. Da dadurch sichergestellt werden kann, dass nach der Neustrukturierung des Quellcodes die Funktionalität und das Verhalten des Programms unangetastet bleibt.

3.14.3 Abweichungen

Schon während der Umsetzungsphase war schnell klar, dass die Ergebnisse des Legacy Systems und der Neuentwicklung keine zu 100% identen Ergebnisse liefern werden. Die Ursache ist wohl in den nicht identen Datentypen zu finden. Wie im Kapitel 3.8 erläutert, können sich die kleinen Differenzen der Datentypen über längere Zeit auch auf die Ergebnisse auswirken.

Beim Vergleichen der Ergebnisse beider Versionen wurden in erster Linie mit WinMerge[29] gearbeitet um festzustellen wo sich Unterschiede in den Ergebnisdateien befinden. Um über die Relevanz der aufgetretenen Unterschiede besser urteilen zu können, wurden die Abweichungen mit Hilfe von Excel (grafisch) aufbereitet.

In der Folge sind ein paar Beispiele aufgelistet, die auch für den Auftraggeber JOANNEUM RESEARCH bei der Übergabe von Bedeutung waren und daher genauer analysiert wurden.

In Abb. 3.11 sind die Abweichungen des Wasserflusses(Flux) in 120 cm Tiefe zu sehen. Neben der Tatsache, dass die Diskrepanzen in einer sehr geringen Höhe ausfallen, kann man feststellen, dass auf eine Abweichung in den positiven Bereich auch immer eine in den negativen folgt. Das bedeutet, dass der Wasserfluss an diesen Stellen um jeweils einen Tag versetzt ist. Dies stellt aber für die Ergebnisse der Simulation kein Problem dar. In Tab. 3.13 sind die genauen Zahlen für die in Abb. 3.11 mit Prozentangaben versehenen Tage angeführt.

[29] WinMerge, Verfügbar unter: http://winmerge.org/, [Datum des Zugriffs: 10.06.2014].

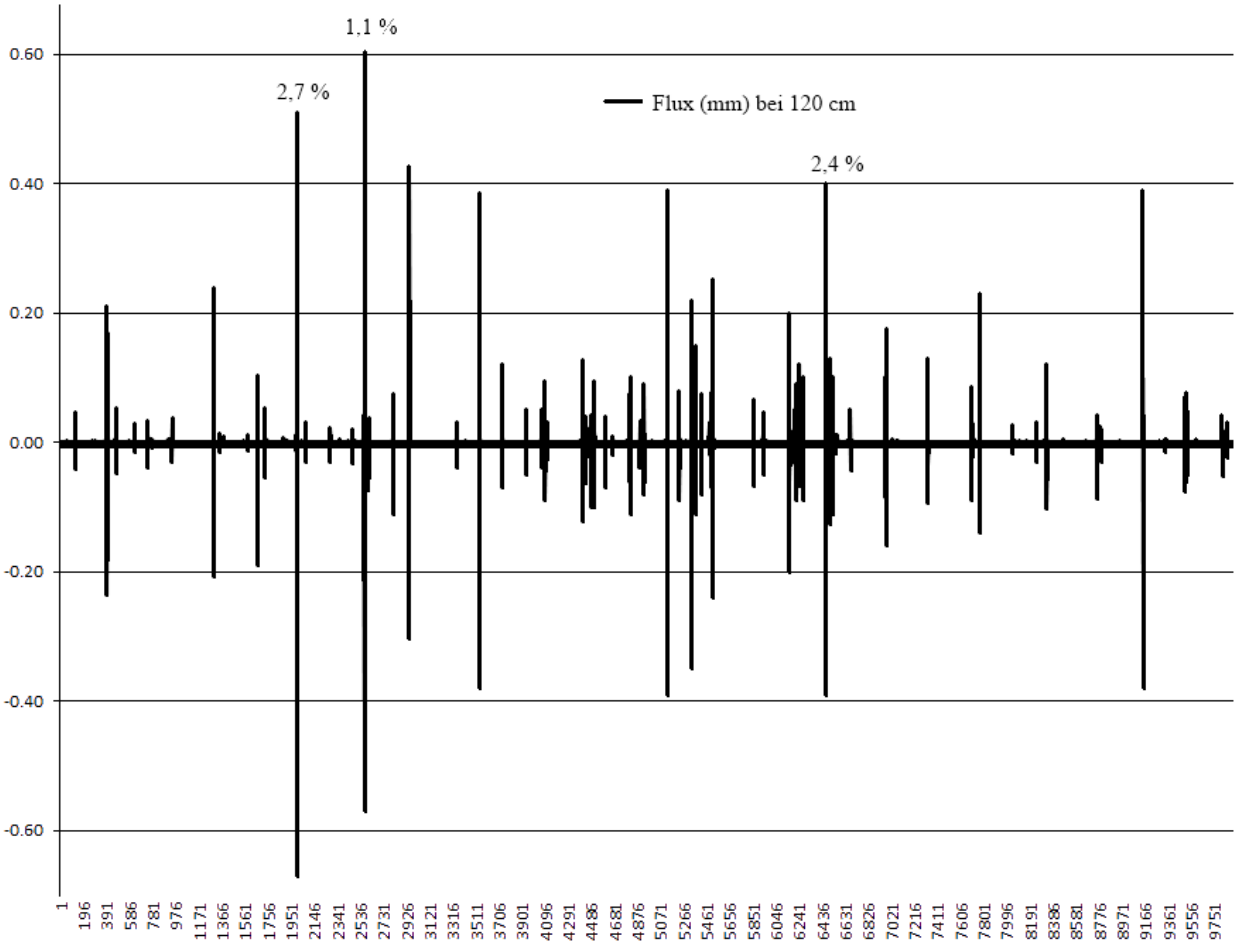

Abb. 3.11 Abweichungen des Wasserflusses (mm) in 120 cm

Simulationstag	Flux FORTRAN (mm)	Flux C# (mm)	Differenz (mm)	Abweichung
1999	18,3	17,79	0,51	2,7 %
2571	50,03	50,6	0,57	1,1 %
6471	16,2	15.8	0,4	2,4 %

Tab. 3.13 Abweichungen Wasserflusses in 120 cm

In den Abbildungen Abb. 3.12 und Abb. 3.13 sind weitere Vergleiche zu sehen, die belegen, dass die Abweichungen mit weniger als einem halben Prozent sehr gering ausfallen. Bei den Ergebnissen handelt es sich um die Werte für den Wassergehalt in 30 cm Tiefe. Sie stammen aus dem Testgebiet Güssing im Burgenland.

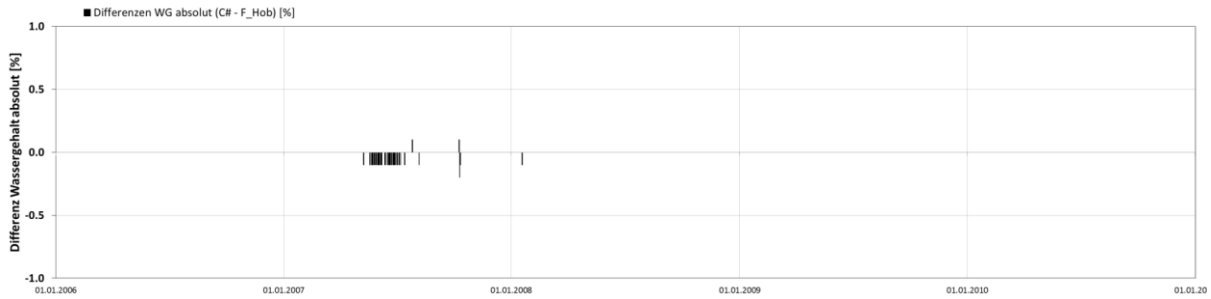

Abb. 3.12 Differenz des Wassergehalts bei 30cm (Burgenland)

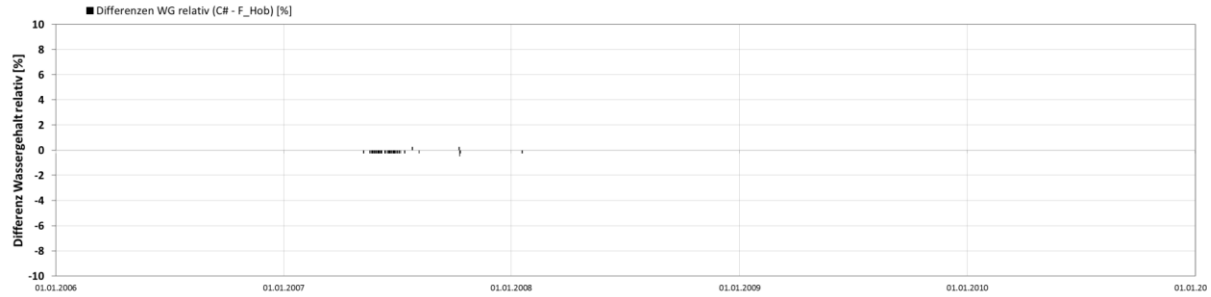

Abb. 3.13 Differenz des Wassergehalts bei 30cm in % (Burgenland)

Bei mehreren Treffen mit Verantwortlichen sowohl von Seiten des Institutes für Kulturtechnik und Bodenwasserhaushalt als auch von Seiten von JOANNEUM RESEARCH wurden die Abweichungen präsentiert und diskutiert. Dabei wurde die Relevanz der Abweichungen auf das Ergebnis der Simulation abgewogen. Bei den meisten Abweichungen herrschte sofort Einigkeit darüber, dass sie schon alleine wegen der geringen Unterschiede nicht relevant seien. Nur bei einzelnen Fällen, beispielsweise wenn einer der Diskussionsteilnehmer der Meinung war, dass die Herkunft der Abweichung genauer zu analysieren sei, weil sie eventuell noch andere Seiteneffekte auslösen könnte, wurden den Abweichungen zusammen mit einem Entwickler von JOANNEUM RESEARCH genauer auf den Zahn gefühlt. Erst als alle Zweifel über negative Auswirkungen der Abweichung ausgeräumt waren, wurde diese als tolerierbar hingenommen.

4 Ergebnisse und Ausblick

4.1 Ergebnisse

Als Ergebnis dieser Diplomarbeit entstand zwischen Oktober 2012 und Februar 2014 eine C# Version des Altsystems STOTRASIM. Neben der Abdeckung des Funktionsumfanges der FORTRAN 77 Version konnte dank Multithreadings die Rechendauer bei Projekten mit mehreren Läufen erreicht werden. Darüber hinaus wurde auch der ein oder andere kleine Fehler in der Simulation behoben und die Konsolenausgabe übersichtlicher gestaltet.

4.2 Ausblick

Neben Erweiterungen, wie zum Beispiel neuen Beregnungs- bzw. Bodenbearbeitungsmaßnahmen, welche durch Hinzufügen neuer Klassen problemlos durchgeführt werden können, gibt es für die Zukunft auch Pläne für weitreichendere Veränderungen.

4.2.1 Manipulation zur Laufzeit

Bei JOANNEUM RESEARCH besteht der Wunsch die Daten der Simulation zur Laufzeit, genauer nach jedem Rechentag, zu manipulieren. Auf diesen Wunsch wurde bei der Neuentwicklung bereits Rücksicht genommen. Der Ablauf der Simulation wurde bereits so umgebaut, dass parallel laufende Projekte nach jedem Tag ohne großen Aufwand angehalten

können. Es gilt nur mehr zu klären auf welche Art und Weise die Daten von einem anderen Programm manipuliert werden können.

4.2.2 Grafisches User Interface

Prinzipiell wäre eine Umsetzung eines einfachen grafischen User Interfaces, welches die momentane Konsolenausgabe ersetzt, kein Problem gewesen. Doch im Rahmen dieser Arbeit sollte fürs erste die Funktionalität der FORTRAN 77 Simulation in eine C# Anwendung überführt werden. Für die Zukunft ist man seitens JOANNEUM RESEARCH nicht abgeneigt zumindest für den Start der Simulation, also für die Konfiguration, ein einfaches User Interface zu erstellen. Dabei soll die Möglichkeit bestehen nach dem Laden der Projektdatei noch Änderungen an den Einstellungen durchzuführen.

Ein weiterer Punkt hierbei wäre dem User eine bessere Rückmeldung über den Fortschritt der Berechnungen zu liefern. Denkbar wären Informationen über den aktuellen Rechentag und die momentan berechnete Frucht.

Da bisher die Ergebnisse nur in Ergebnisdateien geschrieben werden, wäre auch die grafische Aufbereitung der Ergebnisse anhand von Diagrammen für die Zukunft denkbar.

4.2.3 Client/Server-System

Ein weiterer Punkt für zukünftige Entwicklungen wäre die Überführung in ein Client/Server-System. Wobei ein Großteil der bestehenden Anwendung den Serverteil darstellen würde und nur die Clientanbindung neu entwickelt werden müsste. Interessant wäre dies vor allem als App für mobile Endgeräte, die im Zuge von Außeneinsätzen zum Einsatz kommen könnten.

4.2.4 Unit Tests

Hinsichtlich zukünftiger Änderungen, seien es Refactoring-Schritte oder Erweiterungen, wäre eine Einführung von Unit Tests empfehlenswert. Einerseits könnte man dadurch sicherstellen, dass keine ungewollten bzw. versehentlichen Veränderungen am Programm durchgeführt

werden. Andererseits würden sich Entwickler auch leichter tun, anhand der vorhandenen Beispiele, den bestehenden Quellcode zu verstehen.

4.2.5 Bewässerungsart Turnus/Bedarf

Bisher sind in der Simulation nur die Bewässerungsarten Turnus- sowie Bedarfsbewässerung ausformuliert. In der Projektdatei ist aber seit jeher auch die Kombination aus beiden Bewässerungsarten zu finden. In der C# Version wurde darauf Rücksicht genommen, dass hier in Zukunft die Ausformulierung folgt.

4.2.6 Grundwasseranschluss

Wie der Testfall „JR_3_rechenaufwaendigerBoden“ zeigt, sind bei der Durchführung der Simulation mit Grundwasseranschluss in den Ergebnissen teilweise auffallende Sprünge zwischen den Werten von aufeinander folgenden Tagen zu finden. Auch bei JOANNEUM RESEARCH ist man sich über mögliche Probleme bei Berechnungen mit dem Grundwasseranschluss bewusst. Hier gilt es besonders die Zeitschrittschleife für diese Berechnungen genauer zu betrachten, da dort bei der Recherche nach der Ursache außergewöhnlich hohe Schwankungen in den Zwischenergebnissen aufgetreten sind.

5 Literaturverzeichnis

[BALZERT1] BALZERT, Helmut: Lehrbuch der Softwaretechnik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung. Spektrum akademischer Verlag, Heidelberg, 1998.

[BELLAY1] BELLAY Berndt, GALL Harald: A comparison of four reverse engineering tools, in Reverse Engineering. Proceedings of the Fourth Working Conference on - Digital Object, 1997.

[BENNICKE1] BENNICKE Marcel, RUST Heinrich: Programmverstehen und statische Analysetechniken im Kontext des Reverse Engineering und der Qualitätssicherung. Report. TU Cottbus, 2004.

[CHIKOFSKY] CHIKOFSKY Elliot J., CROSS James H: Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, 7(1):13–18, 1990.

[CHU1] CHU William C., LU Chih-Wei, CHANG Chih-Hung, CHUNG Yeh-Ching, LIU Xiaodong, YANG Hongji. Reverse Engineering, in S.K. Chang (editor): Handbook of Software Engineering & Knowledge Engineering, Volume 2 - Emerging Technologies, World Scientific Publishing Co. Pte. Ltd., 2002, Seiten 447-466.

[DIJKSTRA1] DIJKSTRA Edsger Wybe: The Humble Programmer. ACM Turing Lecture, 1972.

[FEICHTINGER1] FEICHTINGER Franz: STOTRASIM - Ein Modell zur Simulation der Stickstoffdynamik in der ungesättigten Zone eines Ackerstandortes. Schriftenreihe des Bundesamtes für Wasserwirtschaft, Band 7, 14-41, 1998.

[FRANZ1] FRANZ Klaus: Handbuch zum Testen von Web-Applikationen. Springer-Verlag Berlin Heidelberg, 2007.

[HAMMER1] HAMMER Michael, CHAMPY James: Reengineering the corporation: A manifesto for business revolution. HarperBusiness, New York, 1993.

[KÜBECK1] KÜBECK Sebastian: Software-Sanierung - Weiterentwicklung, Testen und Refactoring bestehender Software. mitp, Heidelberg, 2009.

[SIEDERSLEBEN1] SIEDERSLEBEN Johannes: Softwaretechnik: Praxiswissen für Software-Ingenieure. Hanser, München, 2003.

[STERNITZER1] STERNITZER Elmar: SIMWASER - Ein numerisches Modell zur Simulation des Bodenwasserhaushaltes und Pflanzenertrages eines Standortes. Mitt. Aus der Bundesanstalt für Kulturtechnik und Bodenwasserhaushalt, Nr. 31, A-3252 Petzenkirchen, 1988.

[VISAGGIO1] Giuseppe Visaggio: Ageing of a data-intensive legacy system: symptoms and remedies, JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE, J. Softw. Maint. Evol.: Res. Pract. 2001; 13:281–308 (DOI: 10.1002/smr.234).

[WAGNER1] Bernardo Wagner: Reverse-Engineering: Sanierung von Software-Altsystem, Technische Rundschau, Heft 10, 1992.