Abschlussprüfung Sommer 2016

Fachinformatiker für Anwendungsentwicklung

Dokumentation zur betrieblichen Projektarbeit

Parsen eines Schemas in eine Baumstruktur

und zergliedern eines Datenstroms anhand dieses Schemas

Abgabetermin: Nürnberg, den 15.05.2016

Prüfungsbewerber:
René Ederer
Identnummer 4135980
Steinmetzstr. 2
90431 Nürnberg

Ausbildungsbetrieb:
Phoenix Group IT GmbH
Sportplatzstr. 30
90765 Fürth

Ausbildende:
Frau Birgit Günther

Projektbetreuer:
Herr Marco Kemmer

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

GUI	Graphical User Interface
Linq	Language Integrated Queries
Regex	Regular Expression
SVN	Subversion
WPF	Windows Presentation Foundation
VS	Visual Studio

1 Projektdefinition

1.1 Auftraggeber

Die Phoenix Pharmahandel GmbH & Co. KG und ihre Tochtergesellschaften sind europaweit unter dem Namen “Phoenix group” mit etwa 30000 Mitarbeitern im Pharmagroßhandel tätig. Ihre Haupttätigkeiten sind der Großhandel mit Pharmaprodukten und die Belieferung von Apotheken mit Medikamenten.

Auftraggeber des Projektes ist die Phoenix group IT GmbH. Sie hat etwa 200 Mitarbeiter und unterstützt die Phoenix group durch die Bereitstellung von IT-Dienstleistungen.

1.2 Projektumfeld

Die Phoenix group IT GmbH ist weiter unterteilt in die Abteilungen Inbound, Outbound und Warehouse. Das Projekt findet in der Abteilung Warehouse statt, in der überwiegend mit COBOL gearbeitet wird.

Phoenix verwendet ein firmeneigenes Dateiformat als Schnittstelle zu verschiedenen Programmen und Diensten. Es wird im Folgenden 1920Schema genannt und spielt eine zentrale Rolle für das Projekt.

1.2.1 1920Schemas

1920Schemas sind Textdateien, die einen Satz hierarchisch gegliederter Variablen beschreiben. Phoenix nutzt diese Schemas unter anderem als Schnittstelle, um Daten von seinem Mainframe zum Lagerrechner zu schicken, als Vorlage für Copybooks[1] und als Schnittstelle zu SSORT[2].

Beispiel für ein 1920Schema

Level	Name	Typ	Bytes	Wiederholzahl	Kommentar
01	Daten				Level und Name sind Pflichtangaben
03	Personendaten				Level gibt die Hierarchiestufe an
05	Vorname	C	5		fünf Bytes vom Typ char
05	Nachname	C	4		
R03	Gesamter-Name	C	9		redefiniert Personendaten
03	Bestellungen			2	Ein “Array”, Länge 2
05	Artikelnr	N	3		drei Byte lange Artikelnummer

Tabelle 1: Beispiel für ein 1920Schema

Anhang A.2: Beispiel für ein echtes 1920Schema auf Seite ii zeigt ein echtes 1920Schema.

[1] COBOL-Datei, in der eine Variablenstruktur definiert wird
[2] IBM-Programm, zeigt Copybooks an

1.2.2 Datenströme

Anhand von 1920Schemas werden Datenströme zergliedert und erhalten dadurch eine Bedeutung. Datenströme können prinzipiell beliebige Zeichen[3] enthalten, nur die Anzahl muss ausreichen, um jeder Schema-Variablen einen Wert zuzuweisen. Das Schema aus Tabelle 1 beschreibt einen 15 Byte langen Datenstrom (5+4+2*3), der so aussehen könnte:

Beispiel für einen Datenstrom

VN~~~NN~~123456

Datenströme sind aufgrund der größe des Terminal Windows fast immer 1920 Bytes lang.

1.3 Projektziel

Es soll ein Programm geschrieben werden, das nach Eingabe von Datenstrom und 1920Schema den Datenstrom dem Schema entsprechend zergliedert und anzeigt. Das Programm wird im Folgenden 1920Parser genannt.

Gewünschte Ausgabe (mit den Beispielwerten der Abschnitte 1.2.1 und 1.2.2)

```
01 Daten
  03 Personendaten
    05 Vorname=VN~~~
    05 Nachname=NN~~
 R03 Gesamter-Name=VN~~~NN~~
  03 Bestellungen(1)
    05 Artikelnr=1234
  03 Bestellungen(2)
    05 Artikelnr=5678
```

1.4 Projektbegründung

Bei Kundenreklamationen, Änderungen an Programmen und Neuentwicklungen stehen die Warehouse-Programmierer oft vor den Problemen:

- Wert einer Schemadatei-Variablen in einem Datenstrom finden.
- Datenstrom-Bytes einer Schemadatei-Variablen zuordnen.

Gegenwärtig zählen die Programmierer die passende Anzahl von Bytes in Schema und Datenstrom ab, einige erfahrene kennen die wichtigsten Schemadateien auch teilweise auswendig.

[3] Ein Byte entspricht einem Zeichen. Die Angaben in 1920Schemas haben die Einheit Byte, im Zusammenhang mit dem Datenstrom ist der Begriff Zeichen aber manchmal anschaulicher. Eine Unterscheidung der beiden Begriffe ist für das Projekt unwesentlich.

1.5 Projektschnittstellen

Benutzer des Programms sind die Programmierer der Phoenix Group IT GmbH, hauptsächlich die aus der Abteilung Warehouse.

1920Parser soll nicht unmittelbar mit anderen Systemen interagieren. Vorgesehen ist, dass die Benutzer die notwendigen Angaben in eine Eingabemaske hineinkopieren.

Projektgenehmigung und die Bereitstellung von Ressourcen erfolgt durch die Ausbildende Frau Birgit Günther, die Projektbetreuung und die Abnahme des Programms durch Herrn Marco Kemmer. Herr Kemmer arbeitet in der Abteilung Warehouse als COBOL-Entwickler, er kommuniziert die Kundenwünsche und will das Programm in Zukunft auch selbst nutzen.

2 Projektplanung

2.1 Projektphasen

Für das Projekt standen 70 Stunden zur Verfügung. Es findet im Zeitraum vom 11.04.2016 - 15.05.2016 statt.

Beispiel Tabelle 2 zeigt die grobe Zeitplanung für das Projekt.

Projektphase	Geplante Zeit
Analysephase	6 h
Entwurfsphase	8 h
Implementierungsphase	44 h
Abnahmetest der Fachabteilung	2 h
Erstellen der Dokumentation	10 h
Gesamt	**70 h**

Tabelle 2: Zeitplanung

Eine detailliertere Zeitplanung findet sich im Anhang A.1: Detaillierte Zeitplanung auf Seite i.

2.2 Ressourcenplanung

Zur Projektdurchführung werden folgende Ressourcen eingesetzt: Windows 7, Visual Studio Professional 2010, Microsoft Visio 2010, TexMaker, XMLSpy 2007, OpenText HostExplorer, Büro mit PC mit Verbindung zum Mainframe und Verbindung zum Internet, Strom, Projektbetreuer

2.3 Entwicklungsprozess

Es wurde ein agiler Entwicklungsprozess angewendet, der an Extreme Programming angelehnt war. Der im Projekt angewandte Prozess beinhaltete die Extreme Programming-Praktiken, testgetriebene Entwicklung, häufige Kundeneinbeziehung, häufiges Refaktorisieren, kurze Iterationen und einfaches Design.

3 Analysephase

3.1 Ist-Analyse

Die Ist-Analyse fand durch Befragung von Herrn Kemmer statt. Die Entwickler der Abteilung Warehouse müssen bei Kundenreklamationen, Programmänderungen und Neuentwicklungen die Werte in den beschriebenen Datenströme analysieren. Diese stammen aus den Logdateien des zentralen Lagerrechners, aus der Eingabemaske des Mainframes oder auch aus erhaltenen Email-Anhängen. Um die Bedeutung der Werte im Datenstrom herauszufinden, zählen die Entwickler gegenwärtig die Bytes in Schema und Datenstrom ab, einige erfahrenere kennen die wichtigsten Schemadateien auch zum Teil auswendig. Im Warehouse arbeiten etwa 20 Entwickler, etwa 10 davon haben regelmäßig mit solchen Datenströmen zu tun. Es werden Dutzende verschiedene 1920Schemas für verschiedenste Zwecke verwendet, aber mit nur etwa 10 arbeiten die Entwickler regelmäßig.

3.2 Wirtschaftlichkeitsanalyse

Das Programm soll den Entwicklern das bisher notwendige, fehleranfällige Abzählen von Zeichen in Schema und Datenstrom abnehmen. Das Projekt verspricht dadurch nicht nur, den Entwicklern Zeit zu sparen, sondern auch Zählfehler wirkungsvoll zu verhindern.

3.2.1 „Make or Buy“-Entscheidung

Die Anforderungen sind so speziell, dass fast auszuschließen ist, dass es außerhalb von Phoenix ein Programm gibt, das die Anforderungen erfüllt. Zu bemerken ist aber, dass viele Programme bei Phoenix mit Schemadateien arbeiten. Der Autor fragte deshalb nach, ob Phoenix eventuell schon ein Programm hat, das die Anforderungen erfüllt. Der Auftraggeber antwortete, dass die Datenströme aus so unterschiedlichen Umgebungen stammen (Unix-Lagerrechner, Mainframe, Email), dass die COBOL-Programme, die sie verarbeiten, nur eingeschränkt eingesetzt werden können. Es wurde entschieden, das Programm selbst neu zu schreiben.

3.2.2 Projektkosten

Die Kosten für die Durchführung des Projekts setzen sich sowohl aus Personal-, als auch aus Ressourcenkosten zusammen. Der Projektersteller ist Umschüler und erhält deshalb von seinem Ausbildungsbetrieb keine Vergütung.

$$7,7\ \text{h/Tag} \cdot 220\ \text{Tage/Jahr} = 1694\ \text{h/Jahr} \tag{1}$$

$$0\,€/\text{Monat} \cdot 13,3\ \text{Monate/Jahr} = 0\,€/\text{Jahr} \tag{2}$$

$$\frac{0\,€/\text{Jahr}}{1694\ \text{h/Jahr}} = 0{,}00\,€/\text{h} \tag{3}$$

Dadurch ergibt sich also ein Stundenlohn von 0,00 € Die Durchführungszeit des Projekts beträgt 70 Stunden. Für die Nutzung von Ressourcen[4] wird ein pauschaler Stundensatz von 12 € angenommen. Für die anderen Mitarbeiter wird pauschal ein Stundenlohn von 23 € angenommen. Eine Aufstellung der Kosten befindet sich in Tabelle 3 und sie betragen insgesamt 1015,00 €.

Vorgang	Zeit	Kosten pro Stunde	Kosten
Entwicklungskosten	70 h	0,00 € + 12 € = 12,00 €	840,00 €
Fachgespräch	3 h	23 € + 12 € = 35 €	105,00 €
Abnahmetest	2 h	23 € + 12 € = 35 €	70,00 €
			1015,00 €

Tabelle 3: Kostenaufstellung

3.2.3 Amortisationsdauer

Es wird davon ausgegangen, dass ausschließlich Entwickler der Abteilung Lager das Programm nutzen werden. Nach Einschätzung von Herrn Kemmer arbeitet die Hälfte der 20 Entwickler regelmäßig mit Schemadateien und dass das Programm jedem täglich 10 Minuten einsparen kann. Bei einer Einsparung von 10 Minuten am Tag für 10 Entwickler an 220 Arbeitstagen im Jahr ergibt sich eine gesamte Zeiteinsparung von

$$10 \cdot 220\ \text{Tage/Jahr} \cdot 10\ \text{min/Tag} = 22000\ \text{min/Jahr} \approx 366,67\ \text{h/Jahr} \tag{4}$$

Dadurch ergibt sich eine jährliche Einsparung von

$$366,67\text{h} \cdot (23 + 12)\,€/\text{h} = 12833{,}45\,€ \tag{5}$$

[4] Räumlichkeiten, Arbeitsplatzrechner etc.

Die Amortisationsdauer beträgt also $\frac{1015{,}00\,€}{12833{,}45\,€/\text{Jahr}} \approx 0{,}08$ Jahre ≈ 1 Monat.

3.3 Anwendungsfälle

Der mit Abstand wichtigste Anwendungsfall ist, dass das Programm nach Angabe von Datenstrom und Schema den Datenstrom entsprechend dem Schema zergliedert anzeigt. Der Kunde nannte noch einige Wünsche mehr zur Funktionalität. Abbildung **??** stellt die Anwendungsfälle in einem Usecase-Diagramm dar, sie befindet sich im Anhang.

3.4 Qualitätsanforderungen

Schemas müssen frei angebbar sein und Datenströme richtig zergliedert werden. Beim Abnahmetest soll dies anhand der 5 wichtigsten Schemadateien überprüft werden. Die Performance des Programmes ist ziemlich egal, es soll aber flüssig benutzbar sein (Zergliederung von Datenstrom und Anzeige in unter 5 Sekunden). Weil die Benutzer Profis sind, ist es nicht unbedingt notwendig, für alles eine Eingabemaske bereitzustellen. Es genügt auch, wenn eine Einstellung durch Editieren einer Konfigurations-Datei geändert werden kann.

3.5 Zwischenstand

Tabelle 4 zeigt den Zwischenstand nach der Analysephase.

Vorgang	Geplant	Tatsächlich	Differenz
1. Analyse des Ist-Zustands	3 h	3 h	
2. Wirtschaftlichkeitsprüfung und Amortisationsrechnung	1 h	2 h	+1 h
3. Erstellen des Lastenhefts	2 h	2 h	

Tabelle 4: Zwischenstand nach der Analysephase

4 Entwurfsphase - Zergliedern des Datenstroms

4.1 Zielplattform

Das Programm soll auf den Entwicklerrechnern der Phoenix unter Windows 7 laufen. Die Wahl der Programmiersprache wurde zunächst auf die bei Phoenix eingesetzten Sprachen COBOL, C++ und C# eingegrenzt. COBOL schied als Programmiersprache für ein Windows-Tool aus, aber C++ mit Qt und C# waren beide geeignet. Die hohe Performance, die C++ verspricht, wurde für 1920Parser aber nicht wirklich benötigt. Die Wahl fiel auf C# aufgrund von dessen Garbage Collection, Language Integrated Queries (Linq) und gutem GUI-Designer.

4.2 Aufbau der Schemadateien

Zur Analyse des Aufbaus der 1920Schemas fragte der Autor den Auftraggeber nach den Namen der 10 wichtigsten Schemadateien und ludt diese unter Verwendung von OpenText HostExplorer 2014 vom Mainframe herunter.

Level und Name sind Pflichtangaben, alle anderen Felder dürfen leer sein. Typ und Byteanzahl treten nur zusammen auf, solche Variablen sind Wertvariablen (sie definieren Bytes aus dem Datenstrom). Variablen ohne Typ und Bytezahl sind Gruppenvariablen. Zusätzlich zu diesen Variablenzeilen enthalten 1920Schemas Zeilen mit Metainformationen zum Schema. Die Metainformationen sind für 1920Parser nicht relevant und werden ignoriert. Die Angabe zur Byte-Anzahl kann auch die Form “AnzahlGesamtstellen,AnzahlNachkommastellen” haben.

Es ergaben sich unerwartet Schwierigkeiten beim Feld Kommentar. In der Schemadatei IOVK92 gab es eine Variable, deren Kommentar-Feld mit "0"begann. Es musste eine Regel festgelegt werden, damit diese 0 nicht als Wiederholzahl interpretiert wird, sondern als Kommentar-Beginn. Es gab die Möglichkeiten, 0 als Wiederholzahl zu verbieten oder die untereinander-stehende Anordnung der Variablenfelder auszunutzen. Falsche Annahmen hätten zu Fehlfunktion des Programms geführt; es war deshalb nötig, Rücksprache mit dem Auftraggeber zu halten. Auf dessen Vorschlag und nach erneuter Prüfung der Schemadateien wurde festgelegt, dass ein Kommentar mit mindestens 2 Leerzeichen vom vorhergehenden Feld getrennt sein muss.

4.3 Architekturdesign

Während der Projekterstellung wurde auf Kohärenz der Klassen geachtet und gegebenenfalls refaktorisiert. Auf ein explizites Design der Architektur wurde verzichtet.

4.4 Entwurf der Benutzeroberfläche

Der Kunde hatte keine Vorgaben bezüglich der Benutzeroberfläche gemacht. Nachdem als Programmiersprache C# feststand, kamen für die Graphical User Interface (GUI) nur Plattformen aus dem .NET-Framework in Betracht. Da Benutzer Schemas speichern können sollen, wäre ein Webinterface (ASP.NET) mit zusätzlichem Aufwand verbunden gewesen (Benutzerverwaltung/Datenbank). Windows Presentation Foundation (WPF) und Winforms waren beide geeignet. Der Autor entschied sich für Winforms, weil er damit mehr Erfahrung hatte.

4.5 Geschäftslogik

Die hierarchische Aufbau von 1920Schemas lässt sich gut mit einer rekursiven Baumstruktur im Programm abbilden. Für die Baumstruktur wurde die abstrakte Klasse AbstractNode entworfen. Von dieser

Klasse erben die Klassen GroupNode und ValuNode, die Gruppen- und WerteVariablen darstellen. GroupNode erhielt ein Attribut children vom Typ List<AbstractNode>, mit dem es auf seine Kind-Knoten verweisen kann. GroupNodes erlauben durch ihre rekursive Definition eine beliebig tiefe Verschachtelung von AbstractNodes.

Ein Klassendiagramm, welches die Klassen der Anwendung und deren Beziehungen untereinander darstellt kann im Anhang **??**: **??** auf Seite ?? eingesehen werden.

4.6 Maßnahmen zur Qualitätssicherung

Das korrekte Parsen von 1920Schemas und die richtige Zergliederung des Datenstroms sind zentral für 1920Parser. Um sicherzugehen, dass dieser Programmteil funktioniert, wurden die Methoden der Klassen Schema, Abstract-, Group- und ValueNode testgetrieben entwickelt.

4.7 Pflichtenheft/Datenverarbeitungskonzept

Ein Auszug aus dem Pflichtenheft ist im Anhang A.4: Pflichtenheft (Auszug) auf Seite iv zu finden.

4.8 Zwischenstand

Tabelle 5 zeigt den Zwischenstand nach der Entwurfsphase.

Vorgang	Geplant	Tatsächlich	Differenz
1. Analyse des Aufbaus der Schemadateien 2 h	2 h	2 h	
2. Entwurf der Klassen für die Baumstruktur	4 h	4 h	
3. Erstellen des Pflichtenhefts	2 h	2 h	

Tabelle 5: Zwischenstand nach der Entwurfsphase

5 Implementierungsphase - Zergliedern des Datenstroms

Die Implementierung begann mit dem Anlegen eines neuen Subversion (SVN)-Repositories mit der vorgegebenen Verzeichnisstruktur (branch, tag, trunk) und dem Erstellen einer neuen Solution in Microsoft Visual Studio Professional 2010 mit einem C# Winforms Projekt. Der Solution wurde ein NUnit-Testprojekt für die Unit-Tests hinzugefügt.

5.1 Implementierung der Benutzeroberfläche

Visual Studio erstellte mit Neuanlage des Winform-Projekts automatisch eine Form-Klasse. Mit dem GUI-Editor von Winforms wurden dieser Klasse drei Textfelder hinzugefügt, jeweils eines für Datenstrom, Schema und Ergebnis. Es wurde noch ein Button zum Starten der Verarbeitung eingefügt. Dieser wurde im Laufe der Implementierung entfernt und die Verarbeitung bei jeder Eingabe in Schema- oder Datenstrom-Textfeld gestartet.

5.2 Implementierung der Geschäftslogik

5.2.1 Parsen des Schemas in eine Baumstruktur

Parsen einer Variablenzeile zu einer AbstractNode Aus jeder Variablenzeile im Schema wird eine Group- oder ValueNode erstellt. Um dies Umzusetzen wurde in der Klasse Schema die Methode ParseLine geschrieben. Die Methode splittet die Angaben einer Variablenzeile mit Hilfe einer Regular Expression (Regex) in ihre Felder auf. Die Regex wurde auf https://regex101.com angepasst und getestet, bis sie richtig funktionierte.

Erstellen einer Baumstruktur aus den Variablenzeilen Anhang A.6: Objektdiagramm - Parsen des Schemas auf Seite vi zeigt ein Objektdiagramm mit der Struktur, die nach dem Erstellen des Baumes hergestellt sein soll. Dies war die schwierigste Aufgabe des Projektes. Die Schemazeilen mussten durchlaufen werden, aus ihnen AbstractNodes erstellt und diese in eine Baumstruktur gebracht werden. Schwierig war, dass mit jeder neuen Zeile ein Kindknoten, ein Geschwisterknoten, oder ein Geschwisterknoten eines früheren Vorfahrens auftauchen konnte. In jedem dieser Fälle konnte unterhalb des Knotens erneut eine Hierarchie von Knoten sein. Es musste diese Hierarchie, falls vorhanden, erstellt werden, danach musste die Verarbeitung mit der richtigen Schemazeile und dem richtigen Elternknoten weitergehen. Schwierig war insbesondere auch, dass eine Schemazeile zu mehreren AbstractNodes werden konnte, abhängig von ihrer Wiederholzahl. In dieser Hinsicht waren besonders GroupNodes problematisch, da sich bei GroupNodes mit der Wiederholzahl der gesamte Baum unterhalb von ihnen mit-wiederholt. Außerdem konnten Wiederholzahlen auch bei mehreren Vorfahren-Knoten vorkommen, so dass sich die Wiederholungen verschachtelten.

Nach vielen vergeblichen Versuchen, mit Rekursion und verschachtelten Schleifen (zum Durchlaufen der Schemazeilen und Erstellen der Wiederholungen von Nodes) zu arbeiten, führten schließlich zwei Ideen zur Lösung des Problems. Die erste war, nicht mit Rekursion zu arbeiten sondern explizit mit einem Stack. Damit wurde es einfacher den Verlauf des Algorithmus nachzufolgen und vorherzusehen. Die zweite und entscheidende Idee war, Nodes mit Wiederholzahl nicht sofort zu kopieren, sondern sie erst einmal fertigzustellen und danach zu kopieren. Hierzu wurden die Node-Klassen um die Methode CreateCopyWithIndex(: int) erweitert.

Etwas ärgerlich war, dass die Methode zuerst nur so aussah, als würde sie funktionieren. Es waren nur flache Kopien der Nodes erzeugt worden, ohne Kopien der Kindknoten zu erzeugen. Dadurch erhielten alle Nodes von Wiederholgruppen beim Zuweisen die gleichen Werte. Der Fehler war durch Unittests nicht entdeckt worden. Zur Behebung des Fehlers musste CreateCopyWithIndex() seine Kind-Nodes rekursiv kopieren.

Anhang A.8: Quelltext Schema.cs auf Seite viii, Anhang A.9: Quelltext GroupNode.cs auf Seite ix und Anhang A.10: Quelltext ValueNode.cs auf Seite x zeigen Quelltextausschnitte.

5.2.2 Grundschema der Methoden von AbstractNode

Group- und ValueNode erben von AbstractNode Methoden. Die Implementierung all dieser Methoden ähnelt sich, sie folgt für Group- und ValueNodes immer diesem Prinzip:

ValueNode macht eine Aktion und gibt danach einen Wert zurück.

GroupNode macht eine Aktion und ruft danach für jedes seiner Kinder die gleiche Methode erneut auf (Rekursion). Aus den Rückgabewerten der Kinder wird ein Wert akkumuliert und dieser zurückgegeben.

Die von einer GroupNode angestoßenen Rekursionen enden bei ValueNodes und GroupNodes ohne Kindknoten, die Abbruchbedingung der Rekursion ist polymorph.

Es ist kein Zufall, dass alle Methoden rekursiv sind, denn auch die Klassenstruktur ist rekursiv (GroupNodes können auf GroupNodes verweisen).

5.3 Zwischenstand

Tabelle 6 zeigt den Zwischenstand nach der Implementierungsphase.

Vorgang	Geplant	Tatsächlich	Differenz
1. Implementierung der Klassen	1 h	1	
2. Schreiben einer Methode zum Vergleichen von Nodes	0 h	1 h	+1 h
3. Erstellen von Tests	5 h	12 h	+7 h
4. Entwickeln einer Regular Expression zum Parsen von Schemazeilen	1 h	3 h	+2 h
5.Schreiben einer Methode zum Kopieren von Nodes	0 h	1 h	+1 h
6. Parsen der Schema-Datei in eine Baumstruktur	14 h	17 h	+3 h
7. Durchlaufen des Baumes und Zuweisen von Werten	12 h	2 h	−10 h
8.Ausgabe des Baumes als String	5 h	1 h	−4 h
9. Erstellen der Benutzeroberfläche	6 h	3 h	−3 h

Tabelle 6: Zwischenstand nach der Implementierungsphase

6 Entwurfsphase - Schemas speichern und gespeicherte Schemas auswählen

Um Schemas zu speichern, wurde entschieden, einen Unterordner namens Schemas zu erstellen. Der Benutzer kann in diesen Schemas als Textdateien hineinkopieren. Diese sollen in der GUI angezeigt werden, indem neben der Überschrift für das Schema-Textfeld eine Combobox eingerichtet wird, die die Namen der Dateien in diesem Ordner anzeigt. Wenn sich der Index der ComboBox ändert, soll die Datei mit dem Namen des ComboBox-Eintrags gelesen werden und das Schematextfeld mit dem Dateiinhalt gefüllt werden.

6.1 Zwischenstand

Tabelle 7 zeigt den Zwischenstand nach der Entwurfsphase des Anwendungsfalls Schemas speichern.

Vorgang	Geplant	Tatsächlich	Differenz
1. Entwurfsphase Schema speichern	0 h	1 h	+1 h

Tabelle 7: Zwischenstand nach der Entwurfsphase des Anwendungsfalls Schemas speichern

7 Implementierungsphase - Schemas speichern und gespeicherte Schemas auswählen

Da die Funktionalität zum Lesen der “schemas”-Ordners in keine der vorhandenen Klassen passte, wurde die neue Klasse SchemaManager erstellt.

Es wurde eine Combobox zur GUI hinzugefügt. Der Konstruktor von 1920ParserWindow wurde so angepasst, dass er die Methode getSchemas() der Klasse SchemaManager aufruft und die Combobox die zurückgegebenen Dateinamen als Einträge erhält.

Zur Auswahl eines Schemas wurde über den Visual Studio (VS) GUI-Designer die Methode cmbSchemas_SelectedIndexChanged() erstellt, die die Methode getFileContent() Klasse SchemaManager bei jeder Combobox-Indexänderung aufruft und den Dateiinhalt als string zurückgegeben bekommt. Das Schema-Textfeld zeigt diesen string an.

7.1 Zwischenstand

Tabelle 8 zeigt den Zwischenstand nach der Implementierung des Anwendungsfalls Schemas speichern.

Vorgang	Geplant	Tatsächlich	Differenz
1. Entwurfsphase Anwendungsfall Schema speichern	0 h	1 h	+1 h

Tabelle 8: Zwischenstand nach der Implementierung des Anwendungsfalls Schemas speichern

8 Abnahmephase

8.1 Abnahmetest

Der Abnahmetest erfolgte durch den Projektbetreuer Herrn Kemmer. Herr Kemmer ließ sich die erstellten Unit-Tests zeigen und überprüfte, dass alle Tests erfolgreich waren. Danach ludt er die 10 am Häufigsten verwendeten Schemas zusammen mit passenden Datenströmen vom Mainframe. 1920Parser zeigte alle getesteten Schemas richtig an und zergliederte die Datenströme in gewünschter Weise. Im Anschluss testete Herr Kemmer die Funktionalität zum Speichern und zur automatischen Auswahl von Schemas. Nach Löschung der XML-Konfigurationsdatei trat bei der Auswahl des Menüpunktes "Config editierenëine FileNotFoundException auf. Diese wurde behoben, danach testete Herr Kemmer noch einmal.

8.2 Zwischenstand

Tabelle 9 zeigt den Zwischenstand nach der Abnahmephase.

Vorgang	Geplant	Tatsächlich	Differenz
1. Abnahmetest	2 h	2 h	

Tabelle 9: Zwischenstand nach der Abnahmephase

9 Einführungsphase

Die Einführung von 1920Parser wurde von Herrn Kemmer übernommen. Diesem wurde per Email ein Link zum SVN-Repository, das gezippte Programm und die gezippten Dokumentationen zugeschickt. Er will das Programm im Intranet verfügbar zu machen.

10 Dokumentation

Für 1920Parser wurde eine Benutzerdokumentation und eine Entwicklerdokumentation geschrieben. Die Entwicklerdokumentation wurde aus Kommentaren autogeneriert. Ein Screenshot befindet sich im Anhang A.11: Entwicklerdokumentation auf Seite xi

10.1 Zwischenstand

Tabelle 10 zeigt den Zwischenstand nach der Dokumentation.

Vorgang	Geplant	Tatsächlich	Differenz
1. Erstellen der Benutzerdokumentation	1 h	1 h	
2. Erstellen der Projektdokumentation	7 h	8 h	+ 1
3. Erstellen der Entwicklerdokumentation	2 h	1 h	− 1

Tabelle 10: Zwischenstand nach der Dokumentation

11 Fazit

11.1 Soll-/Ist-Vergleich

Wie gewünscht können im Programm Datenstrom und Schema frei angegeben werden. Das Ergebnis wurde bei allen Tests richtig angezeigt.

Zur Zeitplanung ist zu sagen, dass insbesondere das Zuweisen der Werte aus dem Datenstrom und die Ausgabe des Baumes als String deutlich weniger Zeit in Anspruch nahmen, als ursprünglich geplant, dafür dauerten die Unit-Tests länger als geplant. Herr Kemmer ist mit dem Programm sehr zufrieden.

Phase	Geplant	Tatsächlich	Differenz
Entwurfsphase	19 h	19 h	
Analysephase	9 h	10 h	+1 h
Implementierungsphase	29 h	28 h	-1 h
Abnahmetest der Fachabteilung	1 h	1 h	
Einführungsphase	1 h	1 h	
Erstellen der Dokumentation	9 h	11 h	+2 h
Pufferzeit	2 h	0 h	-2 h
Gesamt	70 h	70 h	

Tabelle 11: Soll-/Ist-Vergleich

11.2 Lessons Learned

Interessant war, wie sich die Anforderungen erweiterten (ursprünglich sollte nur ein bestimmtes Schema zergliedert werden). Unit-Tests zu schreiben hat sich ausgezahlt, einmal durch höhere Chancen, dass das Programm tut, was es soll, aber auch dadurch, dass es vermutlich Debugging gespart hat. Der Autor konnte durch das Projekt sein Schulwissen vertiefen und wichtige Erfahrungen für die Planung und Umsetzung größerer Projekte sammeln. Er ist ein noch größerer Fan von Stacks als vorher.

11.3 Ausblick

Im Rahmen des Projektes wurden nicht alle Wunschkriterien erfüllt. Der Autor hofft aber, dass diese Aufgabe an einen nachfolgenden Auszubildenden übergeben wird und sich 1920Parser noch weiterentwickelt. Phoenix erwägt seit Jahren, seine COBOL-Programme zu portieren. Dann könnte vielleicht der Algorithmus von 1920Parser für eine Teilaufgabe verwendet werden.

A Anhang

A.1 Detaillierte Zeitplanung

Analyse		**6 h**
Ist-Analyse	3 h	
Wirtschaftlichkeitsprüfung und Amortisationsrechnung	1 h	
Erstellen des Lastenheftes	2 h	
Entwurf		**8 h**
Analyse des Aufbaus der Schemadateien, Treffen von Annahmen	2 h	
Entwurf der Klassen für die Baumstruktur	4 h	
Erstellen des Pflichtenhefts	2 h	
Implementierung		**44 h**
Implementierung der Klassen	1 h	
Erstellen von Tests	5 h	
Entwickeln einer Regular Expression zum Parsen der Schemadatei	1 h	
Parsen der Schema-Datei in eine Baumstruktur	14 h	
Durchlaufen des Baumes und Zuweisen der Werte aus dem Datenstrom	12 h	
Ausgabe des Baumes als String	5 h	
Erstellen der Benutzeroberfläche	6 h	
Abnahme		**2 h**
Abnahmetest	2 h	
Dokumentation		**10 h**
Erstellen der Projektdokumentation	7 h	
Erstellen der Entwicklerdokumentation	2 h	
Erstellen der Benutzerdokumentation	1 h	

A.2 Beispiel für ein echtes 1920Schema

```
.fo off
.pa
+----------------+-----------------------------------+---------------+
I                I                                   I               I
I                I                                   I NACHFUELL     I
I  P H A R M O S  I      Copy-Book-Beschreibung       I               I
I                I   Schnittstelle : Host --> PC     I               I
I                I                                   I     IOVK91    I
+----------------+-----------------------------------+---------------+
I  P H A R M O S  I     Froh zu sein bedarf es        I Ausg.: 1 Wink I
I  8510 Fuerth 2  I          P H A R M O S            I Vom  :13.12.99I
I  0911-9300-695  I                                   I Wink :29:10.02I
+----------------+-----------------------------------+---------------+
Dateiname: PH326591 OPUS

Satzlaenge:        Bytes
:h6.IOVK91 Schnittstelle Host->PC
.sp 2
.fo off
.tr * 40
.cm FELD-BESCHREIBUNG IOVK91
.pi /Bereich/IOVK91
.bx 2   9                27  31     38  42                          &$11
  Stufe   Feldname          Typ Laenge Tab  Kommentar
.bx
    ****                                    Schnittstellenbereich
    03    IO91-AREA                         Host --> PC
     05   IO91-TRANID       C   4           Transaktionscode (VK91)
     05   IO91-FUNKID       C   2           Funktionscode
     05   IO91-EBENE        N   1           Lagerebene
    ***
     05   IO91-DATEN-AREA                   Struktur fuer Wannendaten
    ***
     10   IO91-WANNE                  4     Wannendaten
      15  IO91-SA           C   1           SATZART 1 = VORAB, 2 = LA
      15  IO91-FIL          N   2           Filiale
      15  IO91-L1RRN        N   9           Re. Satznr. L1-Satz = LA-Nr
      15  IO91-VZEIT        N  11           Vorlaufzeit in SEK
      15  IO91-IPUNKT-WNR   N   7           Wannennr vom I-Punkt
      15  IO91-WANREST      C   6           Filler
      15  IO91-POSTAB                 40    Tabelle der Positionen
       20 IO91-KANAL        N   8           Kanalnummer
      R20 IO91-KANAL-C      C   8           LAGERORT
       20 IO91-MENGE        N   2           Menge
       20 IO91-S-REST       C   1           Filler
     10   IO91-REST         C   9           REST
    ***
.bx
   R05    IO91-FEHLER-AREA                  Struktur fuer Fehlermeldung
     10   IO91-FEHLMELD     C   80          Meldungstext
   R10    IO91-FEHLERMELDR                  Struktur fuer Fehlermeldung
      15  IO91-FEHLMELD2    C   20          Meldungstext
      15  IO91-FEHLNR       N    5          L1-Nummer usw.
      15  filler            C    1
      15  IO91-FEHLNR2      N    5          L1-Nummer usw.
      15  filler            C   49
     10   IO91-REST2        C 1831          REST
    ***
.bx
   R05    IO91-einzel                       Struktur fuer Fehlermeldung
     10   IO91-E-L1RRN      N   5           Taktnummer
     10   IO91-E-stpl       N   2           Platznummer im takt
     10   IO91-REST3        C 1906          REST
.bx
   R05    IO91-STATION-AREA                 STRUKTUR FUER FEHLERMELDUNG
     10   IO91-STTAB                  40    TABELLE DER STATIONEN
      15  IO91-VSTATION     N   2           STATION VOR AUTOMAT
    ***
.bx
```

Abbildung 1: Beispiel für ein echtes 1920Schema

A.3 Use Case-Diagramm

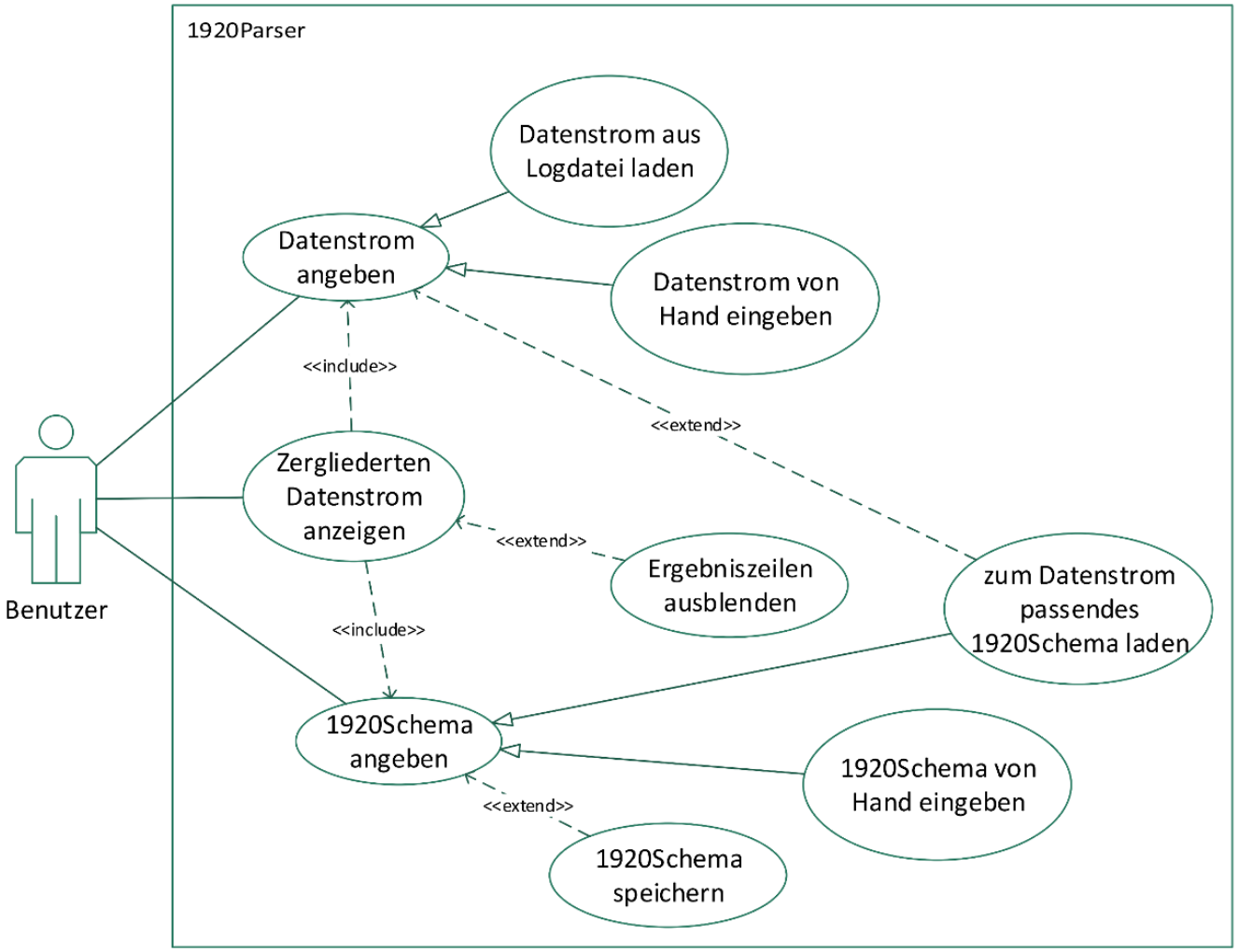

Abbildung 2: Use Case-Diagramm

A.4 Pflichtenheft (Auszug)

Die Anwendung muss folgende Anforderungen erfüllen:

1. Muss-Kriterien:
   1.1. Datenstrom und Schema müssen frei angegeben werden können.
   1.2. Das Programm muss den Datenstrom anhand des Schemas zergliedert anzeigen.
2. Soll-Kriterien:
   2.1. Benutzer sollen ein eingegebenes Schemas speichern können.
   2.2. Benutzer sollen gespeicherte Schemas laden können.
3. Kann-Kriterien:
   3.1. Anhand des Transaktionscodes im Datenstrom soll das richtige Schema - falls vorhanden - automatisch ausgewählt werden.
   3.2. Der Benutzer soll ein Kriterium angeben können, anhand dessen die passenden Datenströme aus der Logdatei ausgewählt werden.
   3.3. Wenn das Länge-Feld eine Angabe zur Anzahl der Nachkommastellen hat, soll das Programm in der Ergebnis-Anzeige an der richtigen Stelle ein Komma einfügen.
   3.4. Variablenzeilen sollen ausgeblendet werden können.
4. Ausschluss-Kriterien:
   4.1. Das Programm soll Schema und Datenstrom nicht auf Korrektheit oder Plausibilität prüfen (z. B. doppelte Variablennamen, Redefines ohne Vorgänger, ungültige Stufennummern, Bytelänge 0, Char mit Nachkommastellen).

A.5 Klassendiagramm

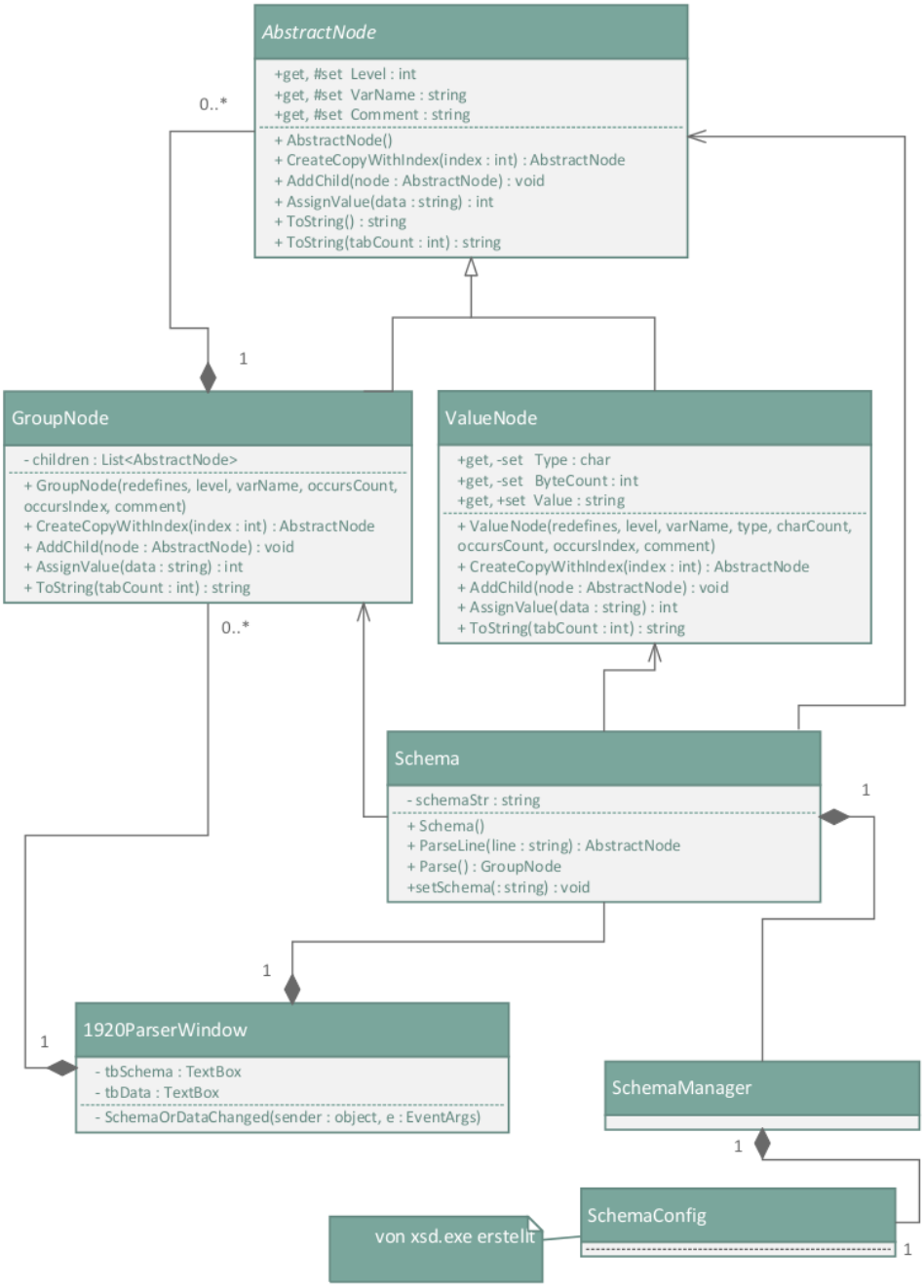

Abbildung 3: Klasseniagramm

A.6 Objektdiagramm - Parsen des Schemas

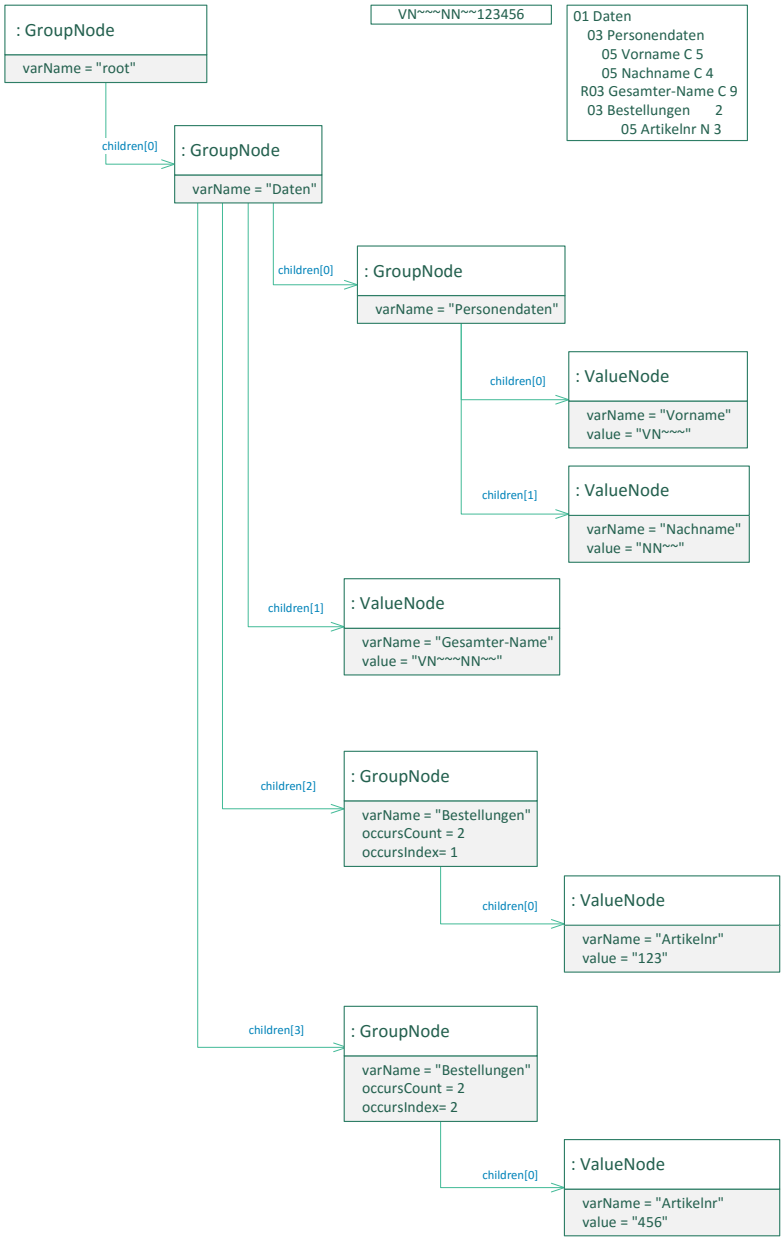

Abbildung 4: Objektdiagramm nach dem Parsen und dem Zuweisen der Werte

A.7 Aktivitätsdiagramm - Parsen eines 1920Schemas

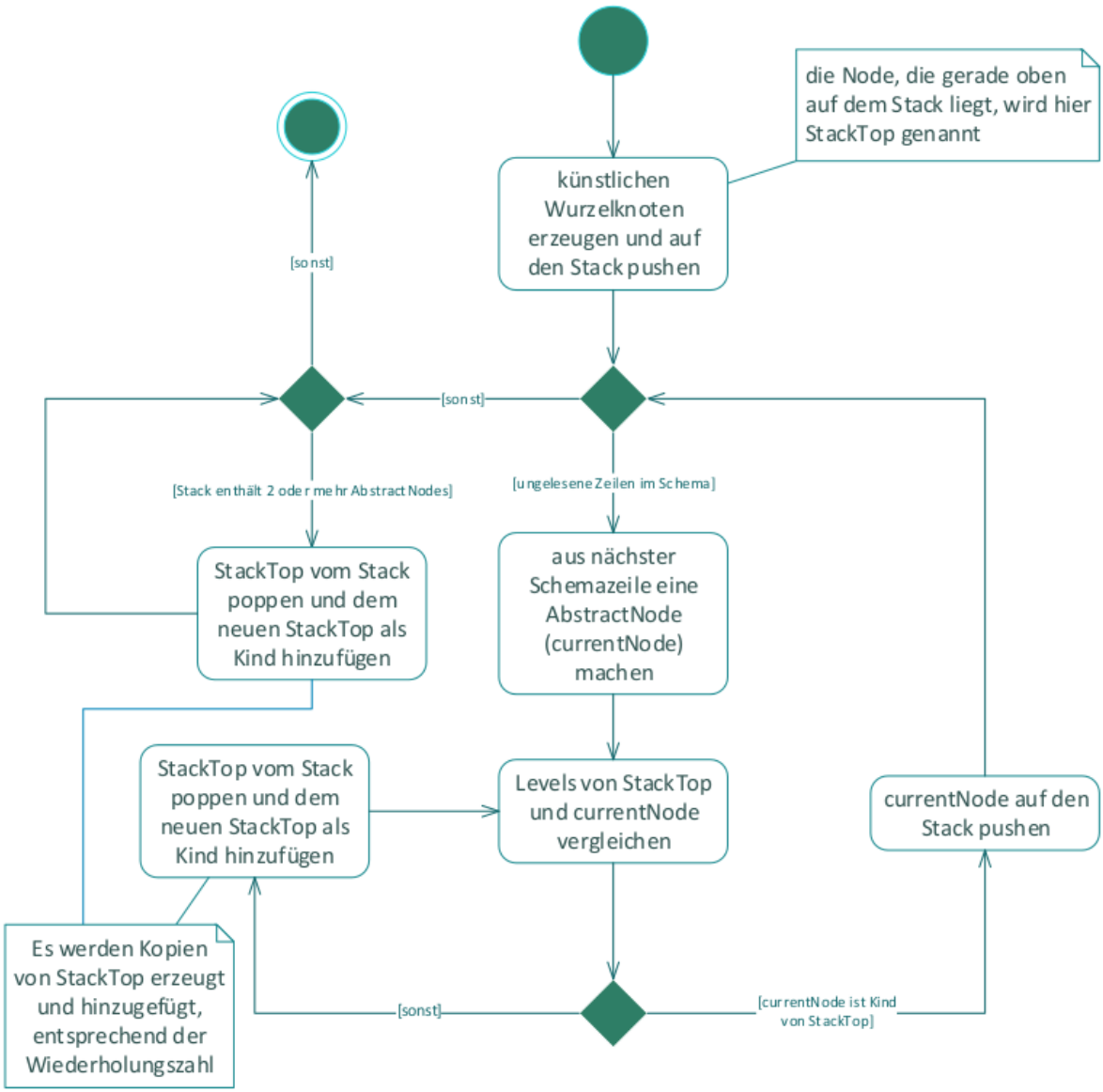

Abbildung 5: Aktivitätsdiagramm - Parsen eines 1920Schemas

A.8 Quelltext Schema.cs

```
1  class Schema
2  {
3      private string pattern = @"^\s*(?<redefines>R?)(?<level>\d+)\s+(?<varName>
         \S+)((\s+(?<type>[xcnpXCNP])\s+((?<length>\d+))(\,(?<decimalPlaces>\d
         +))?)?(\s+(?<repeatCount>\d+))?)?(\s{2,}(?<comment>.*))?$";
4      private string schemaStr = "";
5
6      public AbstractNode ParseLine(string line)
7      {
8          // gibt eine Value- oder GroupNode zurück, je nachdem ob line
9          // Angaben zu Typ und ByteAnzahl hat oder nicht
10         // wenn pattern nicht matcht wird null zurückgegeben
11     }
12
13     public GroupNode Parse()
14     {
15         var stack = new Stack<AbstractNode>();
16         var root = new GroupNode(false, 0, "root", 1, 1, "");
17         stack.Push(root);
18
19         Action addChildFromStackToParent =
20             () =>
21             {
22                 var child = stack.Pop();
23                 var parent = stack.Peek();
24                 parent.AddChild(child);
25                 for (int currentRepeatIndex = 2; currentRepeatIndex <=
                     child.RepeatCount; ++currentRepeatIndex)
26                 {
27                     parent.AddChild(child.CreateCopyWithIndex
                       (currentRepeatIndex));
28                 }
29
30             };
31
32         var schemaLines = schemaStr.Split(new string[] { "\r\n", "\n" },
             StringSplitOptions.RemoveEmptyEntries);
33         foreach (var currentLine in schemaLines)
34         {
35             var currentNode = ParseLine(currentLine);
36             if (currentNode == null) { continue; }
37             while (currentNode.Level <= stack.Peek().Level)
38             {
39                 addChildFromStackToParent();
40             }
41             stack.Push(currentNode);
42         }
43         while (stack.Count >= 2)
44         {
45             addChildFromStackToParent();
46         }
47         return root;
48     }
49 }
```

Abbildung 6: Schema.cs

A.9 Quelltext GroupNode.cs

```
1  class GroupNode : AbstractNode
2  {
3      private List<AbstractNode> children = new List<AbstractNode>();
4      public GroupNode(bool redefines, int level, string varName, int
         repeatCount, int repeatIndex, string comment)
5          : base(redefines, level, varName, repeatCount, repeatIndex, comment)
6      { }
7      public override int AssignValue(string data)
8      {
9          int currentShift = 0;
10         int totalShift = 0;
11         foreach (var child in children)
12         {
13             if (!child.Redefines)
14             {
15                 currentShift = child.AssignValue(data.Substring(totalShift));
16                 totalShift += currentShift;
17             }
18             else
19             {
20                 child.AssignValue(data.Substring(totalShift - currentShift));
21             }
22         }
23         return totalShift;
24     }
25
26     public override string ToString(int tabCount)
27     {
28         StringBuilder strBuilder = new StringBuilder();
29         if (Level != 0)
30         {
31             strBuilder.Append(string.Format("{0}{1}{2} {3}{4}\r\n",
32             new string(' ', tabCount * 4 - (Redefines ? 1 : 0)),
33             Redefines ? "R" : "",
34             Level.ToString().PadLeft(2, '0'),
35             VarName,
36             RepeatCount > 1 ? string.Format("({0})", RepeatIndex) : ""));
37         }
38         foreach (var child in children)
39         {
40             strBuilder.Append(child.ToString(tabCount + (Level == 0 ? 0 :
                 1)));
41         }
42         return strBuilder.ToString();
43     }
44     public override AbstractNode CreateCopyWithIndex(int index)
45     {
46         GroupNode g = new GroupNode(Redefines, Level, VarName, RepeatCount,
             index, Comment);
47         foreach (var child in children)
48         {
49             g.AddChild(child.CreateCopyWithIndex(child.RepeatIndex));
50         }
51         return g;
52     }
53 }
```

Abbildung 7: GroupNode.cs

A.10 Quelltext ValueNode.cs

```
1  class ValueNode : AbstractNode
2  {
3      public string Type { get; private set; }
4      public int Length { get; private set; }
5      public string Value { get; private set; }
6
7      public ValueNode(bool redefines, int level, string varName, string type,
         int length, int repeatCount, int repeatIndex, string comment)
8          : base(redefines, level, varName, repeatCount, repeatIndex, comment)
9      {
10         Type = type;
11         Length = length;
12     }
13
14     public override AbstractNode CreateCopyWithIndex(int index)
15     {
16         return new ValueNode(Redefines, Level, VarName, Type, Length,
             RepeatCount, index, Comment);
17     }
18
19     public override int AssignValue(string data)
20     {
21         Value = data.Substring(0, Length);
22         return Length;
23     }
24
25     public override string ToString(int tabCount)
26     {
27         if (Level == 0)
28         {
29             return "";
30         }
31         return string.Format("{0}{1}{2} {3}{4}={5}\r\n",
32             new string(' ', tabCount * 4 - (Redefines ? 1 : 0)),
33             Redefines ? "R" : "",
34             Level.ToString().PadLeft(2, '0'),
35             VarName,
36             (RepeatCount > 1) ? ("(" + RepeatIndex + ")") : "",
37             Value);
38     }
39
40     public override void AddChild(AbstractNode child)
41     {
42         throw new InvalidOperationException(this.VarName + ": " + "Werteknoten
             können keine Kindknoten haben!");
43     }
44 }
```

Abbildung 8: ValueNode.cs

A.11 Entwicklerdokumentation

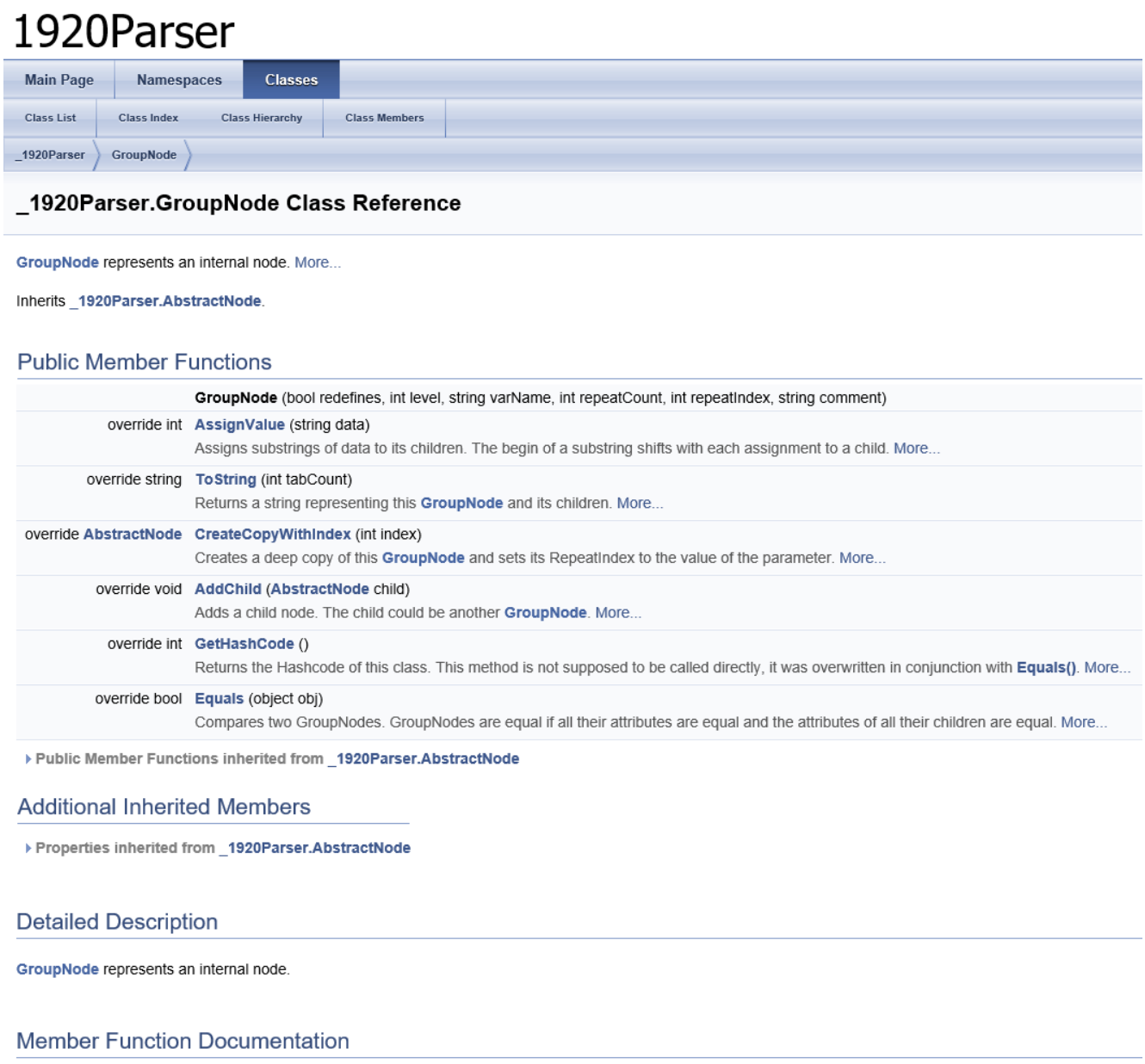

Abbildung 9: Entwicklerdokumentation