AKAD Hochschule für Berufstätige
Diplomstudium Wirtschaftsinformatik FH

Diplomarbeit

Anwendung der Monte-Carlo-Methode zur Simulation eines Business-Prozesses

Patrick Heusser

Matrikel-Nr.	98-910-235
Studiengang Vertiefungsrichtung	Wirtschaftsinformatik FH Business Engineer
Einreichungsort und Datum	Zürich, 28. Januar 2008
Korrektor und Betreuer	Martin Seelhofer
Autor	Patrick Heusser

Abstract

Diese Arbeit zeigt, wie die Monte-Carlo-Methode auf den Business-Prozess eines Modellunternehmens angewendet werden kann.

Ein zentraler Punkt ist dabei die vollständige Umsetzung in der Programmiersprache Java. Es werden zwei Hauptkomponenten implementiert: Das State-Machine-Framework, als generische Basiskomponente zur flexiblen Modellierung einer State-Machine, sowie darauf aufbauend die Monte-Carlo-Komponente, welche die Basiskomponente benutzt, um den Business-Prozess mittels der State-Machine abzubilden. Zusätzlich wird eine im Internet zugängliche Webapplikation erstellt, um die Monte-Carlo-Simulation des Modellunternehmens mit selbst definierten Parametern zu starten und abschliessend detaillierte Resultate des Simulationsdurchlaufs zu erhalten.

Ein weiterer Schwerpunkt der Arbeit liegt darin, aus der erstellten Simulation wirtschaftlich relevante Schlüsse zu ziehen. So werden die Auswirkungen von gelieferter Software-Nonkonformität untersucht und die finanziellen Folgen auf den Ertrag quantifiziert. Es wird gezeigt, wie die Simulation weitere Möglichkeiten zur Prozessoptimierung bietet: Es wird hergeleitet, in welchem optimalen Rhythmus Korrekturen an der Software vorgenommen werden und eine Auslieferung der Software stattfinden soll.

Es bleibt offen, inwiefern sich der gewählte Ansatz auf ein beliebiges Unternehmen anwenden lässt. Es wird vermutet, dass eine Hauptschwierigkeit darin besteht, einen real existierenden Prozess und seine Auswirkungen vollständig zu modellieren, um die Basis für die Monte-Carlo-Simulation zu schaffen.

Vorwort

Die Studienrichtung Wirtschaftsinformatik schlägt die Brücke zwischen Wirtschaft und Informatik. Gerade diese Verflechtung ist Anreiz für diese Arbeit gewesen. Die „Monte-Carlo-Methode“ wird daher zum einen in wirtschaftlich theoretischer Hinsicht angewendet, zum anderen mittels Informatik auf praktische Fragestellungen übertragen. Die Monte-Carlo-Methode ist nach Ansicht des Autors ein interessanter, verhältnismässig einfacher allgemeiner Ansatz zur Simulation von komplexen, schwer überschaubaren, nicht deterministischen Prozessen.

Informationen für den Leser

Dieses Dokument setzt beim Leser Grundkenntnisse in Wirtschaft und Informatik voraus. Die wichtigsten verwendeten Begriffe und Zusammenhänge werden in der Arbeit nur kurz beschrieben. Fachausdrücke werden entweder im Text erklärt oder mittels Fussnoten definiert. Für Letztere wird oftmals der erste Satz des entsprechenden Wikipedia-Artikels verwendet.

In der Informatik sind englische Ausdrücke feste Bestandteile der Fachsprache. Sie werden in dieser Arbeit selten ins Deutsche übersetzt. Ebenso werden Zitate oder Textausschnitte aus englischer Literatur nicht wörtlich übersetzt, sondern höchstens paraphrasiert.

Zur Erhöhung der Lesbarkeit des Textes wird immer nur die männliche Form verwendet. Sie steht gleichberechtigt auch für die weibliche Form.

Referenzen auf Bücher, Journale, Papers etc. folgen dem APA-Standard und werden mittels Kurzbeleg im Text integriert. In Abbildungen erfolgt die Angabe der Quelle direkt in der Abbildungsbezeichnung. Wird keine Quelle angegeben, so wurde die Abbildung selbst erstellt.

Teilweise wurden zur Recherche Online-Bibliotheken verwendet, die bei Büchern keine Seitenangaben anzeigen. Zur genauen Identifizierung der Textstelle wird in diesen Fällen die Kapitelnummer anstelle der Seitenzahl angegeben. Verweise auf Webseiten ohne ausdrücklich ausgewiesenen Autor (z. B.: Unternehmen, Non-Profit-Organisationen), erfolgen nach dem IEEE-Standard und mit dem vorangestellten Buchstaben „W“ für „Web“, z. B.: [W2].

INHALTSVERZEICHNIS

Abbildungsverzeichnis

1. AUSGANGSLAGE UND FRAGESTELLUNG

Wenn ein Unternehmen Software zur Unterstützung seiner eigenen betriebsinternen Prozesse entwickelt, stellt die Qualität der Software einen entscheidenden Erfolgsfaktor dar. Diese Qualität kann durch ein Testing der Software überprüft werden. Aus wirtschaftlicher Sicht wäre es interessant zu wissen, wie stark sich eine Verbesserung der Softwarequalität auf den finanziellen Erfolg eines Unternehmens auswirkt.

Die nachfolgenden Ausführungen untersuchen, ob die Monte-Carlo-Methode geeignet ist, Aufwand und Ertrag für eine Verbesserung der Softwarequalität abzuschätzen. Am Beispiel eines Modellunternehmens wird die Monte-Carlo-Methode exemplarisch angewendet mit dem Ziel, ihre Praxistauglichkeit zu überprüfen.

Daraus lassen sich folgende Fragestellungen ableiten:

- Kann die Monte-Carlo-Methode für ein Modellunternehmen in Java implementiert werden? Welche Erkenntnisse können bei der praktischen Umsetzung gewonnen werden?

- Lassen sich für das Modellunternehmen mittels der so angewendeten Monte-Carlo-Methode quantitative Aussagen über die finanziellen Auswirkungen durch eine Nonkonformität in der gelieferten Software machen?

- Können weitere wirtschaftsrelevante Schlüsse aus dieser Anwendung der Monte-Carlo-Methode gezogen werden?

2. GRUNDLAGEN

In diesem Kapitel werden die Begriffe „Software-Nonkonformität“, „Testing“, „Risikomanagement“ und „Monte-Carlo-Methode“ voneinander unabhängig definiert. Im anschliessenden Kapitel („3. Konzept zur Anwendung der Monte-Carlo-Methode“) werden die Grundlagen auf das Thema der Arbeit bezogen miteinander verknüpft.

2.1 Software-Nonkonformität

In dieser Arbeit wird der Begriff „Software-Nonkonformitätskosten“ häufig verwendet. Der Begriff ist aus dem Englischen „cost of nonconformance“ übernommen. Eine kurze Definition des Begriffs liefert Slaughter (Slaughter et al. 2002, S. 68): „The cost of nonconformance includes all expenses that are incurred when things go wrong.” Ausführlicher definiert Wagner (Wagner 2005, S. 2) in einem Paper:

„The nonconformance costs come into play when the software does not conform to the quality requirements. These costs are divided into internal failure costs and external failure costs. The former contains costs caused by failures that occurred during development, the latter describes costs that result from failures at the client.”

Die Software-Nonkonformität umfasst demzufolge alle Abweichungen der Software vom spezifizierten Verhalten. Die Software-Nonkonformitätskosten können also als finanzieller Schaden, verursacht durch die Software-Nonkonformität, verstanden werden. Das im folgenden Kapitel beschriebene Testing ist ein Mittel, die Software-Nonkonformität und die dadurch verursachten Kosten messen und steuern zu können.

2.2 Testing

2.2.1 Definition

Der Begriff „Testing“ oder „Software-Testing“ beschreibt Vorgänge im Prozess der Software-Entwicklung. In der Literatur sind unterschiedliche Definitionen für Testing zu finden.

Es werden zwei Testing-Techniken und zwei Testing-Verfahren unterschieden. Die Testing-Techniken legen fest, nach welchem Grundprinzip ein Testing durchgeführt wird. Die Testing-Verfahren definieren die Art des Vorgehens (Hetzel 1988, Kapitel 1.1): „Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.”

Diese Definition einer Testing-Technik beschreibt einen klassischen Ansatz. Ein zu entwickelndes System soll sich grundsätzlich genau so verhalten, wie es vorher spezifiziert wurde. Mit dem Testing wird sichergestellt, dass die Software alle Anforderungen erfüllt, die an sie gestellt wurden. Diese Vorgehensweise wird „Positives Testing“ genannt (Myers 2004, Kapitel 2.5): „Testing is the process of executing a program with the intent of finding errors.”

Dieser Ansatz unterscheidet sich von der vorhergehenden Definition erheblich. Hier ist das Ziel die Eliminierung von Fehlern, nicht die Erfüllung von Anforderungen. Diese Vorgehensweise wird „Negatives Testing“ genannt.

In der Praxis werden beide Testing-Techniken kombiniert angewendet. Zum einen wird sichergestellt, dass die Software alle Anforderungen erfüllt (Positives Testing), zum anderen muss gewährleistet sein, dass keine negativen Nebeneffekte auftreten, die eine korrekte Funktionsweise beinträchtigen würden (Negatives Testing).

Daneben werden zwei Testing-Verfahren, das White- und das Blackbox-Verfahren, unterschieden (Watkins 2001, Kapitel 3.2). Die Unterscheidung erfolgt nach der Vorgenes-Methode, allfällige Fehler zu finden. Beim Whitebox-Verfahren wird vorausgesetzt, dass die genauen Abläufe innerhalb des erstellten Codes in einer Software bekannt sind. Das Testing wird aufgrund dieses Wissens durchgeführt. Genau gegensätzlich wird beim Blackbox-Verfahren vorgegangen. Hier verfügt der Tester über keinerlei Wissen über den internen Programmaufbau. Das Testing wird aufgrund der Anforderungsspezifikation erstellt.

2.2.2 Testing von grossen Softwareprojekten

Besonders das Testing von grossen Softwareprojekten ist eine Herausforderung. Ein Programm von grosser Komplexität umfasst unzählige Möglichkeiten des Programmflusses. Dazu kommt, dass Eingabeparameter bestimmte Wertebereiche abdecken müssen. Werden alle Freiheitsgrade ausgeschöpft, führt dies kombinatorisch betrachtet zu unzähligen Varianten des Programmablaufs.

Um sicherzustellen, dass das Programm exakt so funktioniert wie spezifiziert, müssten alle Varianten getestet werden, was in der Praxis unrealistisch ist. Daraus lässt sich schliessen, dass selbst eine sorgfältig getestete Software kaum fehlerfrei sein wird. Scott Loveland (Loveland 2005, Kapitel 20.3) überschreibt in seinem Buch gar ein Kapitel pointiert mit der Aussage: „The Final Lesson – Bugs: You'll Never Find Them All". Das Kapitel lässt sich etwa folgendermassen zusammenfassen: Es ist in komplexen Programmen unmöglich, alle Fehler zu finden. Deshalb sollen vorrangig die Fehler gesucht werden, die beim Softwareanwender den grössten Schaden verursachen würden.

2.3 Risikomanagement

2.3.1 Definition

Risikomanagement ist der Vorgang zur systematischen Erfassung, Bewertung und Steuerung von Risiken. Eine Definition im Kontext des Projektmanagements liefert Smith (Smith 2002, Kapitel 1.4):

"Applied to a project, a risk is the possibility that an undesired outcome – or the absence of a desired outcome – disrupts your project. Risk management, then, is the activity of identifying and controlling undesired project outcomes proactively."

Ein Risiko ist demnach eine Eventualität, dass ein bestimmtes Ereignis eintreten oder nicht eintreten kann.

2.3.2 Tätigkeiten beim Risikomanagement

Das Risikomanagement soll Risiken proaktiv erkennen und behandeln. Um mit Risiken umgehen zu können, sind nach Delhees (Delhees, Scheuring 2004, S. 42-45) folgende Schritte notwendig:

- Risiken bewerten
- Risiken klassifizieren
- Risiken bewältigen

Zur Risikobewertung wird die Risikomatrix vorgeschlagen. Dies ist eine einfache Methode, um einen Überblick zu erhalten und die Risiken gewichten zu können. Folgende Grafik (Abbildung 1) veranschaulicht dies:

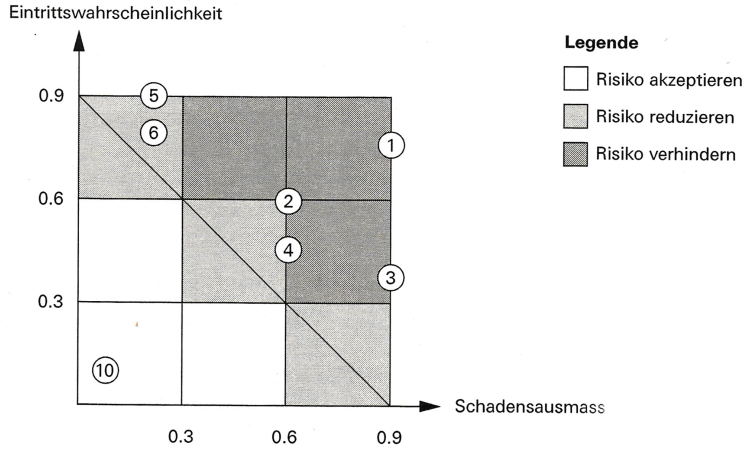


Abbildung 1: Beispiel einer Risikobeurteilung mittels Riskomatrix (Delhees 2004, S. 44)

In dieser Matrix werden die Risiken anhand des zu erwartenden Schadensausmasses (X-Achse) und der Eintrittswahrscheinlichkeit (Y-Achse) eingetragen. Im Beispiel erfolgt dies mit Nummern. Die Risiken in den dunklen Feldern stellen die grössten Problembereiche dar und sollten prioritär angegangen werden. Beispielsweise hat das Risiko mit „Nummer 1" sowohl ein hohes zu erwartendes Schadensausmass als auch eine hohe Eintrittswahrscheinlichkeit. Risiken, die im hellgrauen Bereich liegen, haben ungefähr dieselbe Gewichtung.

Eine weitere Möglichkeit, Risiken zu bewerten und zu klassifizieren, ist formelbasiert. Die Formel wird in der Risikomanagement-Literatur oft verwendet und findet sich beispielsweise bei Smith (Abbildung 2):

Abbildung 2: Formel zur Berechnung des zu erwartenden Verlustes (Smith 2002, Kapitel 3.3)

Die Formel hat Ähnlichkeiten zur Risikomatrix. So entspricht „Risk likelihood" der Y-Achse der Risikomatrix und der dritte Faktor „Total amount of loss if risk occurs" der X-Achse. Mathematisch betrachtet, wird der aus der Statistik bekannte Erwartungswert errechnet.

Das Ergebnis von beiden Mitteln zur Risikobewertung ist qualitativ ähnlich. Gemäss der Risikomatrix löst das Risiko „Nummer 1" den höchsten Handlungsbedarf aus. Würde das gleiche Risiko in die Formel eingesetzt, führte dies zu einem ähnlichen Resultat. Da beide Faktoren gross sind, wird daraus auch ein grosses Produkt resultieren. Das Produkt wiederum entspricht dem „zu erwartenden Schaden" und da selbiges gross ist, wird dies eine Handlung zur Verhinderung des Risikos verlangen.

2.4 Die Monte-Carlo-Methode zur Berechnung der Zahl PI

2.4.1 Einführung

Die Monte-Carlo-Methode wurde zur Zeit des Zweiten Weltkrieges im Rahmen des amerikanischen „Manhattan-Projektes" (Entwicklung der Atombombe) entwickelt, um bei chemischen Abläufen gewisse Voraussagen über deren Ausgang machen zu können. Die Methode wurde benutzt, um mathematische Funktionen, die gar nicht oder nur sehr schwer analytisch gelöst werden konnten, mittels Simulation zu approximieren.

Ausgehend von den militärischen Anwendungsgebieten wurde die Monte-Carlo-Methode erst in den 90er Jahren in der Wirtschaft angewendet, zuerst in der Erdölindustrie, kurze Zeit später in der Finanzwelt zur Berechnung des Preises von Optionen an der Börse. Weitere Einzelheiten zur historischen Entwicklung führt Wanka (Wanka 2006, S. 160) aus. Heute findet die Monte-Carlo-Methode immer breitere Anwendung vor allem im Risikomanagement zur quantitativen Bewertung von Risiken. An amerikanischen Universitäten wird die Monte-Carlo-Methode gemäss Frey (Frey 2001, S. 17) bei MBA-Lehrgängen bereits fix im Studienprogramm integriert.

Das Interesse an dieser Methode wird aber auch in der breiten Bevölkerung immer grösser. So hat der Tagesanzeiger[1] im September 2007 zwei Beiträge im Wirtschaftsbund abgedruckt. Beide Artikel füllten jeweils eine ganze Seite und behandelten die Monte-Carlo-Methode und deren Möglichkeiten, Investment-Risiken kalkulierbar zu machen. Beide Artikel sind in dieser Arbeit im Anhang (Kapitel „9.5 Artikel im Tages-Anzeiger zur Monte-Carlo-Methode“) angefügt.

2.4.2 Ein einfaches Anwendungsbeispiel: Berechnung der Zahl PI

Die deutsche Wikipedia-Seite [W3] zeigt mit der Berechnung der Zahl PI ein gutes, leicht verständliches Anwendungsbeispiel der Monte-Carlo-Methode. Da das Beispiel aber nur sehr rudimentär erläutert wird, wurde dieses verfeinert und die mathematische Herleitung dazu ausgearbeitet. Abschliessend wurde ein einfaches Java-Programm implementiert, um die erarbeitete Theorie praktisch zu überprüfen.

Zur Herleitung wird mit bekannten Formeln und messbaren Vorgängen gearbeitet. Grundlage sind ein Viertelkreis und ein Quadrat, das in den Viertelkreis eingeschrieben ist (siehe Abbildung 3). Auf die Fläche des Quadrates werden virtuelle Pfeile (kleine Dreiecke) auf eine zufällige Position geworfen. Es wird anschliessend untersucht, ob ein Pfeil jeweils im Bereich des Quadrates gelandet ist, welcher ausserhalb (graue Dreiecke) oder innerhalb (rote Dreiecke) des Kreises liegt.

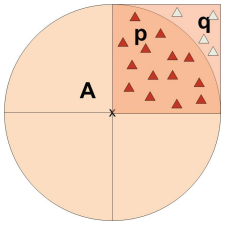


Abbildung 3: Virtuelle Pfeile innerhalb oder ausserhalb des Kreisbogens

Die eigentliche Berechnung von PI erfolgt abschliessend über den Vergleich zweier spezieller Verhältnisse: eines Verhältnisses von Anzahl Pfeilen und eines Verhältnisses von Flächen. Durch mathematische Gleichsetzung dieser Verhältnisse erhält man eine Formel für PI, die nur von der Anzahl Pfeilen („p“ und „q“) abhängig ist. Die genaue Herleitung der Formel findet sich im Anhang im Kapitel „9.1 Herleitung der Formel zur Berechnung von PI mittels der Monte-Carlo-Methode“. Die nachfolgende Abbildung 4 zeigt das Ergebnis dieser Herleitung:

[1] Der Tages-Anzeiger ist eine überregionale Zürcher Tageszeitung und gilt als eine der führenden Zeitungen der deutschsprachigen Schweiz.

$$\pi = \frac{4p}{p+q}$$

Abbildung 4: Formel zur Approximation
von PI mittels Monte-Carlo-Methode

Umsetzung in Javacode
Die hergeleitete Formel wird nun in einem Java-Programm benutzt, um PI zu approximieren und die erarbeitete Theorie praktisch zu verifizieren.

Voraussetzung dafür ist, dass die Pfeile wiederum nach dem Zufallsprinzip auf das Quadrat mit eingeschriebenem Viertelkreis treffen und zudem gemessen werden kann, in welchem Flächenteil („p“ oder „q“) der Pfeil aufgetroffen ist. Der Einfachheit halber wird ein Kreis mit Radius 1 angenommen. Mittels der Koordinaten „x“ und „y“, die auf einer Zufallszahl zwischen 0 und 1 beruhen, wird der aufgetroffene Pfeil festgelegt (Abbildung 5). Nun muss überprüft werden, ob der Pfeil zur Kategorie „p“ oder „q“ gezählt werden soll.

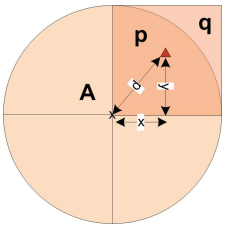


Abbildung 5: Überprüfung, ob ein Punkt
innerhalb des Kreisbogens liegt

Dazu wird ausgerechnet, ob der Pfeil innerhalb des Viertelbogens aufgetroffen ist. Dies ist immer dann der Fall, wenn sein Abstand vom Mittelpunkt des Kreises kleiner ist als 1. Der Abstand „d“ lässt sich mit Pythagoras berechnen:

$$d = \sqrt{x^2 + y^2}$$

Das Programm zur praktischen Überprüfung der Theorie wurde wie folgt implementiert (Abbildung 6):

```
Random rnd = new Random();
long p = 0; long q = 0; long loops = 10000; long reportFrequency = 10;

DecimalFormat format1 = new DecimalFormat(" 0.00000000;-0.00000000");
DecimalFormat format2 = new DecimalFormat("000000");
double estimatedPI = 0;
for (long i = 0; i < loops; i++) {

    double x = rnd.nextDouble(); // number between 0..1
    double y = rnd.nextDouble(); // number between 0..1
    double r = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
    if (r <= 1) {
        p++;  // we are inside the circle
    } else {
        q++;  // we are outside the circle
    }
    estimatedPI = ((double) (4 * p)) / (p + q);
    if ((i % reportFrequency) == 0) {

        System.out.println("loop:\t" + format2.format(i) + "\tcurrent PI=\t"
                + format1.format(estimatedPI) + "\t error[%]=\t"
                + format1.format(Math.PI / estimatedPI * 100 - 100));
}}
```

Abbildung 6: Javacode zur PI-Approximation mittels der Monte-Carlo-Methode

Das Programm zählt fortlaufend die Anzahl Pfeile „p“ und „q“ und berechnet mit der hergeleiteten Formel die Zahl PI. Wird die Ausgabe übernommen und das Resultat in Excel dargestellt, ergibt sich bei insgesamt 10'000 virtuellen Pfeilen folgende Kurve (Abbildung 7).

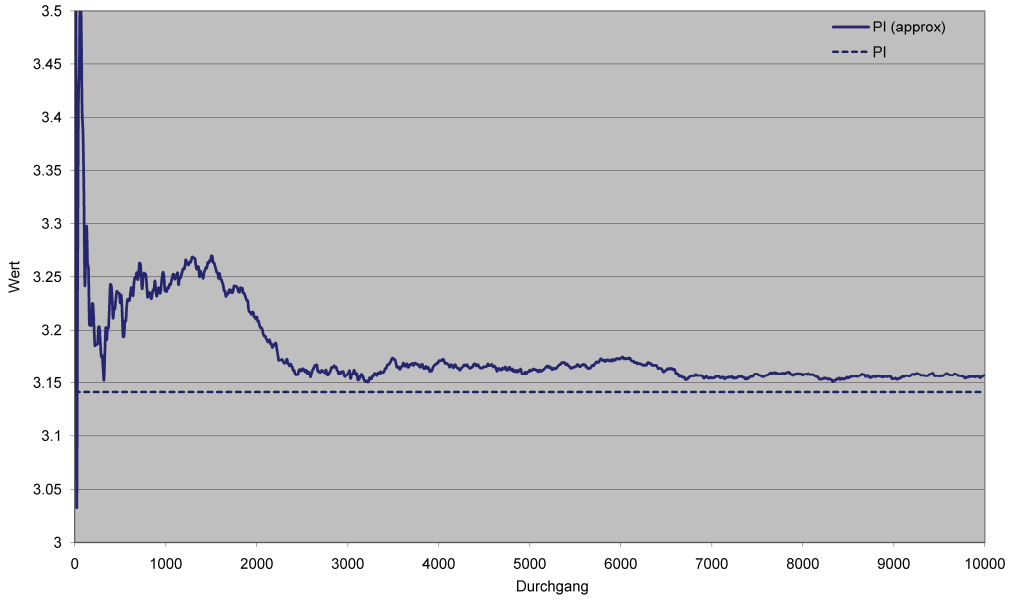


Abbildung 7: Annäherungsverhalten bei der PI-Approximation

Die Abbildung zeigt auf der Y-Achse den Wert PI, der in der entsprechenden Iteration berechnet wurde. Die X-Achse zeigt die Anzahl der virtuellen Pfeile.

Die ersten Werte waren ziemlich ungenau. Nach 1'000 Pfeilen lag der Wert für PI bei ungefähr 3.25. Nach 10'000 Iterationen lag der Wert bereits bei 3.15724152, was der Zahl PI ziemlich nahe kommt. Es ist zu erkennen, dass sich der Algorithmus immer genauer PI nähert, je mehr Iterationen ausgeführt werden. Wird die Simulation über 1'000'000 Iterationen ausgeführt, verbessert sich der Wert abermals. PI entspricht nun dem Wert von 3.14210296. Dies ist eine Genauigkeit von 99.98 %.

Erkenntnis aus dem Anwendungsbeispiel
Die Nützlichkeit der Monte-Carlo-Methode wird aufgezeigt. Es war möglich, PI zu approximieren, ohne genaue Kenntnisse über die Kreiszahl zu haben. Dies wurde erreicht, indem der Vorgang viele Male wiederholt wurde. Ein Mangel an Wissen kann hier also mit einer hohen Anzahl an Iterationen kompensiert werden.

2.5 Die Monte-Carlo-Methode zur Planung und Vorhersage

Das vorangehende Kapitel gab einen ersten Eindruck über eine Anwendungsmöglichkeit der Monte-Carlo-Methode. Die Einsatzmöglichkeiten sind aber sehr vielfältig. Besonders zur Erstellung von Prognosen mittels Modellen eignet sich dieser Ansatz zum Bestimmen der Einstrittswahrscheinlichkeit gewisser Szenarien. Dieses Kapitel zeigt nach Frey (Frey 2001, S. 21-44), welches die Vorteile der Schätzung mittels Monte-Carlo-Methode gegenüber zwei klassischen Schätzmethoden sind. Die Terminologie ist ebenfalls von Frey übernommen.

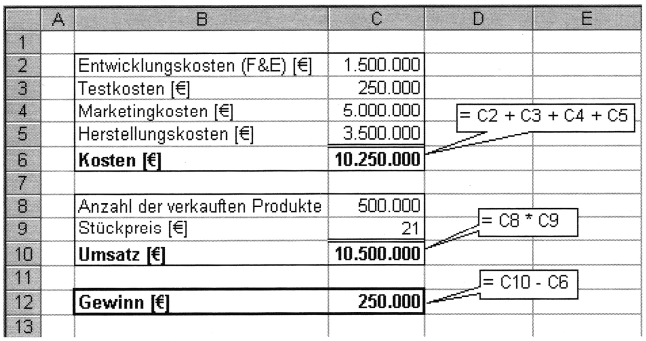
2.5.1 Ausgangslage

Ein Unternehmen möchte ein Produkt entwickeln. Dabei entstehen durch die Entwicklung zuerst Kosten, später durch den Verkauf Einnahmen. Mehrere Faktoren bestimmen den Gewinn, z. B. Entwick-

lungskosten, Marketingkosten, erzielbarer Stückpreis, Absatz etc. In der Planung ist es essenziell zu wissen, ob sich die Entwicklung eines Produktes lohnt. Dies kann mittels verschiedener Schätzmethoden berechnet werden.

2.5.2 Die deterministische Methode

Die deterministische Methode ist nach Frey das einfachste Schätzmodell. Als Grundlage dienen fixe Schätzwerte für jeden Budgetposten. Die Werte für Kosten und Ertrag werden jeweils aufsummiert. Als Differenz resultiert der Gewinn. Siehe als Beispiel dazu die nachfolgende Abbildung (Abbildung 8):



	A	B	C	D	E
1					
2		Entwicklungskosten (F&E) [€]	1.500.000		
3		Testkosten [€]	250.000		
4		Marketingkosten [€]	5.000.000		
5		Herstellungskosten [€]	3.500.000		
6		**Kosten [€]**	**10.250.000**		
7					
8		Anzahl der verkauften Produkte	500.000		
9		Stückpreis [€]	21		
10		**Umsatz [€]**	**10.500.000**		
11					
12		**Gewinn [€]**	**250.000**		
13					

Abbildung 8: Planung mit der deterministischen Methode (Quelle: Frey 2001, S. 19)

Grosser Vorteil dieser Methode ist ihre einfache Anwendung. Aus diesem Grund wird sie auch oft eingesetzt. Leider ist sie aber im Gegenzug auch ungenau. Pro Budgetposten wird ein einziger Schätzwert angenommen. Es ist sehr unwahrscheinlich, dass die einzelnen Schätzwerte die Wirklichkeit genau treffen. Weiter bleibt unbeantwortet, wie sich eine Veränderung der Planung auswirken wird. Diese Fragestellung führt zur nächsten Schätzmethode.

2.5.3 Methode der Szenarienbetrachtung

Die Methode der Szenarienbetrachtung kann genutzt werden, um über die Auswirkungen von möglichen Entwicklungen genauere Angaben zu erhalten. Frey erweitert dazu das deterministische Modell um zwei Szenarien: den „schlechtesten Fall", den „wahrscheinlichsten Fall" und den „besten Fall".

	A	B	C	D	E	F
1						
2			Schlechtester Fall	"Wahrschein-lichster" Fall	Bester Fall	
3						
4		Entwicklungskosten (F&E) [€]	2.300.000	1.500.000	1.300.000	
5		Testkosten [€]	320.000	250.000	200.000	
6		Marketingkosten [€]	7.000.000	5.000.000	4.500.000	
7		Herstellungskosten [€]	4.700.000	3.500.000	3.000.000	
8		**Kosten [€]**	**14.320.000**	**10.250.000**	**9.000.000**	
9						
10		Anzahl der verkauften Produkte	430.000	500.000	590.000	
11		Stückpreis [€]	21	21	21	
12		**Umsatz [€]**	**9.030.000**	**10.500.000**	**12.390.000**	
13						
14		**Gewinn [€]**	**-5.290.000**	**250.000**	**3.390.000**	
15						

Abbildung 9: Planung basierend auf „Szenarienbetrachtung" (Quelle: Frey 2001, S. 22)

Obenstehendes Beispiel (Abbildung 9) zeigt mögliche Szenarien auf. Im schlechtesten Fall (Spalte C) wurde die Annahme getroffen, dass sich alle Faktoren negativ entwickeln, d. h. die Kosten gegenüber dem „wahrscheinlichsten Fall" steigen und die Umsätze sinken. Es wird bei diesem Szenario ein Verlust von 5'290'000 EUR resultieren. Umgekehrt verhält es sich im besten Fall (Spalte E), mit der tiefsten

Kostenschätzung und den gleichzeitig höchsten Umsatzerwartungen. Es würde ein Gewinn von 3'390'000 EUR erreicht.

Auch diese Methode hat Nachteile. Zwar werden mögliche Szenarien aufgezeigt, es kann aber nicht abgeleitet werden, mit welcher Wahrscheinlichkeit ein solches Szenario eintreffen wird. So bedingt das Eintreffen des „schlechtesten Falles" eine Konstellation, in der jeder einzelne Faktor seine schlechteste Ausprägung annimmt. Dies ist aufgrund der Kombinatorik zwar möglich, jedoch unwahrscheinlich. Gleiches gilt für den „besten Fall". Einen Lösungsansatz bietet die Anwendung der Monte-Carlo-Methode.

2.5.4 Die Monte-Carlo-Methode als Erweiterung der Szenarienbetrachtung

Die Szenarienbetrachtung wird nun ausgeweitet. Anstelle von nur drei Szenarien werden unzählige evaluiert, ähnlich wie dies schon im Einführungsbeispiel mit PI gemacht wurde (siehe Kapitel „2.4.2 Ein einfaches Anwendungsbeispiel: Berechnung der Zahl PI").

Das Zufallselement „virtueller Pfeil" aus der PI-Approximation entspricht nun den einzelnen Budgetposten. Im Gegensatz zur Berechnung von PI wird nicht mit einem gleichmässig zufälligen Eintreffen gerechnet. Die Eintreffenswahrscheinlichkeit der Kosten wird basierend auf einer zuvor festgelegten mathematischen Verteilung erfolgen. Dies wird die Realität eher abbilden. Am Beispiel der Marketingkosten könnte eine Verteilung folgendermassen (Abbildung 10) aussehen.

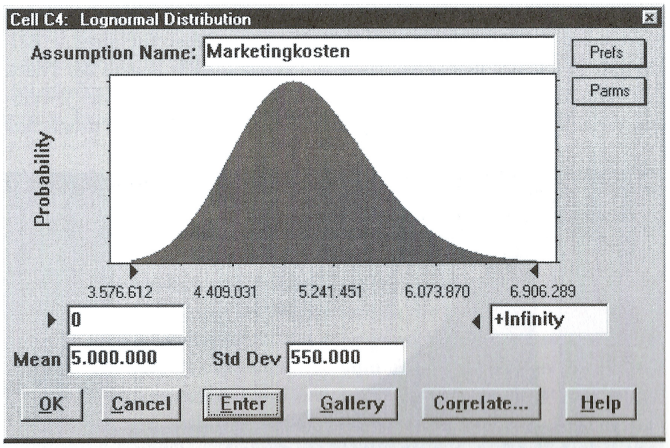


Abbildung 10: Beispiel einer logarithmischen Normalverteilung (Quelle: Frey 2001, S. 38)

Die geschätzten Kosten werden, basierend auf einer logarithmischen Normalverteilung, um den maximalen Wert von 5'000'000 EUR variiert. Dieser wurde zuvor als wahrscheinlichste Variante festgelegt. Analog werden allen Budgetposten Verteilungen zugewiesen. Frey verwendet ein Excel Add-In (Crystalball[2]), das die verschiedenen Szenarien simuliert. Dabei werden für jeden Budgetposten Zufallswerte gemäss der Verteilung verwendet. Daraus resultierend wird der zu erwartende Gewinn berechnet. Zur Veranschaulichung wird eines dieser vielen Szenarien exemplarisch herausgegriffen (Abbildung 11).

[2] Crystalball: Ist eine auf Microsoft Excel basierende Software, welche die Monte-Carlo-Methode mit Hilfe von Excel-Tabellen ausführt. http://www.crystalball.com.

	A	B	C	D
1				
2		Entwicklungskosten (F&E) [€]	1.920.359	
3		Testkosten [€]	236.780	
4		Marketingkosten [€]	4.570.428	
5		Herstellungskosten [€]	3.683.460	
6		**Kosten [€]**	**10.411.027**	
7				
8		Anzahl der verkauften Produkte	501.148	
9		Stückpreis [€]	21	
10		**Umsatz [€]**	**10.524.108**	
11				
12		**Gewinn [€]**	**113.081**	
13				

Abbildung 11: Herausgegriffenes Szenario der Monte-Carlo-Simulation (Quelle: Frey 2001, S. 33)

Gegenüber dem vorangehenden „wahrscheinlichsten Fall“ (Abbildung 9) zeigt sich in diesem Szenario, dass die Marketing- und Testkosten günstiger als erwartet, die Entwicklungs- und Herstellungskosten jedoch höher ausfallen. Trotzdem resultiert ein Gewinn von 113'081 EUR. Dieses Szenario ist aber nur eines von vielen Szenarien, das die Software durchrechnet. Insgesamt rechnet Crystalball in diesem Beispiel 5'000 Szenarien durch. Nach Berechnungsschluss werden in einem Histogramm (Abbildung 12) alle Szenarien zusammengestellt. Auf der X-Achse wird der Ertrag gegenüber der Eintrittswahrscheinlichkeit/Eintrittshäufigkeit (Y-Achse) aufzeigt.

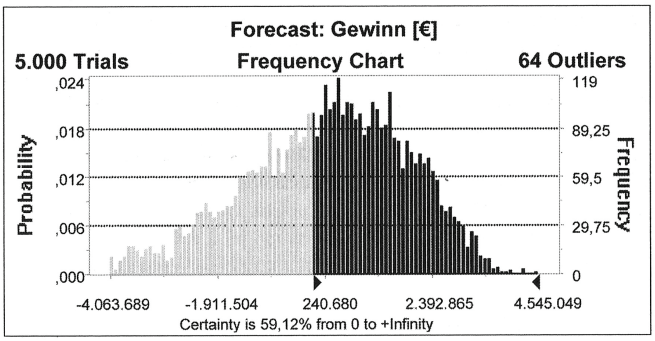


Abbildung 12: Histogramm des zu erwartenden Gewinns (Quelle: Frey 2001, S. 44)

Die Abbildung 12 zeigt, dass in ganz wenigen Fällen ein erheblicher Verlust von etwa 4 Millionen Euro resultiert. Die Eintrittswahrscheinlichkeit dafür ist jedoch gering. Ebenso verhält es sich mit dem maximalen Gewinn von etwa 4.5 Millionen Euro. Zur Errechnung der Wahrscheinlichkeit eines positiv ausfallenden Gewinns werden alle Eintrittswahrscheinlichkeiten, die in der Gewinnzone liegen (dunkle Balken rechts vom Null-Gewinnspunkt), summiert. In 59% aller Fälle ist demnach mit einem Gewinn grösser als 0 zu rechnen.

Gegenüber der Szenarienbetrachtung aus dem vorangehenden Kapitel werden nun genauere Angaben geliefert, zum Beispiel darüber, mit welcher Wahrscheinlichkeit wie viel Gewinn resultiert. Das Szenario für den „besten Fall“ tritt nur sehr selten auf und hat deshalb nur eine sehr begrenzte Aussagekraft.

Das Histogramm stellt ein geeignetes Mittel zur Visualisierung der verschiedenen Eintrittswahrscheinlichkeiten dar und wird deshalb später in dieser Arbeit erneut aufgegriffen.

3. KONZEPT ZUR ANWENDUNG DER MONTE-CARLO-METHODE

3.1 Verknüpfung der Grundlagen

Die unabhängigen Themenbereiche aus dem vorangehenden Grundlagenkapitel werden nun zusammengeführt. Dies wird zeigen, auf welchen theoretischen Grundlagen die Monte-Carlo-Methode in dieser Arbeit eingesetzt werden soll und welche grundsätzlichen Vorteile daraus entstehen.

3.1.1 Gedanken zur theoretischen Anwendbarkeit der Monte-Carlo-Methode

Es wurde einführend gezeigt (Kapitel „2.2.1 Definition"), dass Testing einen komplexen Teilprozess im gesamten Entwicklungsprozess darstellt. Zudem wurde festgestellt, dass es nahezu unmöglich ist, eine Software fehlerfrei auszuliefern. Die Gesamtheit unerwünschten Verhaltens der Software wird Software-Nonkonformität genannt (siehe Kapitel „2.1 Software-Nonkonformität").

Die Auslieferung von Software ist also risikobehaftet. Der mögliche finanzielle Schaden, ausgelöst durch Software-Nonkonformität, erfüllt dabei die Kriterien der Definition eines Risikos (Kapitel „2.3.1 Definition"). Im Kontext des Projektmanagements ist Risikomanagement das Instrument zur Handhabung von Risiken. Die Bewertung und Klassifizierung der Risiken erfolgt dabei mittels der Risikoformel. Zur Erinnerung die simplifizierte Version der Formel aus Abbildung 2:

$$RiskLikeli\,hood \times TotalAmoun\,tOfRisk = ExpectedLo\,ss$$

Es soll nun gezeigt werden, wie die Monte-Carlo-Methode angewendet werden kann, um Risikomanagement präziser zu betreiben. Dazu soll die obige Formel mit Schätzmethoden von Frey verglichen werden (Frey 2001, siehe dazu Kapitel „2.5 Die Monte-Carlo-Methode zur Planung und Vorhersage").

Die obige Formel berechnet einen Erwartungswert (genannt „ExpectedLoss"). Es wird also mit der wahrscheinlichsten Variante gerechnet, um daraus einen möglichen Schaden auszurechnen. Ausser Acht gelassen werden aber die verschiedenen Szenarien, die auftreten können, beispielsweise der „beste" und der „schlechteste anzunehmende Fall", wie sie bei der Szenarienbetrachtung erläutert wurden (siehe Kapitel „2.5.3 Methode der Szenarienbetrachtung"). Eine noch genauere Vorhersage, ob ein Risiko eintrifft, könnte durch Anwendung der Monte-Carlo-Methode gemacht werden. Sie würde unzählige Szenarien aufgrund der gegebenen Wahrscheinlichkeiten durchrechnen und dann einen präziseren Erwartungswert für das Risiko liefern.

Aus dieser Herleitung lässt sich schliessen, dass sich Kosten und Risiken, die die durch Software-Nonkonformität entstehen, basierend auf der Monte-Carlo-Methode vorhergesagt bzw. berechnet werden können. Die Monte-Carlo-Methode wäre somit ein geeignetes Instrument, um effizienter und genauer Risikomanagement zu betreiben, was sich folglich in einem erfolgreicheren Projektabschluss manifestieren würde.

3.2 Anwendung der Monte-Carlo-Methode auf die gegebene Fragestellung

Nachdem gezeigt wurde, dass die Fragestellungen dieser Arbeit grundsätzlich mittels der Monte-Carlo-Methode gelöst werden können, wird nun erläutert, wie die Simulation aufgebaut werden soll.

Im Grundlagenkapitel werden zwei Anwendungsbeispiele der Monte-Carlo-Methode aufgezeigt: zum einen die Approximation der Zahl PI (Kapitel „2.4. Ein einfaches Anwendungsbeispiel: Berechnung der Zahl PI"), zum anderen die Errechnung des zu erwartenden Gewinns (Kapitel „2.5.4. Die Monte-Carlo-Methode als Erweiterung der Szenarienbetrachtung").

Dabei ist letzteres als Erweiterung des ersten zu sehen. Beim ersten Anwendungsbeispiel konnte mittels nur eines Faktors, nämlich des zufällig platzierten, virtuellen Pfeils, der gesuchte Zielwert durch Anwendung der Monte-Carlo-Methode berechnet werden. Bei der Szenarienbetrachtung mussten hingegen bereits mehrere Faktoren simuliert werden.

In beiden Anwendungsbeispielen zeigt sich die Eigenheit der Monte-Carlo-Methode: die Voraussage eines gewissen Ausgangs durch unzählige Simulationsdurchläufe. Auf dieser Eigenheit basiert auch der in dieser Arbeit gewählte Lösungsansatz, der die Szenarienbetrachtung aber nochmals erweitert. In einem Simulationsdurchgang wird der gesamte Business-Prozess des Modellunternehmens über einen gewählten Zeitraum durchgerechnet.

Die folgende Tabelle gibt einen Überblick, welches Ziel durch die Simulationen mittels der Monte-Carlo-Methode angestrebt wird und was in einem Simulationsdurchgang geschieht.

Ziel	*Was geschieht in einem Simulationsdurchgang?*
PI berechnen.	Einen virtuellen Pfeil werfen.
Zu erwartenden Gewinn und dessen Eintreffenswahrscheinlichkeit vorhersagen.	Veränderung jedes Budgetpostens anhand einer vorgegebenen Verteilung.
Auswirkungen von Software-Nonkonformität simulieren.	Kompletter Durchlauf des zugrunde liegenden Business-Prozesses zur Entwicklung von Software, inklusive all seiner Einzelschritte und der damit verknüpften Entscheidungen im Modellunternehmen während eines definierten Zeitraums.

Abbildung 13: Vergleich der verschiedenen Simulationsarten

Das folgende Beispiel soll zeigen, welche Vorteile die in dieser Arbeit gewählte Methodik gegenüber dem Ansatz von Frey bietet.

Beispiel: Abhängigkeit und gegenseitige Beeinflussung der Faktoren
Als Ausgangslage dient abermals das Unternehmen von Frey (siehe Kapitel „2.5.4 Die Monte-Carlo-Methode als Erweiterung der Szenarienbetrachtung“) mit seinen Budgetposten (z. B. Abbildung 11). Angenommen wird ein starker Anstieg der Nachfrage nach dem produzierten Produkt. Folglich wird entschieden, die Produktion zu steigern. Dies bedingt die Anschaffung einer zweiten Maschine, was wiederum die Rekrutierung von zusätzlichem Betriebspersonal verlangt.

Frey führt in der Simulation mehrere Budgetposten. Zwei davon werden in diesem Beispiel aufgegriffen. Dies sind einerseits die „Herstellungskosten“ – auf diesen Budgetposten müssen die Kosten für die zusätzliche Maschine gebucht werden – und andererseits die „Anzahl der verkauften Produkte“; diese steigt an, wenn mehr Produkte verkauft werden.

Diese beiden Budgetposten werden bei Frey durch Verteilfunktionen beschrieben, die voneinander aber unabhängig sind und somit voneinander isoliert simuliert werden. Eine *Verknüpfung* der Faktoren und deren gegenseitige Beeinflussung ist nicht vorgesehen. Genau diese Verknüpfung der Faktoren kann mit dem in dieser Arbeit vorgeschlagenen Ansatz erreicht werden. So wird der gesamte Business-Prozess während einer definierten Zeit simuliert und immer wieder mit den gleichen Startbedingungen gestartet. Aufgrund dieser Startbedingungen werden dann zufallsbasiert Entscheidungen simuliert. Dies führt zu einem bestimmten Zustand, der selbst wieder weiterführende Entscheidungen zur Folge hat.

In einem späteren Kapitel (3.5.1 Detaillierte Spezifikation des Business-Prozesses) werden der zu simulierende Prozess und alle darin enthaltenen Entscheidungskriterien detailliert spezifiziert.

3.3 Überlegung zur Notwendigkeit eines Modellunternehmens

Wie im vorangehenden Kapitel erarbeitet wurde, bedingt die Anwendung der Monte-Carlo-Methode ein Modell. Bezogen auf die Fragestellung dieser Arbeit bedeutet dies ein Abbild des Entwicklungsprozesses von Software. Software-Entwicklung wird in der heutigen Wirtschaft unterschiedlich betrieben. Zwar gibt es gebräuchliche Standards, doch schlagen diese teilweise unterschiedliche Methoden vor oder widersprechen sich.

Ein in grossen Unternehmen oft eingesetzter Standard ist ITIL[3]. Dieser ursprünglich von der englischen Regierung entworfene Standard deckt die gesamte IT-Prozesslandschaft eines Unternehmens ab (Pengelly 2004), unter anderem auch den Bereich der Entwicklung und Qualitätssicherung, worin auch das Testing enthalten ist. Ein ebenfalls weit verbreiteter Standard ist COBIT[4]. Es nicht ausgeschlossen, dass ein Unternehmen sich aus bestimmten Gründen gar nicht an solchen Standards orientiert und den Software-Entwicklungsprozess komplett individuell gestaltet.

Es kann folglich kein universell gültiger Standard zur Entwicklung und zum Testing von Software angenommen werden. Um die Monte-Carlo-Methode zur Berechnung der durch Software-Nonkonformität entstehenden Kosten in dieser Arbeit dennoch anwenden, implementieren und simulieren zu können, wird ein Modellunternehmen aufgebaut und ein Business-Prozess zur Entwicklung und Wartung der Software entworfen. Es werden Elemente aus beiden oben genannten Standards verwendet.

3.4 Erarbeitung des Business-Prozesses im Modellunternehmen

In diesem Kapitel werden eingangs das Modellunternehmen und seine Funktionsweise beschrieben. Der Text enthält Kleinbuchstaben in Klammern, die den gelben Sternen in der anschliessenden grafischen Darstellung entsprechen.

3.4.1 Beschreibung des Business-Prozesses

Umfeld und Architektur

Das Modellunternehmen bietet Dienstleistungen an. Es existiert eine Softwareabteilung, die für das eigene Unternehmen Software entwickelt, um die im operationellen Geschäft tätigen Mitarbeiter bei ihren Aufgaben zu unterstützen. Die entwickelte Software ist also in einen bestehenden Business-Prozess eingebunden, der eine gewisse Wertschöpfung erarbeitet. Die Architektur der Software ist eine Client/Server-Lösung. Alle Anwender greifen auf eine zentrale Serverinstanz zu.

Der produktive Betrieb (a) wird durch ein Team von Systemadministratoren sichergestellt. Diese sind für den unterbrechungsfreien Betrieb verantwortlich. Im besten Fall läuft das System ohne Fehler (c). Korrekturen am Programm werden durch das Entwicklungsteam vorgenommen.

Mögliche Störungen

Es gibt zwei Szenarien, die den störungsfreien operationellen Betrieb (c) der Software beeinträchtigen können:

- Störungen, die den laufenden Betrieb (a) komplett verunmöglichen (d). Die zentrale Serverinstanz fällt aus. Als Folge können die Benutzer nicht mehr arbeiten.
- Störungen, die den Arbeitsablauf eines einzelnen Benutzers behindern, diesen aber nicht verunmöglichen (b).

---

[3] ITIL ist eine Abkürzung für Information Technology Infrastructure Library.

[4] CobiT (Control Objectives for Information and Related Technology) ist das international anerkannte Framework zur IT-Governance und gliedert die Aufgaben der IT in Prozesse und Control Objectives.

Nachfolgend werden die zwei Arten von Störungen näher beschrieben.

Verfahren bei Störungen der zentralen Serverinstanz
Diese Störungen werden ausgelöst durch schwerwiegende Programmierfehler (z. B. Memory-Leak, Datenbank-Fehlmanipulationen etc.), die das gesamte System zum Erliegen bringen. Ein derartiger Fehler ist sehr kostspielig, da alle Benutzer unmittelbar betroffen sind und ihre Tätigkeiten nicht weiter ausführen können. Das System ist in einem solchen Fall so lange blockiert (d), bis der Fehler gefunden oder eine Umgehungslösung ausgearbeitet wurde. Im besten Fall kann auch ein simpler Restart des Systems helfen. Ein derartiger Fehler hat massive Auswirkungen auf die Wertschöpfung des Prozesses oder anders formuliert, der Fehler und dessen Behebung verursachen Kosten.

Verfahren bei Störungen, die den Arbeitsablauf behindern
Diese Störungen behindern den Arbeitsprozess. Der Benutzer muss mit einer Umgehungslösung arbeiten. Ein Fehler dieser Art hat einen moderaten Einfluss auf die Geschäftstätigkeit. Die Wertschöpfung wird etwas verkleinert, da jede Massnahme zur Umgehung Zeit kostet und dies die Effizienz des betroffenen Benutzers senkt.

Wenn ein derartiger Fehler auftritt (e), so wird dieser zuerst durch das Entwicklungsteam analysiert. Zusammen mit den Verantwortlichen des operationellen Geschäfts wird abgeschätzt, welche Kosten der Fehler verursacht (f). Sind diese kleiner als ein definierter Schwellenwert, so wird der Fehler vorläufig hingenommen und nicht behoben. Der Fehler wird auf eine Merkliste (g) gesetzt. Möglich ist, dass mehrere „kleine“ Fehler auf der Merkliste zusammen einen erheblichen Schaden verursachen, so dass für diese gemeinsam eine Korrektur (h) eingeleitet wird.

Wird bei der Abschätzung hingegen festgestellt, dass der Fehler gravierende Kosten verursacht, so wird eine Korrektur unmittelbar eingeleitet (h). Die Korrektur eines Fehlers benötigt Zeit und hat Kosten zur Folge. Zuerst muss das Entwicklungsteam den Fehler ausfindig machen und korrigieren (h). Die korrigierte Software wird dann den internen Richtlinien folgend ins Testing weitergegeben, um zu überprüfen, ob die Korrektur wirklich greift (k). So kann eine vermeintliche Korrektur unter Umständen noch immer nicht das gewünschte Verhalten des Systems bringen. Dies würde im Testing (k) entdeckt und der Fehler würde zum erneuten Korrigieren zurück an das Entwicklungsteam (h) geleitet.

Anschliessend wird sichergestellt, dass sich die Software auf einem produktionsähnlichen System installieren lässt (m). Dabei kann entdeckt werden, dass die Software nicht installiert werden kann. Ein möglicher Grund dafür kann ein Fehler in der Installationsroutine sein. Es wäre auch denkbar, dass gewisse Komponenten der Software mit anderen bereits installierten Komponenten Inkompatibilitäten aufweisen. In beiden Fällen wird das gesamte Softwarepaket an das Entwicklungsteam zurückgegeben (h).

Schliesslich wird die Software eingeführt. Da dies nicht während des laufenden Betriebs geschehen kann, wird dies zu Rand- und Wochenendzeiten erledigt. Dies hat ausserordentliche Einsätze der Systemadministratoren zur Folge, was ebenfalls Kosten nach sich zieht. Nach einem erfolgreichen Release gibt es eine Reihe von administrativen Tätigkeiten zu erledigen. Beispielsweise soll das ausgebreitete Release dokumentiert werden (n).

3.4.2 Grafische Darstellung des Business-Prozesses

Anmerkung zum verwendeten Darstellungsstandard
Für die grafische Darstellung des eben beschriebenen Prozesses wird der von der „Object Management Group“ (OMG) vorgeschlagene Standard „State Machine Diagram“ [W4] verwendet. Möglich gewesen wäre auch eine Darstellung unter Verwendung der „Business Process Modeling Notation“ (BPMN), ein Darstellungsstandard, der mittlerweile vom gleichen Gremium gewartet wird. Der abgebildete Business-

Prozess wird aber später mit einer State-Machine implementiert (Kapitel „4.3.1 Aufbau“). Somit ist es konsistenter, den Prozess bereits als State-Machine aufzuzeichnen.

Der Business-Prozess im Bild
Es können zwei Teilprozesse identifiziert werden, die parallel und unabhängig voneinander ablaufen.

Der erste Teilprozess (a) implementiert das „daily business“, also den operationellen Betrieb zur Generierung der Wertschöpfung. Dieser Teilprozess läuft infinit.

Der zweite Teilprozess (b) beschreibt den Fall, dass ein Fehler den Arbeitsablauf im operationellen Betrieb behindert. Dieser Prozess wird durch den Teilprozess (a) ausgelöst. Die ablaufenden Prozessschritte (z. B. Fehlerkorrektur, Testing etc.) tangieren den operationellen Betrieb nicht und werden unabhängig davon ausgeführt. Der Prozess kann auf unterschiedliche Arten beendet werden (siehe Endpunkt in der Abbildung). Es ist möglich, dass der zweite Teilprozess nicht läuft; dies wäre vorstellbar, wenn die Software zufriedenstellend läuft und keine weiteren Verbesserungen notwendig sind.

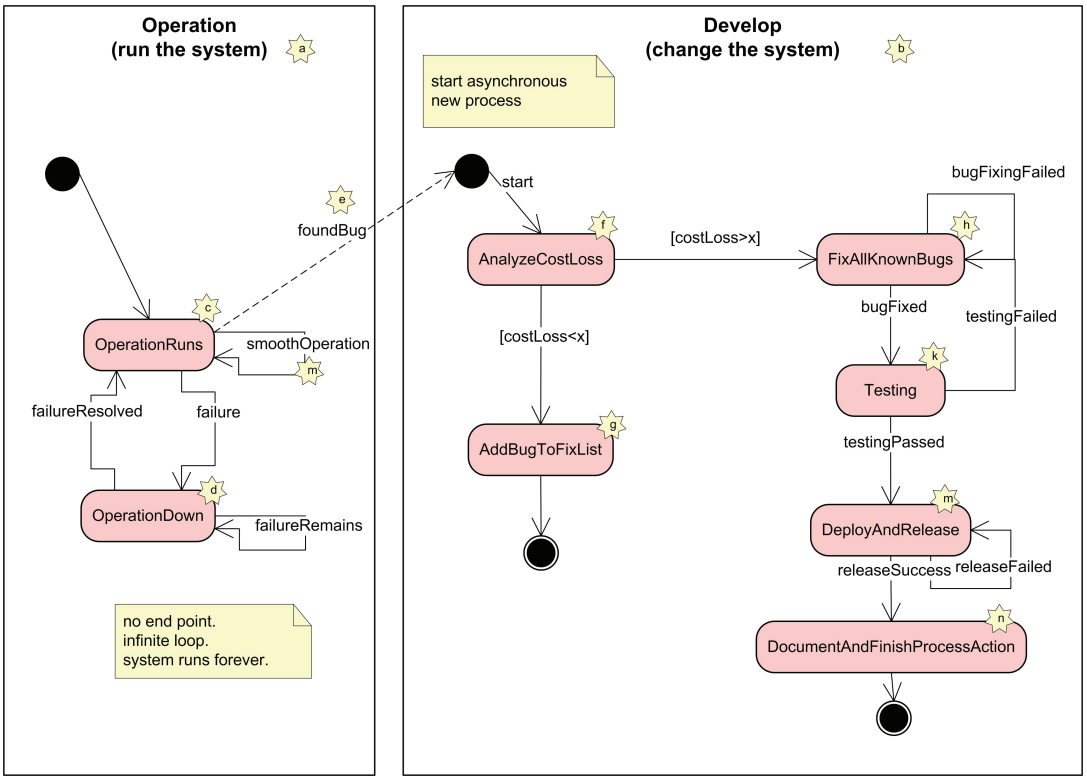


Abbildung 14: Business-Prozess des Modellunternehmens

3.4.3 *Antiscope des Modells*

Jedes Modell hat seine Grenzen, einige Umstände werden bewusst ausgeklammert. Die ausgeschlossenen Faktoren sollen nachfolgend erläutert werden.

Nicht materielle Schäden
Es gibt eine ganze Reihe von möglichen Schäden, die nicht materieller Natur sind. So ist es denkbar, dass ein nicht funktionierendes System sich negativ auf die Motivation der Benutzer im Unternehmen auswirkt. Es kann sogar sein, dass selbst nach einer erfolgreichen Korrektur ein Motivationsdefizit bestehen bleibt und somit die Wertschöpfung des Prozesses nicht mehr gleich hoch ist wie zuvor.

Ebenfalls negative Auswirkungen kann ein nicht funktionierendes System auch auf der Kundenseite haben. Wird der Ausfall vom Kunden wahrgenommen, so kann dies den Kunden verunsichern und künftige Geschäftsbeziehungen beeinträchtigen. Ausgelöst durch einen Softwarefehler kann neben dem einzelnen Kundenschaden gar ein Imageschaden für das betroffene Unternehmen entstehen. Werden beispielsweise durch einen Systemfehler die Börsengeschäfte einer Bank tangiert und dies wird in negativen Pressemitteilungen publik gemacht, so kann grosser immaterieller Schaden entstehen. Solche schwer messbaren Faktoren werden im Modellunternehmen nicht berücksichtigt, obwohl diese unter Umständen einen sehr hohen Einfluss auf die Wertschöpfung haben könnten.

Weitere Störungen des Betriebs
Im aufgestellten Modell werden nur Störungen betrachtet, welche durch die gelieferte Software ausgelöst werden. Vorstellbar sind aber auch Hardware-Ausfälle, physische Störungen wie Stromausfall, Wasserschäden oder mutwillige Manipulationen (Hacker, frustrierte Mitarbeiter, Viren etc.). All diese Arten von Störungen werden nicht betrachtet.

Personelle Umstände
Allfällige personelle Ressourcenengpässe (Krankheit, Ferien etc.), die eine schnelle Behebung eines operationellen Ausfalls verzögern können, sind nicht berücksichtigt. Gleiches gilt für das Entwicklungsteam; es wird angenommen, dass immer Mitarbeiter zur Verfügung stehen.

Mögliche weitere Einflussgrössen
Die obige Auflistung ist nicht abschliessend. Jedes Unternehmen ist vielfältig mit seinem Umfeld vernetzt. Die „Grundkategorien" des St. Galler Management-Modells [W5] zeigen, dass ein Schaden potenzielle Auswirkungen auf viele Anspruchsgruppen haben kann, z. B. Kunde, Kapitalgeber, Lieferanten, Öffentlichkeit, Mitarbeitende und Staat.

3.5 Weitere Prozessaspekte: Zeit, Kosten und Eintrittswahrscheinlichkeit

Bislang lag der Fokus auf den Prozessabläufen im Modellunternehmen und auf der Frage, wie auf Software-Nonkonformität reagiert werden soll. Um das oben beschriebene Modellunternehmen simulieren zu können, ist eine Vervollständigung jedes Prozessschrittes durch Angabe eines genauen Parameters notwendig. Dieses Vorgehen gleicht der „Monte-Carlo-Methode als Erweiterung der Szenarienbetrachtung" von Frey (Kapitel 2.5.4), bei dem allen Budgetposten ein zu erwartender Wert und eine Wahrscheinlichkeitsverteilung hinterlegt wird. Bezogen auf das Modellunternehmen, reicht diese eindimensionale Betrachtung aber nicht. Da der gesamte Business-Prozess simuliert werden soll, werden umfassendere Kennzahlen benötigt. Folgende Aspekte müssen bekannt sein:

- Parameter Zeit: Dieser Parameter definiert, wie lange es dauert, bis ein Prozessschritt abgearbeitet ist. Zum Beispiel: Wie lange wird es dauern, bis ein Fehler und seine finanziellen Auswirkungen (f) analysiert worden sind?
- Parameter Kosten: Er definiert, wie viele Kosten für einen entsprechenden Prozessschritt zu erwarten sind und auf welches Buchungskonto sie verbucht werden.
- Parameter Eintrittswahrscheinlichkeit: Diese Kennzahl legt fest, aufgrund welcher Wahrscheinlichkeiten ein Ereignis eintritt.

3.5.1 Detaillierte Spezifikation des Business-Prozesses

Die nachfolgende Tabelle präzisiert den Business-Prozess des Modellunternehmens und legt fest, welche Werte für die Kennzahlen angenommen werden. Zusammen mit dem vorangehenden Kapitel kann die Tabelle als Spezifikation für die spätere Implementierung verstanden werden.

Jede Kennzahl wird in der Tabelle als Parameter mit einem eindeutigen Namen, der „Param-ID", dargestellt (z. B. „G002"). Diese „Param-ID" findet sich ebenfalls im Quellcode (hauptsächlich in der Klasse

com.x8ing.mc.Configuration) und in der erstellten Demo-Webapplikation [W1] (siehe Kapitel „9.2 Diplomarbeit-Webseite und Demo-Webapplikation"). Verweise innerhalb der Tabelle auf bestimmte „Prozessschritte" beziehen sich immer auf die Abbildung 14 aus dem vorangehenden Kapitel „3.4.2 Grafische Darstellung des Business-Prozesses".

In der Spalte „Wert/Konto" finden sich zwei Angaben: einerseits der in dieser Arbeit verwendete Wert, wie er für die Simulation und die daraus gewonnenen Resultate verwendet wurde. Dieser Wert ist ein realistischer Schätzwert. Andererseits zeigt die Spalte in eckigen Klammern das „Konto", das einer vereinfachten Buchhaltung entspricht, die während der Simulation erstellt wird. Ist der beschriebene Parameter kostenrelevant, so werden die Kosten auf das entsprechende Konto gebucht. Geführt werden folgende Konten:

- Prod Production
- Dev Development
- Dep Deployment
- Ana Bug-Analyzing
- Test Testing

Param-ID	*Beschreibung*	*Anmerkungen*	*Wert/ [Konto]*
Parameter zu Kosten und Wertschöpfung (in CHF)			*[CHF]*
G001	Initial-Wert der Software-Nonkonformitätskosten (z. B. verursacht durch verminderte Effizienz).	Dieser Wert kann im Laufe einer Simulation durch Korrekturen (Bug-Fixes) der Software vermindert werden.	500
G002	Initial geplante Wertschöpfung (pro Tag) des Prozesses; dies entspricht den geplanten Einnahmen. Die reale Wertschöpfung wird um die Software-Nonkonformitätskosten vermindert.	Die Wertschöpfung wird durch den Prozessschritt „OperationRuns" generiert. Durch erfolgreiche Releases der Software wird die Wertschöpfung im Laufe der Simulation verbessert.	2100 [Prod]
G003	Schwellenwert, den die Software-Nonkonformitätskosten überschreiten müssen, damit die Entscheidung gefällt wird, die Fehler zu beheben und ein neues Release der Software zu liefern.	Diese Entscheidung wird im Prozessschritt „AnalyzeCostLoss" getroffen. Es ist aus unternehmerischer Sicht nicht sinnvoll, den Softwareentwicklungsprozess zu starten, solange nur Fehler mit insgesamt vernachlässigbaren Auswirkungen bekannt sind.	300
G004	Kosten (pro Tag), die durch Störungen der zentralen Serverinstanz entstehen.	Dieser Wert muss nicht der geplanten Wertschöpfung (G002) entsprechen. Einerseits kann der Wert durch mögliche Folgeschäden erhöht werden. Andererseits ist es vorstellbar, dass der Wert kleiner ist, weil ein Ausfall mit Nacharbeiten (Überzeit) wieder kompensiert werden kann. Diese entstandenen Kosten fallen im Prozessschritt „Operation-Down" an.	1000 [Prod]

G010	Kosten für die Korrektur eines Fehlers werden durch eine Gauss'sche Normalverteilung modelliert. Für die mathematische Formulierung werden mehre Parameter benötigt: „G010", „G011", „G012".	Dieser Parameter beschreibt die Standardabweichung[5] der Kostenverteilung. Die Verteilung wird im Prozessschritt „FixAllKnownBugs" verwendet.	2 [Dev]
G011	Minimale Kosten für die Korrektur eines Fehlers innerhalb des von „G010" definierten Intervalls.	(siehe G010)	100 [Dev]
G012	Maximale Kosten für das Beheben eines Fehlers innerhalb des von G010 definierten Intervalls.	(siehe G010)	500 [Dev]
F001	Fixkosten für das Testing der auszuliefernden Software bei einem neuen Release.	Bei einem neuen Release werden immer alle Funktionen (auch bestehende) neu getestet (Regressionstest). Der zugehörige Prozessschritt heisst „Testing".	2500 [Test]
F002	Fixkosten, die nach erfolgter Einführung einer neuen Software für Dokumentations- und Projektabschlusstätigkeiten anfallen.	Diese Kosten fallen beim Prozessschritt „DocumentAndFinishProcess" an.	380 [Dep]
F003	Fixkosten, die bei der Installation und Ausbreitung einer neuen Software anfallen. Sie umfassen Arbeiten, die vom Admin- und Support-Team zu Randzeiten ausgeführt werden müssen, um den laufenden Betrieb nicht zu tangieren.	Diese Kosten fallen beim Prozessschritt „DeployAndRelease" an.	2400 [Dep]
F004	Fixkosten, die für die Bestimmung der Software-Nonkonformitätskosten anfallen. Die Vertreter des operationellen Geschäfts und des Entwicklungsteams analysieren zusammen die Folgekosten eines Fehlers. Diese Tätigkeit löst Kosten aus.	Diese Kosten fallen beim Prozessschritt „AnalyzeCostLoss" an.	200 [Ana]
F005	Fixkosten administrativer Art, die entstehen, wenn ein entdeckter Fehler auf die Merkliste der später zu korrigierenden Fehler gesetzt wird.	Diese Kosten fallen beim Prozessschritt „AddBugToFixList" an.	170 [Dev]
Parameter zu Eintrittswahrscheinlichkeiten (in Prozent)			*[%]*
P001	Wahrscheinlichkeit einer Störung der zentralen Serverinstanz.	Dieser Parameter wird im Prozessschritt „OperationRuns" benutzt.	20
P002	Wahrscheinlichkeit, dass eine Störung der zentralen Serverinstanz durch das Admin- und Supportteam innerhalb eines Tages erfolgreich behoben werden kann.	Wenn das Team das Produktionssystem nicht gleich am Tag des Ausfalls reparieren kann, wird es dies am nächsten Tag mit exakt gleicher Erfolgswahrscheinlichkeit erneut versuchen. Dieser Parameter wird im Prozessschritt „OperationDown" benutzt.	90

[5] Zur Erinnerung aus der Statistik: nDev: 1=66% Intervall. 2=95%. 3=99.7%. 4=99.9%.

P003	Wahrscheinlichkeit, dass innerhalb eines Tages eine Störung, die den Arbeitsablauf behindert, entdeckt wird.	Ein Ablauffehler (Bug) ist im Gegensatz zu einem Totalausfall nur behindernd und mindert die Effizienz. Dieser Parameter wird im Prozessschritt „OperationRuns“ benutzt, um möglicherweise den Teilprozess „Develop“ zu starten.	60
P004	Wahrscheinlichkeit, dass das Entwicklungsteam eine Fehlerkorrektur innerhalb eines Tages erfolgreich ausführen kann.	Dieser Parameter beschreibt implizit die Effizienz des Entwicklungsteams und wird im Prozessschritt „FixAllKnownBugs“ verwendet.	75
P005	Wahrscheinlichkeit, dass im Testing eine Korrektur eines Fehlers erfolgreich verifiziert werden kann.	Dieser Parameter beschreibt implizit die Arbeitsqualität des Entwicklungsteams. Der Parameter wird im Prozessschritt „Testing“ verwendet.	80
P006	Wahrscheinlichkeit, dass ein geplantes Release vom Admin- und Support-Team auf Anhieb erfolgreich eingespielt werden kann.	Wenn die Ausbreitung eines Releases scheitert, wird der gleiche Vorgang mit gleich bleibender Erfolgswahrscheinlichkeit am folgenden Tag wiederholt. Der Parameter wird im Prozessschritt „DeployAndRelease“ verwendet.	70
Parameter zu technischen Aspekten			
T001	Technische Variable, die festlegt, für wie viele Tage das Unternehmen mittels Monte-Carlo-Methode simuliert werden soll.	Das Unternehmen und all seine Prozesse werden jeweils für eine bestimmte Zeit simuliert.	120
T002	Technische Variable, welche die Anzahl der Gesamtdurchläufe, die in einer Simulation durchgeführt werden sollen, festlegt, das heisst, wie viele Male das Modellunternehmen während der definierten Tage (T001) durchgerechnet wird. Die Durchführung erfolgt dabei jeweils mit veränderten Zufallsbedingungen und wird daher zu einem unterschiedlichen Endzustand führen.	Je höher dieser Parameter, desto genauer und zuverlässiger wird das erhaltene Resultat ausfallen. Dies hat jedoch eine längere Simulationsdauer zur Folge. (Vergleiche dazu die Approximation von PI, welche die Erhöhung der Genauigkeit während der Durchläufe zeigt: Kapitel 2.4.2.)	1000
T003	Während eines Simulationsdurchlaufs werden automatisch die benötigten Daten für die spätere Resultatauswertung gesammelt. Zusätzlich zu diesen können weitere detaillierte Protokolle aller Abläufe im Modellunternehmen erstellt werden. Diese Aufzeichnungen können der Plausibilitätskontrolle und Verifikation der Implementierung dienlich sein. Diese technische Variable definiert, wie viele detaillierte Protokolle gesammelt werden sollen. Die Auswahl geschieht zufällig.	Bei einer Simulation mit mehreren Durchläufen (siehe T002) fallen sehr viele Daten an. Aus Gründen der limitierten Speicherkapazität können aber nicht alle aufgezeichnet und aufbewahrt werden.	3

Abbildung 15: Detaillierte Aufstellung der Parameter im Business-Prozess

3.6 Konsequenzen des Monte-Carlo-Ansatzes: Diskretisierung im Zeitbereich

Die Simulation des Modellunternehmens soll in konstanten Zeitabschnitten geschehen. Ein Prozessschritt ist immer gleichzusetzen mit dem Voranschreiten der Simulationszeit um einen Tag. Dieser Ansatz wird von Ledin (Ledin 2001, Kapitel „2.2 Dynamic Systems“) genauer beschrieben. Innerhalb eines Prozessschrittes können durchaus mehrere Tätigkeiten erfolgen, aber die Entscheidung, welcher Prozessschritt als Nächstes ausgeführt wird, bedingt das Voranschreiten der Simulation um einen Zeitabschnitt.

Beispiel
Am Modellunternehmen bedeutet dies Folgendes: Es kann vorkommen, dass im Prozessschritt „FixAllKnownBugs“ mehrere Fehler korrigiert werden, also mehrere Tätigkeiten ablaufen. Der Entscheid, ob alle Fehler erfolgreich korrigiert werden konnten oder nicht (entspricht der Condition „bugFixingFailed“ bzw. „bugFixed“), geschieht aber immer erst zu Beginn des nächsten Simulationsschrittes, also 24 Stunden nachdem mit der Korrektur des ersten Fehlers begonnen wurde.

Um die Genauigkeit der Simulation zu erhöhen und detaillierter auf Ergebnisse reagieren zu können, wäre es vorstellbar, das Zeitintervall zu verkürzen (z. B. 15-Minuten-Rhythmus). Da der Business-Prozess des Modellunternehmens eher träge ist, scheint ein Simulationsschritt von einem Tag aber angemessen.

Die Simulation des Modellunternehmens erfolgt immer über eine bestimmte Zeitdauer (N) in mehreren Durchgängen (D). Wenn eine Anzahl Durchgänge simuliert wird, dann resultiert daraus die Anzahl „D“ an Endzuständen des Unternehmens. Die Ausgangssituation ist dabei immer die gleiche, die Verläufe sind aber verschieden. Würde bei jedem Simulationsdurchgang der Simulationsschritt „X“ herausgegriffen, so wäre dieser mit hoher Wahrscheinlichkeit immer ein anderer. Dabei sind folgende Punkte zu beachten:

Veränderung der Software-Nonkonformitätskosten (G001) und der Wertschöpfung des Prozesses (G002)
Wie in der Spezifikation beschrieben (Kapitel „3.5.1 Detaillierte Spezifikation des Business-Prozesses“), können die Software-Nonkonformitätskosten durch Ausbreitung eines neuen Software-Releases verkleinert werden. Dies hat zur Folge, dass die Wertschöpfung des Prozesses (G002) genau um die Veränderung des Wertes „G001“ ansteigt. Die Variablen verändern sich also während eines Simulationsdurchlaufes, wobei „G001“ den Parameter „G002“ beeinflusst. Eine Veränderung tritt z. B. dann auf, wenn der Entscheid für ein neues Release der Software gefällt wurde und dieses geliefert wird. Für einen beliebig herausgegriffenen Simulationsschritt „X“ werden die Parameter „G001“ und „G002“ in den meisten Fällen unterschiedliche Werte aufweisen.

Ausgeführter Prozessschritt
Ähnlich verhält es sich bei einem ausgeführten Prozessschritt. Dieser wird bei einem herausgegriffenen Simulationsschritt „X“ in den meisten Fällen ein unterschiedlicher sein.

Innerer Zustand des Unternehmens
Der aktuelle Zustand des Unternehmens wird durch eine Reihe von weiteren internen Variablen abgebildet. Dieser Zustand wird in der Klasse „BusinessContext“ gehalten und wird später im Kapitel „4.4 Design der applikatorischen Monte-Carlo-Komponente und des Modellunternehmens“ genauer erklärt. Unter anderem enthält dieser Zustand Angaben über alle bekannten Software-Fehler, den Zustand der Produktion (Störung oder fehlerfreier Betrieb), die laufende Buchhaltung etc. Auch hier würde ein herausgegriffener Zustand „X“ unterschiedliche Werte für all diese Variablen aufzeigen.

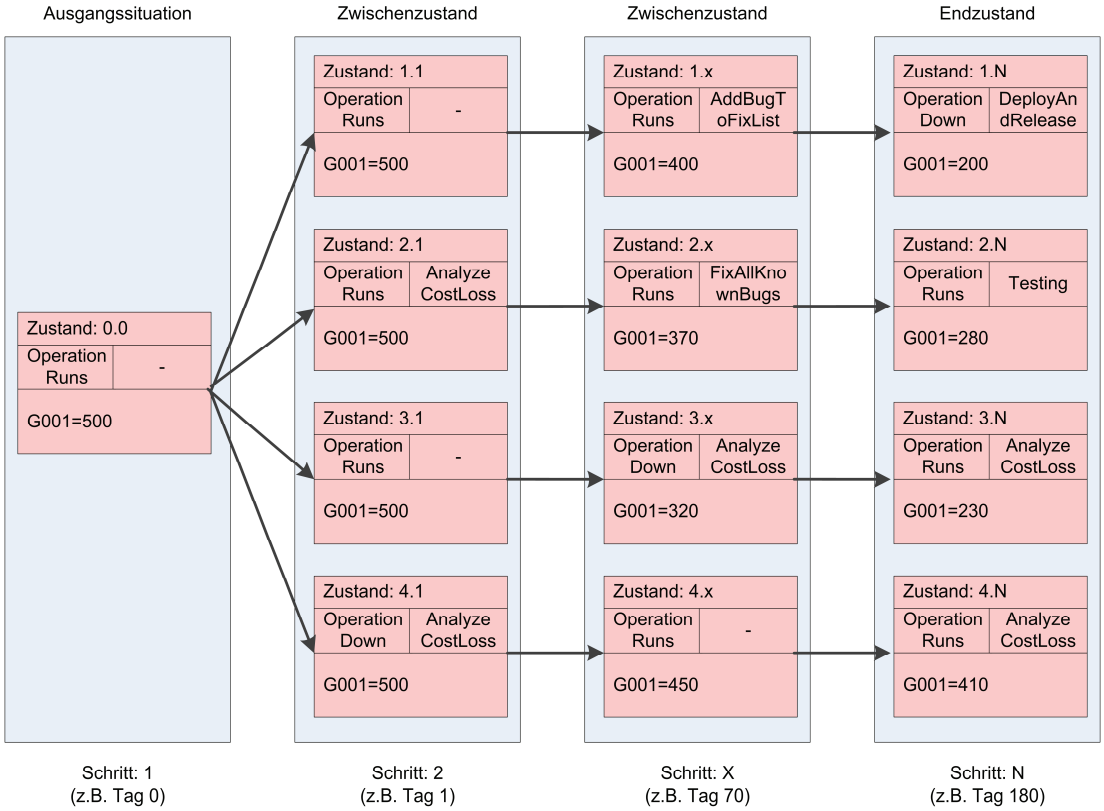


Abbildung 16: Visualisierung von vier Simulationsdurchläufen

Anmerkung

Die erstellte Demo-Applikation [W1] kann dazu genutzt werden, einen Eindruck über die verschiedenen Verlaufsmöglichkeiten des Modellunternehmens während der Simulationsdurchläufe zu gewinnen. Mittels des erstellten Protokolls (Logbooks) (siehe Anhang „9.2.3 Resultat Detailsicht: “) lässt sich der Verlauf gut nachvollziehen. Ein Auszug eines solchen Protokolls findet sich ebenfalls im Anhang (Kapitel „9.3 Beispiel eines Simulationsdurchlaufes“).

4. IMPLEMENTIERUNG DER MONTE-CARLO-METHODE

4.1 Überlegungen zum Einsatz von „Crystalball“

Zur Implementierung von Simulationen empfiehlt Frey (Frey 2001) eine Software namens „Crystalball“. Diese wird von „Desioneering Inc“ hergestellt und vertrieben [W6]. Erst kürzlich wurde das Unternehmen vom Softwarehaus Oracle [W7] übernommen. Das Produkt wird aber als eigenständige Marke weitergeführt. „Crystalball“ ist als Excel Add-in implementiert. Mittels Excel-Tabellen werden verschiedene Parameter definiert. Jedem Parameter wird eine vordefinierte Wahrscheinlichkeitsverteilung hinterlegt. Dies dient der Software anschliessend als Basis der Simulation. Es stehen vielfältige Möglichkeiten zur Auswertung der gewonnenen Daten zur Verfügung.

In der Standard-Variante fehlt die Möglichkeit, den Simulationsprozess den eigenen Bedürfnissen anzupassen. Die bei dieser Arbeit vorliegende Anforderung, einen gesamten Business-Prozess zu simulieren, wäre so nicht umsetzbar. Erst in den professionellen Varianten der Software („Crystal Ball Professional“ und „Crystal Ball Premium“) existiert eine Entwicklungsschnittstelle, welche via VisualBasic[6] oder einer anderen OLE2[7]-fähigen Programmiersprache angesprochen werden kann. Die Software ist allerdings relativ teuer. Die Professional Edition kostet 1'402 USD und die Premium Edition 2'217 USD (Stand 9.2007). Die vergünstigte „Academic Edition“ für Studenten ist für nichtamerikanische Studenten schwer zu beschaffen und umfasst nur die Standard Edition, welche keine Entwicklungsschnittstelle bietet und somit hier nicht eingesetzt werden könnte.

Da es ein Ziel dieser Arbeit ist es, einen Monte-Carlo-Simulator eigenständig zu implementieren, wird „Crystalball“ nicht eingesetzt. So kann besser abschätzt werden, wie viel Aufwand für die Umsetzung benötigt wird und welche Probleme dabei auftreten können. Der Einsatz von „Crystalball“ in einem realen Projekt wird im Diskussionsteil nochmals analysiert (siehe Kapitel „6.1 Zur Umsetzbarkeit der Monte-Carlo-Methode in Java-Code“).

4.2 Eigenbau: Grundsätzliches zur Implementierung

Die im Eigenbau umgesetzte Lösung genügt den folgenden Grundsätzen:

- Das Programm basiert auf Java (J2SE 1.4).
- Alle verwendeten Komponenten sind Opensource (oder ähnliche Lizenzen).
- Als Entwicklungsplattform wird Eclipse[8] verwendet.
- Eine Webseite [W1] mit Demo-Webapplikation wird erstellt. Diese Applikation ist webbasiert und im Internet öffentlich abrufbar. Alle Ressourcen des Projekts (Source, Dokumentation und Build) sind ebenfalls auf der Seite publiziert.

Das Design der Applikation wird in verschiedene Komponenten aufgeteilt (siehe Abbildung 17). Dabei sind zwei Hauptkomponenten wichtig: das State-Machine-Framework (siehe nachfolgendes Kapitel), als generische Basiskomponente zur flexiblen Modellierung einer State-Machine, sowie – auf der Ersteren aufbauend – die applikatorische Monte-Carlo-Komponente (siehe Kapitel „4.4 Design der applikatorischen Monte-Carlo-Komponente und des Modellunternehmens “), welche die Basiskomponente be-

---

[6] Visual Basic (Abk. VB) ist eine proprietäre objektorientierte Programmiersprache von Microsoft.

[7] Object Linking and Embedding (OLE, engl. Objekt-Verknüpfung und -Einbettung) ist ein von Microsoft entwickeltes Objektsystem und Protokoll, das die Zusammenarbeit unterschiedlicher (OLE-fähiger) Applikationen und damit die Erstellung heterogener Verbunddokumente ermöglichen soll.

[8] Eclipse ist ein Open-Source-Framework zur Entwicklung von Software nahezu beliebiger Art. http://www.eclipse.org/.

nutzt, um den Business-Prozess mittels der State-Machine abzubilden. Die Demo-Webapplikation (Kapitel „4.7 Design der Demo-Webapplikation“) und das Konsolenprogramm zur Generierung der Ergebnisse, die für diese Arbeit benötigt wurden, sind Benützer der Hauptkomponenten. Bei der Schichtung der Komponenten gilt, dass jede darunterliegende Komponente keine Abhängigkeit von der nächsthöheren Schicht aufweist, jede höhere Schicht jedoch die darunterliegende benutzt.

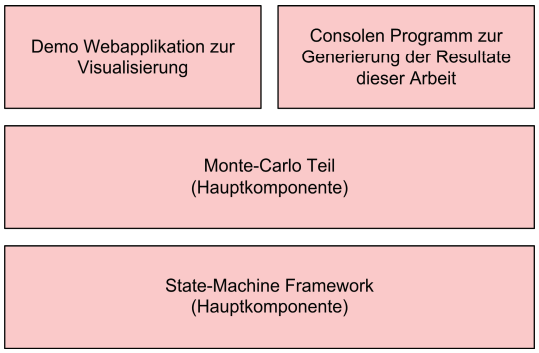


Abbildung 17: Schichtung der Applikation

4.3 Design des State-Machine-Framework

4.3.1 Aufbau

Das Modellunternehmen besteht aus einer Reihe miteinander verknüpfter Prozesse (siehe Abbildung 14). Eine solche Struktur ist als State-Machine modellierbar. Das entworfene State-Machine-Framework kann eine State-Maschine abbilden. Die Implementierung ist dabei generisch gehalten und bietet dem Entwickler viel Freiraum und mehr Möglichkeiten, als für diese Arbeit benötigt werden. So ist die Abbildung des Business-Prozesses nur eine Möglichkeit, das Framework einzusetzen. Aus diesem Grund werden die State-Machine und ihre Grundelemente in der für State-Machines üblichen Terminologie erklärt. Die Übersetzung der Begriffe auf den Anwendungsfall des „Business-Prozesses“ erfolgt laufend im Text.

Grundelemente der State-Machine

Beim gewählten Design sind folgende Grundelemente beteiligt (siehe Abbildung 18):

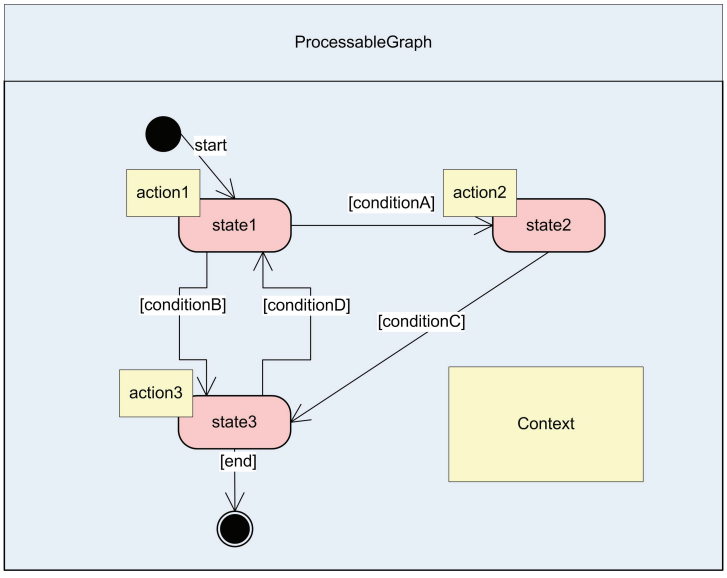


Abbildung 18: Beteiligte Elemente der State-Machine

Ein Graph ist eine Ansammlung einer beliebigen Anzahl von States (Zuständen), die mit Transitions (Verbindungen) miteinander verknüpft sind. Ein Kontroller arbeitet diesen Graphen ab, indem er sich von State zu State bewegt und beim Erreichen eines State die dazugehörige Action aufruft. Eine Transition kann nur traversiert werden, wenn ihre zugrunde liegende Condition (Bedingung) erfüllt (wahr) ist.

ProcessableGraph
Die Klasse „com.x8ing.lsm4j.state.ProcessableGraph“ stellt den Haupteinstiegspunkt dar, um einen Graphen und somit ein Netz von States aufzubauen und zu prozessieren.

Technisch erbt die Klasse „„ProcessableGraph“ von der Elternklasse „StaticGraph“. Die Elternklasse kapselt dabei die Funktionalität für die Struktur Graphen. Dies umfasst die einzelnen States und deren Verbindungen über die Transitions. Die Klasse „ProcessableGraph“ erweitert diese Funktionalität um die dynamischen Aspekte. Sie bietet Kontroller-Funktionalität zum Prozessieren des Graphen. Der Kontroller evaluiert ausgehend von einem State alle weiterführenden Conditions. Wenn eine Condition gefunden wird, die „wahr“ zurückgibt, so wird diesem Weg gefolgt, um zum nächsten State zu gelangen. Beim nächsten State ankommend, wird zuerst die zugehörige Action aufgerufen. Für den Ausnahmefall, dass alle Conditions „falsch“ zurückgeben, wird vom Kontroller eine „Exception“ vom Typ „NoMatchingTransitionConditionFoundException“ geworfen und die Ausführung wird unterbrochen. Es gibt verschiedene Möglichkeiten, einen Graph zu prozessieren, entweder schrittweise oder automatisch, bis ein Endzustand gefunden wird. Weiter können dem Kontroller, dem Observer Pattern (Gamma 1995, S. 293) folgend, Informationsobjekte (Klasse „GraphListener“) registriert werden. Diese werden dann über die Vorgänge im Graphen mittels „Events“ informiert.

Übertragen auf den Business-Prozess entspricht der „ProcessableGraph“ dem Ablaufplan des Business-Prozesses, d. h., er hält fest, wie die einzelnen Prozessschritte miteinander verbunden sind. Die Klasse wird daher benutzt, um die Struktur des Business-Prozesses zu definieren. Ebenfalls wird der eigentliche Ablauf des Business-Prozesses von dieser Klasse kontrolliert.

ProcessableState
Ein State ist ein Knoten auf einem Graphen. Er enthält eine Referenz auf die auszuführende Action (siehe unten). Ein State kann als Endzustand definiert werden. Trifft der Kontroller auf einen solchen Endzustand, wird die Prozessierung ordnungsgemäss beendet. Der State ist als Klasse mit Namen „com.x8ing.lsm4j.state.ProcessableState“ realisiert und erbt von „com.x8ing.lsm4j.state.StaticState“.

Der State hat keine direkte Entsprechung im Business-Prozess. Er kann im weitesten Sinne als „Phase“ im Business-Prozess gesehen werden, z. B. „Testing“. Er darf dabei aber nicht mit den Tätigkeiten verwechselt werden, die in dieser Phase des Prozesses ausgeführt werden; dies entspräche der Action (siehe unten).

Condition
Eine Condition ist eine Bedingung, die erfüllt sein muss, damit der Kontroller die Transition von einem Zustand in den nächsten verfolgt. Nur wenn die Bedingung wahr (true) ist, folgt der Kontroller dem Weg zum nächsten State. Wenn in der obigen Grafik also die Bedingung mit dem Namen „conditionA“ nicht erfüllt ist, hingegen „conditionB“ sich als wahr herausstellt, dann wird der Kontroller vom „state1“ zum „state3“ wechseln. Eine Condition kann dabei vom Kontext (siehe unten) abhängig sein. Die Condition ist als Interface implementiert (com.x8ing.lsm4j.Condition). Dabei ist die Methode „conditionTrue(StateContext currentContext)“ relevant, die abhängig vom Kontext entscheidet, ob die Condition wahr oder falsch ist. Die Methode wird vom Kontroller aufgerufen, um den weiteren Verlauf der Prozessierung zu bestimmen.

Die „Condition“ entspricht im Business-Prozess einer Entscheidung, die über den weiteren Verlauf des Prozesses bestimmt. Am Beispiel des Modellunternehmens wird nach erfolgtem Testing die „Entscheidung“ gefällt, ob das Testing erfolgreich war oder nicht. Abhängig von dieser Entscheidung folgt entwe-

der der Prozessschritt zur Einführung der Software, oder die Software muss vom Entwicklungsteam nachgebessert werden.

Action
Die Action „com.x8ing.lsm4j.Action“ definiert, was beim Erreichen eines State ausgeführt werden soll. Sie ist daher immer an einen State gebunden. Sobald ein bestimmter State erreicht wird, ruft der Kontroller automatisch die zugehörige Action auf. Dies entspricht dem „Strategy-Pattern“ (Gamma 1995, S. 315). Der Kontroller ruft dabei die Arbeitsmethode „execute“ der Action auf und übergibt dabei einige Informationen, unter anderem den aktuellen Kontext (siehe unten). Somit ist es der Action möglich, auf den aktuellen Zustand des Automaten zu reagieren. Die Action ist als Interface umgesetzt.

Bezogen auf den Business-Prozess stellt die „Action“ alle Tätigkeiten dar, die in einem bestimmten Prozessschritt ausgeführt werden sollen. Beim Modellunternehmen entspricht dies beim Testing, dem Durchführen des Software-Tests, dem Protokollieren der Fehler und dem Verrechnen der entstandenen Kosten für die Buchhaltung.

StateContext
Der „StateContext“, oder kurz Kontext, ist ein zentraler, von mehreren Elementen gemeinsam benutzter Sammelbehälter für Informationen aller Art, der State-Machine selbst und ihrer Umwelt. Der Kontroller stellt dabei sicher, dass aufgerufene Actions und Conditions immer eine aktuelle Referenz auf den Kontext zur Verfügung gestellt bekommen. Technisch betrachtet ist der „StateContext“ ein einfaches, leeres Interface (com.x8ing.lsm4j.StateContext), das keine Methoden vorgibt. Jede konkrete Implementierung einer Action oder Condition muss selbst eine Typenumwandlung (engl. cast) des „StateContext“ auf die gewünschte Implementierung vornehmen.

Angewendet auf einen Business-Prozess, könnte der „StateContext“ beispielsweise Informationen halten, die dem Prozess als Eingabe-Werte dienen. Im Modellunternehmen hält der Kontext unter anderem folgende Informationen: alle bekannten Software-Fehler (Bug-Liste), die Buchhaltung, den Zustand der Produktion (Störung oder fehlerfreier Betrieb), das aktuelle Simulationsdatum etc.

StateMetaInformation
Dieses Interface hat eine ähnliche Funktion wie der „StateContext“. Es bietet einen universellen Container zur Haltung von Informationen an. Im Gegensatz zum „StateContext“ können diese Informationen aber nicht von allen Actions und Conditions abgerufen werden. Die „StateMetaInformationen“ gehören immer genau zu einem State und stehen somit nur den Actions und Conditions zur Verfügung, die mit dem entsprechenden State direkt verknüpft sind.

4.3.2 Klassen- und Package-Diagramme

Überblick der Packages

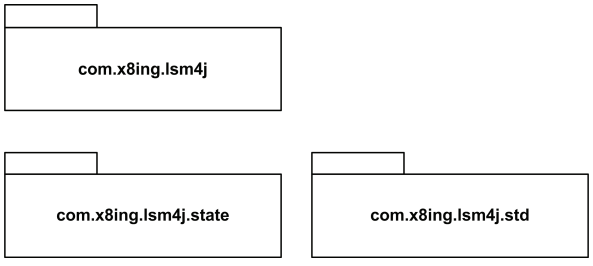


Abbildung 19: Package-Übersicht der State-Machine-Framework-Komponente

Die Abbildung 19 zeigt wie die Komponenten des State-Machine-Framework in drei Packages aufgeteilt sind. Das Package „com.x8ing.lsm4j“ (Abbildung 20) beinhaltet dabei Interfaces, die vom Entwickler

implementiert werden müssen. Im Gegensatz dazu werden Klassen aus dem Package „com.x8ing.lsm4j.state“ (Abbildung 22) vom Framework zur Verfügung gestellt und sollen nicht erweitert, sondern nur benutzt werden, um z. B. einen Graphen aufzubauen. Das Package „com.x8ing.lsm4j.std“ (Abbildung 21) bietet einige vorgefertigte Implementierungen für häufig gebrauchte Funktionen. Nachfolgend wird der Inhalt jedes dieser Packages mittels Klassendiagramm visualisiert, dabei wird nicht auf jede Klasse detailliert eingegangen.

Klassendiagramm des Package „com.x8ing.lsm4j“

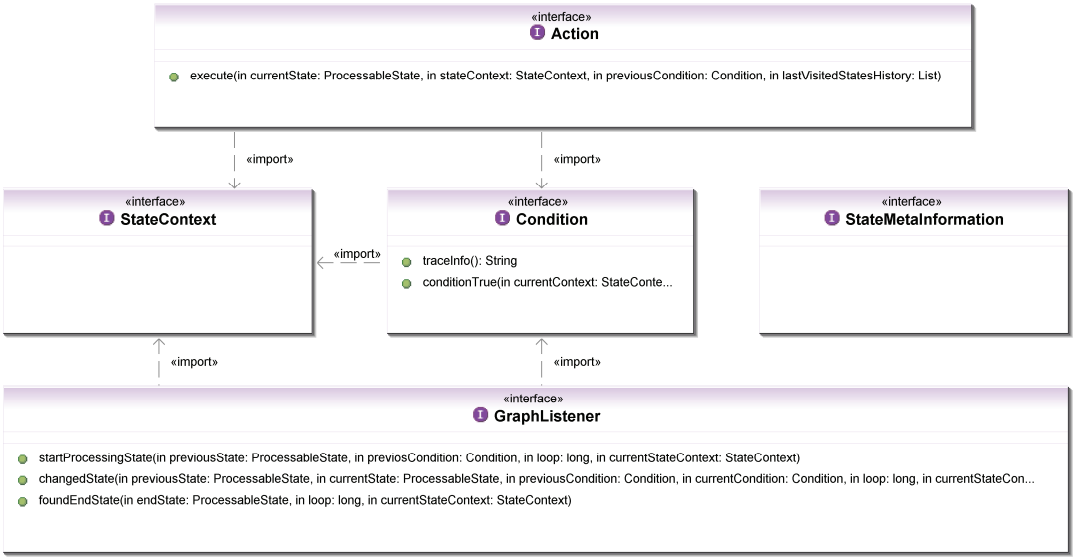


Abbildung 20: Basis-Interfaces des State-Machine-Framework

Klassendiagramm des Package „com.x8ing.lsm4j.std“

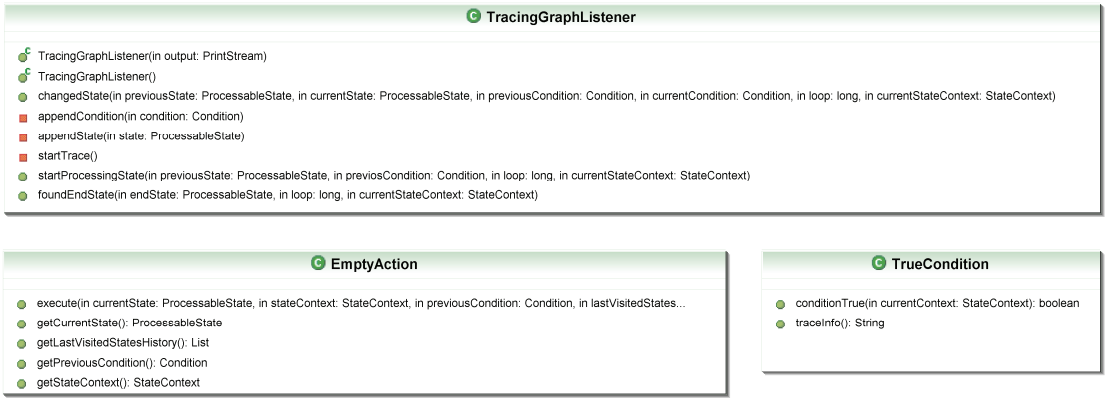


Abbildung 21: Hilfsklassen des State-Machine-Framework

Klassendiagramm des Package „com.x8ing.lsm4j.state“

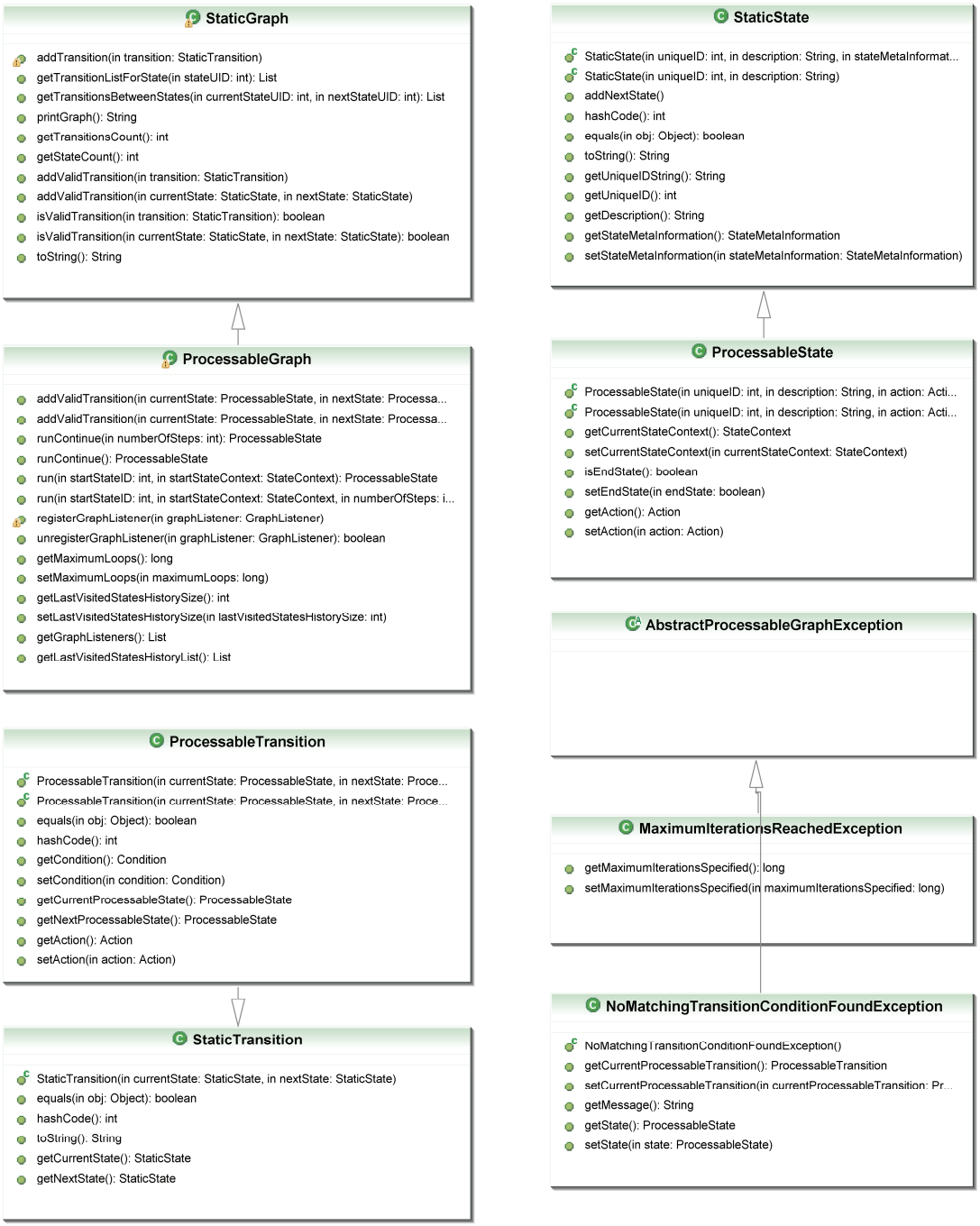


Abbildung 22: Klassen, die vom State-Machine-Framework zur Verfügung gestellt werden

Anmerkung zur Offenlegung des Quellcodes

Das State-Machine-Framework wurde im Zuge der Implementierung für diese Diplomarbeit erstellt. Es ist geplant, den Quellcode unter dem Projektnamen LSM4J (Lightweight State Machine Framework For Java) offenzulegen. Das Open-Source-Portal SourceForge[9] hat LSM4J bereits als Projekt akzeptiert. In den nächsten Monaten wird der Inhalt der SourceForge-Seite [W8] mit Inhalten versehen werden.

[9] Das Webportal „SourceForge.net“ dient zur Entwicklung von Open-Source-Programmen und wird von vielen Software-Entwicklern zur Verwaltung ihrer Projekte genutzt.

4.4 Design der applikatorischen Monte-Carlo-Komponente und des Modellunternehmens

Zur Simulation des Modellunternehmens wird dessen Abbild als Software benötigt. Als Grundlage dient das vorher beschriebene State-Machine-Framework. Darauf aufbauend sollen nun das Modellunternehmen und sein Business-Prozess implementiert werden. Dieses Modell liefert die Basis der Simulation mittels der Monte-Carlo-Methode.

4.4.1 Design

Das State-Machine-Framework gibt ein bestimmtes Design vor und zwingt den Entwickler, eine vorgegebene Struktur einzuhalten. Dies hat den Vorteil, dass der erstellte Code eine klare Struktur aufweist und somit überschaubar bleibt. In den Package- und Klassendiagrammen werden konkrete Implementierungen der vorher eingeführten „Actions“ und „Conditions“ wiederzuerkennen sein.

Überblick der Packages
Die Implementierung des Business-Prozesses ist in mehrere Packages aufgeteilt. Die Abbildung 23 zeigt einen Grobüberblick über die erstellten Packages. Nachfolgend wird auf jedes dieser Packages detaillierter eingegangen.

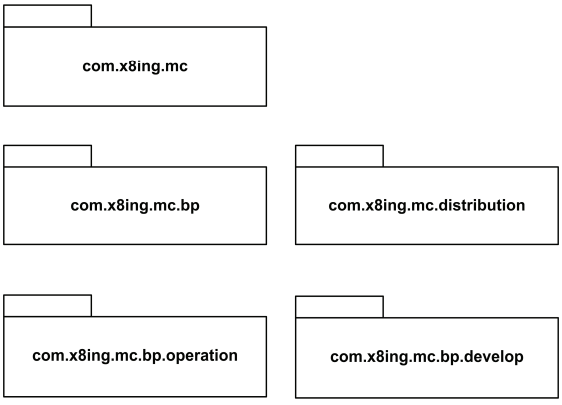


Abbildung 23: Package-Übersicht der Monte-Carlo-Komponente

Package „com.x8ing.mc“
In diesem Package (Klassendiagramm siehe Abbildung 24) befinden sich die Kernkomponenten der Simulation.

Eine zentrale Rolle übernimmt der „MonteCarloControler“, der als Steuereinheit die gesamte Simulation kontrolliert. Er startet die Simulation und hält das Ergebnis des Durchlaufes für die spätere Auswertung fest. Ein Simulationsdurchlauf wiederum umfasst eine komplette Simulation des Unternehmens während einer definierten Zeitperiode (von z. B. 180 Tagen). Der „MonteCarloControler“ darf nicht mit dem Kontroller des State-Machine-Framework verwechselt werden, der für die Steuerung beim Prozessieren des Graphen, also der State-Machine, verantwortlich ist. Diese Funktionalität wird komplett von der vorher beschriebenen „ProcessableGraph“-Klasse übernommen. Der „MonteCarloControler“ liefert die Basisdaten für den wirtschaftlichen Erfolg der Unternehmen. Während einer Simulation sammelt er die Resultate aller Bilanzen (siehe unten Klasse „BalanceSheet“), um diese später auswerten zu können.

Während eines Durchlaufes werden weitere Informationen erstellt, beispielsweise das Protokoll aller Aktivitäten im Unternehmen (siehe später Klasse „LogBook“). Selbst für heutige leistungsfähige Computer wäre aber das Aufbewahren aller Protokolle eine grosse Datenmenge (Gigabyte-Bereich). Somit ist es eine weitere Aufgabe des „MonteCarloControler“, nur eine definierte Anzahl Protokolle zu sammeln. Die Anzahl der aufzubewahrenden Protokolle wird durch die technische Variable mit der Param-ID „T003“ festgelegt.

Eine weitere zentrale Klasse heisst „Configuration“. Sie beinhaltet alle wirtschaftlichen Parameter für die Simulation, wie sie in Kapitel „3.5.1 Detaillierte Spezifikation des Business-Prozesses“ spezifiziert wurden. In der JavaDoc sind alle Parameter mit der in der Tabelle verwendeten „Param-ID“ gekennzeichnet, um ein einfaches Wiederfinden zu ermöglichen. Die Klasse „Statistic“ bietet, wie der Name vermuten lässt, Methoden zur mathematischen Analyse von Daten an. Die Klasse „Main“ ist nur eine Hilfsklasse, um die Simulation zu Testzwecken zu starten. Wird die Demo-Webapplikation benutzt (Design siehe Kapitel „4.7 Design der Demo-Webapplikation“), so wird die Simulation direkt im Webcontainer aus der Struts[10]-Action (Klasse ProcessMCAction) gestartet.

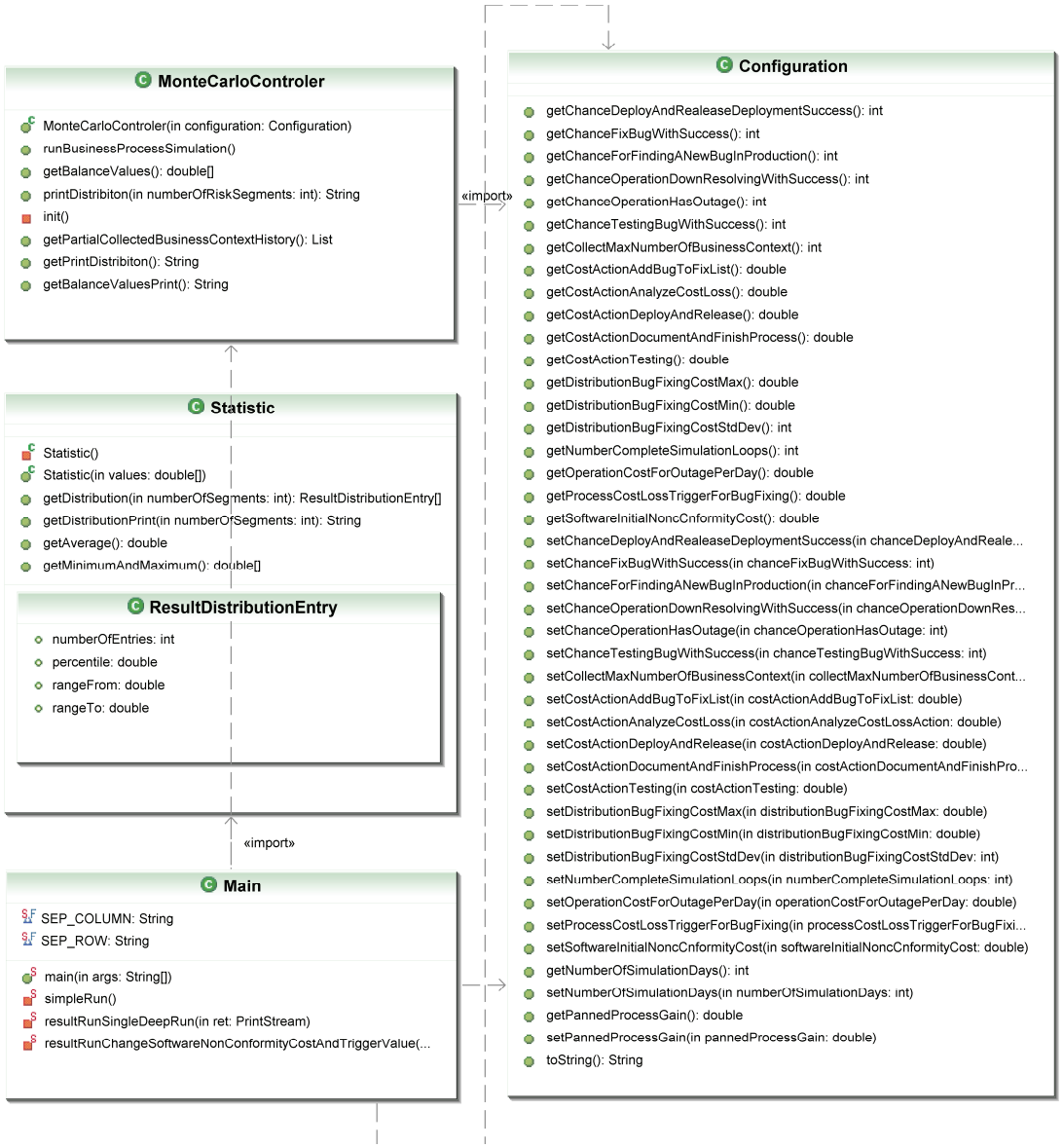


Abbildung 24: Das Basis-Package der applikatorischen Monte-Carlo-Komponente (com.x8ing.mc)

[10] Struts ist ein Open-Source-Framework für die Präsentations- und Steuerungsschicht von Java-Webanwendungen.

Package „com.x8ing.mc.bp“
Dieses Package (Klassendiagramm siehe Abbildung 25, Seite 42) bietet Komponenten zur Modellierung eines Business-Prozesses an. Verschiedene Klassen sind konkrete Implementierungen der State-Machine-Framework-Komponente.

Der „BusinessContext“ implementiert das Interface „StateContext“ und beinhaltet somit relevante Informationen über den aktuellen Zustand des simulierten Unternehmens. Die Klassen „Bug“ und „BugList“ bieten die Funktionalität zum Verwalten aller aktuell im Unternehmen bekannten Fehler der Software. Jeder Fehler wird dabei als Objekt des Typs „Bug“ in der „BugList“ geführt. Dabei besitzt jeder Fehler eine Reihe von Attributen, die ihn beschreiben, beispielsweise das Entdeckungsdatum, die geschätzten Software-Nonkonformitätskosten etc.

Der wirtschaftliche Erfolg des Unternehmens wird durch eine vereinfachte Buchhaltung festgehalten. Die Umsetzung „BalanceSheet“ umfasst dabei mehrere Konten (BalanceAccount), worauf die einzelnen Buchungsvorgänge (MoneyTransaction) gebucht werden. Somit lassen sich am Ende eines Simulationsdurchlaufes detaillierte Aussagen über die Kosten machen, die für die einzelnen Positionen entstanden sind. Gemäss der detaillierten Spezifikation (Kapitel „3.5.1. Detaillierte Spezifikation des Business-Prozesses“) gibt es fünf Konten, auf die gebucht werden kann. In der Implementierung sind die Konten nicht als Instanzen der Klasse „BalanceAccount“ umgesetzt, sondern ein Buchungsvorgang (MoneyTransaction) erfolgt immer mit Angabe eines bestimmten Kontos. Die Konto-Referenzen sind dabei statische Instanzen auf der Klasse „BalanceAccount“ und heissen „Production, Bug_Analyzing, Develop, Testing“.

Die abstrakte Klasse „AbstractBusinessAction“ implementiert die State-Machine-Framework-Klasse „Action“ und dient selbst wiederum als Basisklasse für alle „Actions“, die in einer Simulation ablaufen. Die „AbstractBusinessAction“ bietet eine Funktion an, um die Simulationszeit voranschreiten zu lassen. Eine weitere zentrale Klasse ist „BusinessProcesses“. Sie benutzt die Klasse „ProcessableGraph“ des State-Machine-Framework, um den Business-Prozess und seine Struktur aufzubauen. Alle Übergänge werden konfiguriert und die entsprechenden „Actions“ mit den „Conditions“ verknüpft. Der Code, der dies implementiert, ist in Kapitel „4.6.1 Aufbau der State-Machine“ abgebildet.

Die Klasse „LogBook“ dient hauptsächlich der Nachvollziehbarkeit und Plausibilitätskontrolle der programmierten Abläufe. Sie hält Einträge protokollartig fest und kann diese später in verschiedenen Formaten wieder ausgeben. Die „Constants“-Klasse stellt globale Konstanten zur Verfügung, so z. B. Formatierungsvorlagen für Währung, Zeit etc.

Package „com.x8ing.mc.bp.develop“
Alle Klassen dieses Package (Klassendiagramm siehe Abbildung 26, Seite 43) implementieren die abstrakte Basisklasse „AbstractBusinessAction“ (siehe oben), die selbst wieder die State-Machine-Framework „Action“ implementiert. Jede dieser spezialisierten Action-Klassen modelliert einen dedizierten Business-Prozessschritt des Modellunternehmens. Die Klassen von diesem Package bilden alle Prozessschritte des Modellunternehmens ab, die im Teilprozess „Develop“ geschehen (siehe Abbildung 14). Zusammen mit dem später beschriebenen Package „com.x8ing.mc.operation“ werden alle ablaufenden Prozesse des Modellunternehmens umgesetzt. Die vier „Conditions“, in der Abbildung, implementieren das „Condition“-Interface des State-Machine-Framework. Sie bestimmen die Wahl des nächsten auszuführenden Prozessschrittes.

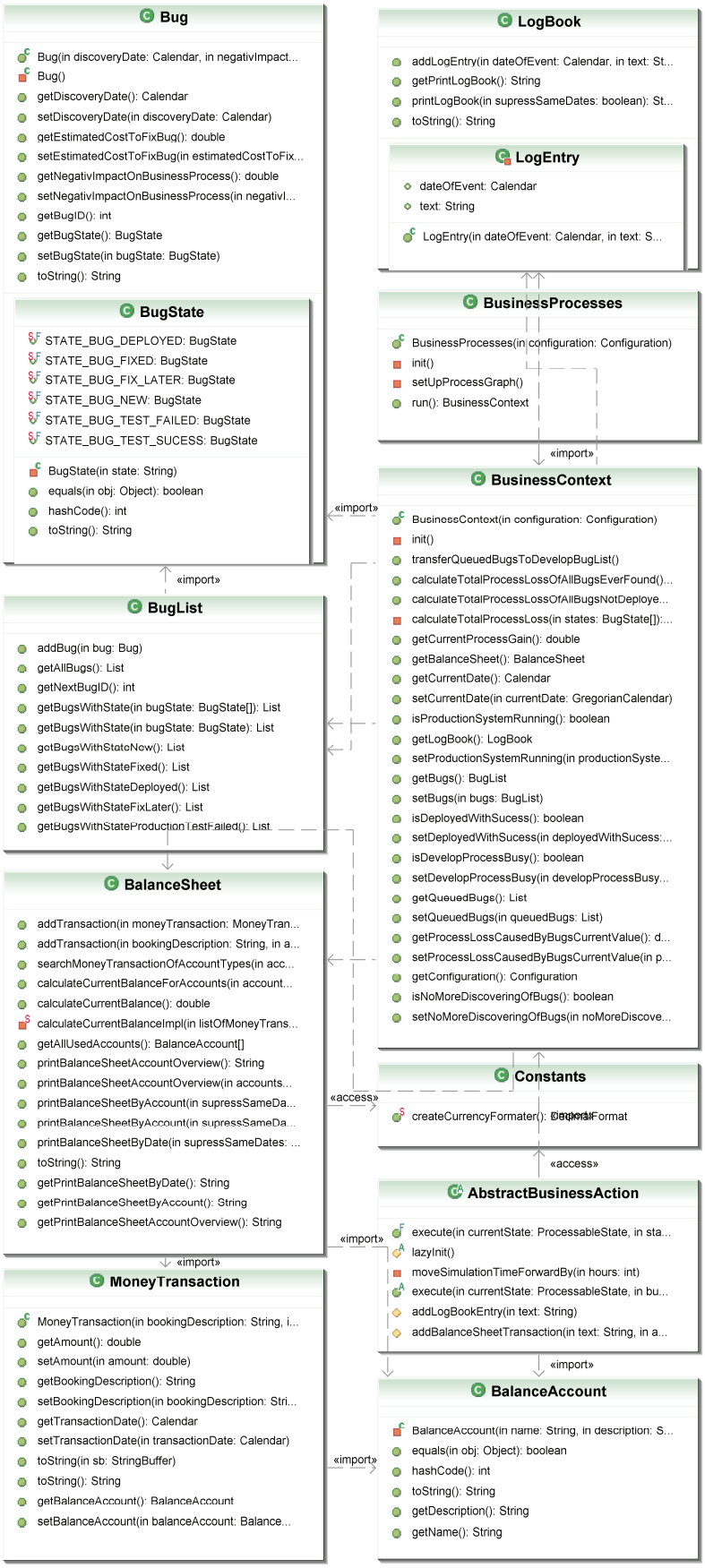


Abbildung 25: Package zur Modellierung des Business-Prozesses (com.x8ing.mc.bp)

Abbildung 26: Package zur Modellierung des Entwicklungsprozesses (com.x8ing.mc.bp.develop)

Package „com.x8ing.mc.operation“
Analog zum vorangehenden Package bilden dieses Package (Klassendiagramm siehe Abbildung 27) alle Prozessschritte und Übergänge ab, die im Modellunternehmen zum Betrieb „Operation“ gehören.

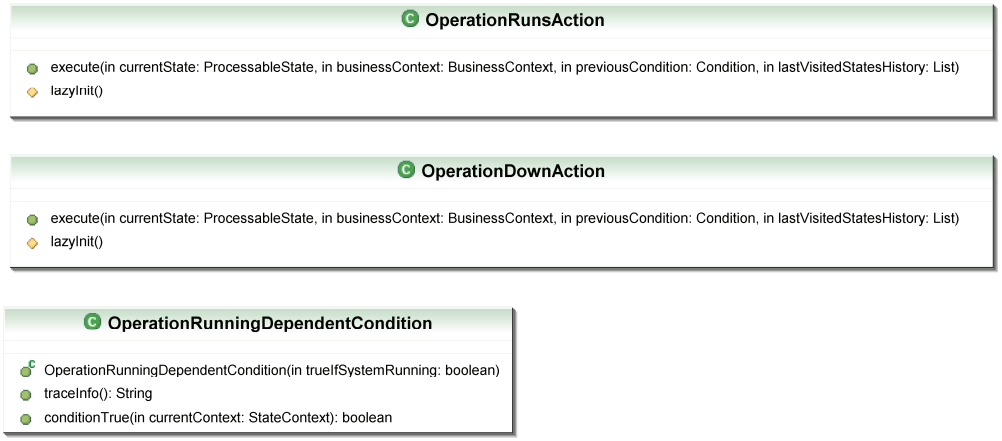


Abbildung 27: Package zur Modellierung des Operation-Prozesses (com.x8ing.mc.bp. operation)

Package „com.x8ing.mc.distribution“
Dieses Package (Klassendiagramm siehe Abbildung 28) stellt Methoden zur Generierung von Zufallszahlen zur Verfügung, die einer angestrebten statistischen Verteilung folgen. Ross (Ross 2002, S. 37) erläutert dabei die Wichtigkeit von möglichst natürlichen Zufallszahlen bei der Monte-Carlo-Simulation. Diese scheinbar einfache Aufgabe kann mathematisch komplex werden. In dieser Arbeit werden darum bestehende Helferklassen einer Library von CERN[11] verwendet [W9], um natürliche Zufallszahlen und gewünschte Verteilungen zu generieren. Um ein einfaches Austauschen der gewünschten Verteilung zu ermöglichen, implementieren alle Verteilungen das Interface „RandomDistribution“. Im Modellunternehmen werden nur zwei Verteilungsarten benutzt: die Gauss- und die Linearverteilung. Name und Eigenschaften dieser Verteilungen sind dabei von Manno (Manno 1999, S. 20 und S. 29) übernommen.

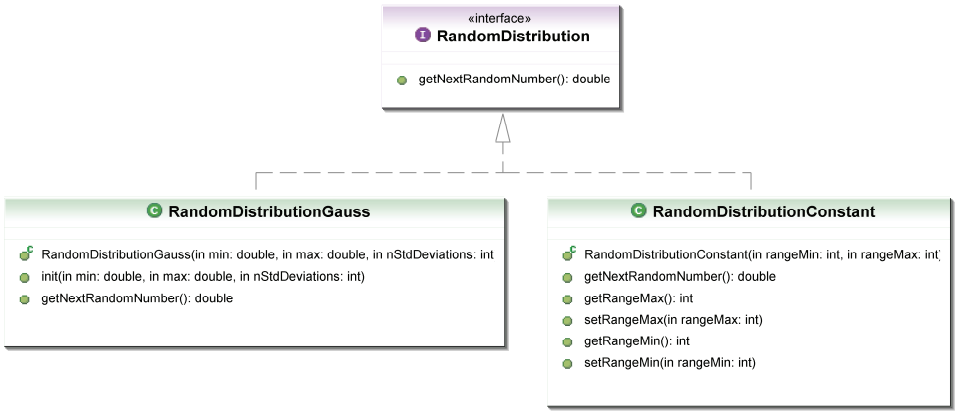


Abbildung 28: Package mit mathematischen Funktionen (com.x8ing.mc.distribution)

[11] CERN, die Europäische Organisation für Kernforschung, ist eine Grossforschungseinrichtung in der Nähe von Genf in der Schweiz.

4.5 Interaktion der Komponenten

Das nachfolgende Sequenz-Diagramm (Abbildung 29) zeigt die verschiedenen involvierten Klassen beider Hauptkomponenten und wie sie miteinander interagieren. Die Darstellung ist schemenhaft und fokussiert sich auf den wesentlichen Ablauf, einige Aufrufe wurden weggelassen.

Klassen mit dem Präfix „Concrete“ stehen für beliebige Implementierungen einer Klasse; z. B. kann eine „ConcreteBusinessAction“ stellvertretend stehen für „AnalyzeCostLossAction“, „TestingAction“ etc. Die in den Text eingebetteten Zahlen in Klammern referenzieren den jeweiligen Schritt in der Grafik.

Der Ablauf im Detail
Der „MonteCarloControler“ erhält bei seiner Erstellung eine Referenz auf die „Configuration“. Er hält (1) sich diese, um sie später an alle beteiligten Komponenten weitergeben zu können. Die „BusinessProcesses“-Klasse wird erstellt. Sie benutzt die State-Machine-Framework-Klasse „ProcessableGraph“ (4), um die Struktur des Business-Prozesses abzubilden. Die „AbstractBusinessAction“ implementiert die „Action“ des State-Machine-Framework und bildet selbst wieder die Elternklasse aller „ConcreteMonteCarloAction“-Instanzen, die in der Monte-Carlo-Hauptkomponente verwendet werden. Der Graph wird durch die Angabe aller Transitions und der darin beteiligten „Actions“ und „Conditions“ (7) aufgebaut. Abschliessend wird der „BusinessContext“ (9) erstellt, eine Implementierung des „StateContext“. Somit ist der Business-Prozess vollständig aufgebaut.

Der Start des Business-Prozesses erfolgt vom „MonteCarloControler“ wie Delegation (10) und kommt beim „ProcessableGraph“ (10.1) an. Dieser beginnt, alle „Conditions“ zu evaluieren (10.1.1), um den nächsten State festzulegen. Die „Condition“ kann zur Bestimmung ihres Rückgabewertes auf den „BusinessContext“ (10.1.1.1.1) oder die „Configuration“ (nicht speziell eingezeichnet) zugreifen. Wird eine „Condtion“ gefunden, die „true“ zurückgibt (10.1.1.1), so wird dieser Pfad verfolgt.

Beim nächsten State ankommend, wird die dem State angehängte „Action“ des State-Machine-Framework ausgeführt (10.1.3). Die „AbstractBusinessAction“ erhält diesen Aufruf (10.1.3) und führt zuerst einige interne Arbeiten aus. Anschliessend delegiert sie den Aufruf an eine eigene abstrakte Funktion weiter. Diese Funktion (abstract) muss somit von allen abgeleiteten Klassen implementiert werden. Somit erhält auch die „ConcreteMonteCarloAction“ den Aufruf (10.1.3.1). Zur Abarbeitung kann diese wiederum auf den „BusinessContext“ und die „Configuration“ zugreifen.

Die Abarbeitung ab Schritt (10.1) erfolgt anschliessend iterativ, bis eine gültige Endbedingung gefunden wird.

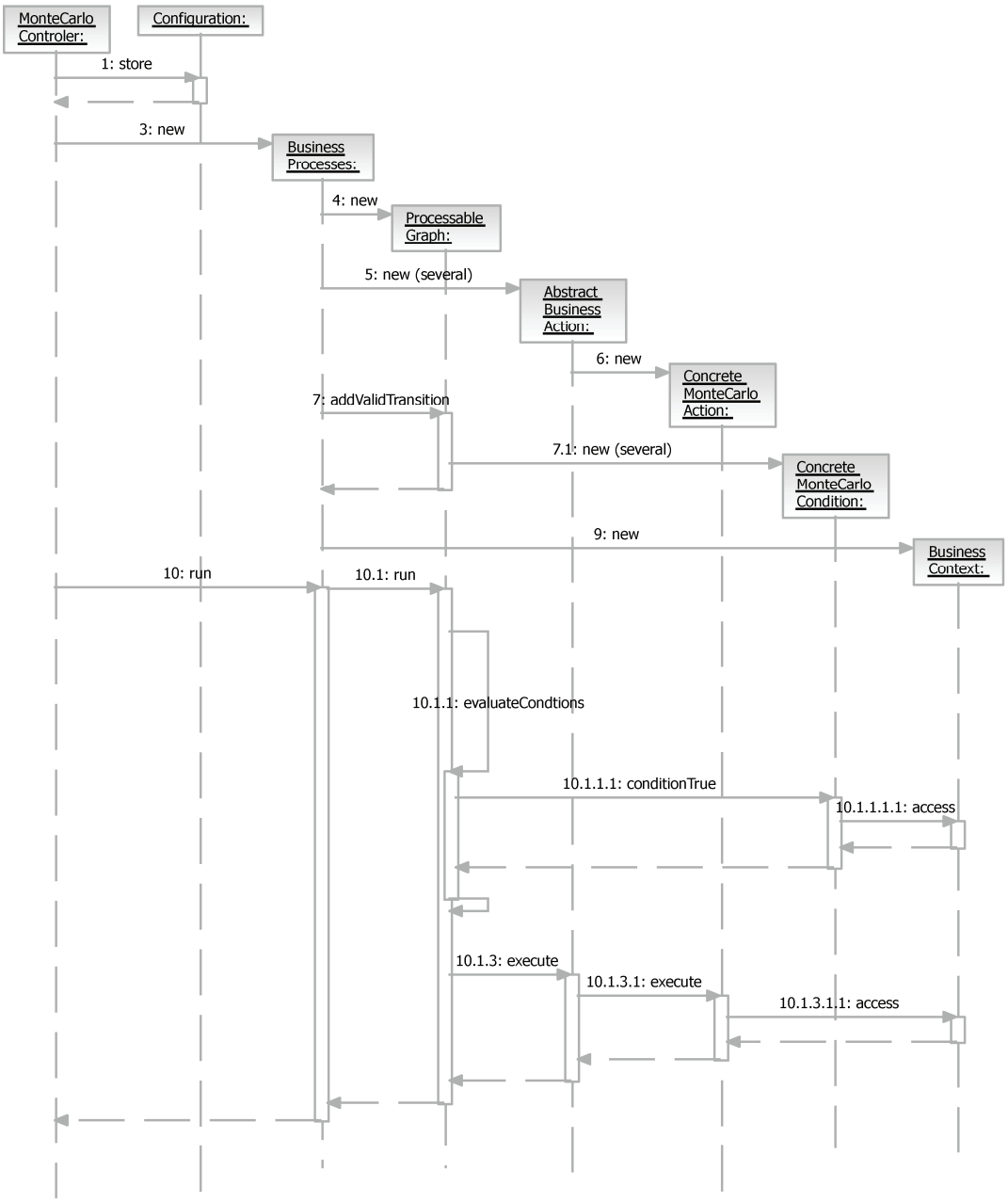


Abbildung 29: Sequenzdiagramm mit dem schemenhaften Ablauf beider Hauptkomponenten

4.6 Code-Beispiele

Die nachfolgenden zwei Code-Beispiele sind Ausschnitte aus dem erstellten Code, der für die Simulation des Modellunternehmens verwendet wurde. Aus Übersichtlichkeitsgründen wird auf die Abbildung von weiterem Code verzichtet. Der Code der gesamten Simulation ist aber vollumfänglich von der Webseite dieser Arbeit abrufbar [W1].

4.6.1 Aufbau der State-Machine

Das nachfolgende Beispiel (Abbildung 30) zeigt, wie die Framework- und die Business-Prozess-Komponente miteinander angewendet werden. Der abgedruckte Code-Ausschnitt zeigt, wie der Teil-Business-Prozess der „Operation“ (siehe Abbildung 14) aufgebaut wird. Zu Beginn wird ein „ProcessableGraph“ erstellt, der die Struktur des Graphen beinhaltet.

Dann (Code-Kommentar „create actions“) werden die „Action“-Instanzen erstellt: „OperationRuns-Action“ und „OperationDownAction“. Diese implementieren die Vorgänge im Unternehmen beim Eintreffen des entsprechenden Prozessschrittes.

In einem weiteren Schritt (Code-Kommentar „create states“) werden die „States“ erstellt, dabei wird jedem „State“ eine Action-Referenz zugewiesen. Für die State-Machine-Framework-Komponente ist nun definiert, welche Action sie beim Erreichen eines „State“ auszuführen hat.

Als letzter Schritt (Code-Kommentar: „create transitions“) wird die eigentliche Struktur definiert, das Layout des Graphen, indem die Übergänge (Transitions) zwischen den einzelnen States hinzugefügt werden. Um einen solchen Übergang genau zu spezifizieren, muss der Start- und End-State der Transition angegeben werden. Weiter kann eine Condition mitgegeben werden. In diesem Fall wird immer der gleiche Typ von Condition mitgegeben, nämlich die „OperationRunningDependentCondition“. Diese Condition ist einzig vom aktuellen Zustand der Produktion abhängig und reagiert darauf, ob die Produktion fehlerfrei läuft oder gerade einen Ausfall hat. Im Konstruktor definiert ein Flag (true oder false), ob die Condition bei Produktionsausfall „true“ oder „false“ zurückgeben soll. Das Flag hat also invertierende Wirkung auf den Rückgabewert. Dies hat den Vorteil, dass eine Instanz dieser Condition für beide benötigten Transitions (siehe „failureResolved“ und „failure“ in Abbildung 14) verwendet werden kann.

Mit diesem Code ist der „Operation“-Teilprozess abgebildet. Auf die Abbildung des komplexeren „Develop“-Teilprozesses wird verzichtet. Das Vorgehensprinzip ist aber identisch.

```
// _CUT_START
private void setUpProcessGraph() {

int uidCounter = 100;

// scope: set up operation process

bpOperation = new ProcessableGraph();

// ////////////////////////////////////////////////////////////////////////////
// create actions
OperationRunsAction operationRunsAction = new OperationRunsAction();
OperationDownAction operationDownAction = new OperationDownAction();

// ////////////////////////////////////////////////////////////////////////////
// create states
ProcessableState operationRuns = new ProcessableState(bpStartID, "operationRuns",
                operationRunsAction, false);
ProcessableState operationDown = new ProcessableState(uidCounter++,
                "operationRuns", operationDownAction, false);

// ////////////////////////////////////////////////////////////////////////////
// create transitions
bpOperation.addValidTransition(operationRuns, operationRuns,
                new OperationRunningDependentCondition(true));
bpOperation.addValidTransition(operationRuns, operationDown,
                new OperationRunningDependentCondition(false));
bpOperation.addValidTransition(operationDown, operationDown,
                new OperationRunningDependentCondition(false));
bpOperation.addValidTransition(operationDown, operationRuns,
                new OperationRunningDependentCondition(true));

// _CUT_END
```

Abbildung 30: Codeausschnitt zum Aufbau der State-Machine

4.6.2 Beispiel einer Action-Implementierung

Beim Aufbau des Graphen werden „Actions“ verwendet. Der in Abbildung 31 gezeigte Codeausschnitt gehört zur „OperationDownAction“, die simuliert, was geschehen soll, wenn die Produktion einen Fehler in der zentralen Serverinstanz erleidet und allen Benutzern das Weiterarbeiten verunmöglicht. Die Implementierung ist einfach und kurz und eignet sich daher gut als Beispiel für den Aufbau einer Action-Implementierung.

Erläuterung der Vererbungshierarchie

Die „OperationDownAction“ erbt von der „AbstractBusinessAction“. Diese Eltern-Klasse stellt verschiedene oft gebrauchte Methoden zur Verfügung. Diese sind:

- addBalanceSheetTransaction: Führt eine Buchung aus. Auf einem bestimmten Buchungskonto wird automatisch eine Transaktion angelegt. Diese wird mit dem aktuellen Zeitstempel versehen.
- addLogBookEntry: Dies ist eine Hilfsfunktion, die im Logbuch (Protokoll) des Unternehmens einen Eintrag erstellt.

Die „AbstractBusinessAction“ erbt wiederum vom State-Machine-Framework „Action“ und kann daher, wie im vorherigen Kapitel erwähnt, einem State angehängt werden. So ist es auch das Framework, das die „execute“-Methode beim Erreichen eines State automatisch aufruft. Die „AbstractBusinessAction“ implementiert diese „execute“-Methode. Es werden einige interne Vorbereitungen getroffen, unter anderem wird eine Typenumwandlung (engl. „cast“) des übergebenen „StateContext“ auf die konkrete Klasse „BusinessContext“ durchgeführt. Anschliessend delegiert die „AbstractBusinessAction“ die weitere Prozessierung an eine eigene „abstrakte“ Methode „execute()“, die mit einer überladenen Methoden-Signatur um den „BusinessContext“ erweitert ist. Diese abstrakte Methode zwingt alle vererbenden Klassen, selbige zu implementieren, so auch die „OperationDownAction“, die so den Methodenaufruf erhält. So kommt der Einstieg in die abgebildete Klasse zustande.

Trennung der Aufgaben

Es gilt zu beachten, dass in der „OperationDownAction“ nichts programmiert wird, was bestimmen würde, was im Modellunternehmen als Nächstes passieren soll. Es werden lediglich die Prozessschritte ausprogrammiert, die ausgeführt werden sollen. Die Entscheidung darüber, was als Nächstes geschieht, wird durch das State-Machine-Framework gefällt. Als Membervariable der Klasse findet sich einzig die lineare Verteilung „failureResolveChance“, die gemäss der Spezifikation des Modellunternehmens (siehe Kapitel „3.5.1. Detaillierte Spezifikation des Business-Prozesses“) festlegt, mit welcher Wahrscheinlichkeit ein Produktionsausfall behoben werden kann. Im „BusinessContext“ wird ein möglicher Erfolg oder Misserfolg beim Beheben des Produktionsausfalls abgespeichert, um später von der Condition ausgewertet zu werden.

Es fällt auf, dass keine Zahlenwerte in der Klasse „hart“ codiert sind. Alle Parameter, die das Unternehmen definieren, sind zentral in der Klasse „Configuration“ ausgelagert. Beim Start einer Simulation wird diese Klasse mit den gewünschten Werten abgefüllt und steht nachher allen „Actions“ und „Conditions“ zur Verfügung. Diese zentrale Datenhaltung ist für die Implementierung der Webapplikation (siehe Kapitel „9.2. Diplomarbeit-Webseite und Demo-Webapplikation“) vorteilhaft, da dort automatisch per JSP[12]und Struts[13] mittels Formpopulierung die „Configuration“ mit den gewünschten Werten abgefüllt werden kann.

---

[12] JSP: JavaServer Pages, eine von Sun Microsystems entwickelte Technologie, die im Wesentlichen zur einfachen dynamischen Erzeugung von HTML- und XML-Ausgaben eines Webservers dient.

[13] Struts ist ein Open-Source-Framework für die Präsentations- und Steuerungsschicht von Java-Webanwendungen.

Ablauf innerhalb der Action
Wie die bisherige Erläuterung der Klasse zeigt, wird viel Funktionalität durch die mitbeteiligten Klassen implementiert. Dies macht die abgebildete Klasse kurz. In der Klasse selbst laufen die folgenden Schritte ab:

- Es erfolgt eine Buchung mit den Ausfallkosten (Code-Kommentar: „book money“).
- Es wird ein Protokoll (LogBook) Eintrag erstellt.
- Es wird überprüft, ob die Störung in diesem Zeitabschnitt behoben werden kann.
- Konnte das Problem gelöst werden, wird dies im „BusinessContext“ festgehalten, indem das Flag „productionSystemRunning“ auf den Wert „true“ gestellt wird.

Nach Abarbeitung dieser „Action“ wird der State-Machine-Framework-Kontroller (ProcessableGraph) die verschiedenen „Conditions“ evaluieren. Diese sind in diesem Fall (gemäss Abbildung 14 in Verbindung mit Abbildung 27) verschiedene Instanzen der „OperationRunningDependentCondition“. Diese Condition prüft einzig, ob das vorher gesetzte Flag zur Anzeige des Zustandes der Operation (productionSystemRunning) „true“ oder „false“ ist, und wird diesen Wert entsprechend dem Kontroller zurückgeben.

```
public class OperationDownAction extends AbstractBusinessAction {

private RandomDistribution failureResolveChance = new RandomDistributionConstant(0,
                100);

public void execute(ProcessableState currentState, BusinessContext businessContext,
                Condition previousCondition, List lastVisitedStatesHistory) {

        // book money
        addBalanceSheetTransaction("Production down: Outage costs.", -businessContext
                        .getConfiguration().getOperationCostForOutagePerDay(),
                        BalanceAccount.PRODUCTION);

        // log info
        addLogBookEntry("Production system is down.");

        // check if we can fix the fug
        if (failureResolveChance.getNextRandomNumber() < businessContext

        .getConfiguration().getChanceOperationDownResolvingWithSuccess()) {

                // we solved the problem
                businessContext.setProductionSystemRunning(true);

                addLogBookEntry("production failure resolved.");

        } else {
                addLogBookEntry("production failure not yet resolved.");

        }

}
}
```

Abbildung 31: Codeausschnitt der OperationDownAction

4.7 Design der Demo-Webapplikation

Im Gegensatz zu den vorherigen Packages, die für die eigentliche Implementierung der Monte-Carlo-Methode zuständig sind, steht in diesem Package (Abbildung 32) die Darstellung und Visualisierung im Vordergrund. Die Monte-Carlo-Simulation ist komplett unabhängig von der hier beschriebenen Webkomponente, wird aber durch diese angestossen.

Der Benutzer erhält in der erstellten Webapplikation die Möglichkeit, alle Eckdaten der Simulation (Kapitel „3.5.1 Detaillierte Spezifikation des Business-Prozesses“) per Web-GUI einzugeben, die Simulation

zu starten und abschliessend eine visuelle Auswertung in Form eines Histogramms der Wahrscheinlichkeiten zu erhalten. Weiter bekommt er Einblick in die ausgeführten Abläufe und erhält eine detaillierte finanzielle Auswertung aller Aktivitäten. Detaillierte Screenshots der Applikation finden sich im Anhang in Kapitel „9.2 Diplomarbeit-Webseite und Demo-Webapplikation“.

Die Webapplikation wurde mittels einer Java EE[14]-Anwendung umgesetzt, die auf dem Apache Struts[15]-Opensource-Projekt [W10] basiert. Verwendet wurde Struts in der Version 1.3.8. Zum Anzeigen der Resultat-Grafik (Histogramm) wird die Opensource Library JFreeChart[16] [W11] in der Version 1.0.6 verwendet. Betrieben wird die Anwendung von Apache Tomcat[17] [W12] Webcontainer in der Version 6.0.10. Der Container wiederum läuft auf einem OpenSUSE[18] Server in der Version 10.2.

Package: com.x8ing.mc.web
Dieses Package zeigt alle Klassen (Abbildung 32), die von der Webapplikation benutzt werden, welche die Simulation visuell darstellt. Die Klasse „MCForm“ erbt von der Struts-Framework-Klasse „org.apache.struts.action.ActionForm“. Weiter erben „ProcessMCAction“ und „OpenMCAction“ von „org.apache.struts.action.Action“. Der „GlobalRequestListener“ basiert direkt auf der Java EE-Klasse „javax.servlet.ServletRequestListener“. Nicht abgebildet sind die entwickelten JSP Files.

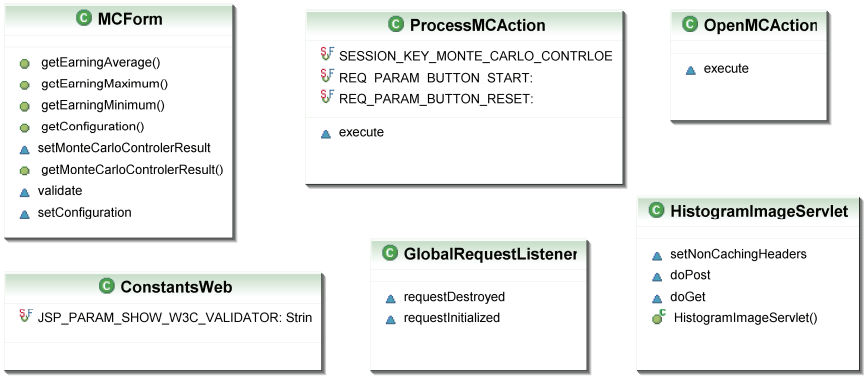


Abbildung 32: Package für die Webkomponente (com.x8ing.mc.web)

[14] Java Platform, Enterprise Edition, abgekürzt Java EE oder früher J2EE, ist die Spezifikation einer Softwarearchitektur für die transaktionsbasierte Ausführung von in Java programmierten Anwendungen.

[15] Struts ist ein Open-Source-Framework für die Präsentations- und Steuerungsschicht von Java-Webanwendungen.

[16] JFreeChart ist ein Framework für die Programmiersprache Java, mit welchem auf einfache Weise auch komplexe Diagramme erstellt werden können.

[17] Apache Tomcat stellt eine Umgebung zur Ausführung von Java-Code auf Webservern bereit.

[18] OpenSUSE, ehemals SUSE Linux, ist eine Linux-Distribution der Firma Novell.

5. RESULTATE

5.1 Überlegungen zur Implementierung der Monte-Carlo-Methode in Java-Code

Die Arbeit stellt die Frage nach der Umsetzbarkeit der Monte-Carlo-Methode in Java. Das vorangehende Kapitel „4. Implementierung der Monte-Carlo-Methode“ beantwortet diese Frage bereits implizit durch das beschriebene Design und dessen Umsetzung. Auf die nochmalige Ausführung dieser technischen Aspekte wird deshalb verzichtet. Es kann grundsätzlich festgehalten werden, dass die Monte-Carlo-Methode für ein Modellunternehmen erfolgreich in Java umgesetzt werden kann.
In den nachfolgenden zwei Unterkapiteln werden jene Aspekte der Umsetzung dargelegt, die noch nicht im vorangehenden Kapitel behandelt wurden.

5.1.1 Allgemeines Laufzeitverhalten der Applikation

Die Hardwareansprüche der Applikation sind erheblich. Während der Laufzeit werden ungefähr 50 Megabyte an Rechenspeicher benötigt. Ausserdem liegt der Zeitaufwand für einen Simulationsdurchlauf mit Variation mehrerer Parameter im Stunden- oder gar Tagesbereich.

5.1.2 Erfahrungen bei der praktischen Umsetzung

Die Kapitel „3.4 Erarbeitung des Business-Prozesses im Modellunternehmen“ und „3.5 Weitere Prozessaspekte: Zeit, Kosten und Eintrittswahrscheinlichkeit“ können als Spezifikation des zu implementierenden IT-Systems verstanden werden. Die Requirements waren somit klar definiert und die Anforderungen konnten mit Standardmitteln gelöst werden. Während der Implementierungsphase sind keine technischen Schwierigkeiten aufgetaucht, welche die Entwicklungsarbeiten aufgehalten, verzögert oder gar blockiert hätten. Zudem verfügte der Autor dieser Arbeit bereits über Erfahrungen mit allen eingesetzten Technologien. Die Implementierung konnte deshalb zeiteffizient umgesetzt werden. Dennoch benötigte die komplette Implementierung, inklusive ausgiebigen Testings, mehr als vier volle Arbeitswochen.

5.2 Erfolgsauswirkung einer definierten Software-Nonkonformität

Es werden die finanziellen Auswirkungen einer definierten Software-Nonkonformität auf das Ergebnis des Modellunternehmens untersucht. Das der Monte-Carlo-Simulation zugrunde liegende Modell bildet ab, wie sich das Unternehmen ausgehend von der initial gelieferten Software-Nonkonformität entwickeln wird.

Analog der in Kapitel „2.5.3. Methode der Szenarienbetrachtung“ beschriebenen Methode wird ein Histogramm zur grafischen Auswertung der gewonnenen Ergebnisse benutzt.

5.2.1 Datenbasis

Grundlage der numerischen Datenbasis bildet der aufgestellte Business-Prozess (siehe Kapitel „3.4 Erarbeitung des Business-Prozesses im Modell“). Sofern nicht ausdrücklich erwähnt, wurden für alle Parameter ihre Defaultwerte gemäss der Spezifikation in Kapitel „3.5.1 Detaillierte Spezifikation des Business-Prozesses“ beibehalten.

Datenbasis bilden 10'000 Simulationsdurchläufe des Modellunternehmens. Pro Simulationsdurchlauf werden jeweils 180 Tagesdurchläufe simuliert. Es wird angenommen, dass die Software-Nonkonformitätskosten 400 CHF betragen. Dieser Parameter hat die Param-ID „G001“ und wird in Kapitel „3.5.1 Detaillierte Spezifikation des Business-Prozesses“ beschrieben. Die numerische Datenba-

sis dieser Simulation findet sich im Anhang in Kapitel „9.4.1 Einmaliger Simulationsdurchlauf mit detaillierter Risiko-Ausgabe“. Der Quellcode, der zu diesen Daten führt, findet sich in der Klasse „com.x8ing.mc.Main#resultRunSingleDeepRun“.

Diese Simulation führt zu 10'000 Endzuständen. Dabei wird über die kosten- und ertragsrelevanten Transaktionen während des ganzen Verlaufs Buch geführt. Nach Erstellen einer Erfolgsrechnung werden die Saldi der Endzustände in einem Histogramm dargestellt (Abbildung 33). Die X-Achse bildet den Ertrag (in tausend CHF) ab. Die Y-Achse zeigt, mit welcher Auftrittswahrscheinlichkeit ein solches Ergebnis erzielt wird. Jeder Balken in der Grafik wird durch ein festgelegtes Intervall definiert, in dem die jeweiligen Ergebniswerte liegen. Dargestellt werden 100 Balken, was ein Intervall von 1'715 CHF zur Folge hat. Der dunkelblaue Balken definiert das arithmetische Mittel aller Daten. Die hellblauen Balken definieren das 95%-Sicherheitsintervall.

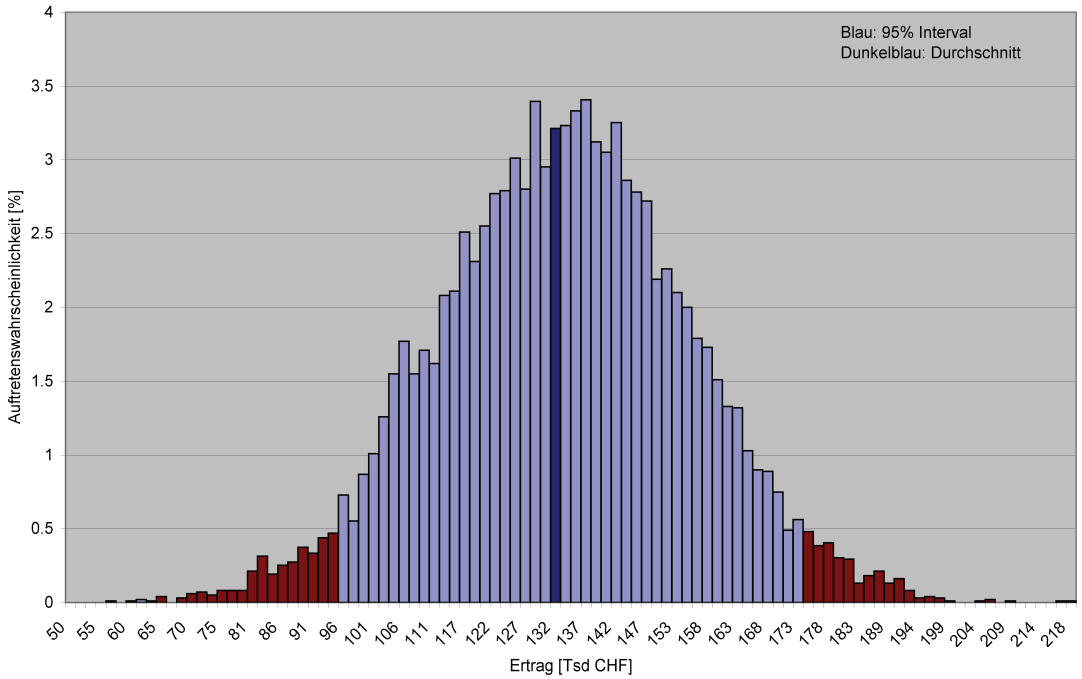


Abbildung 33: Histogramm der Ertragsergebnisse

5.2.2 Betriebswirtschaftliche Betrachtung

Beim Betrachten des Histogramms fällt auf, dass alle Werte auf der X-Achse ein positives Vorzeichen haben. Dies bedeutet, dass in keinem einzigen Simulationsdurchlauf ein Verlust hingenommen werden musste. Durchschnittlich wird das Modellunternehmen einen Gewinn von 132'000 CHF erzielen (dunkelblauer Balken).

Die Auswertung anhand der Monte-Carlo-Methode liefert aber noch mehr Informationen. So kann dem Diagramm entnommen werden, dass im schlechtesten Fall mit einem Gewinn von ca. 47'000 CHF zu rechnen ist (siehe numerische Datenbasis oben). Dieser Fall tritt zwar nur in 0.01% aller Durchläufe auf, kann durch eine Verkettung von vielen negativen Ereignissen aber trotzdem eintreten. Ebenfalls in 0.01% wird ein erfreuliches Ergebnis von 218'000 CHF erzielt. Zudem ist erkennbar, dass die Streuung der Resultate relativ gross ist. So liegt das 95%-Sicherheitsintervall zwischen 94'000 und 175'000 CHF. Die gelieferte Software wird also mit 95%iger Wahrscheinlichkeit einen Gewinn generieren, der zwischen 94'000 und 175'000 CHF liegt.

5.2.3 *Laufzeitverhalten der Simulation*

Die Simulation benötigt auf einem Intel Pentium IV 3.0 GHz System mit einer Java JRE 6 im Server-Modus ungefähr 106 Sekunden.

5.3 Erfolgsauswirkung von variabler Software-Nonkonformität

In der vorangehenden Simulationsauswertung lag der Fokus auf der Vorhersage des finanziellen Erfolges bei definierten Software-Nonkonformitätkosten. In dieser Simulation wird der Simulationsumfang erweitert: Der Initialwert der Software-Nonkonformitätskosten (Parameter „G001“) wird bei der Ausgangssituation jedes Simulationsdurchlaufes verändert. Das Augenmerk liegt dabei auf dem Durchschnitt der Ergebnisse. Dies ermöglicht es, den Ertrag in Abhängigkeit der Software-Nonkonformität zu bestimmen.

5.3.1 *Datenbasis*

Wie bei der vorangehenden Simulation, haben alle Parameter ihren Default-Wert. Der Initialwert des Parameter „Software-Nonkonformitätskosten“ (in der vorangehenden Simulation 400 CHF) wird nun für jede Ausgangssituation eines Simulationsdurchlaufes von 100 bis 2'000 CHF in 50er-Schritten erhöht. Insgesamt bestehen also 39 Ausgangssituationen für Simulationsdurchläufe. Jeder Simulationsdurchlauf wird 10'000 Mal über jeweils 180 Tage durchgerechnet. Es resultieren 390'000 Endzustände (39 * 10'000). Von jedem der 39 Simulationsdurchläufe wird der arithmetische Mittelwert des jeweiligen Erfolges berechnet und im Diagramm (Abbildung 34) eingetragen. Somit entspricht jeder Datenpunkt des Diagramms dem durchschnittlichen Erfolg aus 10'000 Simulationsdurchläufen (über 180 Tage) für die entsprechende Software-Nonkonformität. Abschliessend werden die Datenpunkte mit einer Trendlinie (lineare Regression) ergänzt.

Die numerische Datenbasis dieser Simulation findet sich im Anhang in Kapitel „9.4.2 Mehrere Durchläufe mit Variation zweier Simulationsparameter“. Der Quellcode, der zu diesen Daten führt, findet sich in der Klasse „com.x8ing.mc.Main#resultRunChangeSoftwareNonConformityCostAndTriggerValue“.

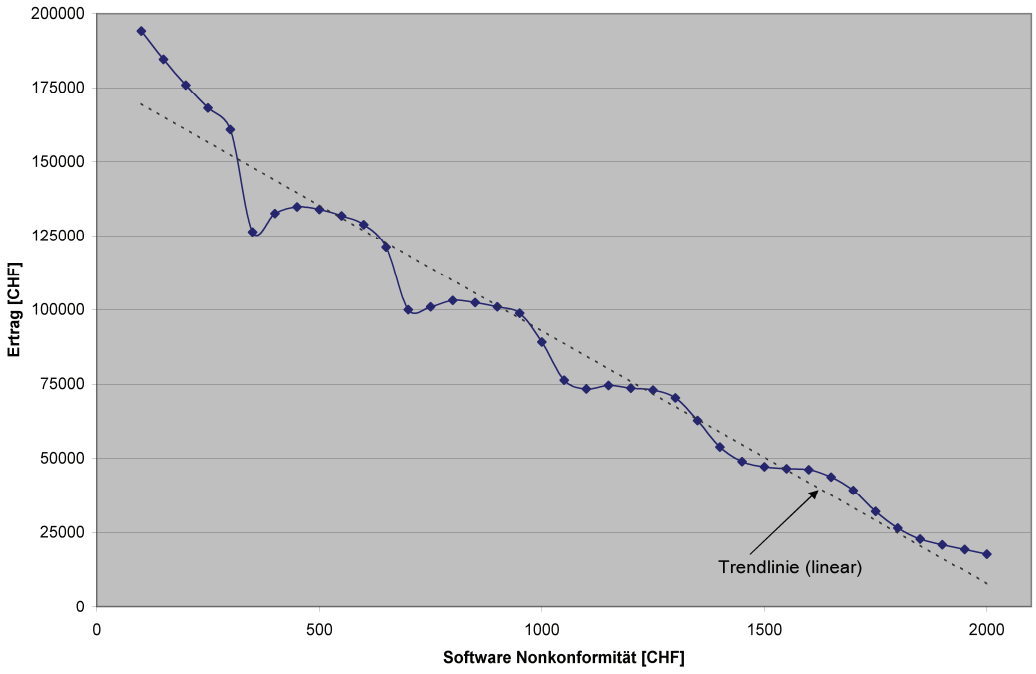


Abbildung 34: Ertrag in Abhängigkeit zur Software-Nonkonformität

5.3.2 Betriebswirtschaftliche Betrachtung

Es fällt auf, dass der finanzielle Ertrag mit dem Abnehmen der Software-Nonkonformität steigt. Je weniger Fehler oder prozessbehindernde Elemente in der gelieferten Software existieren, desto weniger Probleme tauchen auf und umso effizienter können die Mitarbeiter ihre Aufgaben erfüllen. Dies wiederum führt zu einer kosteneffizienteren Erledigung der Arbeiten, was zu einem höheren Ertrag führt.

Weiter zeigt das Diagramm, dass die Ertragskurve in Wellen verläuft. Die Ursache für dieses Verhalten kann vielfältiger Natur sein und ist analytisch schwer herzuleiten. Dies ist hier aber auch nicht notwendig. Aus betriebswirtschaftlicher Sicht ist wichtig, dass die Charakteristik überhaupt erkannt wird und daraus Schlüsse gezogen werden können. Wird beispielsweise die Veränderung des Ertrages zwischen den Datenpunkten der Software-Nonkonformitätskosten von 300 und 350 CHF (X-Achse) betrachtet, so ist ein erheblicher Ertragsunterschied von rund 34'000 CHF (160'000–126'000) feststellbar. Dies bedeutet, dass eine Verringerung der Software-Nonkonformitätskosten von 350 auf 300 CHF (also eine Verbesserung der Softwarequalität) eine Ertragssteigerung von 34'000 CHF zur Folge hat. Daraus kann beispielsweise geschlossen werden, dass es sich lohnen könnte, in zusätzliche Ressourcen zur Steigerung der Softwarequalität zu investieren.

Wird die Kurve mit der linearen Trendlinie verglichen, so fällt auf, dass die Kurve um die Trendlinie schwankt. Erst wenn der Wert der Software-Nonkonformitätskosten gegen CHF tendiert, also nahezu fehlerfreie Software ausgeliefert wird (Software-Nonkonformitätskosten kleiner als 200 CHF), deutet sich ein exponentiell steigendes Ertragswachstum an.

5.3.3 Laufzeitverhalten der Simulation

Details zum Laufzeitverhalten werden im späteren Kapitel „5.4.3 Laufzeitverhalten der Simulation“ beschrieben.

5.3.4 Exkurs: Überlegungen zum Gesamterfolg

Es folgt ein Einschub mit Diskussionscharakter. Da sich der Exkurs aber nicht direkt auf die ursprüngliche Fragestellung dieser Arbeit bezieht, werden die Ausführungen dazu in diesem Kapitel dargelegt. Die Fragestellung dieser Arbeit bezieht sich auf die finanziellen Auswirkungen von gelieferter Software-Nonkonformität. Es wird davon ausgegangen, dass die Software bereits entwickelt und initial ausgeliefert wurde. Es liegt ausserhalb des gesetzten Rahmens dieser Arbeit, die Entwicklungskosten mitzusimulieren. Das obige Resultat muss deshalb vorsichtig interpretiert werden. Es muss klar darauf hingewiesen werden, dass im ausgewiesenen Erfolg die initialen Entwicklungskosten nicht eingerechnet sind. Zwar umfasst die Simulation Wartungsarbeiten (Bugfixing, Testing), aber nicht den Aufwand zur Erstellung der Software.

Beispiel
Die obige Grafik erweckt den Eindruck, es sei am ertragsreichsten, fehlerfreie Software zu liefern. Es soll nun hypothetisch angenommen werden, dass die Entwicklungskosten exponential zur Qualität steigen (pinkfarbene Kurve, Y-Achse links in Abbildung 35). Die Erfolgskurve wird aus dem vorangehenden Simulationsergebnis (dunkelblaue Kurve) übernommen. Der Gesamterfolg (grüne Kurve, Y-Achse rechts) würde sich dann aus der Differenz der beiden Kurven ergeben. Die Gesamterfolgskurve zeigt nun ein anderes Bild als die in der zuvor dargestellten Auswertung (Abbildung 34). Eine fast fehlerfreie Software ist mit sehr hohen Kosten verbunden, die den Gesamterfolg mindern. Demnach wäre es rentabler, gewisse Fehler hinzunehmen. Im gezeigten Fall gibt es zwei nahezu identische Optima: Eines liegt bei Software-Nonkonformitätskosten von 900 CHF, das andere bei 1'250.

Es existieren verschiedene Methoden, die Entwicklungskosten von Software zu schätzen (Laird et al. 2006, Kapitel „6.2. Software Estimation Methodologies And Models“). Die Annahme einer exponentiel-

len Kurve ist eine mögliche Methode (Laird et al. 2006, Kapitel „8.6. The Cost Of Reliability“), die Entwicklungskosten in Abhängigkeit der Softwarequalität zu bringen. Für die Anwendung bei einem realen Unternehmen müsste die Formel von Laird auf die tatsächliche Situation angepasst werden. Die vollständige Formel würde zudem noch weitere Parameter (Komplexität der Software, Zeit etc.) beinhalten. Die in diesem Beispiel verwendete exponentielle Kurve ist eine starke Vereinfachung der Formel von Laird und soll nur die Vorgehensweise illustrieren.

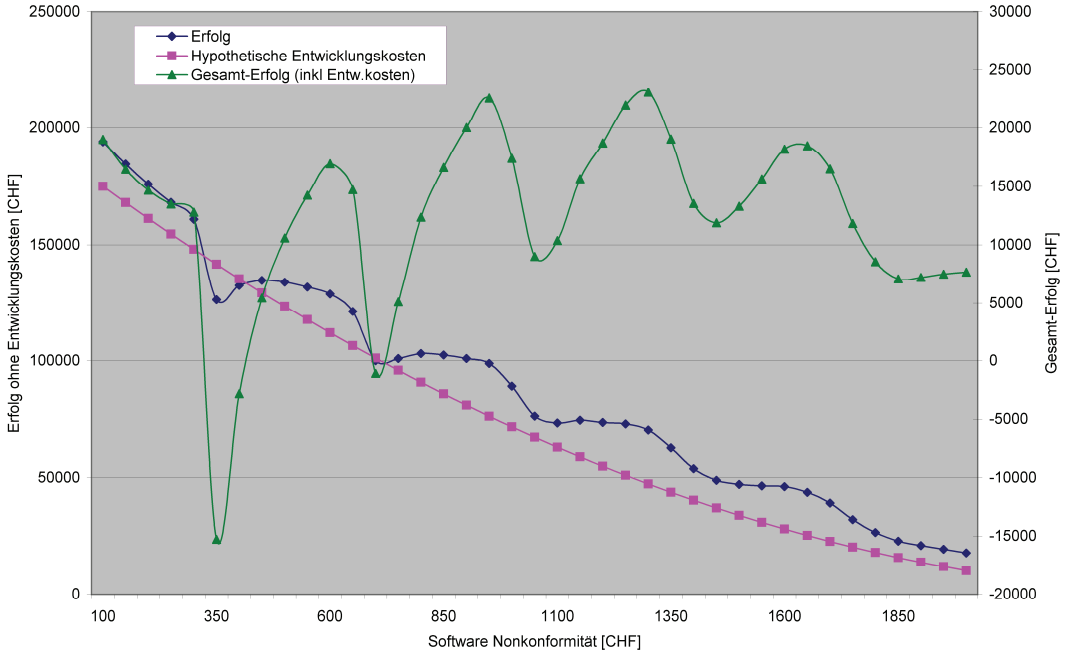


Abbildung 35: Erfolg unter Berücksichtigung von hypothetischen Entwicklungskosten

5.4 Ableitung weiterer wirtschaftlich relevanter Schlüsse

Eine weitere Fragestellung dieser Arbeit lautet, welche weiteren Schlüsse aus dem Ergebnis der Simulation mittels Monte-Carlo-Methode gezogen werden können. Die Antwort wird hier beispielhaft gegeben, indem ein weiterer Parameter im Simulationsdurchlauf verändert wird. Es soll so gezeigt werden, dass daraus weitere betriebswirtschaftlich relevante Informationen gewonnen werden können.

5.4.1 Datenbasis

Die Datenbasis für die Simulation ist ähnlich der vorangehenden. Allerdings wurde die Anzahl der sich verändernden Parameter abermals erhöht.

Die Software-Nonkonformitätskosten sollen dabei gleich wie oben den Werten zwischen 100 und 2'000 CHF in 50er-Schritten folgen. Zusätzlich wird der Parameter „G003“[19] (siehe Kapitel „3.5.1 Detaillierte Spezifikation des Business-Prozesses“) verändert. Dieser wird fortan als „CostLossTrigger“ bezeichnet. Der „CostLossTrigger“ wird für jede Ausgangssituation eines Simulationsdurchlaufes von 100 bis 1'200 CHF in 100er-Schritten erhöht. Insgesamt bestehen nun 468 (12 * 39) Ausgangssituationen für Simulationsdurchläufe. Diese leiten sich aus 39 Ausgangssituationen für die „Software-Nonkonformitätskosten“ und 12 Ausgangssituationen für den „CostLossTrigger“ her. Aus diesen Ausgangssituationen resultieren 4'680'000 Endzustände (39 * 12 * 10'000). Dies bedeutet, dass in diesem Versuch insgesamt fast eine

[19] Zur Erinnerung die Definition des „CostLossTrigger“ gemäss der Spezifikation: „Schwellenwert, den die Software-Nonkonformitäts-Kosten überschreiten müssen, damit die Entscheidung gefällt wird, die Fehler zu beheben und ein neues Release der Software zu liefern.“

Milliarde Tage (genau 842'000'000 = 4'680'000 Endzustände * 180 Tage) simuliert werden. Würde die oben geschilderte Simulation mathematisch formuliert, ergäbe sich folgende Formel:

$$Ertrag(G001, G003) = sim(G001, G003, p1..pn)$$

Die Simulation als Funktion mit Namen „sim“ ist von den Parametern „G001“, „G003“ und weiteren konstanten Parametern abhängig, wobei gilt: 100 <= G001 <= 2000 (in 50er-Schritten) und 100 <= G003 <= 1200 (in 100er-Schritten). Als Ergebnis resultiert ein Wert für den Ertrag. Das Ergebnis der Simulation kann somit im dreidimensionalen Raum als Oberfläche aufgezeichnet werden. Mit Hilfe von Mathematica[20] kann ein dreidimensionales Bild dargestellt werden (Abbildung 36).

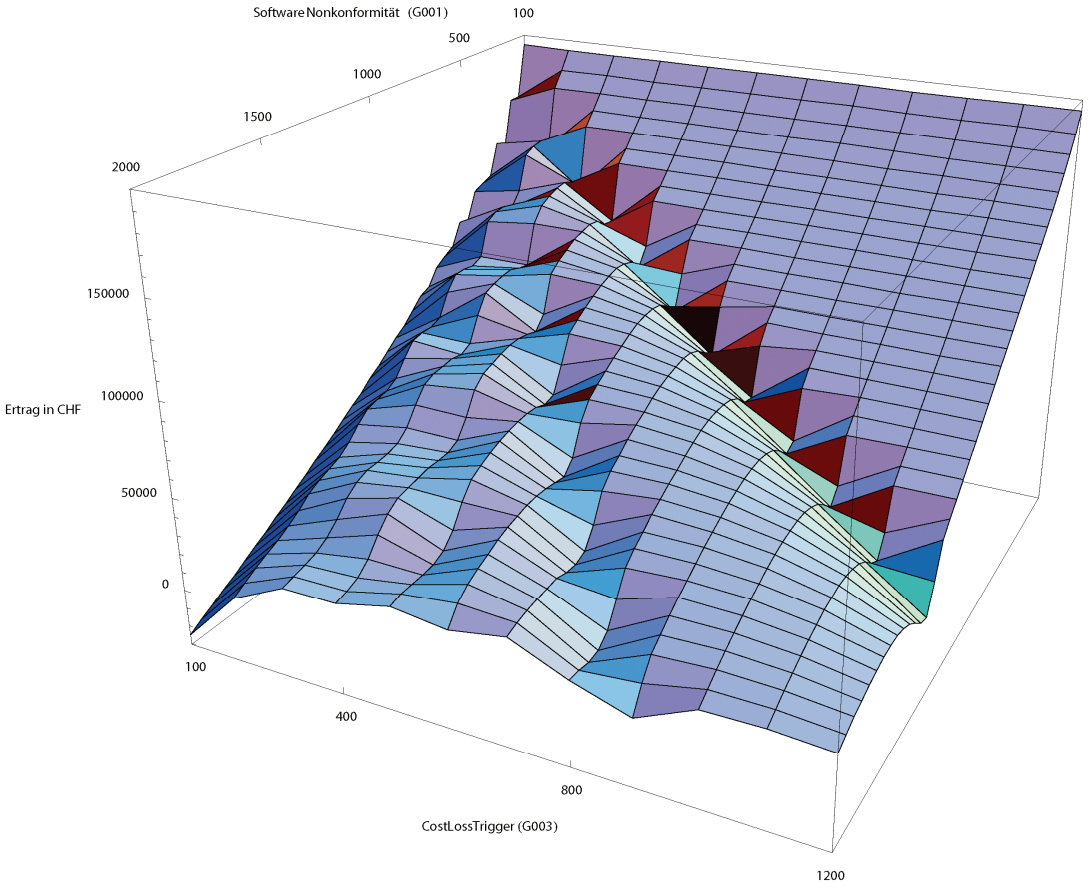


Abbildung 36: Oberflächendiagramm des Ertrages in Abhängigkeit weiterer Parameter

Die dreidimensionale Grafik gibt einen guten Gesamtüberblick über die gewonnenen Ergebnisse. Um den numerischen Wert für den Ertrag besser ablesen zu können, soll ein Teil der Daten noch zweidimensional dargestellt werden, gewissermassen als Schnitte durch die Oberfläche.

Zwecks besserer Übersichtlichkeit wird ein Teil der Datenreihen (in der Dimension der „CostLoss-Trigger“) weggelassen. Ungefähr die Hälfte der Datenreihen für den Parameter „CostLossTrigger“ wird als einzelne Kurven mit verschiedenen Farben gegenüber „Ertrag“ und „Software-Nonkonformitätskosten“ aufgezeichnet (Abbildung 37). In der Legende ist abzulesen, welchen Wert der Parameter des „CostLossTrigger“ in der Schnittebene aufweist.

[20] Mathematica ist ein kommerzielles Softwarepaket der Firma Wolfram Research und stellt eines der meistbenutzten mathematisch-naturwissenschaftlichen Programmpakete dar.

Die numerischen Ergebnisse dieser Simulation finden sich im Anhang Kapitel „9.4.2. Mehrere Durchläufe mit Variation zweier Simulationsparameter“. Der Quellcode, der zu diesen Daten führt, findet sich in der Klasse „com.x8ing.mc.Main#resultRunChangeSoftwareNonConformityCostAndTriggerValue“.

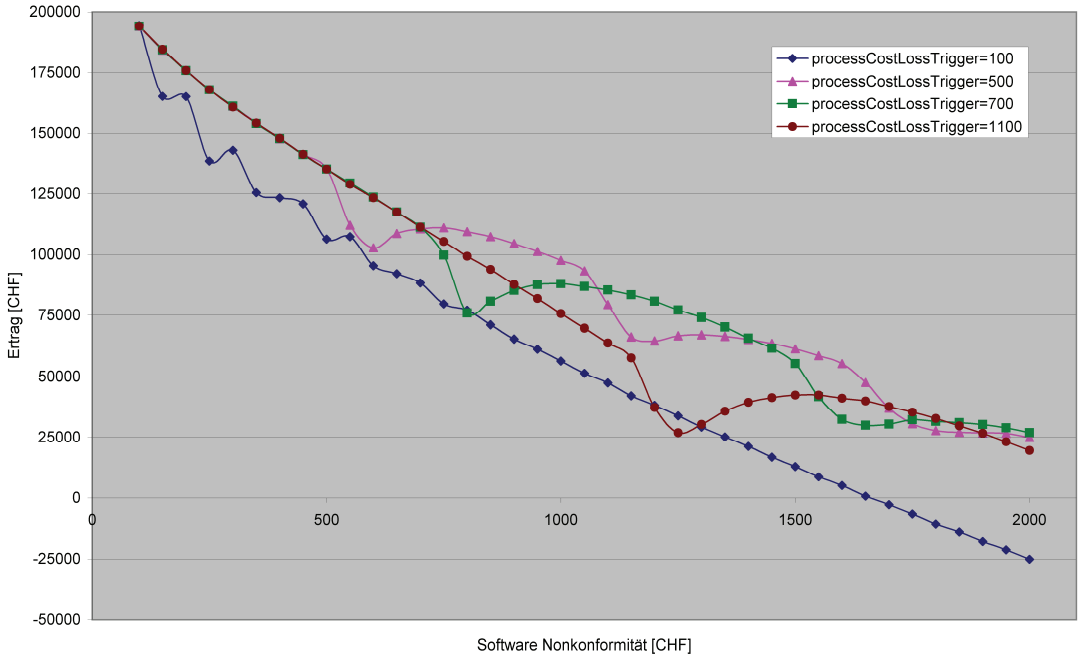


Abbildung 37: Ertrag in Abhängigkeit weiterer Parameter („G001“ und „G003“)

5.4.2 *Betriebswirtschaftliche Betrachtung*

Der Parameter „CostLossTrigger“ definiert gemäss Spezifikation (Kapitel „3.5.1. Detaillierte Spezifikation des Business-Prozesses“) einen Schwellenwert, den die Software-Nonkonformitätskosten überschreiten müssen, damit die Entscheidung gefällt wird, die Fehler zu beheben und ein neues Release der Software zu liefern. In anderen Worten, es wird anhand dieses Parameters entschieden, wie oft und ab welchem Zeitpunkt ein Release gemacht werden soll, bzw. ob Fehler (vorläufig) akzeptiert oder (sofort) behoben werden sollen.

Es stellt sich die Frage, ab welchem Schwellenwert der Entscheid zur Korrektur getroffen werden soll, denn der Business-Prozess gibt vor, dass eine Korrektur eine ganze Reihe von Prozessschritten (z. B. Korrigieren, Testen, Ausliefern und Dokumentieren) und damit verbundenen Kosten auslöst. Die so verursachten Kosten müssen von der Wertschöpfungssteigerung getragen werden können, sonst ist ein Entscheid zur Korrektur finanziell nicht lohnend.

Die Antwort auf diese Frage findet sich in den zwei vorangehenden Grafiken. Wie sich in der Oberflächengrafik in Abbildung 36 erkennen lässt, kann eine ungünstige Wahl des Parameters „CostLossTrigger“ zu einem negativen Ergebnis führen. Die zweidimensionalen Schnitte (Abbildung 37) zeigen, dass der kleinste Wert (100 CHF) fast immer zu einem schlechteren Ergebnis führt als beispielsweise der maximale Wert (1'100 CHF).

Dem Ertragsoptimum nahe kommt ein Parameterwert für den „CostLossTrigger“ zwischen 500 und 700 CHF. Welcher exakte Wert die besten Ergebnisse liefert, hängt wiederum von der gelieferten Software-Nonkonformität ab. Kleine „CostLossTrigger“-Werte (kleiner als 300 CHF) sind zu vermeiden. Dies bedeutet, dass es aus finanzieller Sicht effizienter ist, eine hohe Software-Release-Frequenz zu vermeiden. Es ist also kostengünstiger, die Benutzer kleine Fehler für eine gewisse Zeit hinnehmen zu lassen, als sie jeweils sofort zu beheben.

5.4.3 *Laufzeitverhalten der Simulation*

Die Simulation benötigt auf einem Intel Pentium IV 3.0 GHz System mit einer Java JRE 6 im Server-Modus ungefähr 12 Stunden.

6. Diskussion

In diesem Kapitel werden die gewonnenen Ergebnisse kritisch hinterfragt und es wird überprüft, ob die Fragestellung beantwortet werden konnte.

6.1 Zur Umsetzbarkeit der Monte-Carlo-Methode in Java-Code

Effizienz der Umsetzung
In der Besprechung der Ergebnisse (Kapitel „5.1 Überlegungen zur Implementierung der Monte-Carlo-Methode in Java-Code“) und im Kapitel zur Implementierung (Kapitel „4 Implementierung der Monte-Carlo-Methode“) konnte gezeigt werden, dass die Monte-Carlo-Methode für das Modellunternehmen erfolgreich implementiert werden konnte. Wie im Resultatteil erwähnt, sind dabei keine Probleme aufgetaucht.

Der Zeitaufwand für die Umsetzung ist dennoch beträchtlich, insbesondere unter Berücksichtigung der Ausgangslage. So ist die Komplexität des entworfenen Modellunternehmens sicher erheblich geringer als die eines realen Unternehmens. Es bleibt offen, wie hoch der Aufwand wäre, die Monte-Carlo-Methode für einen realen, komplexen Business-Prozess zu implementieren.

Es ist aber davon auszugehen, dass bei einer erneuten Implementierung bei ähnlicher Ausgangssituation das State-Machine-Framework und eventuell gewisse Elemente der applikatorischen Monte-Carlo-Hauptkomponente wieder verwendet werden könnten. Dieser Umstand würde sich positiv auf die zukünftige Entwicklungszeit auswirken.

Ebenfalls zu einer Reduktion der Entwicklungszeit könnte die Anschaffung einer dedizierten Software wie Crystalball beitragen. Es war Ziel dieser Arbeit, die Implementierung selbst durchzuführen, um daraus weitergehende Schlüsse ziehen zu können. Möchte aber ein reales Unternehmen die Monte-Carlo-Methode anwenden, dann sollte zuerst der Entscheid für oder gegen einen Eigenbau gefällt werden. Die Anschaffung einer dedizierten Software kann sich für ein Unternehmen lohnen, wenn dadurch eigene Entwicklungszeit einspart werden kann. Wie bei jeder Evaluation von Standardsoftware kontra Eigenbau müssten bei einem potenziellen Entscheid aber nicht nur die Kosten betrachtet werden, sondern auch weitere Faktoren. So müsste abgeklärt werden, ob die zu kaufende Software aus dem Blickwinkel der Architektur und des Betriebs in die strategische Systemlandschaft passen würde. Im Falle von Crystalball müsste entschieden werden, ob ein Excel Add-In verwendet werden soll. Weitere mögliche Fragestellungen wären: Möchte sich das Unternehmen in Abhängigkeit von einem externen Lieferanten begeben? Ist die eingekaufte Software genügend auf die eigenen Bedürfnisse anpassbar?

Laufzeitaspekte
In Kapitel „5.1.1 Allgemeines Laufzeitverhalten der Applikation“ wird erklärt, dass die Laufzeit einer Simulation mit mehreren Parametern mehrere Stunden oder gar Tage dauern kann. Es ist vorstellbar, dass eine derart hohe Laufzeit für gewisse Anwendungsfälle kritisch ist.

Da sich die Implementierung der Monte-Carlo-Methode sehr gut parallelisieren lässt, ist eine Verteilung der Simulation auf verschiedene Rechnersysteme ein denkbarer Optimierungsansatz. Diese Verteilung wäre verhältnismässig einfach machbar, da die einzelnen Durchläufe voneinander unabhängig durchgerechnet werden können. Auf jedem Rechnersystem könnte somit eine gewisse Zahl von Iterationen durchlaufen werden. Die Resultate würden auf einem zentralen Rechner gesammelt und die Ergebnisse ausgewertet. Eine weitere Optimierungsmöglichkeit besteht in der Performance-Optimierung des bestehenden Codes. Des Weiteren wird Rechenleistung immer günstiger, da die Hauptrecheneinheit immer leistungsfähiger wird (Moore’s Law).

Es wurde dargelegt, dass die technische Umsetzung der Monte-Carlo-Methode für diese Arbeit keine Probleme bereitete. Weiter darf angenommen werden, dass bei der Anwendung auf ein reales Unternehmen ebenfalls nicht mit Problemen technischer Natur zu rechnen wäre. Weitere wirtschaftliche und theoretische Überlegungen zum Einsatz der Monte-Carlo-Methode in einem realen Unternehmen finden sich im späteren Kapitel „6.5. Umsetzbarkeit auf ein reales Unternehmen".

6.2 Diskussion des Einflusses der Software-Nonkonformität

Für das Modellunternehmen hat die Monte-Carlo-Simulation eine genaue Aussage darüber geliefert, wie sich eine gelieferte Software-Nonkonformität auf den Ertrag auswirkt.

Aus Sicht des Risikomanagements liefert das Histogramm mit den Auftretenswahrscheinlichkeiten der verschiedenen Erträge (Abbildung 33) eine gute Basis zur Einschätzung der möglichen Gefahren und Chancen. Es kann abgeschätzt werden, mit welcher Wahrscheinlichkeit mit einem Gewinn gerechnet werden kann. Das Histogramm kann aber auch aufzeigen, in welchen Fällen mit einem Verlust gerechnet werden muss.

Das Histogramm zeigte eine hohe Streuung, d. h., das 95%-Sicherheitsintervall des Ertragswertes bewegt sich zwischen 96'000 und 173'000 CHF. Dies zeigt den Vorteil gegenüber der klassischen Planung, in der meist nur mit einem Erwartungswert (z. B. dem Durchschnitt) gerechnet wird. Das Ergebnis zeigt, dass für das Modellunternehmen (mit den gegebenen Parametern) mit einer hohen Streuung gerechnet werden muss. Aus planerischer Sicht wäre es wünschenswerter, wenn eine kleinere Streuung resultiert hätte. Die Planung könnte dann einfacher erfolgen, da der zu erwartende Ertrag präzis bekannt wäre. Die Information, die aus dem Histogramm gewonnen werden kann, ist dennoch wertvoll. Der Informationsvorteil liegt gerade im Wissen, dass der zu erwartende Ertrag volatil sein wird.

Weiter wurde der Einfluss von verschiedenen Software-Nonkonformitäten auf den Ertrag untersucht. Das Ergebnis war komplex, d. h., die Ertragkurve war wellenförmig und zeigte teilweise exponentielles Verhalten (siehe Abbildung 34). Es konnte gezeigt werden, dass unter Umständen bereits eine geringfügige Veränderung der Software-Nonkonformität grosse Änderungen des Ertrages zur Folge hat.

Somit lässt sich die zweite Fragestellung positiv beantworten. Es lassen sich quantitative und qualitative Aussagen über die finanziellen Auswirkungen durch die Software-Nonkonformität machen. Weiter wurde dargelegt, dass das Risikomanagement mittels Monte-Carlo-Simulation verbessert werden kann, da eine genauere Einschätzung des zukünftigen Verlaufs des Projektes gemacht werden kann.

6.3 Ableitung weiterer wirtschaftlich relevanter Schlüsse

In Kapitel „5.4 Ableitung weiterer wirtschaftlich relevanter Schlüsse" wurde aufgezeigt, wie sich der Ertrag in Abhängigkeit von Software-Nonkonformität und weiteren Parametern verhält. Das Ergebnis wurde unter anderem als dreidimensionaler Raum dargestellt (Abbildung 36). Die dem Resultat zugrunde liegende Fragestellung zielte darauf ab, mittels Monte-Carlo-Methode eine Ertragsverbesserung durch Optimierung des Business-Prozesses zu erreichen.

Es konnte gezeigt werden (Abbildung 37), dass der Ertrag nicht nur von der Software-Nonkonformität abhängt, sondern ebenfalls vom „CostLossTrigger". Dieser Parameter legt fest, in welchem Rhythmus Veränderungen an der Software vorgenommen werden, und steuert somit den Release-Zyklus der Software. Der Prozess kann also durch einen optimalen Release-Zyklus ressourcenschonender gestaltet werden, was wiederum eine Steigerung des Ertrages zur Folge hat.

Dies legt die Vermutung nahe, dass der Ertrag noch von weiteren Faktoren abhängt. Die Webapplikation zum Testen der Monte-Carlo-Anwendung (siehe Anhang „9.2 Diplomarbeit-Webseite und Demo-

Webapplikation“) kann benutzt werden, um diese These zu untermauern, indem die Simulation immer wieder gestartet wird, mit jeweils genau einem modifizierten Parameter, was die These zumindest heuristisch bestätigt.

Dieser Umstand ist betriebswirtschaftlich interessant, denn er ermöglicht es, den Ertrag weiter zu optimieren. In kurzer Zeit können verschiedene Szenarien virtuell durchgespielt werden. Das so erarbeitete Wissen kann in der Realität genutzt werden, um ertragsoptimierende Massnahmen zu ergreifen, die aufgrund von Erkenntnissen aus der Simulation gewonnen wurden.

Neben potenziellen Ertragsoptimierungen können aber noch weitere Informationen aus der Simulation gewonnen werden. Beim Durchlaufen der Simulation wird das Unternehmen virtuell simuliert. Dabei können beliebige Daten gesammelt werden. Der Ertrag ist nur einer dieser Daten. In der Simulation werden die Budgetposten ebenfalls detailliert aufgestellt. Dies bedeutet, dass der Projektleiter bereits vorab einen Eindruck bekommt, wie viel Kosten wo genau anfallen.

Oftmals sind Kosten gleichbedeutend mit dem geleisteten Arbeitsaufwand, also dem Arbeitseinsatz von Menschen. Ressourcenplanung ist ein wichtiger Teilbereich im Projekt-Management. Die Daten aus der Monte-Carlo-Simulation können dabei unterstützend wirken. Im Beispiel des Modellunternehmens existiert ein Budgetposten „Deployment“ (siehe Kapitel „3.5.1. Detaillierte Spezifikation des Business-Prozesses“). Angenommen, der Budgetposten des „Deployment“ würde sehr hoch, dann liesse sich daraus ein erhöhter Personalbedarf im „Deployment-Team“ ableiten. Eine frühzeitige Allokation der benötigten Ressourcen wäre somit für die Planung unerlässlich und führte zu einer besseren, konstanteren Auslastung der Mitarbeiter und vermindert unvorhergesehene Auslastungsspitzen.

Es lässt sich also zusammenfassen, dass aus der Monte-Carlo-Methode weitere betriebswirtschaftliche Schlüsse gezogen werden können. Diese umfassen zum einen die Optimierung bestehender Prozesse. Des Weiteren ist es denkbar, mittels Monte-Carlo-Simulation Planungen zu verbessern.

6.4 Beurteilung des Modellunternehmens

In diesem Abschnitt wird der Rahmen des Modellunternehmens und die Art der Simulation kritisch hinterfragt.

Bei der Beschreibung des Modells wurde der Rahmen sorgfältig abgesteckt und Faktoren wie Imageverlust, personelle Umstände etc. (siehe Kapitel „3.4.3. Antiscope des Modells“) ausgeschlossen.

Alle ausgeschlossenen Faktoren sind schwer quantifizierbar, die Punkte sind aber nicht unwichtig. Im Gegenteil, eine gute Mitarbeitermotivation kann einen entscheidenden Erfolgsfaktor für ein Unternehmen darstellen. Sollten die Mitarbeiter durch schlechte Software bei ihrer Arbeit einen Motivationsverlust erleiden, wäre die Auswirkung gross. Eine vorstellbare Wirkungskette wäre: Schlechte Software führt zu Ineffizienz, diese zu Frustration, was wiederum zu Kündigungen und somit Wissensabfluss führt. Selbst diese nicht abschliessende Kette zeigt die möglichen kapitalen Folgen auf.

Es bleibt somit offen, wie stark sich die Abwesenheit dieser Faktoren des Antiscope auf das Simulationsresultat auswirkt. In anderen Worten: Das verwendete Modell ist unvollständig und es ist unklar, wie sehr diese Lücke das erhaltene Ergebnis verfälscht.

Ein weiterer kritischer Punkt ist der zeitdiskrete Ansatz der Simulation. Es kann sein, dass Entscheidungen in kürzerer Zeit als einem Tag getroffen werden müssen. Grundsätzlich sieht die Implementierung dies vor, da die Simulationszeit beliebig schnell oder auch langsam voranschreiten kann. Um die Komplexität in Grenzen zu halten, wurde aber ein Tag als minimale Zeit gewählt. Es bleibt offen, inwiefern sich dieser Tagesrhythmus auf die Qualität der Vorhersage auswirkt.

Weiter gibt es bei einem Prozess folgende mögliche Grössen, die den Ablauf charakterisieren: Durchlaufzeit, Liegezeit, Rüstzeit, Arbeitszeit. Das Modellunternehmen berücksichtigt nur die Durchlaufzeit. Aus Sicht der Implementierung wäre es einfach, die weiteren Aspekte zu implementieren. Eine Variante wäre das Einführen von zusätzlichen Zuständen mit Scheinübergängen, die z. B. eine Liegezeit simulieren. Auch hier bleibt offen, inwiefern das Verwenden dieses vereinfachten Prozessmodells die Qualität der Vorhersage beeinflusst.

6.5 Umsetzbarkeit auf ein reales Unternehmen

In dieser Arbeit wird ein Modellunternehmen benutzt, um die Monte-Carlo-Methode ausführen zu können. Die Gründe dazu finden sich in Kapitel „3.3 Überlegung zur Notwendigkeit eines Modellunternehmen“. Möchte aber ein reales Unternehmen die Monte-Carlo-Methode anwenden, so sind einige Punkte zu beachten.

6.5.1 Abbildung des real existierenden Prozesses

Zur Erstellung eines gutes Modells muss zuerst das eigene Unternehmen analysierst werden, um alle ablaufenden Prozesse genau zu kennen. So kann es vorkommen, dass sich Arbeitsabläufe etabliert haben, die unbekannt und möglicherweise sogar ungewollt sind. Bei allfälligen Entscheidungen, die innerhalb eines Prozesses getroffen werden, müssen des Weiteren die genauen Entscheidungskriterien bekannt und entweder statistisch oder logisch erfassbar sein. Nur so können später mittels der Monte-Carlo-Methode die Abläufe simuliert werden.

In einem Unternehmen werden viele Daten laufend erhoben. Dies kann beispielsweise im Rahmen der Buchhaltung, des Controllings, der Prozessüberwachung oder für strategische Balanced Scorecards geschehen. Es ist aber nicht zwingend gegeben, dass alle vorhandenen Daten an einer Stelle zusammenlaufen und verfügbar sind. Ausserdem ist anzunehmen, dass selbst diese Fülle von Informationen nicht ausreichen wird, um das System komplett als Modell zu beschreiben. Dieser Mangel an Informationen muss also zuerst korrigiert werden. Dies bedeutet, dass Aufwand betrieben muss, um überhaupt die Simulationsgrundlage zu schaffen.

Diese Arbeit lässt offen, inwieweit es möglich ist, in einem realen Unternehmen den existierenden Prozess genau genug abzubilden, um darauf aufbauend die Monte-Carlo-Methode in der Praxis anzuwenden.

6.5.2 Qualität des Modells bei realem Einsatz

Das vorangehend beschriebene Problem beim Erstellen des Modells soll nun vorerst ignoriert werden. Es wird davon ausgegangen, dass ein Modell des Unternehmens erfolgreich erstellt wurde. Es ist eine Eigenheit des Modells, dass es die Wirklichkeit nur zu einem bestimmten Grad abbildet, es bleibt immer ein gewisser Fehler zwischen Modell und Wirklichkeit bestehen. Dies ist auch vertretbar, denn das Ziel eines Modells ist es ja nur, Aussagen in einer gewünschten Genauigkeit machen zu können. Das prognostizierte Ergebnis muss der selbst gesetzten Toleranz genügen.

Dies führt aber zu einer weiteren Schwierigkeit. Soll die Monte-Carlo-Methode eingesetzt werden, dann muss gewährleistet sein, dass das Resultat der Simulation auch wirklich innerhalb der Toleranzgrenzen liegt. Das gesamte System muss also getestet werden. Um ein System testen zu können, muss die Prognose des Systems mit dem tatsächlich eintreffenden Resultat verglichen werden. Dies ist im Fall von Software-Projekten schwierig. Erstens haben insbesondere grosse Software-Projekte eine lange Durchführungsdauer, was ein Testen des Systems zeitlich in die Länge zieht. Zweitens ist jedes Projekt ein einzigartiges Unterfangen. Somit ist selbst eine erfolgreich verifizierte Prognose kein Beweis, dass das System auch beim nächsten Mal ein zuverlässiges Resultat liefern wird.

6.5.3 Überlegungen zur Rentabilität bei einem realen Einsatz

Für den Einsatz der Monte-Carlo-Methode in einem realen Unternehmen gibt es verschiedene Gründe. Die Arbeit hat aufgezeigt, dass sowohl Prognosen über Erfolg oder Misserfolg gemacht als auch Erkenntnisse bezüglich Verbesserungen des Prozesses gewonnen werden können. Beide Optimierungen zielen auf eine Steigerung des Ertrages ab.

Dabei darf aber nicht ausser Acht gelassen werden, dass die Durchführung eines Projektes, das die Optimierung zum Ziel hat, selbst auch wieder Ressourcen kostet, d. h., die Analyse des bestehenden Prozesses, die Implementierung einer Monte-Carlo-Simulation und die Umsetzung der daraus gewonnenen Erkenntnisse benötigen Zeit, Geld und binden Arbeitskräfte. Es muss sichergestellt werden, dass die Einsparungen, die mittels eines Monte-Carlo-Projekts erzielt werden, grösser sind als die Kosten für die Durchführung der Optimierung.

Die entstehenden Grundkosten für den Einsatz einer Monte-Carlo-Simulation scheinen also beträchtlich zu sein. Somit muss das Optimierungspotenzial mindestens so gross sein, damit sich das ganze Unterfangen lohnt. Wahrscheinlich wird es sich nur ein Grossunternehmen bei einem umsatzintensiven, optimierungsträchtigen Prozess leisten können, ein solches Projekt zu starten.

7. SCHLUSSFOLGERUNG UND AUSBLICK

7.1 Schlussfolgerung

In dieser Arbeit konnte die Monte-Carlo-Methode erfolgreich auf ein Modellunternehmen angewendet werden. Die Implementierung in Java war zeitaufwendig, aber ohne Probleme durchführbar. Mit Hilfe dieser Umsetzung konnten quantitative und qualitative Aussagen über die finanziellen Auswirkungen von gelieferter Software-Nonkonformität gemacht werden. Ausserdem liessen sich weitere, wirtschaftlich relevante Schlüsse aus der Simulation gewinnen: Es liess sich herleiten, in welchem optimalen Rhythmus Korrekturen an der Software vorgenommen und eine Auslieferung der Software stattfinden sollen.

Es bleibt offen, inwiefern sich der gewählte Ansatz auf ein beliebiges Unternehmen anwenden lässt. Es wird vermutet, dass eine Hauptschwierigkeit darin besteht, einen real existierenden Prozess und seine Auswirkungen vollständig modellieren zu können, um die Basis für die Monte-Carlo-Simulation zu schaffen.

7.2 Weitergehende Forschungsmöglichkeiten

7.2.1 Überprüfung der Anwendbarkeit auf einen real existierenden Prozess

In der Arbeit wurde nicht gezeigt, ob und wie sich die Monte-Carlo-Methode auf einen real existierenden Prozess anwenden lässt. Denkbar wäre eine weiterführende Arbeit, auf der auf theoretischer Basis untersucht wird, ob sich real existierende Prozesse mit genügender Präzision modellhaft abbilden lassen. Alternativ könnte in Form eines Feldversuches an einem praktischen Beispiel die Anwendbarkeit exemplarisch ausprobiert werden.

7.2.2 Automatische Optimierung des Ertrages

In der Arbeit wurde zuerst untersucht, wie der Erfolg der Unternehmen von einem einzigen Parameter (Software-Nonkonformität) abhängt. Die gleiche Frage wurde an zwei Parametern (Software-Nonkonformität, CostLossTrigger) untersucht. Wird dieses Vorgehen weitergedacht, so spricht nichts dagegen, weitere Parameter zu variieren. Betriebswirtschaftlich wäre es interessant, damit den Ertragswert zu maximieren.

Im mathematischen Gebiet der Optimierung gibt es verschiedene Verfahren zur Suche von obig beschriebenen Maxima, wenngleich nach heutigem Forschungsstand keine absolut zuverlässige Methode existiert. Es wäre interessant zu untersuchen, ob es möglich ist, die Monte-Carlo-Methode selbst als System zu betrachten, das Eingabeparameter entgegennimmt und einen Ausgabewert aufweist. Bezogen auf diese Arbeit, wären die Eingabeparameter der Parameterkatalog (siehe Kapitel „3.5.1 Detaillierte Spezifikation des Business-Prozesses“) und die Ausgabevariable der Ertrag, den es zu optimieren gilt. Dieses System gälte es als Ganzes einem Optimierungs-Algorithmus (z. B.: „hill climbing“, „great deluge algorithm“, „simulated annealing“ „stochastic tunneling) zu übergeben, der dann entsprechend die Eingabeparameter selbstständig einstellt.

Wäre eine solche Optimierung möglich, wäre eine auf Monte-Carlo-Simulation basierende, automatisierte Prozess- und Ertragsoptimierung erreicht.

8. QUELLENVERZEICHNIS

8.1 Referenzen zu Literatur

Delhees, Maximilian; Scheuring, Johannes: IT-Projekte planen und initialisieren. Zürich: Compendio Bildungsmedien AG 2004.

Frey, Herbert: Monte-Carlo-Simulation, München: Gerling Akademie Verlag 2001.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: Design Patterns. Boston u. a.: Addison-Wesly 1995.

Hetzel, Bill: The Complete Guide to Software Testing (2nd Edition). Wellesley (MA): QED Information Sciences Inc 1988.

Laird, Linda M.; Brennan, Carol M.: Software Measurement and Estimation. New Jersey: IEEE Press 2006.

Ledin, Jim: Simulation Engineering, Lawrence: CMP Books 2001.

Loveland, Scott u. a.: Software Testing Techniques. Massachusetts: Charles River Media 2005.

Manno, Istvan: Introduction to the Monte-Carlo-Method. Budapest: Akademiai Kaido 1999.

Mooney, Christopher Z: Monte-Carlo-Simulation. West Virginia: West Virginia University Sage Publications 1997.

Myers, Glenford J.: The Art of Software Testing (2nd Edition). New York: John Wiley & Sons 2004.

Pengelly, James: ITIL Foundations. London: GTSLearning 2004.

Ross, Sheldon M.: Simulation (3rd Edition). California: Department of Industrial Engineering University of Berkeley Academic Press 2002.

Slaughter, Sandra A.; Harter E. Donald; Krishnan S. Mayuram: Evaluating the Cost of Software Quality. Paper der ACM Press (Association for Computing Machinery). Carnegie Mellon University 1998.

Smith, Preston G.; Merritt, Guy M.: Proactive Risk Management. New York: Productivity Press 2002.

Wagner, Stefan; Seifert, Tilman: Software Quality Economics for Defect-Detection Techniques Using Failure Prediction. Paper der Technischen Universität München 2005.

Wanka, Rolf: Approximationsalgorithmen. Wiesbaden: B. G. Teubner Verlag 2006.

Watkins, John: Testing IT, An Off-the-Shelf Software Testing Process. New York: Cambridge University Press 2001.

8.2 Referenzen zu Webseiten

[W1] P. Heusser, „Monte Carlo“ (umfasst Demo-Applikation, den ganzen Quellcode und die elektronische Form dieser Arbeit), 2008, http://www.x8ing.com/resources/work/thesis2007.html

[W2] IEEE, „Information for Authors“, 2006, http://www.ieee.org/portal/cms_docs/pubs/transactions/auinfo03.pdf

[W3] Wikimedia Foundation Inc, „Monte-Carlo-Simulation“, 2008, http://de.wikipedia.org/w/index.php?title=Monte-Carlo-Simulation&oldid=39924693

[W4] Object Management Group, „UML“, 2008, http://www.uml.org/

[W5] Universität St. Gallen, „St. Galler Management-Modell“, 2007, http://www.ifb.unisg.ch/org/IfB/ifbweb.nsf/wwwPubInhalteGer/St.Galler+Management-Modell

[W6] Oracel Inc, „Crystal Ball Predictive Modeling Software and Services“, 2008, http://www.crystalball.com/

[W7] Oracle Inc, „Oracle and Hyperion“, 2007,. http://www.oracle.com/hyperion/index.html

[W8] Source-Forge Net., „LSM4J“ (Webseite mit dem State-Machine-Framework, welches für die Arbeit erstellt wurde), 2007, http://sourceforge.net/projects/lsm4j

[W9] Cern, „Colt: Open Source Libraries for High Performance Scientific and Technical Computing in Java“, 2004, http://spi.cern.ch/extsoft/packages.php?pkg=Colt und http://dsd.lbl.gov/~hoschek/colt/

[W10] Apache Software Foundation, „Apache Struts: Welcome“, 2008, http://struts.apache.org/

[W11] Object Refinery Limited, „JFree Chart“, 2008, http://www.jfree.org

[W12] Apache Software Foundation, „Apache Tomcat“, 2008, http://tomcat.apache.org

[W13] M. Gebhard, „Java2Html converter“, 2006, http://www.java2html.de

9. ANHANG

9.1 Herleitung der Formel zur Berechnung von PI mittels der Monte-Carlo-Methode

Im Kapitel „2.4.2 Ein einfaches Anwendungsbeispiel: Berechnung der Zahl PI“ wird die Kreiszahl PI mittels Monte-Carlo-Methode berechnet. Nachfolgend werden die dazu benötigte Formel und deren Herleitung erläutert. Die Ausgangslage für das Beispiel ist dem referenzierten Grundlagenkapitel zu entnehmen.

Es werden virtuelle „Pfeile“ nach dem Zufallsprinzip auf das Quadrat mit eingeschriebenem Viertelkreis geworfen (Abbildung 38). Einige Pfeile werden somit innerhalb des Viertelkreises (in der Grafik: rote Dreiecke), andere ausserhalb auftreffen (graue Dreiecke). Die Summe aller roten Pfeile wird mit „p“, diejenige der grauen mit „q“ bezeichnet.

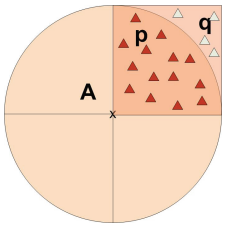


Abbildung 38: Virtuelle Pfeile innerhalb oder ausserhalb des Kreisbogens

In einem zweiten Schritt werden einerseits die geworfenen Pfeile und andererseits die Zielflächen in ein Verhältnis gebracht. Das erste Verhältnis „V1“ ist dabei ein Verhältnis zwischen Pfeilen, nämlich allen Pfeilen p zur gesamten Anzahl aller Pfeile, die geworfen wurden, also „p+q“. Dies führt zu „V1“:

$$V1 = \frac{p}{p+q}$$

Das Verhältnis der Flächen „V2“ wird genau über den gleichen Gebieten gebildet, wie dies bei den Pfeilen getan wurde, nämlich der Fläche des Viertelkreises zu der Fläche des Quadrates. Dabei leitet sich die Fläche des Viertelkreises aus der Kreisfläche ab. Die Fläche des Quadrates ist bekanntlich r^2.

$$V2 = \frac{\frac{1}{4} r^2 \pi}{r^2}$$

Aufgrund der zufälligen Verteilung der einzelnen Pfeile werden sich die beiden Verhältnisse „V1“ und „V2“ nähern, je mehr Pfeile eintreffen. Daher können die Verhältnisse in einer Gleichung gegenübergestellt werden. Somit kann geschrieben werden:

$$V1 = V2$$

In diese Gleichung werden nun die obigen Werte für V1 und V2 eingesetzt. Dies führt zu:

$$\frac{p}{p+q} = \frac{\frac{1}{4} r^2 \pi}{r^2}$$

Die Formel kann vereinfacht und nach PI aufgelöst werden. Dies ergibt die gewünschte Formel. Bei der Formel fällt auf, dass sie nur noch „p“ und „q“ enthält, also die Anzahl Pfeile in der jeweiligen Fläche.

$$\pi = \frac{4p}{p+q}$$

Abbildung 39: Formel zur Approximation von PI mittels Monte-Carlo-Methode

9.2 Diplomarbeit-Webseite und Demo-Webapplikation

Im Zuge der Diplomarbeit wurde eine auf Struts[21] [W10] basierende interaktive Webapplikation entwickelt, welche die praktische Umsetzung der Diplomarbeit zeigt. Das in dieser Arbeit beschriebene Modellunternehmen (Kapitel „3.4. Erarbeitung des Business-Prozesses im Modell“) kann so mittels Monte-Carlo-Methode simuliert werden. Es lassen sich alle Eingabeparameter für die Simulation setzen, aufgrund deren die Simulation anschliessend durchgerechnet wird. Das Resultat wird auf unterschiedliche Arten dargestellt. Unter anderem wird eine Histogramm-Grafik erzeugt, welche die Risikoverteilung für den Simulationsdurchlauf zeigt. Nachfolgend werden die wichtigsten GUIs der Webapplikation beschrieben.

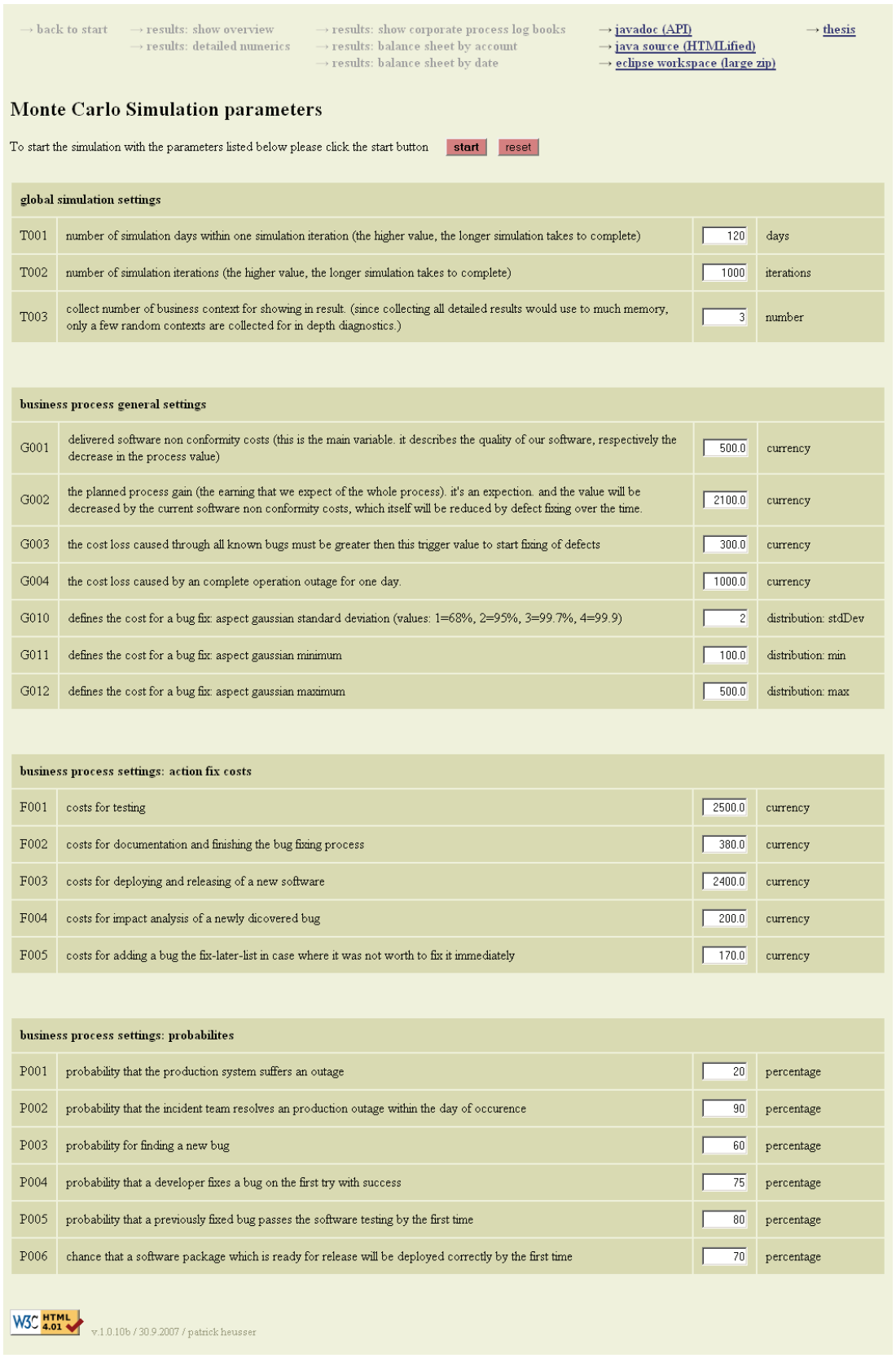
9.2.1 Startseite

Beim Betreten der Webapplikation wird die Startseite angezeigt (Abbildung 40). Im oberen Bereich findet sich die Navigation, die auf allen Seiten der Applikation angezeigt wird. Die Navigation erfolgt mittels Links. Einige dieser Links sind beim Betreten inaktiv (grau) und werden erst aktiv, wenn die Simulation gestartet wird.

Unterhalb der Navigation wird eine Eingabemaske mit allen Parametern, die für die Monte-Carlo-Simulation gemäss Spezifikation des Business-Prozesses (Kapitel „3.5.1 Detaillierte Spezifikation des Business-Prozesses“) relevant sind, angezeigt. Dabei findet sich die Parameter-Kurzbezeichnung (z. B. „G001“) der ersten Kolonne in der erwähnten Spezifikation wieder.

Durch Anklicken des Buttons „start“ wird der Simulationslauf mit den spezifizierten Parametern gestartet. Die Berechnung kann einige Sekunden in Anspruch nehmen.

[21] Struts ist ein Open-Source-Framework für die Präsentations- und Steuerungsschicht von Java-Webanwendungen.



→ back to start → results: show overview → results: detailed numerics → results: show corporate process log books → results: balance sheet by account → results: balance sheet by date → javadoc (API) → java source (HTMLified) → eclipse workspace (large zip) → thesis

Monte Carlo Simulation parameters

To start the simulation with the parameters listed below please click the start button start reset

global simulation settings			
T001	number of simulation days within one simulation iteration (the higher value, the longer simulation takes to complete)	120	days
T002	number of simulation iterations (the higher value, the longer simulation takes to complete)	1000	iterations
T003	collect number of business context for showing in result. (since collecting all detailed results would use to much memory, only a few random contexts are collected for in depth diagnostics.)	3	number

business process general settings			
G001	delivered software non conformity costs (this is the main variable. it describes the quality of our software, respectively the decrease in the process value)	500.0	currency
G002	the planned process gain (the earning that we expect of the whole process). it's an expection. and the value will be decreased by the current software non conformity costs, which itself will be reduced by defect fixing over the time.	2100.0	currency
G003	the cost loss caused through all known bugs must be greater then this trigger value to start fixing of defects	300.0	currency
G004	the cost loss caused by an complete operation outage for one day.	1000.0	currency
G010	defines the cost for a bug fix: aspect gaussian standard deviation (values: 1=68%, 2=95%, 3=99.7%, 4=99.9)	2	distribution: stdDev
G011	defines the cost for a bug fix: aspect gaussian minimum	100.0	distribution: min
G012	defines the cost for a bug fix: aspect gaussian maximum	500.0	distribution: max

business process settings: action fix costs			
F001	costs for testing	2500.0	currency
F002	costs for documentation and finishing the bug fixing process	380.0	currency
F003	costs for deploying and releasing of a new software	2400.0	currency
F004	costs for impact analysis of a newly dicovered bug	200.0	currency
F005	costs for adding a bug the fix-later-list in case where it was not worth to fix it immediately	170.0	currency

business process settings: probabilites			
P001	probability that the production system suffers an outage	20	percentage
P002	probability that the incident team resolves an production outage within the day of occurence	90	percentage
P003	probability for finding a new bug	60	percentage
P004	probability that a developer fixes a bug on the first try with success	75	percentage
P005	probability that a previously fixed bug passes the software testing by the first time	80	percentage
P006	chance that a software package which is ready for release will be deployed correctly by the first time	70	percentage

W3C HTML 4.01 v.1.0.10b / 30.9.2007 / patrick heusser

Abbildung 40: Startseite der Monte-Carlo-Simulation

9.2.2 *Resultat-Übersicht*

Nach erfolgter Berechnung wird die grobe Resultat-Übersicht (Abbildung 41) mit den wichtigsten Kennzahlen aus dem Modellunternehmen angezeigt:

- Durchschnittlicher finanzieller Erfolg.
- Schlechtestes Resultat, das in der Simulation jemals aufgetreten ist.
- Bestes Resultat, das jemals aufgetreten ist.
- Histogramm mit der Auftretenswahrscheinlichkeit für das jeweilige Resultat.

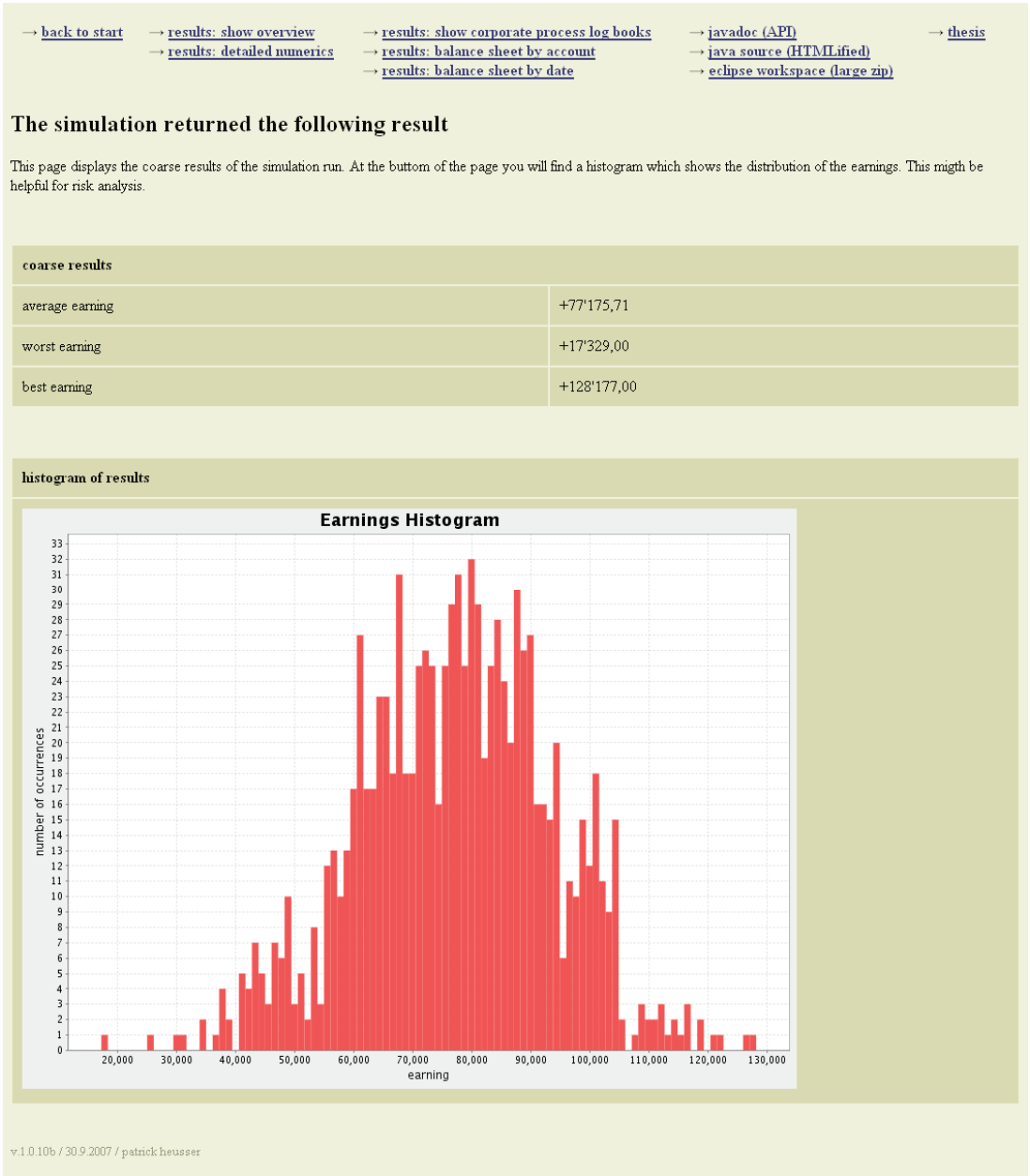


Abbildung 41: Resultat-Übersicht der Simulation

9.2.3 Resultat Detailsicht: Protokoll

Beim Anklicken des Links „results: show corporate process log books“ wird eine Seite mit Details zum Verlauf der Simulation angezeigt. Es ist eine Eigenheit der Monte-Carlo-Simulation, dass ein Vorgang unzählige Male wiederholt wird (Kapitel „2.4.1 Einführung“). In dieser Arbeit entspricht dabei ein Durchgang der Simulation des Modellunternehmens während einer vorgegebenen Anzahl Tage.

In abgebildeten Fall (Abbildung 42) wird das Unternehmen für den Zeitraum von 180 Tagen 1'000 Mal durchgerechnet und die Ergebnisse werden gesammelt. Das gezeigte „Logbook #0“ hält dabei fest, was bei einem zufällig ausgewählten Durchlauf im Unternehmen geschehen ist. Die Anzahl von Logbüchern, die gesammelt werden sollen, ist konfigurierbar. Da die Sammlung dieser Daten relativ viel Hauptspeicher benutzt, ist es selbst mit der heutigen Hardware-Konfiguration nicht möglich, alle Logbücher zu sammeln. Deshalb wird die spezifizierte Anzahl Logbücher per Zufall ausgewählt.

Das Logbuch kann unter anderem dazu dienen, die Plausibilität der implementierten Prozesse zu überprüfen, und hilft, ein Gefühl für die Abläufe in der Simulation zu bekommen.

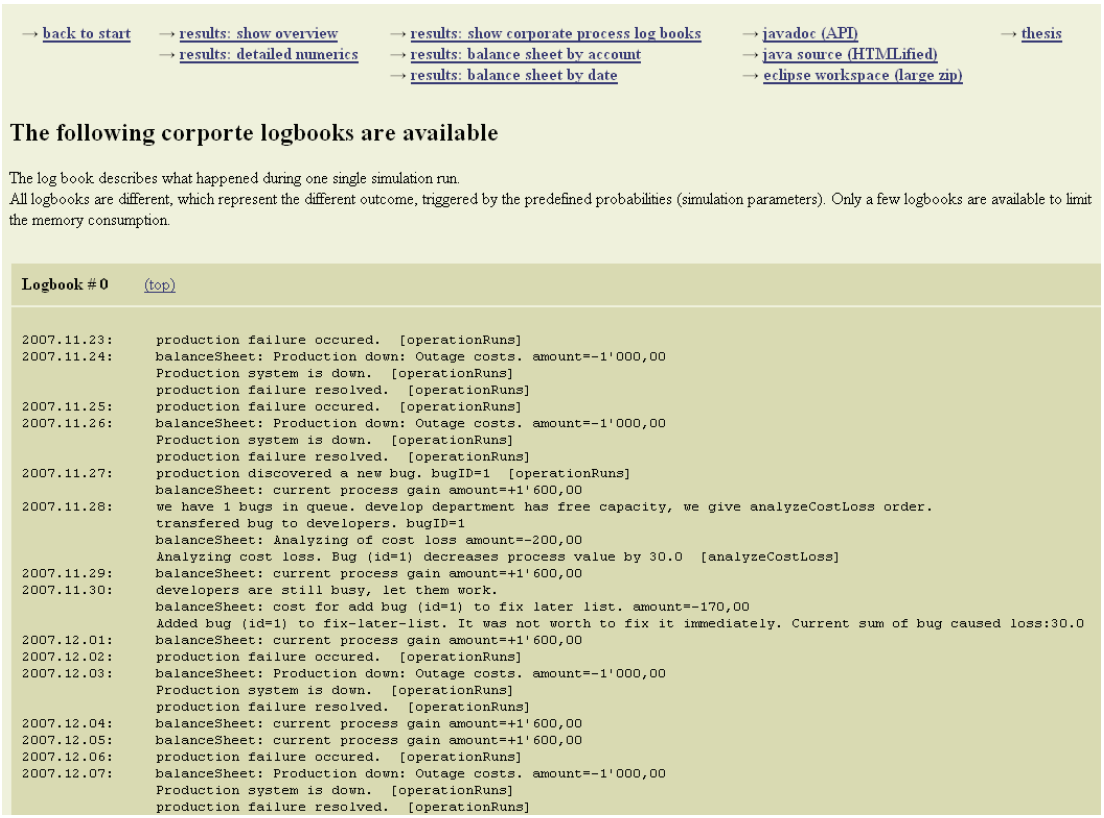


Abbildung 42: Detailansicht über den Verlauf der Simulation

9.2.4 *Resultat Detailsicht: Buchhaltung*

Der Link „results: balance sheet by account“ führt zur finanziellen Detailsicht (Abbildung 43). Das Sammeln von Buchhaltungsergebnissen verhält sich analog zu den Logbüchern. Jeder Endzustand des Modellunternehmens weist einen unterschiedlichen finanziellen Erfolg aus. Dieselben Simulationsdurchläufe, die schon für die Logbücher ausgewählt wurden, werden hier angezeigt, aber diesmal auf ihren finanziellen Erfolg hin ausgewertet. Die Auswertung gibt an, auf welchem Buchhaltungskonto wie viel gebucht wurde. Daraus lässt sich schliessen, wie die finanziellen Mittel investiert wurden.

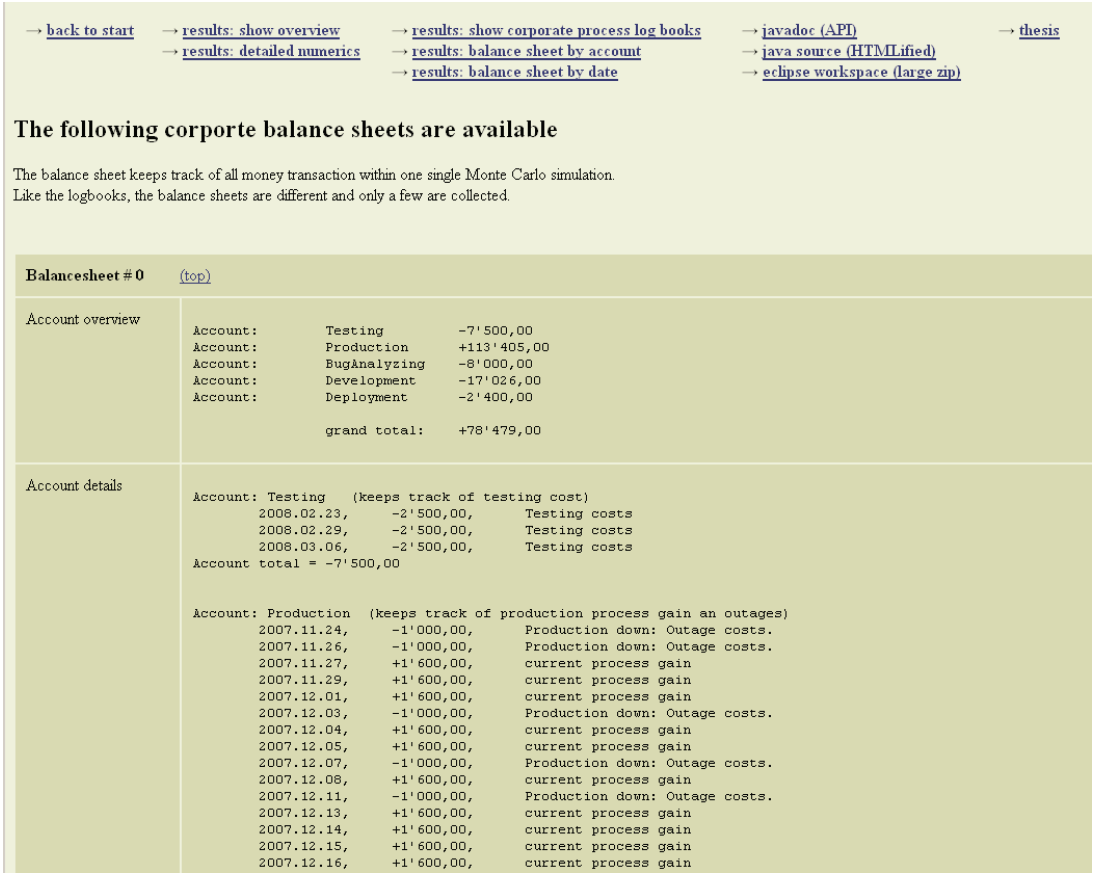


Abbildung 43: Detailansicht über die finanzielle Situation

9.2.5 *Dokumentation: Javadoc*

Ebenfalls auf der Webseite vorhanden (Link „javadoc“) ist die komplette Dokumentation des Quellcodes. Die Dokumentation wurde mittels Javadoc [22] erstellt.

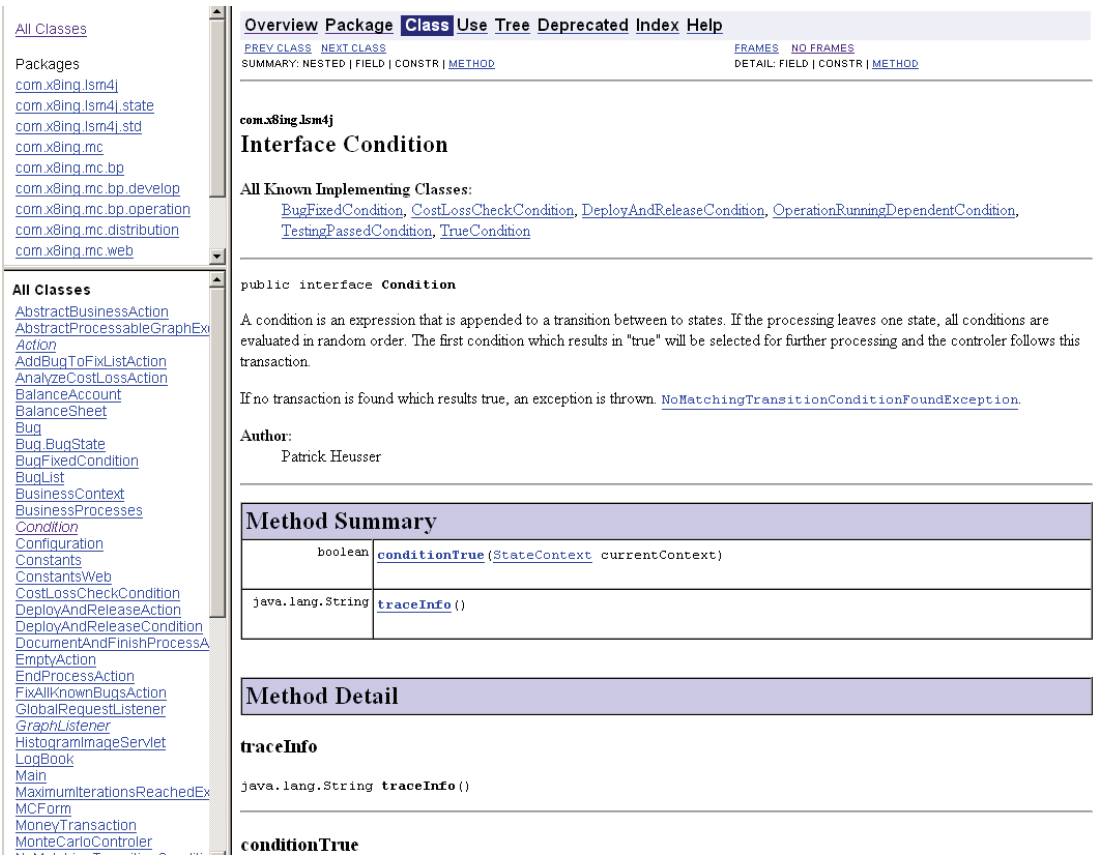


Abbildung 44: Dokumentation des erstellen Quellcodes mittels Javadoc

9.2.6 *Dokumentation: Quellcode*

Der Link „java source“ führt direkt zum Quellcode aller implementierten Klassen. Der Quellcode wurde mittels eines Opensource-Tools namens „Java2Html“ [W13] so aufbereitet, dass er mit Syntax-Highlighting im Browser darstellbar ist.

[22] Javadoc ist ein Software-Dokumentationswerkzeug, das aus Java-Quelltexten automatisch HTML-Dokumentationsdateien erstellt.

GraphListener.java

```
01 /*
02  * Created on May 19, 2007
03  */
04 package com.x8ing.lsm4j;
05
06 import com.x8ing.lsm4j.state.ProcessableGraph;
07 import com.x8ing.lsm4j.state.ProcessableState;
08
09 /**
10  * Will be notified if something happens on the graph.
11  *
12  * The listener must be registred on the observeable graph. see: {@link ProcessableGraph#registerGraphListener(GraphListener)}
13  * <p>
14  * The usage of this interface is optional.
15  *
16  * @author Patrick Heusser
17  */
18 public interface GraphListener {
19
20   // TODO refactor order of params
21   public void startProcessingState(ProcessableState previousState, Condition previosCondition, long loop,StateContext currentStat
22
23   /**
24    * The listener will be notified using this message, if the graph changed it's state. <br>
25    */
26   public void changedState(ProcessableState previousState, ProcessableState currentState, Condition previousCondition,
27       Condition currentCondition, long loop,StateContext currentStateContext);
28
29   // TODO check naming!!! for all..
30   public void foundEndState(ProcessableState endState, long loop, StateContext currentStateContext);
31
32 }
```

Abbildung 45: Möglichkeit der Anzeige des Quellcodes als HTML-Seiten

9.2.7 *Dokumentation: Weitere Download-Möglichkeiten*

Die in der Abbildung 46 markierten Navigationspunkte führen zu einer weiteren Dokumentation des Projekts. Ein Klick auf den Link „eclipse workspace“ startet den Download eines ZIP-Files, das den gesamten Eclipse[23]-Projekt-Workspace beinhaltet. Im Workspace enthalten sind sowohl alle benötigten Bibliotheken als auch die Scripts für den automatisierten Ant[24]-Build. Der Link „Thesis“ führt zum Download dieser Diplomarbeit im Adobe PDF[25]-Format.

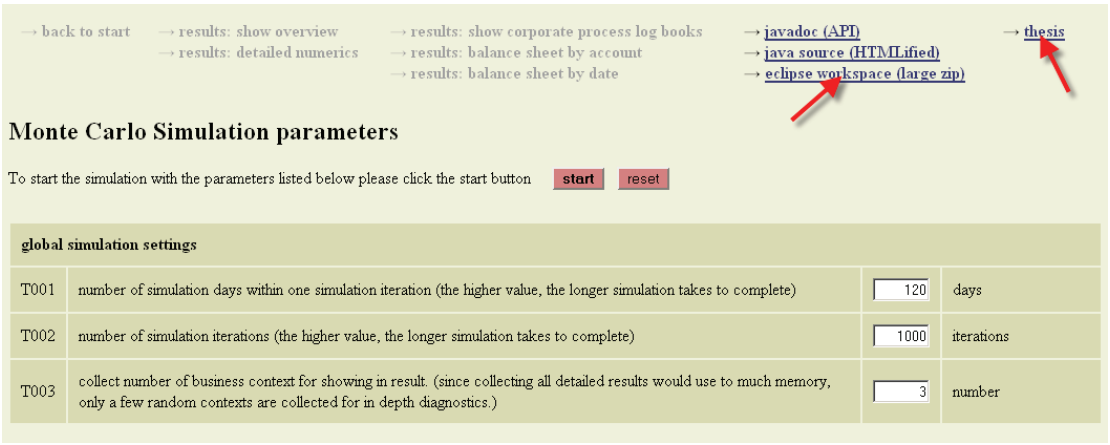


Abbildung 46: Download der gesamten Entwicklungsdaten (ZIP) oder diese Arbeit als PDF

[23] Eclipse ist ein Open-Source-Framework zur Entwicklung von Software nahezu aller Art. http://www.eclipse.org/.

[24] Ant ist ein in Java geschriebenes Werkzeug zur automatisierten Erzeugung von Programmen aus Quelltext.

[25] Das Portable Document Format (PDF) ist ein plattformübergreifendes Dateiformat für Dokumente.

9.3 Beispiel eines Simulationsdurchlaufes

Die nachfolgende Abbildung 47 zeigt einen Simulationsdurchlauf des Modellunternehmens während 180 Tagen. In dieser Zeit wird eine ansehnliche Reihe von Prozessabläufen gemäss der Spezifikation des Unternehmens (Kapitel „3.5.1 Detaillierte Spezifikation des Business-Prozesses“) simuliert. Diese internen Prozessabläufe werden protokolliert (siehe Kapitel „4.4.1 Design“).

Insgesamt wurden in diesem Testlauf über 600 Log-Statements generiert. Die nachfolgende Tabelle enthält einen Teilauszug der Statements. Wenn immer in der ersten Spalte „[CUT]“ steht, bedeutet dies, dass Zeilen entfernt wurden, um die Länge zu kürzen.

[CUT]	
2007.11.30:	we have 1 bugs in queue. develop department has free capacity, we give analyzeCostLoss order.
	transfered bug to developers. bugID=1
	balanceSheet: Analyzing of cost loss amount=-200,00
	Analyzing cost loss. Bug (id=1) decreases process value by 30.0 [analyzeCostLoss]
2007.12.01:	production discovered a new bug. bugID=2 [operationRuns]
	balanceSheet: current process gain amount=+1'600,00
2007.12.02:	developers are still busy, let them work.
	balanceSheet: cost for add bug (id=1) to fix later list. amount=-170,00
	Added bug (id=1) to fix-later-list. It was not worth to fix it immediately. Current sum of bug caused loss:30.0 [addBugToFixList]
2007.12.03:	balanceSheet: current process gain amount=+1'600,00
2007.12.04:	we have 1 bugs in queue. develop department has free capacity, we give analyzeCostLoss order.
	transfered bug to developers. bugID=2
	balanceSheet: Analyzing of cost loss amount=-200,00
	Analyzing cost loss. Bug (id=2) decreases process value by 38.0 [analyzeCostLoss]
2007.12.05:	production discovered a new bug. bugID=3 [operationRuns]
	balanceSheet: current process gain amount=+1'600,00
2007.12.06:	developers are still busy, let them work.
	balanceSheet: cost for add bug (id=2) to fix later list. amount=-170,00
	Added bug (id=2) to fix-later-list. It was not worth to fix it immediately. Current sum of bug caused loss:68.0 [addBugToFixList]
2007.12.07:	production discovered a new bug. bugID=4 [operationRuns]
	balanceSheet: current process gain amount=+1'600,00
2007.12.08:	we have 2 bugs in queue. develop department has free capacity, we give analyzeCostLoss order.
	transfered bug to developers. bugID=3
	transfered bug to developers. bugID=4
	balanceSheet: Analyzing of cost loss amount=-200,00
	Analyzing cost loss. Bug (id=3) decreases process value by 33.0 [analyzeCostLoss]
	balanceSheet: Analyzing of cost loss amount=-200,00
	Analyzing cost loss. Bug (id=4) decreases process value by 9.0 [analyzeCostLoss]
2007.12.09:	balanceSheet: current process gain amount=+1'600,00
[CUT]	
2008.03.04:	developers are still busy, let them work.
	Go to fix all yet known bugs (19) (bugs with states: new, fixLater, productionTestFailed.) [fixAllKnownBugs]
	Fixed bug (id=19). State is now: fixed [fixAllKnownBugs]
	balanceSheet: costs for fix of bug (id=19) amount=-204,00
	Fixed bug (id=1). State is now: fixed [fixAllKnownBugs]
	balanceSheet: costs for fix of bug (id=1) amount=-396,00
	Fixed bug (id=2). State is now: fixed [fixAllKnownBugs]
	balanceSheet: costs for fix of bug (id=2) amount=-348,00
	Fixed bug (id=3). State is now: fixed [fixAllKnownBugs]
	balanceSheet: costs for fix of bug (id=3) amount=-486,00
	Fixed bug (id=4). State is now: fixed [fixAllKnownBugs]
[CUT]	
2008.03.08:	developers are still busy, let them work.
	Started testing [testing]
	balanceSheet: Testing costs amount=-2'500,00
	testing of bug (id=1) failed. Return bug to developers. [testing]
	testing of bug (id=2) sucessful. [testing]
	testing of bug (id=3) failed. Return bug to developers. [testing]
	testing of bug (id=4) failed. Return bug to developers. [testing]

	testing of bug (id=5) sucessful. [testing]
	testing of bug (id=6) sucessful. [testing]
	testing of bug (id=7) sucessful. [testing]
[CUT]	
	balanceSheet: current process gain amount=+1'905,00
2008.04.26:	developers are still busy, let them work.
	balanceSheet: cost for add bug (id=29) to fix later list. amount=-170,00
	Added bug (id=29) to fix-later-list. It was not worth to fix it immediately. Current sum of bug caused loss:74.0 [addBugToFixList]
2008.04.27:	balanceSheet: current process gain amount=+1'905,00
2008.04.28:	we have 1 bugs in queue. develop department has free capacity, we give analyzeCostLoss order.
	transfered bug to developers. bugID=30
	balanceSheet: Analyzing of cost loss amount=-200,00

Abbildung 47: Beispiel der protokollierten Ereignisse während einer Simulation

9.4 Detaillierte numerische Simulationsergebnisse

Die Daten in diesem Abschnitt dienen als Grundlage für die Grafiken des Resultat-Kapitels „5. Resultat“.

9.4.1 Einmaliger Simulationsdurchlauf mit detaillierter Risiko-Ausgabe

Die Abbildung 48 zeigt die Rohdaten für das Risiko-Histogram in Kapitel „5.2 Erfolgsauswirkung einer definierten Software-Nonkonformität“.

Die erste Zeile ist dabei als Kopfzeile zu verstehen. Jede weitere Zeile umfasst die Daten. Aus Platzgründen wurden drei Spalten zur Auflistung verwendet. Die ersten beiden Zahlen einer Zeile beschreiben im Histogramm jeweils den finanziellen Ertrag („earning“) in einem festgelegten Interval (von/bis). Die zweite Zahl zeigt, wie oft die Simulation diesen Ertrag als Endresultat erreicht hat. Die letzte Zahl beschreibt jeweils die Häufigkeit des Auftretens als prozentualer Wert, abgeleitet aus dem vorherigen Wert.

```
rangeFrom   ;rangeTo   ;count   ;percentil
46183;47899;1;0.01
47899;49615;0;0
49615;51331;0;0
51331;53047;0;0
53047;54763;0;0
54763;56478;1;0.01
56478;58194;0;0
58194;59910;1;0.01
59910;61626;2;0.02
61626;63342;1;0.01
63342;65058;4;0.04
65058;66774;0;0
66774;68490;3;0.03
68490;70206;6;0.06
70206;71922;7;0.07
71922;73638;5;0.05
73638;75353;8;0.08
75353;77069;8;0.08
77069;78785;8;0.08
78785;80501;21;0.21
80501;82217;31;0.31
82217;83933;19;0.19
83933;85649;25;0.25
85649;87365;27;0.27
87365;89081;37;0.37
89081;90797;33;0.33
90797;92513;44;0.44
92513;94228;47;0.47
94228;95944;73;0.73
95944;97660;55;0.55
97660;99376;87;0.87
99376;101092;101;1.01
101092;102808;126;1.26
102808;104524;155;1.55
104524;106240;177;1.77
106240;107956;155;1.55
107956;109672;171;1.71
109672;111388;162;1.62
111388;113103;208;2.08
113103;114819;211;2.11
114819;116535;251;2.51
116535;118251;231;2.31
118251;119967;255;2.55
119967;121683;277;2.77
121683;123399;279;2.79
123399;125115;301;3.01
125115;126831;280;2.8
126831;128547;340;3.4
128547;130263;295;2.95
130263;131978;321;3.21
131978;133694;323;3.23
133694;135410;333;3.33
135410;137126;341;3.41
137126;138842;312;3.12
138842;140558;305;3.05
140558;142274;325;3.25
142274;143990;286;2.86
143990;145706;278;2.78
145706;147422;272;2.72
147422;149138;219;2.19
149138;150853;226;2.26
150853;152569;210;2.1
152569;154285;200;2
154285;156001;179;1.79
156001;157717;173;1.73
157717;159433;151;1.51
159433;161149;133;1.33
161149;162865;132;1.32
162865;164581;103;1.03
164581;166297;90;0.9
166297;168013;89;0.89
168013;169728;75;0.75
169728;171444;49;0.49
171444;173160;56;0.56
173160;174876;48;0.48
174876;176592;38;0.38
176592;178308;40;0.4
178308;180024;30;0.3
180024;181740;29;0.29
181740;183456;13;0.13
183456;185172;18;0.18
185172;186888;21;0.21
186888;188603;13;0.13
188603;190319;16;0.16
190319;192035;8;0.08
192035;193751;3;0.03
193751;195467;4;0.04
195467;197183;3;0.03
197183;198899;1;0.01
198899;200615;0;0
200615;202331;0;0
202331;204047;1;0.01
204047;205763;2;0.02
205763;207478;0;0
207478;209194;1;0.01
209194;210910;0;0
210910;212626;0;0
212626;214342;0;0
214342;216058;0;0
216058;217774;1;0.01
```

Abbildung 48: Rohdaten für den Resultatteil

9.4.2 Mehrere Durchläufe mit Variation zweier Simulationsparameter

Die Abbildung 49 zeigt die Datenbasis von mehreren Grafiken in den Kapiteln 5.3 und 5.4. Es wurden zwei Simulationsparameter variiert. Die Abbildung ist als zweidimensionale Tabelle zu lesen, wobei die erste Zeile als Tabellenüberschrift zu interpretieren ist. Jede weitere Zeile enthält das Ergebnis eines Simulationsdurchlaufs (Earning), wobei die einzelnen Durchgänge mittels Semikolon getrennt wurden. Der erste Wert einer Zeile beschreibt jeweils den Parameter „G001“ (siehe Kapitel „3.5.1 Detaillierte Spezifikation des Business-Prozesses“), alle weiteren Zahlen das zugehörige Ergebnis mit dem Parameterwert für „G003“ gemäss der Tabellenüberschrift.

```
-----------------------------------------------------
START: resultRunChangeSoftwareNonConformityCost()
numberOfSimulationDays=180; numberCompleteSimulationLoops=10000
-----------------------------------------------------
softwareNonConfCost,processCostLossTrigger=100,processCostLossTrigger=200,processCostLossTri
gger=300,processCostLossTrigger=400,processCostLossTrigger=500,processCostLossTrigger=600,pr
ocessCostLossTrigger=700,processCostLossTrigger=800,processCostLossTrigger=900,processCostLo
ssTrigger=1000,processCostLossTrigger=1100,processCostLossTrigger=1200;
100,194430,194041,193995,194200,194074,194110,194135,194043,193672,193827,194165,194298;
150,165161,184165,184536,184050,184233,184330,184376,184725,184199,184143,184567,184283;
200,165049,175687,175941,175798,175993,175832,175821,175968,175800,175758,175890,175778;
250,138612,143041,168107,167883,168010,167943,167881,168312,168054,167812,167976,168043;
300,143017,148408,160903,161038,161005,160947,161091,160924,160552,160953,160766,160722;
350,125483,147550,126479,153995,154227,154299,153936,154052,153878,153752,154203,153923;
400,123351,144679,132693,147777,147709,147864,147757,147690,147624,147462,147911,147685;
450,120828,125720,134875,116684,141342,141204,141227,141507,141325,141450,141389,141380;
500,106342,118251,134074,117457,135339,135077,135255,135378,135365,135225,135245,135673;
550,107397,121280,131916,121832,112054,129220,129483,129304,129358,129195,128915,129271;
600,95372,120912,128937,122680,102785,123396,123606,123200,123322,123401,123366,123357;
650,92109,117725,121175,121244,108639,107023,117504,117322,117546,117113,117534,117528;
700,88296,99856,99982,119650,110552,88854,111252,111342,111187,111364,111354,111369;
750,79677,97254,100947,116367,111015,94763,99740,105440,105280,105455,105395,105517;
800,77067,98282,103077,113161,109411,98660,75962,99785,99392,99549,99271,99827;
850,71234,98026,102378,108433,107374,99674,80770,90437,93807,93731,93837,93705;
900,65514,90767,100938,88829,104516,99161,85221,65416,87894,87771,87728,87561;
950,61421,79197,98764,82167,101231,97928,87631,66659,80390,81853,81929,81835;
1000,56488,76802,89080,83947,97599,95407,88002,72194,55584,75857,75746,75625;
1050,51481,76767,76285,84864,93223,92243,86828,75342,52627,69289,69889,70269;
1100,47520,75535,73371,84595,79373,89279,85403,76367,58043,46193,63904,64036;
1150,42143,68710,74509,83117,66337,85614,83454,76425,62488,39273,57791,58031;
1200,38303,60680,73576,81103,64719,81815,80793,75729,64548,44580,37363,52270;
1250,33921,57160,72953,79076,66808,77390,77170,74040,65381,49370,27038,45977;
1300,29317,55862,70351,73778,67167,68557,74110,71522,64827,52071,30461,28828;
1350,25224,54106,62773,62350,66565,53537,70339,68508,63646,53649,35819,14626;
1400,21358,49114,53845,53549,65388,47397,65794,65139,61659,53830,39485,16653;
1450,16923,42901,48910,50766,63740,48422,61747,62119,59511,53024,41426,21868;
1500,13047,38358,47174,50398,61534,49377,55415,58438,56644,51879,42446,26435;
1550,8759,35812,46505,50446,58730,49639,41806,54959,53508,50034,42514,29275;
1600,5181,33435,46169,49168,55434,49202,32524,50431,50360,47693,41211,31005;
1650,808,30074,43733,48052,47640,47795,30082,46414,46666,44705,40060,31191;
1700,-2694,25009,39183,46756,37103,46427,30498,41116,42762,41795,37726,30818;
1750,-6555,20861,32048,42609,30651,44354,32306,29717,39089,38532,35501,29563;
1800,-10653,17224,26494,35860,27823,41361,31661,18514,34960,35133,32847,28028;
1850,-13861,14294,22846,27474,27157,38693,31129,13569,30412,31469,29850,25905;
1900,-17744,11213,20939,22148,26903,35290,30326,12748,25900,27625,26810,24021;
1950,-21165,7487,19343,19708,26709,30307,28998,14068,16925,23380,23272,21100;
2000,-25100,3819,17765,18140,25059,21659,26996,14506,5156,19127,19711,18050;

task done... time[ms]=42367409  time[min]=706
```

Abbildung 49: Rohdaten für mehrere Abbildungen aus dem Resultatteil

9.5 Artikel im Tages-Anzeiger zur Monte-Carlo-Methode

Der Tages-Anzeiger publizierte zwei Artikel zum Thema Monte-Carlo-Methode. Beide Artikel wurden in der Rubrik „Geld“ abgedruckt und füllen jeweils eine ganze Seite. Die Artikel sind hier abgebildet. Ein voller Abdruck war aufgrund ihrer Grösse nicht möglich, es wird jeweils der erste Teil des Artikels gezeigt.

Tages-Anzeiger · Montag, 17. September 2007

GELD

Das Unbekannte in den Griff bekommen

Jeder Anleger muss Risiken auf sich nehmen. Mit Monte-Carlo-Simulationen werden aus ihnen kalkulierbare Grössen.

Von **Erich Solenthaler**

Der erste Teil über Monte-Carlo-Simulationen (TA vom 3. September) beschäftigte sich mit den Grundlagen dieses Verfahrens: Monte-Carlo-Simulationen geben nicht nur die gradlinige voraussichtliche Entwicklung eines Vermögens wieder. Ihre Stärke ist es, dass man auch Risiken einbeziehen und optimistische oder pessimistische Szenarien bilden kann, die mit einer individuell definierten Wahrscheinlichkeit eintreffen.

In der Praxis dürften die pessimistischen Szenarien von besonderer Bedeutung sein. Wenn die Simulation Werte produziert, welche die Verlusttoleranz strapazieren, muss eine konservativere Strategie gewählt werden. Aber auch das Gegenteil kommt vor: Portfolios sind häufig zu vorsichtig konstruiert. Mit einer angepassteren Strategie liesse sich eine höhere Rendite erzielen. «Risiko» – sonst eher ein abstrakter, unfassbarer Begriff – wird auf diese Art eine anschauliche Grösse, die präzise Hinweise für die Portfolio-Zusammenstellung liefert.

Um Szenarien zu berechnen, muss die Volatilität eines Wertpapiers oder der gewählten Anlagestrategie bekannt sein. Die Volatilität ist die Schwankungs-Intensität eines Portfolios. Je grösser sie ist und je ferner das Sparziel in der Zukunft liegt, umso weniger lässt sich sagen, wo man dereinst landen wird. In den Grafiken schlägt sich dieser Effekt in den für Monte-Carlo-Simulationen typischen, sich nach rechts öffnenden Keilen nieder.

Angaben über die Volatilität erhält man von einem Anlageberater, in der Fachliteratur, oder man kann sich an Fonds mit einer ähnlichen Anlagestrategie orientieren. Für gemischte Portfolios sind je nach dem Aktienanteil Volatilitäten von 5 bis 12 Prozent typisch. Es ist wichtig, den Renditeund Volatilitäts-Annahmen genügend Aufmerksamkeit zu schenken. Sonst liefert die Simulation unbrauchbare Resultate. Richtig eingesetzt ist das Monte-Carlo-Verfahren aber ein wertvoller Teil der Finanzplanung.

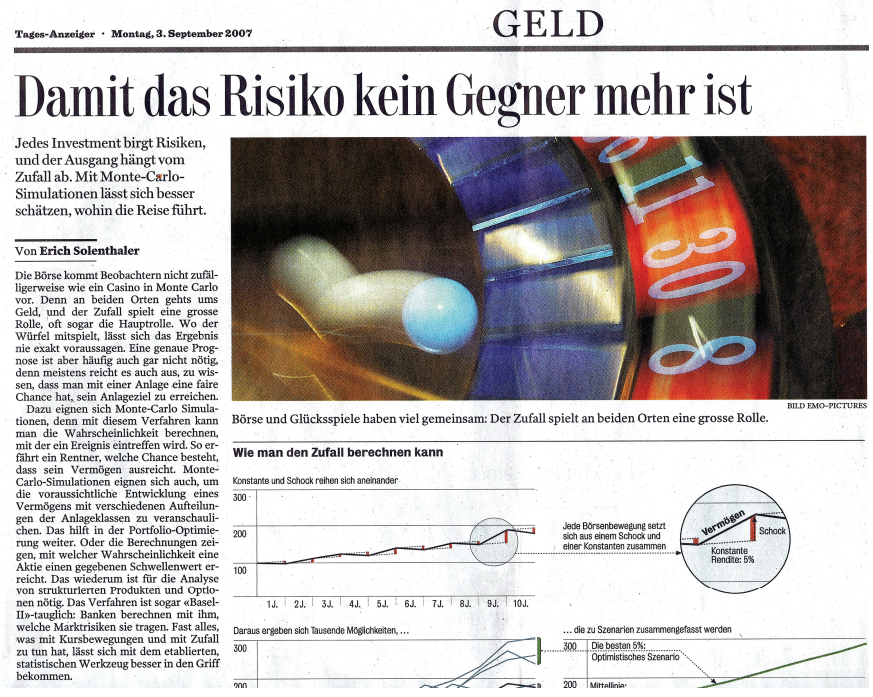
BILD AXEL SEIDEMANN/BLOOMBERG NEWS

Nicht nur im Casino, auch an der Börse spielt der Zufall mit. Seine Auswirkungen zu kennen, bringt viele Vorteile.

Unser Service: Der TA stellt ein Gratis-Excel-Blatt zur Verfügung, mit dem einfache Simulationen geübt werden können. Betreff: MC2

geld@tages-anzeiger.ch

Abbildung 50: Ausschnitt aus Tages-Anzeiger-Artikel vom 17.9.2007 zur Monte-Carlo-Methode



Tages-Anzeiger · Montag, 3. September 2007

GELD

Damit das Risiko kein Gegner mehr ist

Jedes Investment birgt Risiken, und der Ausgang hängt vom Zufall ab. Mit Monte-Carlo-Simulationen lässt sich besser schätzen, wohin die Reise führt.

Von **Erich Solenthaler**

Die Börse kommt Beobachtern nicht zufälligerweise wie ein Casino in Monte Carlo vor. Denn an beiden Orten gehts ums Geld, und der Zufall spielt eine grosse Rolle, oft sogar die Hauptrolle. Wo der Würfel mitspielt, lässt sich das Ergebnis nie exakt voraussagen. Eine genaue Prognose ist aber häufig auch gar nicht nötig, denn meistens reicht es auch aus, zu wissen, dass man mit einer Anlage eine faire Chance hat, sein Anlageziel zu erreichen.

Dazu eignen sich Monte-Carlo Simulationen, denn mit diesem Verfahren kann man die Wahrscheinlichkeit berechnen, mit der ein Ereignis eintreffen wird. So erfährt ein Rentner, welche Chance besteht, dass sein Vermögen ausreicht. Monte-Carlo-Simulationen eignen sich auch, um die voraussichtliche Entwicklung eines Vermögens mit verschiedenen Aufteilungen der Anlageklassen zu veranschaulichen. Das hilft in der Portfolio-Optimierung weiter. Oder die Berechnungen zeigen, mit welcher Wahrscheinlichkeit eine Aktie einen gegebenen Schwellenwert erreicht. Das wiederum ist für die Analyse von strukturierten Produkten und Optionen nötig. Das Verfahren ist sogar «Basel-II»-tauglich: Banken berechnen mit ihm, welche Marktrisiken sie tragen. Fast alles, was mit Kursbewegungen und mit Zufall zu tun hat, lässt sich mit dem etablierten, statistischen Werkzeug besser in den Griff bekommen.

BILD EMO-PICTURES

Börse und Glücksspiele haben viel gemeinsam: Der Zufall spielt an beiden Orten eine grosse Rolle.

Wie man den Zufall berechnen kann

Konstante und Schock reihen sich aneinander

Abbildung 51: Ausschnitt aus Tages-Anzeiger-Artikel vom 3.9.2007 zur Monte-Carlo-Methode

Eigenständige Arbeit

Der Inhalt dieser Arbeit ist komplett selbst gestaltet. Die verwendeten Abbildungen sind entweder selbst erstellt oder mit Quellenangabe versehen. Falls Literatur verwendet wurde, wird diese entweder zitiert oder ebenfalls via Quellenangabe referenziert.