technische universität
dortmund

Diplomarbeit

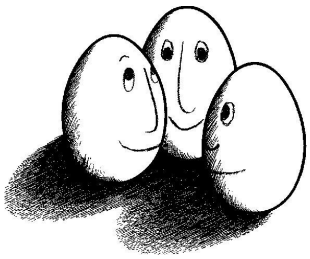
# Erstellung eines XML Adapters zur dynamischen Verarbeitung von ROOT-Daten.

**Marina Schwacke**

**31. Mai 2010**

Betreuer:
Prof. Dr. Katharina Morik
Dipl. Inf. Benjamin Schowe

Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz, LS VIII
Technische Universität Dortmund
http://www-ai.cs.uni-dortmund.de



# Vorwort

Ein großer Bereich der modernen Physik befasst sich mit der Analyse von Daten aus physikalischen Experimenten. Die Analysesoftware ROOT ist eine sehr häufig verwendete Software. Sie ist ein „Data Analysis Framework“, das der Datenspeicherung, der Datenanalyse und der Datenvisualisierung dient. Die zu untersuchenden Daten werden von der Software in einem eigenen Datenformat abgespeichert.

Aufbauend auf dem Design des Datenformats wird in dieser Arbeit ein XML-Konverter und eine RapidMiner-Erweiterung erstellt, welche zusammen das Einlesen des ROOT-Formats in die Software RapidMiner ermöglichen. Die Vorteile und Möglichkeiten der DataMining-Software RapidMiner werden dabei für die Analyse der Physikdaten verfügbar gemacht. In dieser Arbeit werden die Grundlagen und das Konzept des Systems erläutert, die Implementierung und das entwickelte RapidMiner-Plugin vorgestellt. Zusätzlich wird die Handhabung des Systems an drei Beispielen aufgezeigt.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Algorithmenverzeichnis

# Kapitel 1

# Einleitung

Die Teilchenphysik ist ein Bereich der Physik, der sich mit der Erforschung des Aufbaus der Materie aus Teilchen beschäftigt. Diese Teilchen können Moleküle sein, Atome oder Nukleonen bis hin zu Elementarteilchen. Die Ergebnisse aus Experimenten in der Teilchenphysik bestehen meist aus einer sehr großen Anzahl von Einzelergebnissen, die wiederum aus einer Anzahl unterschiedlich gemessener Variablen bestehen. Aus diesen großen Datenmengen müssen seltene, für die Forschung interessante Ereignisse herausgefiltert werden. Das in der Physik dazu häufig benutzte Programm ist das „Data Analysis Framework ROOT“ [7]. Es handelt sich dabei um ein objektorientiertes Softwarepaket, welches auf C++ basiert. Es dient der Datenspeicherung, der Datenanalyse und der Datenvisualisierung.

Ein wichtiges Anwendungsgebiet findet sich in der Astroteilchenphysik, die sich mit der Analyse astronomischer Daten beschäftigt. Dabei wird die Kosmische Strahlung mit Hilfe von Detektoren untersucht. Die kosmische Strahlung besteht überwiegend aus geladenen, hochenergetischen Teilchen, die galaktischen und solaren Ursprungs sind. Aktuelle Projekte sind unter anderem das MAGIC Projekt auf La Palma [23] und das IceCube Projekt am Südpol [20]. Ein weiteres Anwendungsgebiet von ROOT in der Teilchenphysik ist der Teilchenbeschleuniger Large Hadron Collider (LHC) [14] am Europäischen Kernforschungszentrum Conseil Européen pour la Recherche Nucléaire (CERN) bei Genf [32].

Die Analysesoftware ROOT beinhaltet aus verschiedenen Gründen einige Nachteile. Daher besteht der Wunsch, die in ROOT-Dateien gespeicherten Messdaten auch mit anderen Mitteln analysieren zu können.

„RapidMiner“ ist eine Open-Source Lösung für Data Mining Prozesse [41]. Beim Data Mining wird versucht, verborgene Zusammenhänge durch Methoden des maschinellen und statistischen Lernens zu entdecken. RapidMiner gehört zu den führenden Data-Mining-Anwendungen. Sie ist besonders für die Analyse großer Datenmengen geeignet. Diesbezüglich besteht das Anliegen, einen Konverter zwischen dem ROOT- und dem RapidMiner-System zu erstellen.

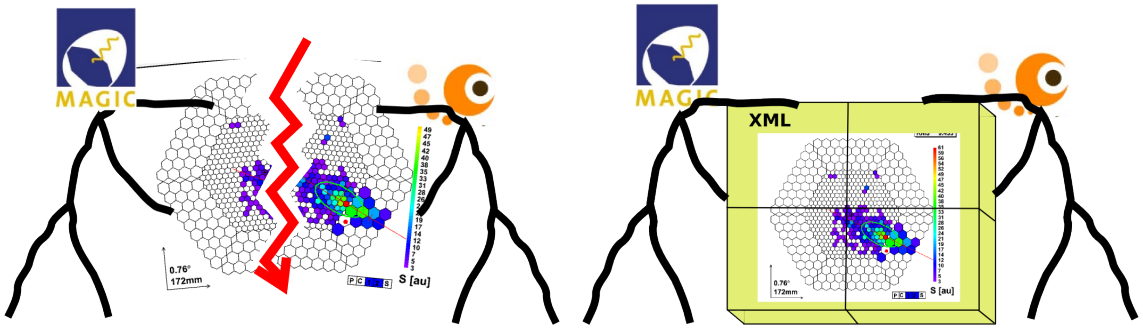
## 1.1 Motivation

Die zu betrachtenden Messdaten besitzen in ihrer rohen Form einen geringen Nutzwert. Dieser entsteht erst durch die Gewinnung von Informationen aus den Daten. Durch Anwendung von Methoden des Data Mining auf solch einen Datenbestand können Muster entdeckt werden. Dabei wird versucht, Regelmäßigkeiten, Wiederholungen, Ähnlichkeiten oder Gesetzmäßigkeiten in diesen Daten zu erkennen.

Um die Daten unter dem Gesichtspunkt des DataMining betrachten zu können, bietet sich die Software RapidMiner [41] an. RapidMiner bietet zur Analyse eine große Zahl an Operatoren an, die nahezu beliebig verschachtelt werden können. Ein Operator erfüllt eine Teilaufgabe des Data Mining-Prozesses. Operatoren stellen Methoden für die für Datentransformationen, -analyse und -visualisierung bereit. Durch diese große Anzahl an Operatoren und deren Kombinationsmöglichkeiten, stehen nicht nur viele Möglichkeiten unterschiedlicher Analysen zur Verfügung, sie werden durch die gemeinsame Ausgangsbasis auch vergleichbar. Dies ist unter ROOT durch die vielen verschiedenen integrierten Anwendungen bisher nicht möglich. [41][7].

Weiterhin erleichtert das benutzerfreundliches Design von RapidMiner die Bedienung, dies ist gegenüber ROOT vom großen Vorteil. Die Benutzung von ROOT erfordert fundierte Vorkenntnisse. Diese umfassen sowohl den Aufbau der verwendeten ROOT-Klassen als auch Kenntnisse der Programmiersprache C++ sowie die Bedeutung des ROOT-System selbst.

Es bietet sich daher an, die Menge an Messdaten nicht nur mit ROOT, sondern auch mit RapidMiner analysieren zu können. Diesbezüglich wird es eine weiterführenden Arbeit geben, die die physikalischen Messdaten mit Hilfe von RapidMiner auf den tatsächlichen Informationsgehalt untersucht [35].




(a) Fehlende Schnitstelle (b) Grundidee

**Abbildung 1.1:** Skizzierung des Problems

## 1.2 Ziel der Diplomarbeit

Die Datenanalysesoftware „ROOT“ besitzt ihr eigenes Dateiformat [27]. Durch diese Diplomarbeit soll die Analyse von Messdaten, die im „ROOT-Dateiformat“ gespeichert wurden, in RapidMiner ermöglicht werden.

Im Rahmen dieser Diplomarbeit soll ein Konzept für eine Schnittstelle zwischen Dateien der physikalischen Datenanalysesoftware „ROOT“ und dem frei zugängliche XML Format erstellt und implementiert werden. Das Problem wird skizzenhaft in Abbildung 1.1 dargestellt. Die konvertierten Daten sollen anschließend dem Datenanalysetool RapidMiner Web-basiert zur Verfügung gestellt werden. Die Nutzbarkeit des Ansatzes soll an Hand von 3 Szenarien demonstriert werden. Bei der Konvertierung der Daten soll ihre Semantik erhalten bleiben.

## 1.3 Kooperation

Diese Diplomarbeit ist eine Zusammenarbeit des Lehrstuhls für Künstliche Intelligenz, der Fakultät für Informatik und des Lehrstuhls Experimentelle Physik 5, Astroteilchenphysik der Fakultät für Physik. Beide Fakultäten gehören der Technischen Universität Dortmund an.

Der Lehrstuhl für Künstliche Intelligenz der Fakultät Informatik beschäftigt sich mit dem Gebiet des maschinellen Lernens [13]. Insbesondere steht dabei die praktische Umsetzung von Lernverfahren auf gängige Probleme im Vordergrund.

Die Astroteilchenphysik der Technischen Universität Dortmund befasst sich mit der Analyse von Signalen von extragalaktischen Quellen, sowohl in Gamma-Strahlung als auch in Neutrino-Strahlung [11]. Hier stehen insbesondere Aktive galaktische Kerne (AGN) und kurzzeitigen Gamma-Strahlungsausbrüche (GRB) im Mittelpunkt.

## 1.4 Verwandte Arbeiten

Die Studienarbeit von Marius Helf „Genetische Merkmalsselektion für die Gamma-Hadron-Separation im MAGIC-Experiment“ im August 2009 setzt sich mit dem Thema der Analyse der ROOT-Daten [35] auseinander. Dabei kombiniert ein evolutionärer Algorithmus vorhandene Parameter mit unterschiedlichen Rechenoperationen und wählt daraus mit Hilfe einer Bewertungsfunktion geeignete Merkmale für die Charakterisierung von Gammastrahlungsereignissen und Hadronenenereignissen aus.

## 1.5 Aufbau der Diplomarbeit

Diese Diplomarbeit ist in mehrere Teile gegliedert. Im ersten Kapitel werden die Einleitung des Themas, die zugrunde liegende Motivation und die Ziele der Arbeit dargestellt.

Im zweiten Kapitel werden die relevanten Hintergründe bezüglich der Motivation eines ROOT-Adapters erläutert. Diesbezüglich werden einige ausgewählte physikalische Hintergründe der ROOT-Software vorgestellt.

Anschließend wird im dritten Kapitel auf die Grundlagen der verwendeten Software eingegangen. Dazu werden die beiden zugrunde liegenden Softwaretools RapidMiner und ROOT vorgestellt und der Aufbau der ROOT-Dateien beschrieben. Des weiteren werden die beiden verwendeten Grundwerkzeuge XML und XSLT dargestellt.

Das vierten Kapitel gibt einen Überblick über die Anforderungen an das System und über das entwickelte Systemdesign.

Im fünften Kapitel der Diplomarbeit wird die Realisierung des implementierten Systems beschrieben. Dieser Bereich der Arbeit beschreibt die Programmarchitektur und verschiedene Implementationsdetails. Zudem vermittelt es notwendiges Wissen für die Möglichkeit der Erweiterung des bestehenden Systems.

Die Möglichkeiten des Systems und seine Bedienung werden im sechsten Kapitel an Hand verschiedener, ausgewählter Beispiele erläutert.

Das letzte Kapitel fasst die Diplomarbeit zusammen. Zudem werden die Ergebnisse der Arbeit diskutiert und ein Ausblick über die Möglichkeiten zukünftiger Erweiterungen aufgezeigt.

# Kapitel 2

# Physikalischer Hintergrund

Im folgenden Kapitel werden die beiden Hauptanwendungsgebiete des ROOT-Analyseprogramms beschrieben. Die Software wird sowohl in der Teilchenphysik (Elementarteilchenphysik) als auch in der Astrophysik (Astroteilchenphysik) eingesetzt. ROOT wurde am CERN für die Speicherung und Analyse der Messdaten des Large Hadron Collider (LHC) entwickelt [15]. Kapitel 2.1 gibt einen Einblick über die Teilchenphysik und den Large Hadron Collider (LHC) des CERNs. Im Kapitel 2.2 soll ein Überblick über den zweiten Anwendungsbereich, die Astrophysik gegeben werden. Die beiden großen Projekten MAGIC und IceCube werden beschrieben. Anhand von ROOT-Dateien aus diesem Bereich wird die Handhabung des ROOT-Adapters dargestellt.

## 2.1 Teilchenphysik / Elementarteilchenphysik

Die Teilchenphysik beschäftigt sich mit dem elementaren Aufbau der Materie und den fundamentalen Wechselwirkungen zwischen den verschiedenen Bausteinen. Der Schwerpunkt liegt dabei auf den Elementarteilchen. Die Elementarteilchenphysik untersucht dabei die kleinsten Bausteine der Materie und deren Wechselwirkungen.

Die verschiedenen Fragestellungen und die Überprüfung der verschiedenen Theorien der Teilchenphysik werden hauptsächlich durch experimentelle Versuche innerhalb von Teilchenbeschleunigern untersucht. Einer der modernsten Teilchenbeschleuniger der Welt ist der Large Hadron Collider (LHC) des Conseil Européen pour la Recherche Nucléaire (CERN). Die Analysesoftware ROOT wurde für die Analyse der Daten die am Cern durch die verschiedenen Experimente entstehen, entwickelt.

### 2.1.1 LHC

Der Large Hadron Collider (LHC) ist ein ringförmiger Teilchenbeschleuniger für Hardronen am Kernforschungszentrum CERN bei Genf in der Schweiz. Der Ring ist etwa 27 Kilometer lang und befindet sich unter der Erde in einer Tiefe von 50 bis 175 Metern. In ihm werden

Hadronen, wie Protonen und Bleiionen auf nahezu Lichtgeschwindigkeit beschleunigt und zur Kollision gebracht. Der LHC besteht aus 9300 Magneten, mit einer maximalen Stärke von von 8,33 Tesla. Die verschiedenen Wechselwirkungen der entstehenden Teilchenschauer werden durch verschiedene Detektoren beobachtet.

Die Hadronen können an vier verschiedenen Punkten zur Kollision gebracht werden. Der Teilchenbeschleuniger besitzt verschiedene Detektoren, die vier großen ATLAS, CMS, LHCb, ALICE und 2 kleinere, TOTEM und LHCf. Mittels der Detektoren können teilchenphysikalischen Messungen durchgeführt werden.

## 2.2 Astrophysik / Astroteilchenphysik

Die Astroteilchenphysik beschäftigt sich mit den größten Strukturen im Universum bis hin zu den kleinsten Bausteinen der Materie und ihre Wechselwirkung. Dabei arbeitet sie eng mit der Astronomie, der Astrophysik, der Kosmologie und der Elementarteilchenphysik zusammen [1].

Die Astronomie und die Astrophysik untersuchen die Himmelskörper. Wurden früher hauptsächlich Himmelskörper im Wellenbereich des sichtbaren Lichtes beobachtet, wird heutzutage ein breiteres Spektrum der elektronischen Strahlung untersucht, die das menschliche Auge nicht mehr wahrnimmt. Diese können Radiowellen, infrarotes oder ultraviolettes Licht, Röntgenstrahlen und auch Gammastrahlen sein.
Die Kosmologie beschäftigt sich mit dem Universum als Ganzes. Dabei werden Fragen bezüglich der Entstehung und Entwicklung des Universums untersucht.

In der Astroteilchenphysik werden seltene hochenergetische Teilchen, die aus dem Universum ständig auf unsere Erde treffen (Kosmische Strahlung), mit empfindlichen Teleskopen nachgewiesen. Die Untersuchung dieser Teilchen, ihre Wechselwirkung und ihre Herkunft sind zentrale Fragen dieses Bereiches.

### 2.2.1 Kosmische Strahlung

Kosmische Strahlung ist eine Strahlung, bei der geladene Teilchen aus dem Weltall auf die Erde treffen. Diese Teilchen bestehen zum größten Teil aus Atomkernen, vor allem Protonen, daneben können es auch Elektronen sein. Diese Teilchen nennt man die „Primärstrahlung“, sie erreichen die Erdoberfläche nicht. Die Teilchen treten dann in Wechselwirkung mit anderen Teilchen aus der Atmosphäre und es entsteht ein „Teilchenschauer“ mit einer hohen Anzahl an so genannten „Sekundärteilchen“. Diese Strahlung wird auch „Sekundärstrahlung“ genannt und ist auf der Erde messbar [1].

Die Messung der Primärstrahlung würde nur durch Weltraumexperimente gelingen. Das Problem besteht aber darin, dass Teilchen mit hohen Energien aus dem Weltall nur

sehr selten auftreten. So wird die Atmosphäre von kosmischen Beschleunigern aus unserer Galaxie mit bis zu $10^{15}$ Elektrovolt gerade mal von einem Teilchen pro Jahr und Quadratmeter getroffen. Bei Energien eines Teilchens oberhalb von $10^{20}$eV nur noch alle hundert Jahre pro Quadratkilometer. Satelliten sind zu klein, um Teilchen mit höheren Energien messen zu können. Detektoren auf der Erde können einen größeren Bereich überwachen. Die entstandenen Teilchenschauer sind besser zu beobachten als ein einzelnes Teilchen. Dieser Zusammenhang wird in Abbildung 2.1 veranschaulicht.

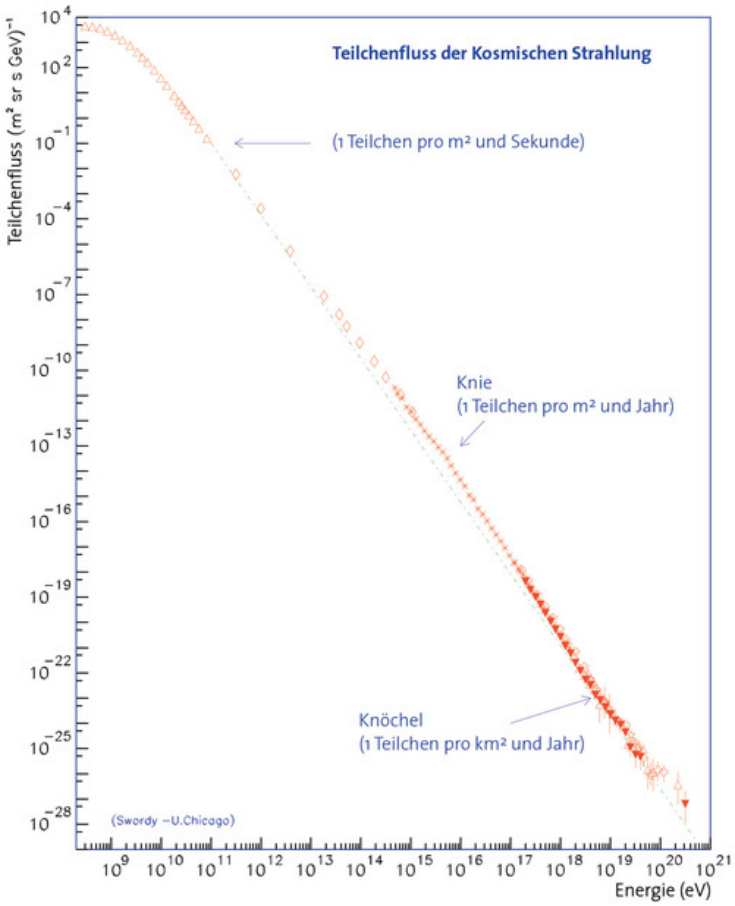



**Abbildung 2.1:** Teilchenfluss der Kosmischen Strahlung [21]

Mit Hilfe von Detektoren wird die „Sekundärstrahlung“ nachgewiesen. Aus den Eigenschaften der Sekundärteilchen werden Rückschlüsse auf die Art und die Energie des Primärteilchens vorgenommen. Es gibt verschiedene Detektoren für Sekundärstrahlung auf der Erde. Zwei Projekte sollen im folgenden kurz vorgestellt werden: Das „Magic-Projekt“ beschäftigt sich mit der hochenergetischen Gammastrahlung aus dem Universum (2.2.2). Das „IceCube-Projekt“ befasst sich mit der Suche nach einzelnen isolierten Objekten, so genannte Neutrinopunktquellen (2.2.3).

### 2.2.2 Das MAGIC-Projekt

Das Major Atmospheric Gammaray Imaging Cherenkov-Teleskop (MAGIC-Teleskop), ist ein Tscherenkow-Teleskop, das auf der Kanarischen Insel La Palma (28°45'4"N, 17°53'2"W) steht. Es befindet sich 2250m über den Meeresspiegel. Dieses Teleskop ermöglicht den erdgebundenen Nachweis von hochenergetischer Gammastrahlung aus dem Universum. Dabei wird der Energiebereich der elektromagnetischen Strahlung zwischen 30 und 300 GeV untersucht. Das MAGIC-Teleskop besitzt außerdem die Fähigkeit, kurzlebige Gammablitze, so genannte „Gamma Ray Bursts (GRBs)", zu messen.

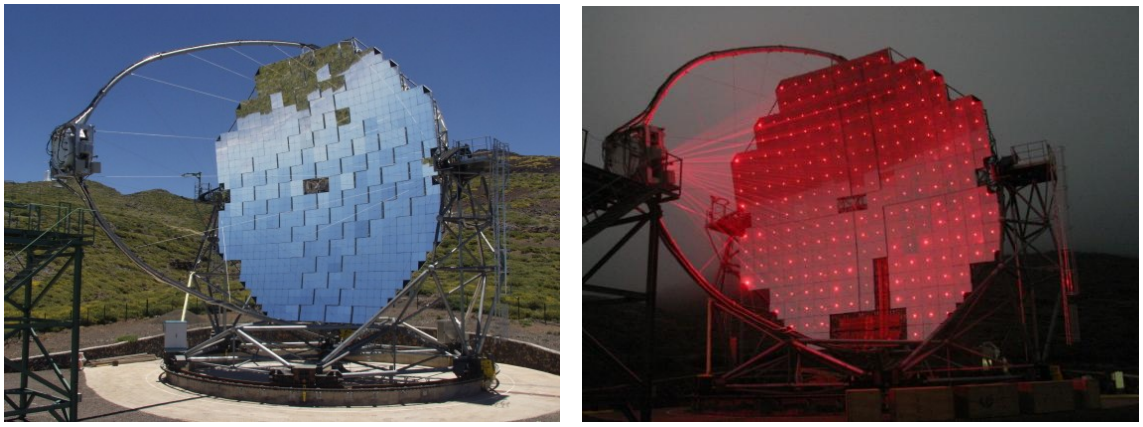


**Abbildung 2.2:** Aufnahmen des MAGIC-Teleskops [22]

MAGIC untersucht verschiedene Bereiche in der Astronomie. Es werden Aktive Galaxiekerne, Supernovaüberreste, Neutronensterne, Schwarze Löcher und ungewöhnliche GRBs beobachtet.

#### Funktionsweise

Tscherenkow-Teleskope können die Gammastrahlung nur indirekt nachweisen, indem die Reaktion der Strahlung mit der Erdatmosphäre beobachtet wird, die „Sekundärteilchenschauer". Diese sekundären Elektronen fliegen als ein eng gebündelter Strahl Richtung Erde und erzeugen dabei einen sichtbaren „Blitz", die so genannte Tscherenkow-Strahlung. Die Dauer des Blitzes beträgt nur einige Milliardstel Sekunden und kann auf der Erde mit Hilfe der Tscherenkow-Teleskope nachgewiesen werden. Die Funktionsweise ist in Abbildung 2.3 dargestellt.

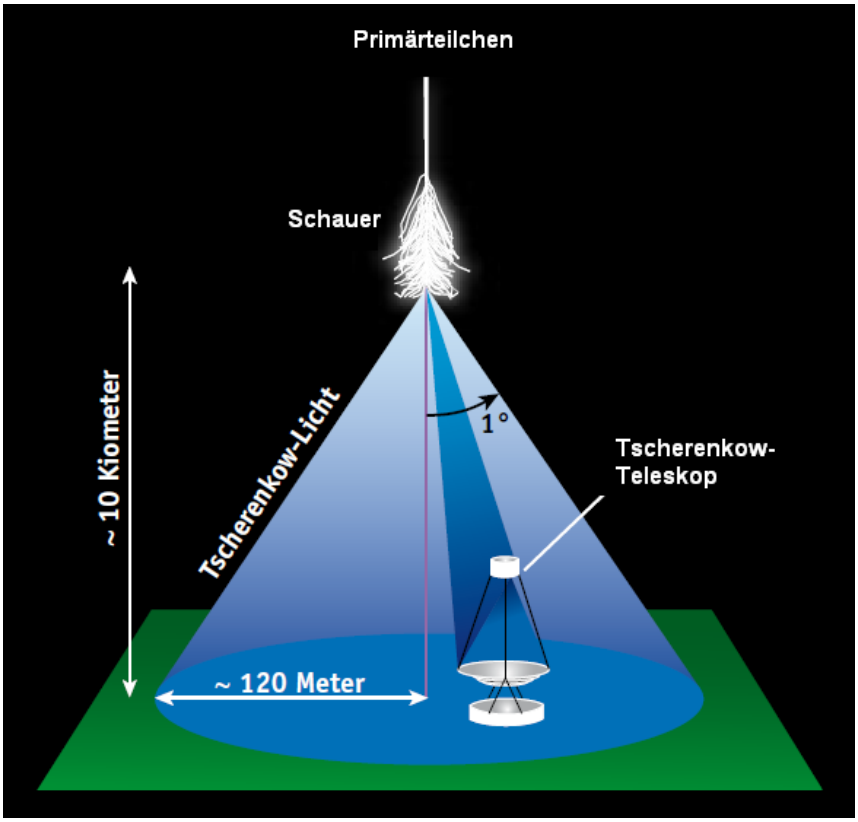



**Abbildung 2.3:** Funktionsweise eines Tschernkow-Teleskopes (auf Basis von [17])

### Tscherenkow-Effekt

Um eine Vorstellung über das Prinzip des Tscherenkow-Effekts zu geben, wird dieser kurz erläutert. Beim Tscherenkow-Effekt erzeugen bewegte Ladungsträger entlang ihren Flugbahn ein Magnetfeld. Die benachbarten Atome richten sich an diesem Magnetfeld aus. Dieser Vorgang wird Polarisation genannt. Die Atome werden zu elektrischen Dipolschwingungen angeregt, dabei werden elektromagnetische Wellen abgestrahlt. Die für die Polarisation benötigte elektromagnetische Strahlung kann sich maximal mit Lichtgeschwindigkeit in diesem Medium ausbreiten. Dadurch können sich die polarisierten Atome symmetrisch anordnen und die Wellen benachbarter Atome interferieren destruktiv, das bedeutet, sie löschen sich gegenseitig aus.

Wenn aber die Geschwindigkeit der bewegten Ladung höher ist als die Lichtgeschwindigkeit in diesem Medium, gibt es keine symmetrische Anordnung der polarisierten Atome. Die elektromagnetischen Wellen interferieren konstruktiv. Dadurch wird Energie in Form von Licht ausgestrahlt. Dieser Licht-"Blitz" kann dann beobachtet werden [10].

### Aufbau

Das Teleskop besteht aus einem großen parabolischen Spiegel. Dieser bewegliche Reflektor besitzt einen Durchmesser von 17 Meter und kann auf die jeweilige Quelle ausgerichtet werden. Der Spiegel besitzt eine Gesamtfläche von $239m^2$ und besteht aus 964 einzelnen quadratischen Spiegeln. Die Aufnahmen werden von einer Kamera gemacht. Dabei wird das in den Spiegeln einfallende Licht auf die Kamera gebündelt.

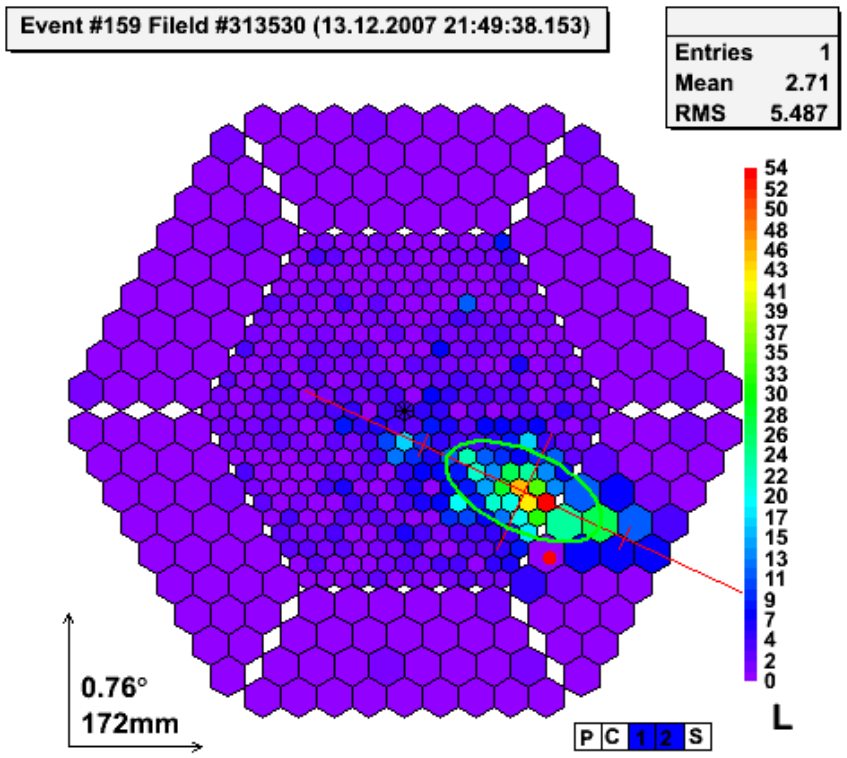



**Abbildung 2.4:** Aufnahme eines Beispielereignisses des MAGIC-Detektors

Die Kamera besteht aus 396 kleinen im Inneren liegenden und 180 größeren am Rand liegenden hexagonalen Photomultipliern. Die kleinen Photomultiplier haben einen Öffnungswinkel von 0.1 Grad, die großen von 0.2 Grad. Dieser Aufbau lässt sich gut in einer Beispielaufnahme in Abbildung 2.4 erkennen. Die Einfärbung in der Beispielaufnahme wird durch die Anzahl der detektierten Photoelektronen bestimmt. MAGIC besitzt eine aktive Spiegelkontrolle, mit Hilfe von Laserstrahlen werden die Spiegelsegmente von MAGIC automatisch justiert. (siehe Abbildung 2.2 [rechts])).

Die durch die MAGIC-Kamera aufgenommenen Bilder können unterschiedliche Gebilde wie Ringe oder Ellipsen enthalten. Aufgenommene Myonenereignisse sind durch eine ringförmige Abbildung zu erkennen. Die Unterscheidung von hadronischen und elektromagnetischen Schauern ist nicht ganz unkompliziert. Zur Unterscheidung wird in das aufgenommene und bereinigte Bild eine Ellipse gefittet (siehe Abbildung 2.4).

Bei jedem aufgenommenen Licht-Blitz wird der Aufnahmemodus des Teleskops getriggert, und auch die Intensität und Ankunftzeit eines jeden Pixels der Kamera. Es gibt 2 verschiedene Aufnamemodi. Diese werden benötigt, da das Tscherenkov-Licht nur sehr schwach ist und sich kaum von der Intensität des Hintergrundes abhebt. Die registrierten Ereignisse werden so von Untergrundsignalen gestört. Um diese herausfiltern zu können, muss noch ein dunkler Bereich des Himmels untersucht werden. Ein Aufnahmemodus besteht durch eine zusätzliche zeitnahe Untersuchung neben dem Beobachtungsraum. Beim zweiten Aufnahmemodus wird der Beobachtungsraums verschoben. Dieser ist dann nicht mehr genau auf die Quelle, die beobachtet werden soll, ausgerichtet und ermöglicht dadurch das Filtern der Daten.

### Magic II

Am 24./25. April wurde offiziell das zweites MAGIC-Teleskop auf der Kanareninsel La Palma der Öffentlichkeit vorgestellt. Es besitzt eine Spiegelgröße von $247m^2$ und steht in 85m Entfernung zum ersten Teleskop (siehe Abbildung 2.5). Die beiden Teleskope unterscheiden sich im Aufbau nur durch neue Aluminium-Reflektoren. Dadurch wird die Seitenlänge um 1m breiter. Die Fläche wird dadurch um das Vierfache größer, dies erlaubt es, niederenergetische Schauer besser beobachten zu können.

Die Nachweisgenauigkeit der Tscherenkow-Blitze wird durch die parallele Beobachtung durch beide MAGIC-Teleskope um den Faktor 3 verbessert. Dadurch erhofft man sich den Nachweis neuer galaktischer und extragalaktischer Quellen hochenergetischer Gamastrahlen [4].

**Abbildung 2.5:** Die MAGIC-Teleskope am Roque-de-los-Muchachos-Observatorium [22]

### 2.2.3 Das IceCub-Projekt

Das IceCube Neutrinoobservatorium ist ein Neutrinoteleskop, das zur Zeit im antarktischen Eis gebaut wird. Es soll bis 2011 fertig gestellt werden. Eines der Hauptziele der IceCube Kollaboration ist die Suche nach einzelnen isolierten Objekten, so genannten Neutrinopunktquellen. Dabei werden unter anderem hochenergetische Neutrinos aus kosmischen Quellen wie Supernovaüberreste, binäre Systeme und Blazare gesucht. Das fertige System soll etwa hundertmal so empfindlich sein wie bereits existierende Instrumente.

#### Aufbau

Das IceCube-Teleskop besteht aus Strängen, an denen Photodetektoren befestigt sind, die im Eis versenkt werden. Insgesamt sollen 4800 Sensoren an 80 Trossen befestigt werden. Dieser Aufbau ist in Abbildung 2.7 zu erkennen. Die Trossen haben etwa einen Abstand von 125 Metern. Die Sensoren an den Trossen sind alle 17 Meter angebracht und haben etwa die Größe eines Medizinballs. Damit deckt IceCube ein Volumen von einem Kubikkilometer ab. Die Sensoren werden in einer Tiefe zwischen 1450m bis 2450m im anarktischen Eis eingefroren. Das Eis besitzt dort eine Dicke von 3 Kilometern. Das Vorgängerprojekt „AMANDA-II“ 2.6 wird in IceCube integriert. Aktuell sind etwa 70% des Detektors fertig gestellt.

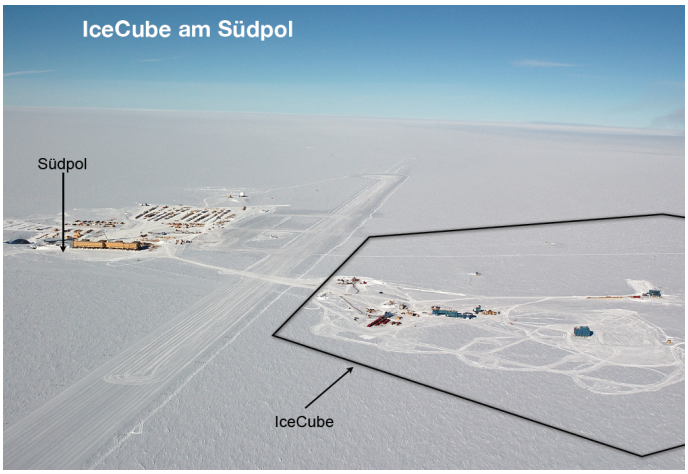



**Abbildung 2.6:** Die Gesamtfläche von IceCube [36]

Die gewonnenen Daten aus den Photodetektoren werden über optische Module digitalisiert und über Kabelstränge an die Oberfläche übermittelt. IceCube wird durch einen Luftschauerdetektor „IceTop“ ergänzt. Dieser besteht aus 160 großen Tanks die mit Eis gefüllt sind. IceTop wird nach der Fertigstellung 360 Photodetektoren besitzen, mit denen das erzeugte Tscherenkow-Licht aus den Luftschauern gemessen werden soll.

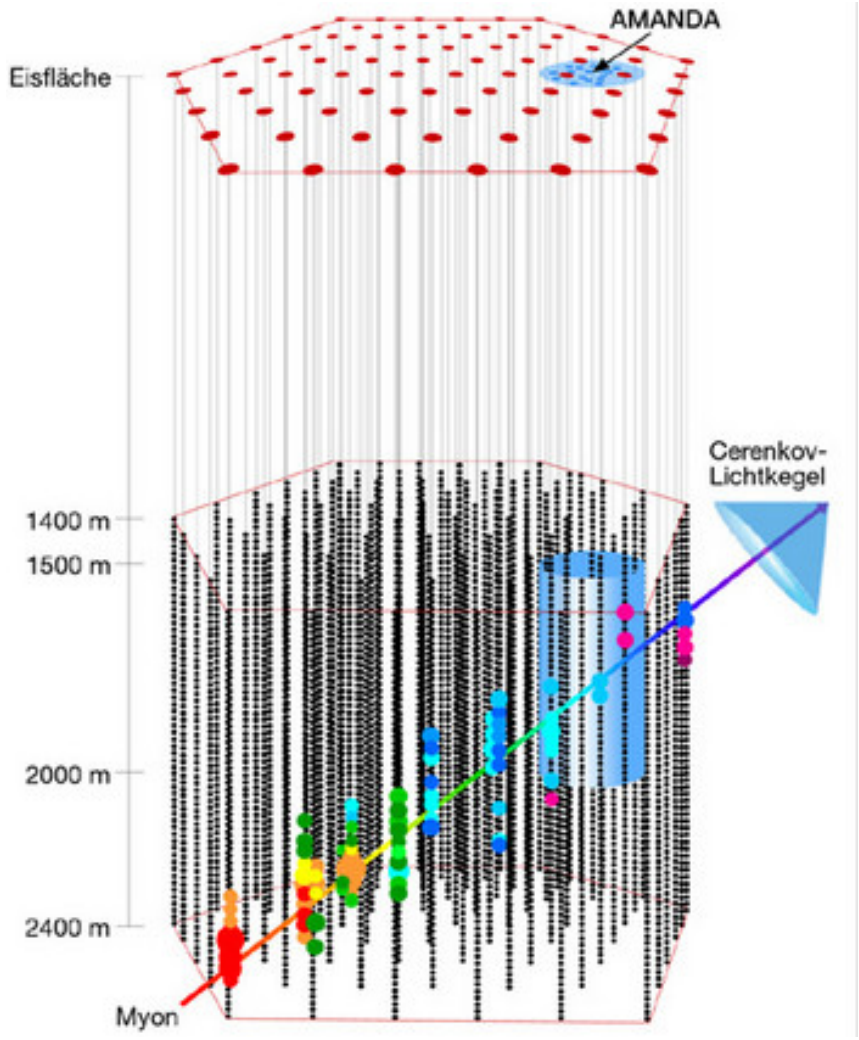



**Abbildung 2.7:** Schematischer Aufbau des IceCube Detektors

### Funktionsweise

Bisher ist es nicht gelungen, die Quelle kosmischer Strahlung zu bestimmen. Kosmische Quellen emittieren Neutrinos, ihr Nachweis erlaubt eine Identifikation der Quelle. IceCube wird für die Suche nach Neutrinos benutzt.

Neutrinos haben einen äußerst kleinen Wirkungsquerschnitt, weshalb sie die Erde erreichen, ohne mit Teilchen im interstellaren Raum in Wechselwirkung zu treten. Andere Teilchen werden auf ihrer Bahn abgelenkt und können aus dieser Grund nicht zum Nachweis der Richtung der Quellen genutzt werden. Dieser Zusammenhang ist in Abbildung 2.8 a) skizziert. Die geringe Wechselwirkung erschwert aber den Nachweis der Neutrinos.

Da diese Teilchen elektrisch neutral sind und auch kein magnetisches Moment besitzen, werden sie weder von elektrischen noch von magnetischen Feldern abgelenkt und erreichen die Erde deshalb ohne von ihrer ursprünglichen Bahn abgelenkt zu werden. IceCube ist

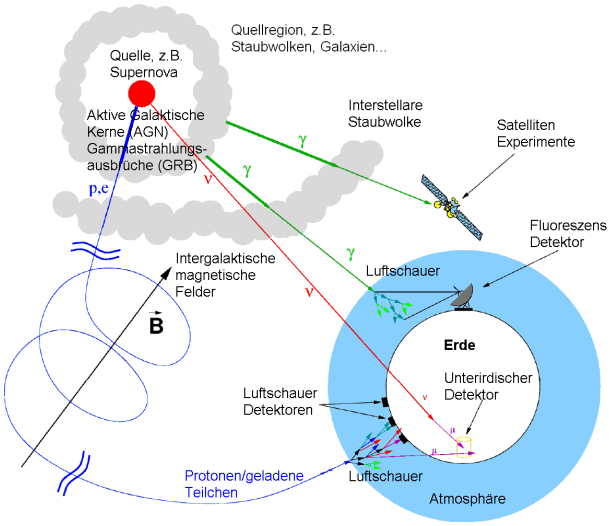



**Abbildung 2.8:** Weg der Kosmischen Strahlung durch das Universum [43]

in der Lage, Neutrinos zu messen, die durch die Erde dringen. Wenn es zu einer Wechselwirkung von Neutrinos mit den Wassermolekülen im antarktischen Eis kommt, werden Myonen erzeugt, die wiederum einen Tscherenkow-Lichtblitz erzeugen. Dieser Licht-Blitz wird von den Photodetektoren erfasst. Dabei wird die Intensität, die Richtung und der Verlauf des Blitzes gemessen. Dadurch lassen sich Rückschlüsse auf die Richtung des Neutrinos ziehen. Denn die Richtung der Myonen, welche identisch mit der der Neutrinos ist, kann dadurch bestimmt werden und somit die Richtung einer möglichen Quelle. Dieser Vorgang ist in Abbildung 2.7 zu erkennen. Das erzeugte Myon durchquert IceCube von unten links in der Abbildung nach oben rechts.

# Kapitel 3

# Software Grundlagen

Dieses Kapitel gibt einen Überblick über die Grundlagen im Bereich der verwendeten Software. Als erstes wird die Data-Mining-Software „RapidMiner“ vorgestellt, anschliessend das „ROOT“-System. ROOT besitzt ein eigenes Dateiformat, der Aufbau dieser Dateien wird in einem eigenen Bereich erläutert. Im weiteren Verlauf dieser Arbeit wird der Begriff „ROOT-Datei“ für die Ein-/Ausgabedateien des ROOT-Systems verwendet. Als letztes wird die „Extensible Markup Language“, kurz XML und die „Extensible Stylesheet Language Transformations“ (XSLT) vorgestellt, die als Grundwerkzeuge dieser Arbeit dienen.

## 3.1 RapidMiner

RapidMiner ist eine Open-Source-Software und steht unter der GNU Affero General Public Lizenz (AGPL) [3]. Es ist eine Umgebung für maschinelles Lernen und Data Mining Prozesse und wurde ursprünglich als YALE (Yet Another Learning Environment) am Lehrstuhl für Künstliche Intelligenz der Technischen Universität Dortmund entwickelt.

In RapidMiner wird ein Data Mining Prozess in Teilaufgaben unterteilt. Die Teilaufgaben werden durch sogenannte „Operatoren“ abgearbeitet. RapidMiner stellt mehr als 500 Operatoren für die verschiedenen Aufgabengebiete des Data Mining bereit [6]. Im folgenden soll ein kurzer Überblick über die Bereiche gegeben werden.

- **Ein- und Ausgabe Operatoren**
  Diese Operatoren stellen Mechanismen für die Ein- und Ausgabe von Daten verschiedenster Formate bereit. Dies sind unter anderem Formate wie „Arff“ und „csv“ aus dem Data Mining Bereich und dem Bereich der Lernverfahren, wie auch allgemeine Formate. Dies können beispielhaft „Excel Tabellen“ oder Daten aus Datenbanken sein.
  Im Rahmen dieser Diplomarbeit wird ein weiterer Eingabe-Operator bereit gestellt, mit dessen Hilfe Daten des „ROOT Formats“ Web-basiert übertragen werden.

- **Datentransformations und -vorverarbeitungs Operatoren**
  Die Operatoren für Datentransformation können unter anderem numerische Attribute in norminale Attribute umwandeln. Weitere Operatoren zur Datenvorverarbeitung erfüllen verschiedene Aufgaben, diese sind beispielhaft neben der Normalisierung die Dimensionsreduktion.

- **Data Mining Operatoren**
  Operatoren des Data Mining Bereichs stellen viele verschiedene Verfahren für die Regression, Klassifikation und das überwachte und unüberwachte Lernen bereit. Operatoren für Lernverfahren sind unter anderem Entscheidungsbäume, Naive Bayes oder Clustering. Zudem gibt es Weka Operatoren, dies sind Lernverfahren und Merkmalsevaluatoren aus der „Machine Learning Bibliothek Weka“. So genannte Merkmaloperatoren stellen Funktionalitäten für die Merkmalsauswahl, die Merkmalsgewichtung, die Merkmalsrelevanz, die Merkmalskonstruktion und die Merkmalsextraktion aus Reihendaten bereit. Ausserdem gibt es noch Meta-Operatoren, die helfen, den Data Mining Prozess zu optimieren. Dazu gehören unter anderen Operatoren für die Parameteroptimierung oder Schleifen und Iterationen.

- **Evaluierungs Operatoren**
  Operatoren, die Validierungs- und Evaluierungsschemata bereit stellen, um die Performanz der Data Mining Prozesse auf den zu Grunde liegende Daten zu schätzen.

- **Visualisierungs Operatoren**
  Diese Operatoren stellen Mechanismen zur Darstellung von Ergebnissen und Daten oder zum Protokollieren von Informationen bereit. Diese beinhalten unter anderem Modellvisualisierungen, Histogramme oder Plots.

Die Operatoren können mittels einer graphische Benutzeroberfläche (GUI) nahezu beliebig verschachtelt werden. Diese Data Mining Prozesse werden in RapidMiner „Experimente“ genannt. Die Experimente können gespeichert werden und stehen zur Wiederverwendung zur Verfügung. Neben den bereitgestellten Operatoren können auch eigene entwickelt werden und mittels eines Plugin-Verfahrens in RapidMiner integriert werden.

RapidMiner kann als Standardsoftware oder als Bibliotheck verwendet werden. Es ist in der Programmiersprache Java geschrieben und dadurch plattformunabhängig. Durch seine Java-API kann RapidMiner auch in eigenen Java-Anwendungen benutzt werden.

Der Prozessaufbau und die Operatoren werden intern durch XML beschrieben. Gemeinsam mit der graphischen Benutzeroberfläche wird eine vollständige integrierte Entwicklungsumgebung (IDE) für das Data Mining bereit gestellt.

In RapidMiner werden importierten Daten als ExampleSets bezeichnet. ExampleSets sind Beispielmengen. Die Ausgabe von Operatoren werden als ResultSets benannt. Sie

stellen die Ergenismengen dar. Innerhalb dieser Diplomarbeit werden die in ROOT enthaltenen Daten in RapidMiner eingelesen und als ExampleSets gespeichert. ExampleSets kann man sich innerhalb von RapidMiner als eine Tabelle vorstellen die, die Daten enthält. Dabei wird jede Spalte durch ein Attribut beschrieben, es definiert die Eigenschaft einer Spalte. Ein Example ist eine Zeile dieser Tabelle. Das Example beinhaltet eine Menge von Werten für alle Attribute.

## 3.2 ROOT

Das ROOT-System ist ein objektorientiertes Softwarepaket zur Datenhaltung, Datenanalyse und Datenvisualisierung, das insbesondere in der Hochenergiephysik verwendet wird [7]. Das System wurde am CERN, der Europäischen Organisation für Kernforschung entwickelt. Ihr Hauptanwendungsgebiet ist das effiziente Speichern, Lesen und Verarbeiten von großen Datenmengen. ROOT bietet dazu eine eine C++-Klassenbibliothek an, einen C++-Interpreter (CINT), der Zugang zu den ROOT-Klassen ermöglicht und eine graphische Benutzeroberfläche.

ROOT besitzt desweiteren folgende Funktionen:

- Datenhaltung
  - Speicherung und Strukturierung von Daten in Bäumen
  - Eigenes ROOT-Dateiformat zur Speicherung von ROOT-Objekten
- Datenanalyse
  - Anpassen von Funktionen an Daten (fitting)
  - mathematische Funktionen
  - Unterstützung von Distributed Computing
  - Weiterverarbeitung der Daten (z.B. MARS 6.1.1)
- Datenvisualisierung
  - Erstellen von Histogrammen (2D/3D) und Graphen
  - Datenvisualisierungsmöglichkeiten (2D/3D)
  - Grafik-Export in verschiedenen Formate (u.a Postscript, EPS, PDF, PNG)

ROOT kann sowohl über eine graphische Benutzeroberfläche (GUI) oder über eine Kommandozeile gesteuert werden, als auch über Batch-Skripte, die ROOT-Befehle enthalten. Benutzerdefinierte Erweiterungen werden mit Hilfe des C++-Kompilierers übersetzt und mit der ROOT-Bibliothek verbunden oder direkt durch den C++-Interpreter ausgeführt.

Die ROOT-Architektur stellt eine Vielzahl an Klassen für die oben genannten Zwecke bereit. So gibt es Basisklassen, die Standardmethoden und Objektstrukturen definieren und Containerklassen, die große Datenmengen verwalten können. Dazu gehören auch so genannten Baumklassen, die zur strukturierten Speicherung der Daten eingesetzt werden. Zudem erlauben die Baumklassen den Einsatz speziell definierter Benutzerklassen und Containerklassen, dessen Methode zur Analyse der Daten verwendet werden können. Weitere Klassen und Funktionen dienen der Datenanalyse sowie der Datenvisualisierung.

Die Grundlage der ROOT-Architektur ist eine geschachtelte Klassenhierarchie mit über 1200 Klassen, die in über 60 Bibliotheken eingeteilt sind. Diese sind in 19 Hauptkategorien eingeteilt. Dabei ist der Großteil der Klassen von der Klasse TObjekt abgeleitet. Diese Klasse stellt eine gemeinsame Basis, meist virtuell deklarierter Methoden bereit.

## 3.3 Aufbau der ROOT-Dateien

In diesem Kapitel wird der allgemeine Aufbau von ROOT-Dateien beschrieben. Der Aufbau der ROOT-Dateien bildet die Grundlage für die Verarbeitung der ROOT-Daten mit Hilfe des in dieser Diplomarbeit zu erstellenden XML-Konverters. Bei der Beschreibung des Aufbaus der ROOT-Dateien wird auf die logische und physikalische Struktur eingegangen.

ROOT ist erstellt worden zur Verarbeitung großer Datenmengen [8]. Die Struktur der ROOT-Dateien folgt dieser Konzeptionierung. Betrachtet man den logischen Aufbau, so ist eine ROOT-Datei wie ein UNIX-Datei-System aufgebaut. Es beinhaltet Verzeichnisse und Objekte. Dabei ist die Verzeichnistiefe unbeschränkt und die Objekte können beliebig tief verschachtelt sein. Der Zugriff auf diese Benutzerdaten erfolgt unter ROOT durch das ROOT-I/O-System.

Das ROOT-I/O-System beinhaltet verschiedene Abstraktionsebenen. Diese sind eine Kernebene, eine Objektebene und eine weitere Ebene, welche optimiert ist für die Speicherung von großen Objektsammlungen. Die Kernebene ist zuständig für das Lesen und Speichern der Dateisystemstruktur. Die Objektebene stellt die Funktionalitäten für das Lesen und Speichern der Objekte bereit. Die unterste Ebene bietet fertige Klassen für die Speicherung der großen Datenmengen.

Der physikalische Aufbau spiegelt das logische Layout wieder. Der exakte Aufbau wird im folgenden beschrieben [30].

### 3.3.1 ROOT-Datei

Eine ROOT-Datei besteht immer aus einem Datei-Header und einem Datenbereich, siehe Abbildung 3.1.

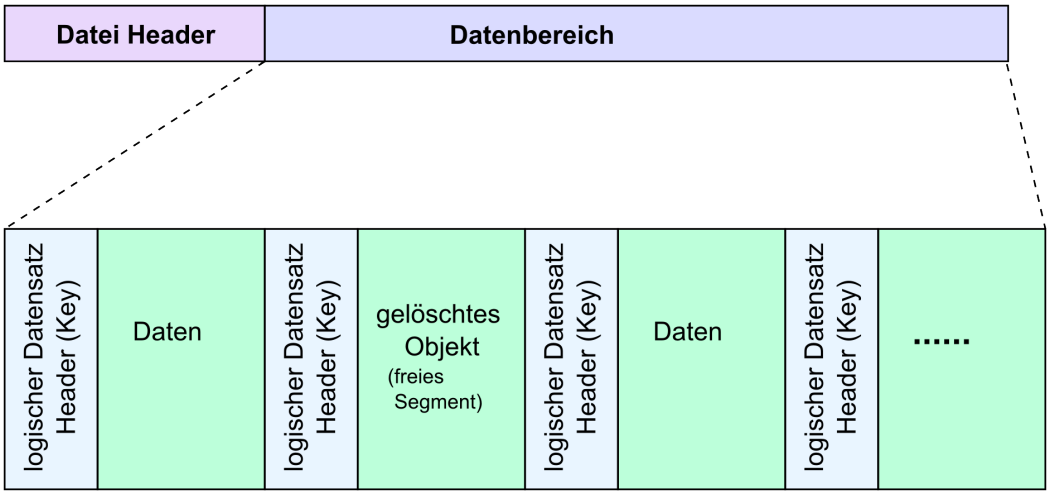



**Abbildung 3.1:** Aufbau der ROOT-Datei

Der Datei-Header steht am Anfang jeder ROOT-Datei und enthält alle grundlegenden Informationen über die Datei. Die Gesamtlänge des Datei-Headers ist abhängig von der Dateigröße, da der Header Sprungadressen enthält. Die Sprungadressen innerhalb der ROOT-Dateien sind 4 Bytes lang. Für sogennante „große Dateien“, dies sind Dateien die mindestens 2 GB groß sind, werden 8 Bytes benötigt. Unterschieden wird mittels der festen Headerinformation „fVersion“, welches die Version des Dateiformates angibt. Tabelle 3.1 gibt den genauen Aufbau des Datei-Headers wieder. So enthält bespielsweise das Feld „fBEGIN“ einen Zeiger auf den Anfang des Datenbereichs. Weitere Zusatzinformationen ermöglichen das Navigieren innerhalb dieser ROOT-Datei. So werden wichtige Sprungadressen wie etwa das Feld „fSeekInfo“ vermerkt. „fSeekInfo“ enthält einen Zeiger auf die Beschreibung benutzerdefinierter Objekte, die so genannten „TStreamerInfos“, siehe Kapitel 3.3.3.

### 3.3.2 Datenbereich

Der Datenbereich enthält einen oder mehrere Datensätze beliebiger Länge. Jeder Datensatz besitzt einen logischen Datensatz-Header, den so genannten „Key“ und die Objektdaten. Der Header enthält beschreibende Informationen zum gespeicherten Objekt, unter anderem Namen, Objekttyp und Länge des Objektes (siehe Tabelle 3.2 ). Er besitzt für alle Objektdaten das gleiche Format.

| Byte (kleine) | Byte (große Datei) | Bezeichner | Beschreibung |
|---|---|---|---|
| 0 -> 3 | | „root“ | ROOT-Datei Bezeichner |
| 4 -> 7 | | fVersion | Version des Dateiformats |
| 8 -> 11 | | fBEGIN | Sprungadresse: erster Datenblock |
| 12 -> 15 | 12 -> 19 | fEND | Sprungadresse: erstes freies Wort am Dateiende |
| 16 -> 19 | 20 -> 27 | fSeekFree | Sprungadresse: FreeSegment-Datenblock |
| 20 -> 23 | 28 -> 31 | fNbytesFree | Anzahl der Bytes im FreeSegments-Datenblock |
| 24 -> 27 | 32 -> 35 | nfree | Anzahl freier Datenblöcke |
| 28 -> 31 | 36 -> 39 | fNbytesName | Anzahl der Bytes in TNamed zur Erstellungszeit |
| 32 -> 32 | 40 -> 40 | fUnits | Anzahl der Bytes für Dateisprungadressen |
| 33 -> 36 | 41 -> 44 | fCompress | Zip-Kompressionsstufe |
| 37 -> 40 | 45 -> 52 | fSeekInfo | Sprungadresse: TStreamerInfo-Datensatz |
| 41 -> 44 | 53 -> 56 | fNBytesInfo | Anzahl der Bytes im TStreamerInfo-Datensatz |
| 45 -> 62 | 57 -> 74 | fUUID | Datei-Identifikationsnummer |

**Tabelle 3.1:** Aufbau des Datei-Headers in ROOT

Die Datenobjekte können aus Instanzen folgender Klassen bestehen:

1. TDirectory
2. TFile
3. KeysList
4. FreeSegments
5. TStreamerInfos
6. TRef,TRefArray, und TProcessID
7. Datensätze mit benutzerdefinierten Objekten

Die oben genannten Datensätze lassen sich in 2 Gruppen unterteilen. Einerseits die von ROOT bereitgestellten Klassen für Objekte, die Instanzen sind so genannte Kernobjekte und anderseits Instanzen benutzerdefinierter Klassen.

### 3.3.3 Repräsentationen von ROOT Objekte

ROOT stellt drei Arten von Klassen von Kernobjekten bereit. Die erste Gruppe wird benutzt für die Beschreibung der ROOT-Datei, ihr gehören die Objekte des Typs TFile, TDirectory, KeysList und FreeSegments an. Die zweite Gruppe enthält das Objekt des

| Offset (Byte) (kleine Datei) | Offset (Byte) (große Datei) | Bezeichner | Beschreibung |
|---|---|---|---|
| 0 -> 3 | | Nbytes | Länge des komprimierten Objektes |
| 4 -> 5 | | Version | TKey Versionsnummer |
| 6 -> 9 | | ObjLen | Länge des unkomprimierten Objektes |
| 10 -> 13 | | Datime | Datum/Zeit der Erstellung des Objektes |
| 14 -> 15 | | KeyLen | Länge des Datensatz-Headers |
| 16 -> 17 | | Cycle | Keyzyklus |
| 18 -> 21 | 18 -> 25 | SeekKey | Zeiger zum eigenen Datensatz |
| 22 -> 25 | 26 -> 33 | SeekPdir | Zeiger zum Directory-Header |
| 26 -> 26 | 34 -> 34 | lname | Anzahl der Bytes im ClassName Feld |
| 27 -> ... | 35 -> ... | ClassName | Klassenname des Objektes |
| 1 Byte | | lname | Anzahl der Bytes im ObjektName Feld |
| ... -> ... | ... -> ... | Name | Name des Objektes |
| 1 Byte | | lTitle | Anzahl der Bytes im ObjektTitel Feld |
| ... -> ... | ... -> ... | Title | Titel des Objektes |
| ... -> ... | ... -> ... | DATA | Objektdaten |

**Tabelle 3.2:** Aufbau des Datensatz-Headers in ROOT

Typs TStreamerInfo, es beschreibt den Aufbau Benutzerdefinierter Objekte. Zur dritten Gruppe gehören die Objekte des Typs TRef, TRefArray und TProcessID. Sie enthalten Objektreferenzen auf persistente Objekte.

Dieser Bereich der Arbeit soll einen kurzen Überblick über die genannten Objekttypen geben (siehe Anhang A).

### TDirectory

Der Datensatz der Klasse TDirectory beschreibt die Verzeichnisstruktur innerhalb der ROOT-Datei. Dabei kann eine ROOT-Datei beliebig viele TDirectory-Datensätze enthalten. Diese stellen jeweils ein Unterverzeichnis des Verzeichnisbaumes dar. Er liegt immer in unkomprimierter Form vor (Anhang A.2).

### TFile

Der Datensatz der Klasse TFile beschreibt entweder die ganze ROOT-Datei oder nur die Wurzel der Verzeichnisstruktur. Pro ROOT-Datei gibt es einen TFile-Datensatz, er steht am Anfang des Datenbereichs. Er liegt immer in unkomprimierter Form vor. TFile ist eine abgeleitete Klasse von TDirectory. Sie beinhaltet Informationen, wann ein Verzeichnis erstellt wurde und wann es zuletzt verändert wurde. Zudem besitzen Objekte des Typs TFi-

le Informationen über Sprungadressen zum Elternverzeichnis sowie zur verzeichniseigenen Liste aller enthaltenen Datensätze (Anhang A.3).

#### KeysList

ROOT-Dateien sind sequentiell aufgebaut. Um wahlfreien Zugriff zu ermöglichen, wird die „KeysList“ benutzt. Der Datensatz der Klasse KeysList existiert einmal für jeden TFile- oder TDirectory-Datensatz. Dieser enthält einen Index aller Datensatz-Header, die so genannten „Keys“. Jeder Key enthält, wie bereits erwähnt, beschreibende Informationen über den jeweiligen Datensatz. Dazu gehören aber keine ROOT interne Datensätze, wie „externe Baskets“. Diese dienen der separaten Speicherung von Daten und werden durch ROOT automatisch in den zugehörigen Datensatz integriert. Ein wichtiges Merkmal der KeysList ein Zeiger auf die Position des Datensatzes in der Datei. Jeder Key in der KeysList ist eine genaue Kopie des jeweiligen Datensatz-Headers (Anhang A.4).

#### FreeSegments

Eine ROOT-Datei kann nicht mehr benutzte Datensätze enthalten. Dieser freie Platz steht ROOT zur Speicherung von neuen Daten zur Verfügung. Der Datensatz des Typs „FreeSegments“ enthält einen Index dieser freien Segmente. Diese sind als sequentielle Liste mit Anfangs- und Endposition der freien Segmente ausgeführt. Diese Liste enthält ein zusätzliches Element, welches den freien Speicherplatz am Ende der Root-Datei kennzeichnet (Anhang A.5).

#### TSreamerInfos

Jede ROOT-Datei enthält genau einen TStreamerInfo-Datensatz.
„TStreamerInfos“ enthält beschreibende Informationen über den Aufbau sich selbst identifizierender Datenobjekte. Zu diesen Datensatzobjekten, die in den TStreamerInfos beschrieben werden, gehören alle Klassen, die in einem der Datensätze dieser ROOT-Datei vorkommen. Die oben beschriebenen Kernobjekte werden in den TStremerInfos nicht beschrieben, es sei denn, dass diese innerhalb anderer Objekte innerhalb der Datensätze benutzt werden.

In Tabelle Tabelle 3.3 ist ein kurzer Auszug der TStreamerInfo einer ROOT-Datei abgebildet. Diese TStreamerInfo Auszug beschreibt sechs verschiedene ROOT-Objekte. Diese sind TNamed, TObject, TTree, TBranchElement, TBranch und MRawRunHeader. Bis auf MRawRunHeader sind es alles von ROOT definierte Objekte. MRawRunHeader ist ein MARS-Objekt. MARS ist eine Erweiterung-Applikation von ROOT und wird in Kapitel 6.1.1 beschrieben. Die verschiedenen Einträge innerhalb der Tabelle werden im folgenden kurz erläutert.

Diese TStreamerInfo-Objekte sind als Liste gespeichert. Zu jedem Objekt wird seine Klassenversion gespeichert, da eine Klasse in verschiedene Versionen vorliegen kann. Siehe Tabelle 3.3 Zeile 1 und 5.

Ein TStreamerInfo-Objekt enthält eine Auflistung seiner Streamer-Elemente. Jedes dieser Elemente beschreibt eine Basisklasse der Objekte oder ein Datenelement des Objektes. Wenn dieses Objekt nicht aus atomaren Typen, sondern weiteren Klassen bestehen, so finden sich auch diese in den TStreamerInfos wieder. Damit lässt sich jedes Objekt in seine atomaren Typen, wie z.B, „Integer“ oder „Boolean“ zerlegen. Siehe Tabelle 3.3 Zeile 2 und 5.

Sollte ein Datenelement auf einen atomaren Typ verweisen, so kann dieser nicht nur als atomarer Typ vorliegen, sondern auch als Array fest definierter Größe. Solche Arrays werden durch das Format „[Länge]“ in den TStreamerInfos gekenzeichnet, siehe Tabelle 3.3 Zeile 19.

Objektklassen, die eigene Ein- und Ausgaberoutinen in ROOT besitzen, sind nicht von den TStreamerInfos erfasst. Einige der internen ROOTIO-Klassen sind zwar in den TStreamerInfos vorhanden werden aber nicht benutzt (Anhang A.6).

#### TRef,TRefArray, und TProcessID

Die drei Objekttypen TRef, TRefArray und TProcessID werden benutzt, um das Speichern von Zeigern auf persistente Objekte zu unterstützen. Persistente Objekte sind Objekte, welche die Ausführungszeit des Prozesses überdauern. In diesem Fall handelt es sich um Objekte die von einem ROOT-Prozess erstellt und gespeichert wurden. Ein ROOT-Prozess ist ein von dem ROOT-System ausgeführtes C/C++ Programm.

Nur „TProcessID“ ist ein eigener Datensatz. Dieser enthält Informationen über einen ROOT-Prozess. Die wichtigste Information des TProcessID ist seine eindeutige Identifikationsnummer, sie ist im Feld „Title“ gespeichert (Anhang A.7).

TRef und TRefArray bilden die abgespeicherte Repräsentation von Zeigern auf Objekte im Speicher. Jedes Objekt enthält ebenfalls eine eindeutige Identifikationsnummer und die Identifikationsnummer des Prozesses, von dem es erzeugt wurde. Die Referenz enthält ebenso diese Identifikationsnummern. Siehe Anhang A.8 und A.9

### 3.3.4 Benutzerdefinierte Objekte

ROOT bietet die Möglichkeit, neben den Objekten von ROOT-Klassen auch Objekte von benutzerdefinierten Klassen zu speichern. Datensätze können diese benutzerdefinierte Objekte oder Sammlung von Benutzerobjekten (7) beinhalten. Die Struktur dieser Objekte wird, bis auf wenige Ausnahmen in den TStremerInfos gespeichert, siehe Kapitel 3.3.3.

| | | | | | |
|---|---|---|---|---|---|
| 1* | StreamerInfo for class: TNamed, version=1, checksum=0xfbe93f79 | | | | |
| 2* | TObject | BASE | offset=0 | type=66 | Basic ROOT object |
| 3* | TString | fName | offset=0 | type=65 | object identifier |
| 4* | TString | fTitle | offset=0 | type=65 | object title |
| 5* | StreamerInfo for class: TTObject, version=1, checksum=0x52d96731 | | | | |
| 6* | UInt_t | fUniqueID | offset=0 | type=13 | object unique identifier |
| 7* | UInt_t | fBits | offset=0 | type=15 | bit field status word |
| 8* | StreamerInfo for class: TTree, version=16, checksum=0xbe994a04 | | | | |
| 9* | TNamed | BASE | offset=0 | type=67 | The basis for a named object |
| 10* | TAttLine | BASE | offset=0 | type=0 | Line attributes |
| 11* | TAttFill | BASE | offset=0 | type=0 | Fill area attributes |
| 12* | TAttMarker | BASE | offset=0 | type=0 | Marker attributes |
| 13* | Long64_t | BASE | offset=0 | type=0 | Number of entries |
| 14* | ... | ... | ... | ... | ... |
| 15* | StreamerInfo for class: MRawRunHeader, version=10, checksum=0x5647c94b | | | | |
| 16* | UShort_t | fRunType | offset=0 | type=12 | Run Type |
| 17* | UInt_t | fRunNumber | offset=0 | type=13 | Run number |
| 18* | UInt_t | fFileNumber | offset=0 | type=13 | Run number |
| 19* | UInt_t | fProjectName[101] | offset=0 | type=13 | Project name |
| 20* | ... | ... | ... | ... | ... |
| 21* | StreamerInfo for class: TBranchElement, version=8, checksum=0xd503eff4 | | | | |
| 22* | StreamerInfo for class: TBranch, version=10, checksum=0xe3725ea2 | | | | |
| ... | ... | ... | ... | ... | ... |

**Tabelle 3.3:** Auszug der Klassenbeschreibungen eines TStreamerInfo-Datensatzes

ROOT stellt einige nützliche Containerklassen für Benutzerdaten bereit, die im folgenden kurz vorgestellt werden.

### TObjArray und TClonesArray

Die Klasse TObjArray unterstützt Arrays von Objekten. Die Objekte müssen nicht alle vom gleichen Typ, jedoch von TObject abgeleitet sein.

Die Klasse TClonesArray ist eine Spezialisierung von TObjArray. Sie enthält nur Objekte die gleichen Typs sind. Der Aufbau von TClonesArray ist im Anhang skizziert (A.10).

### TTree

TTree ist eine hoch spezialisierte Containerklasse, die es ermöglicht Objekte in einer Baumstruktur abzulegen. Sie kann alle Arten von Daten enthalten. Dies beinhaltet nicht nur simple Datentypen, sondern auch Objekte und Arrays. TTrees sind auf Grund der Komprimierbarkeit für die Speicherung von Objekten gleichen Typs optimiert. Für jeden TTree gibt es einen eigenen Datensatz in einer ROOT-Datei.

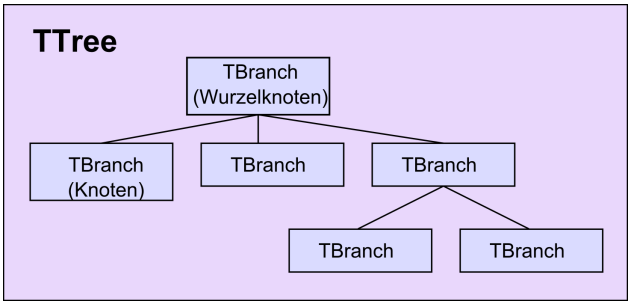



**Abbildung 3.2:** Aufbau eines TTrees

Die TTree-Datenstruktur ist in Abbildung 3.2 veranschaulicht. Ein TTree besteht aus einem oder mehreren Zweigen, den so genannten „TBranches“. Diese wiederum können Unterzweige enthalten. Dabei gibt es keine Beschränkung bezüglich der Tiefe der Verschachtelungen. Ein Zweig wird durch seine Blätter, den sogennate „TLeafs“ beschrieben. Jeder Zweig kann ein oder mehrere Blätter enthalten. Ein TLeaf enthält Informationen über Variablen, aber es enthält selbst keine Variablenwerte.

Die eigentlichen Daten sind in so genannten „TBaskets“ gespeichert. Ein TBasket ist die Kontainerklassen für Benutzerdaten in TTree-Datenstrukturen. Jeder Zweig besitzt ein Array von TBaskets, siehe Abbildung 3.3. Die Größe solcher TBaskets ist standardmäßig auf 32.000 Bytes festgelegt, kann aber von Benutzer verändert werden. Überschreitet die Größe der Daten diesen festgelegten Wert, so werden die Daten in externen TBaskets ausgelagert.

Für jeden zusätzlichen TBasket wird ein eigener TBasket-Datensatz in der ROOT-Datei des TTree-Datensatzes angelegt. ROOT bietet zusätzlich die Möglichkeit für jeden TBranch diese Daten in eine externe Datei zu schreiben. Zudem gibt es keinen Eintrag für TBasket in den TStremerInfos.

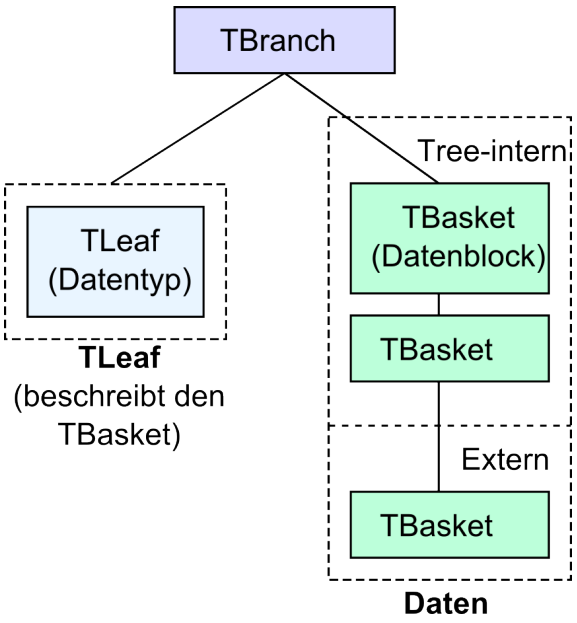



**Abbildung 3.3:** Aufbau eines TBranches

### 3.3.5 Datenkompression

Ein Teil der ROOT-Daten kann in komprimierter Form vorliegen. ROOT benutzt einen Algorithmus der auf dem „gzip“-Algorithmus basiert. Es gibt 10 verschiedene Kompressionsstufen. Von Stufe „0“, keine Kompression, bis Stufe „9“, maximale Kompression. Die in einer Datei verwendete Kompressionsstufe ist im Feld „fCompression“ im Datei-Header gespeichert, siehe Tabelle 3.1 .
Eine ROOT-Datei ist nie vollständig komprimiert. Jede Datei enthält Bereiche, die immer in unkomprimierter Form vorliegen. Dazu gehören folgende Bereiche:

- der Datei-Header
- der KeysList-Datensatz
- der FreeSegments-Datensatz
- jeder Datensatz, dessen Größe 256 Bytes oder weniger beträgt,
  Ausnahme: TTree-Datensätze
- alle Datensatz-Headers

## 3.4 Extensible Markup Language (XML)

XML ist die Spezifikation einer Meta-Sprache, mit der Auszeichnungssprachen für Dokumente erstellt werden können. XML selbst ist keine Sprache, die Inhalte darstellt, sondern es ist die Grundlage für die Definition einer solchen Sprache. XML dient dazu, den Aufbau solcher Dokumente zu definieren. Dabei können beliebige strukturierte Informationen mit einem einfachen Satz an Regeln dargestellt werden. Es ist möglich, logische Strukturen zu formulieren und ihre Aufteilung dabei zu beschränken [33] [37].

Extensible Markup Language beschreibt eine Klasse von Datenobjekten, so genannte XML-Dokumente. Durch das textbasierte und strukturierte Format von XML kann es leicht durch Programmiersprachen und -umgebungen unterstützt werden. Zudem wurde XML durch das World Wide Web Consortium (W3C) standardisiert. Das World Wide Web Consortium nennt seine „Standards“ Recommendations, dies sind Empfehlungen, die keinen rechtlich bindenen Charakter haben. Das W3C ist jedoch eine internationale anerkannte Institution, die viele „De-facto“-Standards hervorgebracht hat. Durch diese Aspekte ist XML eines der wichtigsten Werkzeuge in der Welt des Datenaustausches geworden [19]. Aus diesen Gründen ist es ebenfalls ein Grundbestandteil dieser Arbeit. Diesbezüglich soll in diesem Kapitel ein kurzer Überblick über die Auszeichnungssprache XML gegeben werden.

XML ist eine stark vereinfachte Form der wesentlich komplexeren Standard Generalized Markup Language (SGML), einem Standard zur Beschreibung von Dokumenten. XML besitzt einen reduzierten Merkmalssatz und ist deshalb einfacher einzusetzen.

Zu den Vorteilen von XML gehört neben der einfachen Nutzbarkeit und der großen Flexibilität auch die große Verbreitung. Dadurch haben sich mehrere nützliche Standards, wie DOM, SAX und SOAP etabliert (siehe Kapitel 3.4.4). Diese unterstützen XML-basierende Datenformate, wodurch die Kompatibilität und einen Datenaustausche einfacher zu realisieren sind.

### 3.4.1 Aufbau eines XML-Dokuments

XML-Dokumente besitzen einen physikalischen und einen logischen Aufbau [19] [42].

#### Physikalischer Aufbau

Der physikalische Aufbau eines XML-Dokuments besteht aus einer oder mehreren Speicherungseinheitenen, so genannten Entitäten. Die erste Entität eines Dokuments ist die Haupt-XML-Datei. Weitere Entitäten, wie Zeichenketten, ganze Dateien oder Referenzen auf Zeichenentitäten können über Entitätenreferenzen hinzugefügt werden.

Zum physikalischen Aufbau gehören Dokumenttypdefinition und XML-Deklaration. Sowohl die Dokumenttypdefinition, als auch die XML-Deklaration werden optional verwendet. Sie definieren unter anderem die Dokumentenarten, Sprachversionen und spezifizieren Regeln für den logischen Aufbau der XML-Datei.

#### Logischer Aufbau

Die generelle Struktur eines XML-Dokuments ist im Beispiel 3.1 dargestellt. Der logischer Aufbau eines XML-Dokumentes entspricht einer Baumstruktur.

Ein XML-Dokument kann aus Elementen, Attributen, Verarbeitungsanweisungen, Kommentaren oder so genannten CDATA-Abschnitten bestehen. Sie sind in den 2 Bereichen eines XML-Dokuments, dem Prolog-Bereich (Im Beispiel 3.1 grün dargestellt) und der Elementstruktur (blau dargestellt) zu finden.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="RootStylesheet.xslt"?>

<RootFile>
    <RootHeader>
        <RootFileIdentifier>root</RootFileIdentifier>
        <FVersion>51200</FVersion>
        <FBegin>100</FBegin>
        <FEnd>677869</FEnd>
        <!-- weitere Elemenete -->
    </RootHeader>
    <Root>
        <TKey id="1">
            <RecordHeader>
                <Nbytes>268</Nbytes>
                <Version>4</Version>
                <ObjLen>148</ObjLen>
```

```
                <Datime>900416449</Datime>
                <!-- weitere Elemenete -->
            </RecordHeader>
            <Data/>
        </TKey>
    </Root>
</RootFile>
```

**Listing 3.1:** Beispiel eines XML-Formats

Innerhalb der beiden Bereiche können Elemente („Tags“), Entitäten, als auch allgemeiner Inhalt stehen. Die einzelnen Teile eines XML-Dokuments werden kurz beschrieben.

Ein Element in XML besteht aus einem öffnenden und schließenden „Tag“ sowie dem Inhalt des Elements. Der Name eines Tags kann beliebig gewählt werden. Das erste Zeichen muss aber aus dem Unicode-Zeichensatz stammen oder ein Unterstrich oder Doppelpunkt sein. Leere Elemente können direkt wieder geschlossen werden, so genannte „Empty-Tags“.

```
<FVersion>51200</FVersion>
<FVersion/>
```

**Listing 3.2:** XML-Tags

Die Elemententags können durch Attribute erweitert werden. Sie dienen dazu Name-Wert-Paare mit Elementen zu verknüpfen. Ein Attribut kann nur in öffnenden Tags oder Empty-Tags vorkommen. Die Reihenfolge der Attribute ist beliebig, sie müssen aber Werte enthalten. Ein Wert wird mittels Apostroph(„“) oder einfachen Apostrophs(’ ’) geklammert.

```
<TKey id="1">...</TKey>
```

**Listing 3.3:** XML-Attribut

Verarbeitungsanweisungen sind spezielle Anweisungen für die auslesende Software. Sie werden durch „<?“ und „?>“ markiert und sind allein stehende Tags. Im Beispiel wird eine XSL-Datei eingebunden. Der XML-Parser bindet durch die „xml-stylesheet“-Anweisung die mittels einer „href“-Anweisung referenzierte Datei ein.

```
<?xml-stylesheet type="text/xsl" href="RootStylesheet.xslt"?>
```

**Listing 3.4:** XML-Verarbeitungsanweisung

Entitäten erfüllen die Aufgabe von Textbausteinen, sie dienen als Speicherungseinheiten. So ist es unter anderem möglich, Sonderzeichen in XML benutzen zu können. Als Beispiel wird das &-Zeichen durch eine Zeichenkette dargestellt. Diese Funktionalität lässt sich auch für Abküzungen benutzen oder als Referenz für ganze Dateien.

```
Das "Kaufmanns-Und"' & wird durch &amp; dargestellt
```

**Listing 3.5:** XML-Entität

XML kann auch Kommentare enthalten, diese dürfen mehrzeilig sein und werden durch das Tag <!– ... –> dargestellt

```
<!-- Kommentar -->
```

**Listing 3.6:** XML-Kommentar

XML-Code, der nicht vom XML-Parser interpretiert werden soll, muss in CDATA-Abschnitte verpackt werden. Sie werden über „<![CDATA[“ eingeleitet und mit „]]>“ geschlossen. Im Gegensatz zu Kommentaren werden diese Bereiche mit ausgegeben. Dies hat den Vorteil, dass Tag-Zeichen wie „<“,“ >“ und „&“ enthalten sein können. Diese werden innerhalb der CDATA-Abschnitte vom Parser nicht interpretiert. Aber die schließende Zeichenfolge „]]>“ darf nicht im Abschnitt vorkommen.

```
<![CDATA[<Element>einfache Zeichenfolge</Element>]]>
```

**Listing 3.7:** XML-CDATA-Abschnitt

Jedes XML-Dokument muss ein Wurzelelement enthalten, dieses ist auf der obersten Ebene enthalten. Die Elemente werden hierarchisch je nach Informationsstruktur verschachtelt und an das Wurzelelement angehängt.

Der Aufbau von XML-Dokumenten und die Wahl der Elemente sind bis auf wenige Regeln frei wählbar. Einschränkungen sind neben den oben genannten Regeln, wie das Vorhandensein eines Wurzelelements, auf der Website des World Wide Web Consortium (W3C) [19] beschrieben.

### 3.4.2 Dokumenttypen

Die Struktur von XML-Dokumenten kann in XML beschränkt werden. Dazu stehen verschiedene Dokumenttypen zur Verfügung. Die beiden bekanntesten sind Document Type Definition (DTD) und XML Schema Definition (XSD).

#### Document Type Definition

DTD ist eine Beschreibung für spezifische XML-Dokumente oder eine Gruppe von Dokumenten. Mithilfe einer DTD werden die erlaubten Elemente, die möglichen Attribute und die Anordnung der Elemente festgelegt. Die Anordnung besteht aus der Definition der Schachtelung, der Reihenfolge und der Anzahl der Elemente. Zudem können Textbausteine, so genannte „Entitäten“ definiert werden. Entitäten in DTD können einzelne Zeichen bis hin zu ganzen XML-Dokumenten sein. Sie werden durch den Aufruf des Entitätenname eingebunden.

Eine DTD wird im Prolog-Bereich der XML-Datei angegeben. Die Regeln der DTD können entweder in einer externen Datei festgehalten werden, oder innerhalb der XML-Datei. Externe DTD werden über die Angabe des Dateinamens vorgenommen, wie hier im Beispiel „externe.dtd“. Interne DTD werden in eckigen Klammern [ ] angegeben [2]. DTDs können auch über das Schlüsselwort PUBLIC auf eine „öffentliche“ DTD verweisen. Dafür

wird der Name der öffentlichen DTD und der Name des PUBLICs zur Identifizierung benötigt

```
Externe und interne DTD:
    <!DOCTYPE ElementName SYSTEM "externe.dtd" [ ... interne DTD ... ]>

Mit PUBLIC-Identifier:
    <!DOCTYPE ElementName PUBLIC "publicIdentifier" "externe.dtd" [ ... interne
        DTD ... ]>

Nur externe DTD
    <!DOCTYPE ElementName SYSTEM "externe.dtd">

Nur interne DTD:
    <!DOCTYPE ElementName [ ... interne DTD ...]>
```

**Listing 3.8:** Verschiedene DTD Deklarationen

Das nachfolgende Beispiel in der Abbildung 3.9 gibt die Definition der Elemente und Attribute für die oben skizzierte XML-Datei in der Abbildung 3.1. Das Element „RootFile“ enthält genau einmal die Elemente RootHeader und Root. Das Element „FVersion“ darf nur Text enthalten, gekennzeichnet durch den Eintrag „#PCDATA“. Neben #PCDATA gibt es noch verschiedene Werte die den Inhalt eines Elements definieren können. Über den Wert „EMPTY“ kann ein Elemente auch als immer leer definiert werden. Der Eintrag „ANY“ dagegen besagt, dass Elemente in beliebiger Reihenfolge und Schachtelung vorkommen können. Attribute werden durch den Eintrag „ATTLIST“ definiert. Danach werden der Name, der Typ und die Vorgaben des Attributes angegeben.

```
<!ELEMENT RootFile (RootHeader, Root)>
<!ELEMENT RootHeader (RootFileIdentifier, FVersion, FBegin, FEnd, ...)>
<!ELEMENT RootFileIdentifier (#PCDATA)>
<!ELEMENT FVersion (#PCDATA)>
...
<!ELEMENT Root (TKey+)>
<!ELEMENT TKey (RecordHeader, Data)>
<!ELEMENT RecordHeader (Nbytes, Version, ObjLen, Datime, ...)>
<!ELEMENT FVersion (#Nbytes)>
<!ELEMENT FVersion (#Version)>
...
<!ATTLIST id NMTOKEN '0' >
```

**Listing 3.9:** Beispiel einer DTD

Mithilfe von DTD kann somit zwar die Hierarchie angegeben werden und damit, ob Elemente wieder Elemente enthalten können oder nur Text, aber die genauen Typen von Inhalten kann nicht eingeschränkt werden. Zudem gibt es keine Möglichkeit eine DTD mit Hilfe von Kommentaren zu beschreiben. Ein großer Nachteil von DTDs ist die Eigenschaft, dass DTDs selbst keine XML-Dokumente sind, sondern in einer separaten Sprache verfasst

werden müssen.
Den genauen Aufbau und die Funktionalitäten von DTDs ist auf der Seite des W3C [19] beschrieben.

### XSD

Eine weitere Möglichkeit, XML-Dokumente zu beschreiben, ist das XML Schema. Häufig ist auch die Abkürzung XSD für „XML Schema Definition“ zu finden. XML Schema ist eine W3C Recommendation [25].

Ein XML-Schema kann die Elemente und Attribute definieren, welche in einem XML-Dokument vorkommen dürfen. Dabei kann auch die Reihenfolge, die Hierarchie und auch die Anzahl der nachfolgenden Knoten festgelegt werden. Neben der Struktur ist es mit Hilfe von XML-Schemas auch möglich, den Inhalt von Elemente zu beschränken. Elemente können als leer deklariert werden oder mit Inhalt versehen werden. Der Inhalt von Elementen und Attributen kann bestimmten Datentypen zugewiesen werden. Ausserdem können Standardwerte gesetzt werden.

Das nachfolgende Beispiel, in Abbildung 3.10, gibt einen Teil der Definition, der Elemente und Attribute für die oben skizzierte XML-Datei im Beispiel 3.1 wieder.

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <!-- allgemeine Definition RootFile -->
 <xs:element name="RootFile" type="RootFileTyp"/>

 <xs:complexType name="RootFileTyp">
  <xs:sequence>
      <xs:element maxOccurs="1" minOccurs="1" name="RootHeader" type="RootHeaderTyp
          "/>
      <xs:element minOccurs="1" maxOccurs="1" name="Root" type="RootTyp"/>
     </xs:sequence>
 </xs:complexType>

 <!-- Definition ROOT Header -->
 <xs:complexType name="RootHeaderTyp">
     <xs:sequence>
      <xs:element maxOccurs="1" minOccurs="1" name="RootFileIdentifier" type="
          xs:int"/>
      <xs:element maxOccurs="1" minOccurs="1" name="FVersion" type="xs:int"/>
      <xs:element maxOccurs="1" minOccurs="1" name="FBegin" type="xs:int"/>
      <xs:element maxOccurs="1" minOccurs="1" name="FEnd" type="xs:long"/>
      <!-- weitere Element -->
     </xs:sequence>
 </xs:complexType>

 <!-- Definition ROOT (Datenbereich) -->
 <xs:complexType name="RootTyp">
```

```
    <!-- ... -->
</xs:complexType>

<!-- ... -->

</xs:schema>
```

**Listing 3.10:** Beispiel eines Schemas

Das Wurzelelement „RootFile“ aus dem XML-Beispiel (Abbildung 3.1) wird im Schema als Element angelegt, das vom Typ „RootFileTyp“ ist. „RootFileTyp“ ist ein komplexer Typ, zu erkennen an der Deklaration *<xs:complexType name=“RootFileTyp“>*. Er besteht aus einer Folge von zwei Elemente, das erste ist „RootHeader“, das andere ist „Root“. Beide Elemente müssen genau einmal im XML-Dokument vorkommen, dies spezifiziert die Angabe *maxOccurs=“1“* und *minOccurs=“1“*. Weil die Elemente als *<xs:sequence>* definiert sind, müssen sie in dieser Reihenfolge vorkommen. Diese beiden Elemente sind erneut komplexe Typen. „Root“ spiegelt den Datenbereich einer ROOT-Datei im XML wider. Er ist hier nicht weiter definiert. Der „RootHeader“ besteht aus einer Folge von Elementen. Diese müssen nicht immer komplexe Datentypen sein, es gibt auch einfache Datentypen. Die Elemente können Angabe über atomare Typen enthalten, beim Element „RootFileIdentifier“ wird der Typ über die Angabe *type=“xs:string“* definiert. In unserem Beispiel wurde dem Element ein fester Wert durch den Eintrag *fixed=“root“* zugewiesen. Dies bedeutet, dass ein XML-Dokument welches durch dieses Schema beschrieben wird, das Element „RootFileIdentifier“ mit dem Inhalt „root“ besitzen muss.

XML Schemata vermeiden einige Nachteile von DTDs, siehe Kapitel 3.4.2. So werden XML-Schemata in XML beschrieben. Das hat den Vorteil, das einerseits keine neue Sprache benutzt und erlernt werden muss andererseits vorhandene XML Tools zum editieren, transformieren und parsen benutzt werden können. Zudem erlauben XML Schemata durch die Angabe von Datentypen eine spezifischere Beschränkung von Inhalten, wodurch komplexe Inhalte besser beschrieben werden können. Ein weiterer Vorteil ist die Erweiterbarkeit von XML Schemas. So lassen sich Datentypen von vordefinierten Datentypen herleiten. Ein XML-Dokument kann zudem mehrere Schemata referenzieren.

### 3.4.3 Wohlgeformtheit und Gültikeit

Ein XML-Dokument ist „wohlgeformt“ wenn es der Syntax der XML-Spezifikation des W3C entspricht [19]. Einige notwendige Voraussetzungen hierfür sind:

- Eine XML-Dekleration muss vorhanden sein.
- Genau ein Wurzelelement ist vorhanden.
- Elemente besitzen ein öffnendes und schließendes Tag.
- Geschachtelte Elemente dürfen sich nicht überlappen.

- Ein Element darf keine Attribute mit demselben Namen besitzt (eindeutige Attributennamen).
- Keine Kommentare oder Verarbeitungsanweisungen in Tags.
- Sonderzeichen sind als Entitäten codiert.

Ein XML-Dokument ist gültig, wenn es wohlgeformt ist und auf einer Grammatik (z.B. DTD oder XML-Schema) basiert und dessen Regeln entspricht.

### 3.4.4 XML Parser

Für die Verarbeitung von XML-Dokumente stehen XML-Parser zur Verfügung. Ein XML-Parser ist ein Programm, das ein XML-Dokument einliest, und die enthaltene Informationen (z.B. Elemente & Attribute) der darüber liegenden Schicht einer Anwendung zur Verfügung stellt [9], d.h parst. Dies beinhaltet sowohl den Inhalt, als auch die Struktur. Ein XML-Parser kann zudem das XML-Dokument auf Wohlgeformtheit überprüfen und die Gültigkeit mit Hilfe einer DTD oder eines Schemas validieren.

Für die Verarbeitung der XML-Dokumente gibt es verschiedene Modelle, die von den XML-Parsern unterstützt werden. Die bekanntesten sind Document Object Model (DOM) und Simple API for XML (SAX).

#### Document Object Model

DOM ist ein W3C-Standard [18]. Die DOM-API liest das komplette XML-Dokument ein und repräsentiert es als Baumstruktur. Dabei werden Elemente, Attribute und Inhalte als Knoten oder Blätter dargestellt. Durch das vollständige Einlesen ist ein wahlfreier Zugriff auf die einzelnen Elemente möglich, zudem können Dokumente verändert und abgespeichert werden. Dies beinhaltet den Nachteil des Speicherplatzverbrauchs.

#### Simple API for XML

SAX basiert auf einem anderen Verarbeitungsmodell. Es durchläuft ein XML-Dokument sequentiell und reagiert Ereignisgesteuert. Durch bestimmte Inhalte werden vorher registrierte Funktionen aufgerufen. Dadurch können zwar sehr große XML-Dokumente gelesen werden, aber diese können nicht verändert und gespeichert werden. Zudem ist SAX nicht standardisiert, aber weit verbreitet [16].

## 3.5 Extensible Stylesheet Language Transformations (XSLT)

Extensible Stylesheet Language Transformations (XSLT) dient dazu, XML von einem Format in ein anderes zu konvertieren. So können XML-Dokumente oder XML-Daten so

umgewandelt werden, dass sie in einem Browser, auf dem Handy oder zum Beispiel als PDF-Datei angezeigt werden können. Dazu müssen die Originaldaten in das entsprechende Format gewandelt werden. Eine bekanntes Beispiel dieser Formate ist die Extensible HyperText Markup Language (XHTML). XHTML ist eine Neuformulierung von HTML in XML. Dies ist durch das XML-Konzept der vollständigen Trennung von Form und Inhalt in XML möglich. XSLT ist eine Recommandation des W3C [26] [34].

### 3.5.1 Funktionsweise

Bei der Verarbeitung von XML-Quellen durch einen XSLT-Prozess wird aus den eigentlichen XML-Daten, dem Quelldokument, ein Dokumentenbaum erzeugt und mit Hilfe von vorgegebenen Stylesheet-Dokumenten in einen neuen Zielbaum, das Zieldokument, umgewandelt (siehe Abbildung 3.4). XSLT-Stylesheets werden auch als XSLT-Programme bezeichnet. XSLT-Stylesheets sind selber XML-Dokument und besitzen dessen Format.

XSLT-Prozessoren lesen die XSLT-Stylesheets und wandeln mit dessen Anweisungen die XML-Dokumente in das gewünschte Format um.[34]. Die Transformation besteht aus einer Reihe von Transformationsregeln, den so genannten „Templates“. Ein Template gibt eine Vorlage für den XSLT-Prozess, welche Knoten und wessen Inhalt umgewandelt werden soll. Zudem wird in der Vorlage auch auch die neue Form, in die der Inhalt gebracht werden soll, definiert. Die Knoten werden mit Hilfe der XPath-Sprache adressiert. XML Path Language (XPath) ist eine Entwicklung des W3C-Konsortiums und dient der Adressierung von Teilen eines XML-Dokuments [24].

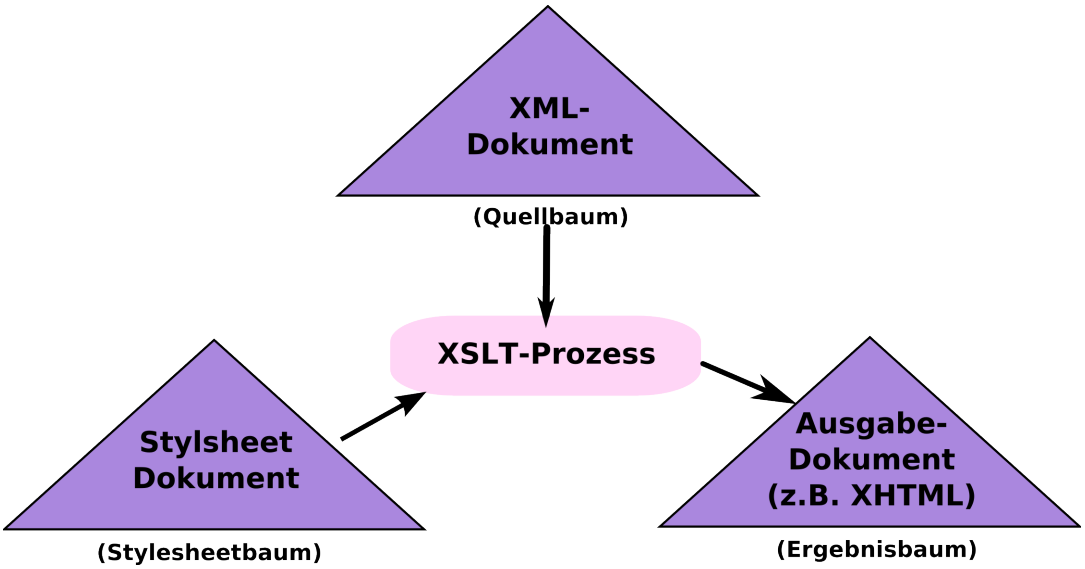



**Abbildung 3.4:** Funktionsweise von XSLT

### 3.5.2 Beispiel

Anhand eines einfachen Beispiels soll die in Kapitel 3.5.1 beschriebene Funktionsweise von XSLT kurz skizziert werden. Dazu werden die drei Dokumentenarten, Quelldokument, Stylesheet-Dokument und Zieldokument dargestellt. Spezifische Informationen über die Möglichkeiten von XSLT und der Definition der Templateregeln könnnen auf der Webseite des W3C [26] nachgelesen werden.

#### Quelldokument

Das Quelldokument ist in Abbildung 3.11 dargestellt. Es ist ein einfaches XML-Dokument, das eine Liste mit Büchern beschreibt. Zu jedem Buch ist der Titel und der Autor, mit Vor- und Nachname gespeichert.

```
<!-- Quell-Dokument -->
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="Buch.xslt"?>

<Buchliste>
    <Buch>
        <Titel>XSLT leicht gemacht</Titel>
        <Autor>
            <Vorname>Michael</Vorname>
            <Nachname>Mustermann</Nachname>
        </Autor>
    </Buch>
    <Buch>
        <Titel>XML für Anfänger</Titel>
        <Autor>
            <Vorname>Ingrid</Vorname>
            <Nachname>Isemann</Nachname>
        </Autor>
    </Buch>
</Buchliste>
```

**Listing 3.11:** Beispiel eines XML-Quelldokuments

Auf die Strukturen und den Inhalt des Quelldokuments beziehen sich die Transformationsregeln. Der Eintrag *<?xml-stylesheet type="text/xsl" href="Buch.xslt"?>* ist eine Verarbeitungsanweisung. Durch sie wird die Verbindung zwischen dem XML-Dokument und dem Stylesheet, welches die Transformationsregeln enthält, hergestellt. Im Beispiel wird das Stylesheet „Buch“ verwendet, welches in Abbildung 3.12 dargestellt wird.

#### Stylesheet

Das Stylesheet-Dokument in Abbildung 3.12 ist ein XML-Dokument. Im Prolog-Bereich ist sowohl die verwendete XML-Version, als auch die Verwendete XSLT-Version definiert.

Zudem wurde das Ausgabeformat durch den Aufruf *<xsl:output method="html"/>* auf „Hypertext Markup Language (HTML)“ festgelegt.

Die Elementenstruktur enthält die Anweisungen für den Transformationsprozess. Das Beispiel besitzt ein Template, das durch den Aufruf *<xsl:template match="/">* erstellt wird. Das „match“-Attribut ordnet einem Template einen Knoten zu, für den die Transformationsregel gilt. *<xsl:template match="/">* wählt das tiefste Tag im Quelldokument, in diesem Beispiel das Tag „Buchliste“, und wendet darauf die nachfolgenden Transformationen an.

Als erstes werden die angegebenen Elemente und Inhalte der Zieldatei, *<html>* bis *<ul>* erzeugt. Danach werden die Transformationsregeln durchgeführt und anschließend die Tags *</ul></body></html>* erzeugt.

Die Transformationsregel *<xsl:for-each select=".//Buch">* verarbeitet alle Elemente des Typs „Buch“ im Quelldokument und erzeugt einen Listeneintrag „<li></li>“ für jedes Element von Typ „Buch“. Der Listeneintrag enthält als erstes den Titel des Buches. Er wird durch die Anweisung *<xsl:value-of select="Titel" />* ausgegeben. Zudem wird der Vorname und der Nachname des Autors in „( )“ gesetzt. Dies geschieht mit Hilfe der Anweisungen *<xsl:value-of select="Autor/Vorname" />* und *<xsl:value-of select="Autor/Nachname" />*. Danach ist der Transormationsprozess abgeschlossen.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

    <xsl:template match="/">
    <html>
        <head>
            <title>
                Buchliste
            </title>
        </head>
        <body bgcolor="#F0F8FF">
        <h1>Liste</h1>
        <ul>
        <xsl:for-each select=".//Buch">
            <li>
                <xsl:value-of select="Titel" />
                (<xsl:value-of select="Autor/Vorname" /> <xsl:value-of select="
                    Autor/Nachname" />)
            </li>
         </xsl:for-each>
        </ul>
        </body>
    </html>
    </xsl:template>

</xsl:stylesheet>
```

**Listing 3.12:** Beispiel eines XSLT-Stylesheets

### Zieldokument

Das erzeugte Zieldokument ist die in Abbildung 3.13 dargestellt XHTML-Seite. In ihr spiegeln sich die Elemente aus dem Quelldokument wieder. Die Ansicht im Browser ist in Abbildung 3.5 dargestellt.

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
      <title>Buchliste</title>
   </head>
   <body bgcolor="#F0F8FF">
      <h1>Liste</h1>
      <ul>
         <li>XSLT leicht gemacht (MichaelMustermann)</li>
         <li>XML f&uuml;r Anf&auml;nger (IngridIsemann)</li>
      </ul>
   </body>
</html>
```

**Listing 3.13:** Beispiel des XSLT-Zieldokuments


**Abbildung 3.5:** Browseransicht des erzeugten Zieldokuments

# Kapitel 4

# Systemdesign

Dieses Kapitel gibt einen Überblick über die Systemanforderungen und das Design des im Rahmen dieser Diplomarbeit entwickelten Systems. Das System lässt sich in drei Konzepte unterteilen, Datenauslese, Datenumwandlung und Datenübertragung. Die einzelnen Konzepte werden im Folgenden vorgestellt.

## 4.1 Systemanforderungen

In dieser Arbeit soll ein Konverter für ROOT-Daten erstellt werden, der unabhängig von der Datenanalysesoftware ROOT das Umwandeln der Daten ermöglicht.

Der Auslesevorgang sowie der Umwandlungsprozess werden durch Dateien gesteuert, welche das ROOT-Datenformat beschreiben. Das Format dieser Steuerdateien soll auf der Basis von XML realisiert werden. Dies ermöglicht den Aufbau eines flexiblen Systems, welches durch Ergänzungen der Steuerdateien erweitert werden kann.

Die Ausgabe des Umwandlungsprozesses soll ebenfalls in XML formatiert sein. Das erzeugte XML-Format muss so definiert sein, dass es der Struktur des RapidMiner Datenformats entspricht.

Die umgewandelten Daten sollen mittels einer Web-Service der DataMining-Software RapidMiner zur Verfügung gestellt werden. Dies bedingt die Erstellung eines entsprechenden RapidMiner-Plugins.

## 4.2 Grundkonzept

Das System besteht aus drei Konzepten, welche benötigt werden, um Daten einer ROOT-Datei in RapidMiner einlesen zu können. In Abbildung 4.1 (Grundkonzept der Datenverarbeitung) sind die verschiedenen Bereiche zu erkennen. Das Konzept der Datenauslese wird durch den „XML-Konverter“ repräsentiert. Das Konzept der Datenumwandlung wird im „XSLT-Prozess“ vorgenommen. Das letzte Konzept ist das der Datenübertragung, die von

einer Steuereinheit vorgenommen wird, dem „ROOT-Navigator“. Die einzelnen Konzepte werden im folgenden kurz erläutert.

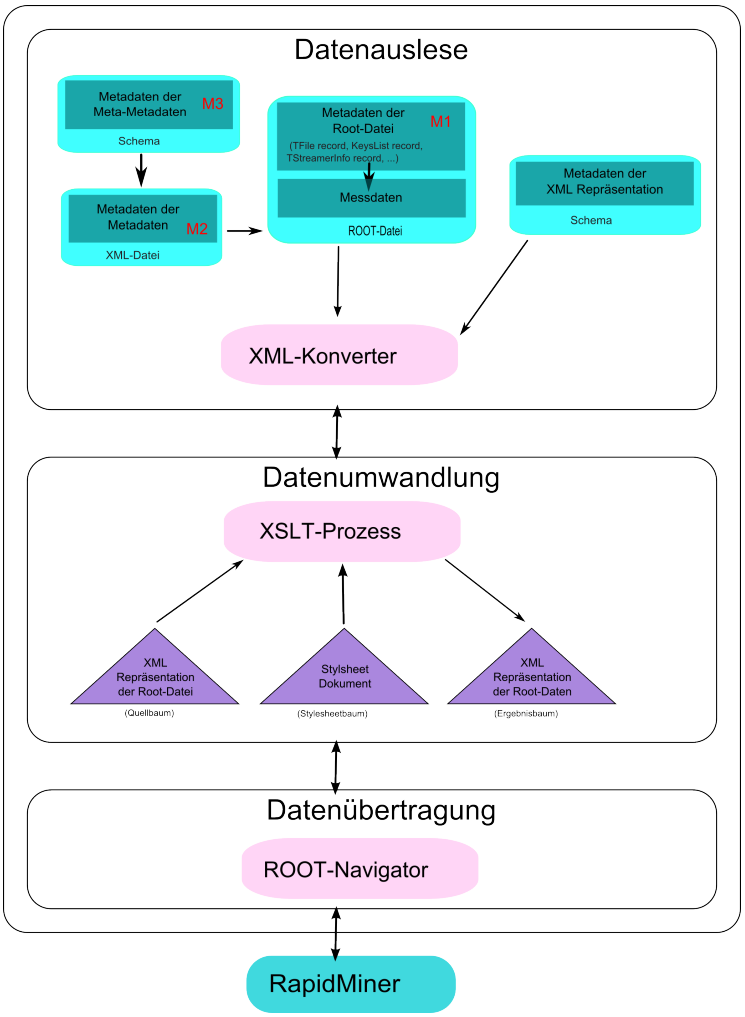



**Abbildung 4.1:** Grundkonzept der Datenverarbeitung

## 4.3 Konzept der Datenauslese

Der Konvertierungsprozess hängt stark vom Aufbau der ROOT-Dateien ab. Eine ROOT-Datei enthält einerseits Messdaten, andererseits Metadaten. Diese ROOT-Metadaten beschreiben den Aufbau der Messdaten innerhalb der ROOT-Datei, siehe Abbildung 4.2 „M1“.

Eine ROOT-Datei beinhaltet verschiedene Bereiche die Metadaten enthalten. Dies sind der Header der Datei, die Header der einzelnen Datensätze, der „TFile“-Datensatz, der „KeysList“-Datensatz und der „FreeSegments“-Datensatz. Der bedeutendste Datensatz ist der „TStreamerInfo“-Datensatz, er enthält Informationen über den Aufbau der verwendeten Objekte der ROOT-Datei. Die TStreamer-Informationen sind notwendige Informationen

für die Verarbeitung von benutzerdefinierten Objekten. Eine Beschreibung über den Aufbau der ROOT-Dateien, ihrer Header und ihrer Datensätze befindet sich im Kapitel 3.3.

Ein Bereich dieser Arbeit ist das Erstellen einer Beschreibung des Aufbaus dieser ROOT-Metadaten. Dies geschieht mittels einer XML-Datei, den so genannten Metadaten der ROOT-Metadaten und Messdaten, siehe Abbildung 4.2 „M2“. Hierfür wird ein XML-Schema erstellt, welches diese XML-Datei beschreibt, „M3“ (die Metadaten der Meta-Metadaten). Die XML-Datei bildet genau den Aufbau der einzelnen Bereiche der ROOT-Datei ab und ermöglicht dadurch das Auslesen der Datei. Dabei muss beachtet werden, dass im Laufe der Entwicklung der Analysesoftware ROOT verschiedene Versionen von ROOT-Objekten entstanden sind. Diese können durch die Metadaten der Metadaten abgebildet und entsprechend ihrer Version angesprochen werden. Die detaillierte Umsetzung des Auslesemechanismus ist im Kapitel 5 zu finden.

Mit diesen Informationen ist es möglich, Daten aus einer ROOT-Datei zu lesen. Um die Daten, die als Binärdaten vorliegen, passend zu konvertieren, wird ein weiteres Schema spezifiziert, welches das Format aller Binärdaten der ROOT-Datei enthält. Dies sind die so genannten Metadaten der ROOT-XML-Repräsentation. Dort sind die Angaben der Datentypen jedes Elements der zu erstellenden XML-Repräsentation erfasst.

Der „XML-Konverter“ benutzt diese Informationen, um die ROOT-Metadaten zu lesen, umzuwandeln und zu nutzen. In einem zweiten Schritt werden diese ROOT-Metadaten in Verbindung mit weiteren Informationen aus der XML-Datei (Metadaten der ROOT-Daten) genutzt, um die Messdaten ebenfalls aus der ROOT-Datei zu lesen, umzuwandeln und abzuspeichern. Die Messdaten werden in einen XML-Baum überführt. Die Repräsentation in XML sichert dabei einen Standard für den Austausch der ROOT-Daten.

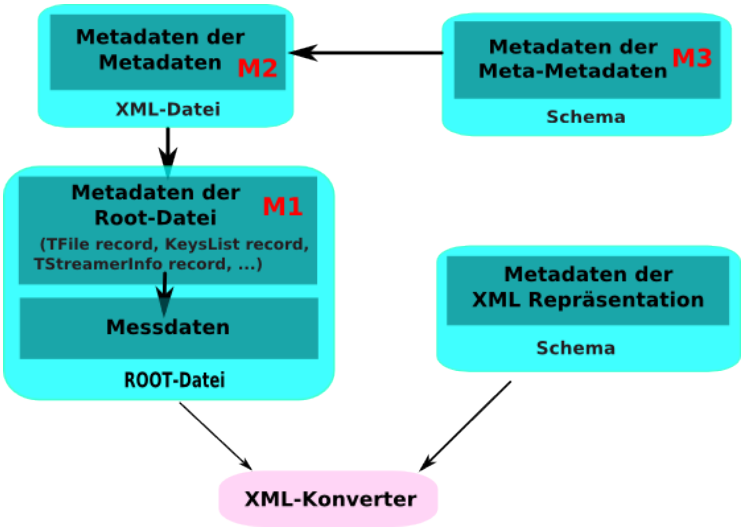



**Abbildung 4.2:** Konzept der Datenauslese

## 4.4 Konzept der Datenumwandlung

RapidMiner benötigt ein bestimmtes Format der übergebenen Daten. So gehören beispielsweise zu einem Ereignis in der Astrophysik verschiedene Messdaten. ROOT speichert alle Messwerte, wie zum Beispiel eine Höhe, in Blöcken innerhalb einer Baumstruktur ab (siehe Kapitel 3.3.4). Innerhalb dieser Blöcke erscheinen die Messwerte sequentiell in der Reihenfolge der Ereignisse. RapidMiner benötigt nach der Umwandlung der ROOT-Daten eine Zuordnung der einzelnen Werte zu jedem Ereignis.

Die Struktur der ROOT-Daten muss so umgewandelt werden, das sie der Struktur der RapidMiner-Daten entspricht. Die Umwandlung der Daten wird durch den XSLT-Prozess vorgenommen. Die Funktionsweise von XSLT ist in Kapitel 3.5 beschrieben.

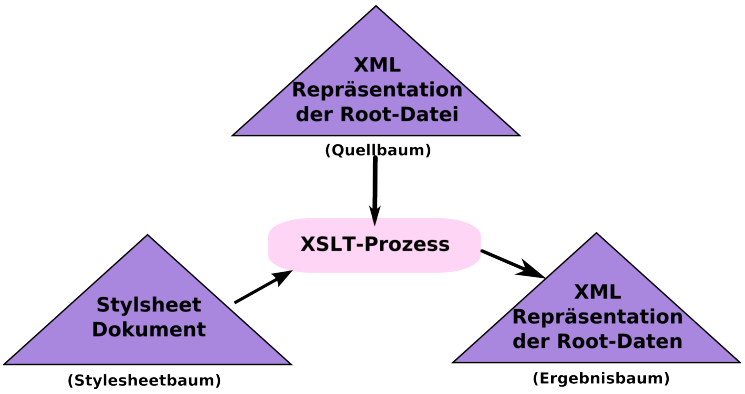



**Abbildung 4.3:** Konzept der Datenumwandlung

Die XML-Repräsentation der ROOT-Datei liegt nach dem XML-Konverter als Baumstruktur vor. Dieser Baum ist der Quellbaum für den XSLT-Prozess, der die Daten in die gewünschte Struktur umwandelt. Der Prozess ist in Abbildung 4.3 dargestellt.

Dazu wird in dieser Arbeit ein Stylesheet-Dokument erstellt, welches die RapidMiner Datenstruktur abbildet. Das Stylesheet-Dokument dient als Vorlage für den Umwandlungsprozess. Zudem soll durch das Stylesheet-Dokument eine Auswahl bestimmter Daten aus der jeweiligen ROOT-Datei ermöglicht werden. Der Benutzer soll die Möglichkeit erhalten, eine Auswahl bestimmter Daten innerhalb der zu lesenden Datei zu treffen. Der erzeugte Ergebnisbaum enthält die gewünschten ROOT-Daten in einer von RapidMiner lesbaren XML-Repräsentation. Die Semantik der Daten bleibt dabei erhalten.

## 4.5 Konzept der Datenübertragung

Als Konzept für die Datenübertragung wurde ein Client/Server Modell gewählt. Hierbei enthält der Server die ROOT-Dateien als Datenquelle und den Konverter als Schnittstelle. RapidMiner übernimmt die Rolle der Clientanwendung. Die Serveranwendung stellt bestimmte Funktionalitäten für RapidMiner bereit, um die passenden Daten aus einer ROOT-

Datei auszulesen und zu übertragen. Die Kommunikation muss dabei folgende Funktionen unterstützen: eine Verzeichnisanfrage aller verfügbaren ROOT-Dateien mit einem Dateiverzeichnis als Antwort, eine Anfrage aller verfügbaren Daten für eine ausgewählte Datei mit einer Liste aller verfügbaren Datenblöcken als Antwort. Zudem eine Anfrage aller Datengruppen innerhalb eines ausgewählten Datenblockes und eine Anfrage spezifizierter Datengruppen und Einschränkungen für den ausgewählten Datenblock mit den entsprechenden Daten als Antwort.

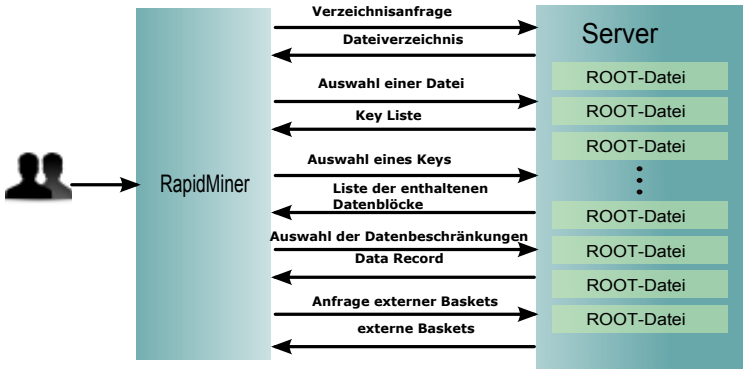



**Abbildung 4.4:** Konzept der Datenübertragung

Die berechneten Daten können noch Sprungadressen zu externen Datensätzen innerhalb der ROOT-Datei enthalten. Diese Addressen beziehen sich auf Baskets innerhalb des zuvor berechneten Datensatzes und weisen auf externe Baskets. Diese enthalten Messdaten vom selben Typ wie der Basket innerhalb des Datensatzes. Wie schon in Kapitel 3.3.4 beschrieben lagert ROOT Daten in separate Baskets, wenn eine gewisse Größe überschritten ist. Für jede enthalte Sprungadresse wird eine Anfrage an den Server geschickt. Als Antwort wird der Datensatz gesendet und die enthaltenen Daten werden ausgelesen.

# Kapitel 5

# Realisierung

Dieses Kapitel der Diplomarbeit beschäftigt sich mit der Umsetzung des zuvor dargestellten Systemdesigns und dessen Anforderungen. Zuerst werden die wichtigsten der für die Diplomarbeit verwendeten Werkzeuge beschrieben. Danach wird die Systemarchitektur beschrieben und auf die einzelnen Bereiche der Client-Server-Architektur eingegangen. Die Serverarchitektur beinhaltet die Beschreibung des Web-Servers „Tomcat“ und des Ablaufs der Kommunikation zwischen Client und Server.

Der Serverdienst, der den XML-Konverter enthält, wird in einem eigenen Bereich erläutert. Dabei wird das Prinzip des Konverters und dessen Umsetzung eingegangen. Des weiteren wird der Aufbau der „Metadaten der Metadaten“ erläutert. Diese Datei enthält eine genaue Definition des Aufbaus von ROOT-Dateien. Zudem wird die Umsetzung der Clientarchitektur beschrieben.

## 5.1 Verwendete Werkzeuge

In diesem Bereich der Arbeit werden die für die Umsetzung des Systemdesigns benötigten Werkzeuge kurz beschrieben. Diese umfassen neben der Programmiersprache „Java“, die Entwicklungsumgebung „Eclipse“ und die verwendeten Standards.

### 5.1.1 Programmiersprache

Die für die Umsetzung der Aufgabe benutzte Programmiersprache ist Java. Java ist eine objektorientierte Programmiersprache und ist ein Warenzeichen der Firma Sun Microsystems. Die Java-Technologie ist in verschiedenen Versionen für verschiedene Anwendungsbereiche kostenfrei erhältlich.

Für die Implementierung des XML-Adapters wurde die Java Version „J2SE 1.6.0 17“ eingesetzt. Die Java 2 Standard Edition besitzt umfangreiche Bibliotheken für die I/O-Programmierung, die Netzwerkprogrammierung, die Verarbeitung von XML-Dokumenten

und die Erstellung von GUIs [38]. Für die Implementierung des XML-Adapters wurden die J2SE Bibliotheken benutzt.

Neben den umfangreichen Bibliotheken besitzt Java den Vorteil plattformunabhängig zu sein. Dadurch können Java-Programme auf allen Computern und Betriebsystemen ausgeführt werden, die eine Java Virtuelle Maschine (JVM) besitzen. Eine Virtuelle Maschine ist eine Umgebung, in der Java-Programme in so genannte Bytecode übersetzt und ausgeführt werden. Dazu wird der Bytecode in der Virtuellen Maschine interpretiert.

Das System RapidMiner verwendet die Programmiersprache Java. Zudem eignet sich Java sehr gut für den Umgang mit XML-Daten. Die vielen zur Verfügung stehenden Bibliotheken bilden eine gute Grundlage für die Umsetzung des XML-Konverters. Des weiteren stellt Java weitere Bibliotheken für die Kommunikation der Web-Schnittstelle zwischen Client und Server bereit. Aus diesen verschiedenen Aspekten wurde die Programmiersprache Java in dieser Arbeit verwendet [40].

Zudem stellt Java eine Logging Bibliothek „Log4j“ bereit, die zentral konfiguriertes Logging erlaubt. Dies beinhaltet die Ausgabe von Anwendungsmeldungen, Laufzeit- und Fehlerinformationen. Durch die Logginghierarchie können verschiedene Ausgabemodi, wie etwa *warn*, *info* und *debug* verwendet werden. Log4j wurde im Projekt für Informationsausgabe sowohl zur Fehlersuche eingesetzt.

Desweiteren wurde „JConsole“, eine Java Anwendung zur Überwachung von Java Prozessen auf lokalen oder entfernten Systemen mit Hilfe der Java Management Extension (JMX), verwendet. JMX ist eine Erweiterung des Java Standards, es ermöglicht die Kommunikation zwischen JVMs.

### 5.1.2 Entwicklungsumgebung

Als Entwicklungsumgebung wurde die Software Eclipse in der Version „Eclipse Java EE IDE for Web Developers“ eingesetzt. Eclipse ist ein Open-Source-Framework, das von IBM entwickelt wurde. Eclipse bietet neben dem eigentlichen Quellcode-Editor verschiedene Werkzeuge zur Projektverwaltung und zur Fehlerbehandlung. Zahlreiche nützliche Funktionen werden für eine einfachere Handhabung bereit gestellt. Beispiele dafür sind die automatische Code-Vervollständigung oder automatische Formatierung des Codes.

Eclipse war bis zur Version 3.0 eine vollständig integrierte Entwicklungsumgebung (IDE). Seit Version 3.0 besteht Eclipse selbst aus einem Kernmodul, welches durch Plugins für Programmiersprachen und anderen Funktionalitäten erweitert werden kann.

Die benutzte Version bietet eine „Eclipse Web Tools Platform“. Die Plattform bietet eine einfache Möglichkeit, Web-Applikationen zu entwickeln. Ausserdem stellt es Funktionen für den Umgang mit XML-Daten bereit. Dadurch ist diese Version von Eclipse besonders für die Umsetzung dieser Diplomarbeit geeignet. Für die Erstellung von UML-Diagrammen wurde die Erweiterung „AgileJ StructureViews“ eingesetzt.

## 5.2 Systemarchitektur

Die Aufbau des Systems basiert auf einer Client-Server-Architektur. Die Client-Server-Architektur ermöglicht die Realisierung von verteilten Anwendungen. Dabei wird vom Server ein Dienst bereit gestellt, der auf Anfragen eines Clients reagiert und entsprechende Funktionalitäten bereit stellt. Die Kommunikation zwischen Server und Client wird durch Protokolle gesteuert.

Der Server ist in Bereitschaft, um jederzeit auf Anfragen von Clients reagieren zu können. Dabei können der Server und die Clients auf der gleichen physikalischen Maschine ausgeführt werden oder auf Maschinen, die durch ein Netzwerk verbunden sind.

Eine Client-Server-Architektur kann aus verschiedenen Gründen eingesetzt werden. Ein Vorteil durch die Bereitstellung eines Dienstes auf einem Server besteht in der Verteilung der benötigten Performance. So ist es möglich, leistungsschwache Clients zu entlasten, indem ein leistungsstarker Server benötigte Berechnungen und Anwendungen ausführt und Ergebnisse oder Zwischenschritte der Berechnungen für den Client bereit stellt.
Ein weiterer Grund für eine solche Architektur besteht in der zentralen Datenhaltung. Auf dem Server sind zentral die Daten gespeichert, die von verschiedenen Clients benutzt werden oder auch verändert werden können.

Für diese Arbeit wurde eine Client-Server-Architektur gewählt, um die ROOT-Dateien zentral verwalten zu können. Die Umwandlung der Daten wird durch den Server ausgeführt. Die Client-Server-Architektur ist in Abbildung 5.1 dargestellt. Bei der Client-Server-Architektur dieser Dipolmarbeit stellt das RapidMiner-Plugin die Client-Anwendung bereit. RapidMiner stellt Anfragen an den Server, um Informationen über verschiedene ROOT-Dateien zu erhalten. Der Server beinhaltet die ROOT-Daten und den Auslesemechanismus des XML-Adapters. Durch den Dienst stehen verschiedene Funktionen je nach Anfrage des Clients bereit, um Informationen aus den ROOT-Dateien zu lesen und an den Client zu senden.

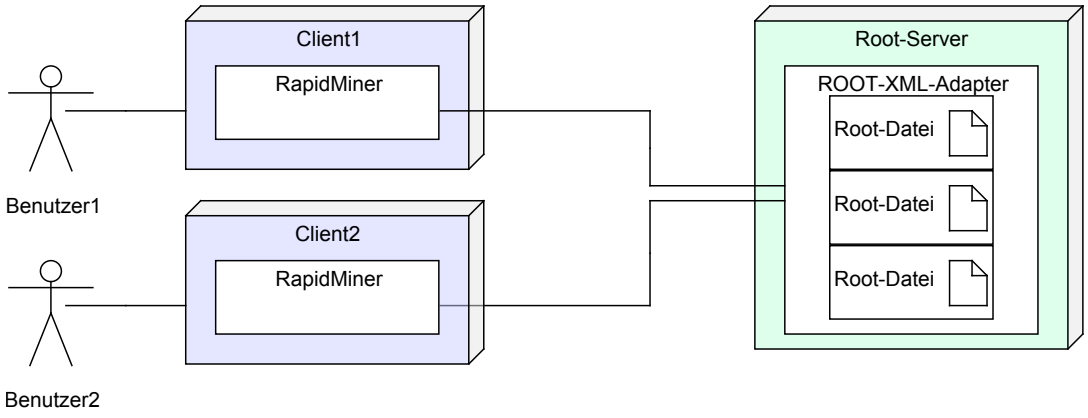



**Abbildung 5.1:** Client-Server-Architektur

## 5.3 Serverarchitektur

Nachdem in den vorigen Abschnitten die zugrunde liegende Architektur beschrieben wurde, wird hier auf die genauere Umsetzung der Serverarchitektur eingegangen. Dabei wird zuerst der verwendete Server und die Struktur der verwendeten Kommunikation erläutert.

### 5.3.1 Tomcat

Als Serverinstanz wurde der in Eclipse integrierte Server „Tomcat-6.0“ verwendet. Tomcat ist ein Web-Server, der eine Umgebung zur Ausführung von Java-Code bereit stellt. Er ist ein Open-Source-Projekt der Apache Software Foundation und stellt einen in Java geschriebenen Servlet-Container bereit. Dieser stellt eine Referenzimplementierung der Java Servlet- und Java Server Pages (JSP) -Spezifikation dar. Tomcat ist ein Server, der Funktionalitäten für Netzwerkdienste zum Empfangen und Versenden von Daten bereitstellt und die Servlets in ihrem gesamten Lebenszyklus verwaltet und erhält, wobei ein Servlet jeweils in einem eigenen Thread bearbeitet wird.

Ein Servlet ist eine Java Klasse, die HTML-Code erzeugt und die von einem speziellen Web-Server geladen und gestartet wird. Durch Servlets bietet Java eine umfangreiche API für die Erstellung von Web-Applikationen mit dynamischen Inhalten. Mithilfe der Servlet API kann auf die HTTP-Protokollebene und deren Eigenschaften zugegriffen werden. So ist es möglich, Clientanfragen an den Web-Server so abzuarbeiten, dass die Antwort als dynamische HTML-Seite durch entsprechende Übertragungsprotokolle an den Client gesendet wird.

### 5.3.2 Kommunikation

Die Interaktion eines Servlet mit den Client basiert auf dem HTTP-Request-Response-Modell [39]. Dafür muss der Servlet-Container das HTTP-Protokoll unterstützen. Ein Benutzer stellt durch seinen Browser einen Request an den Server. Der Server reicht den Request an den Servlet-Container weiter, der wiederum das zuständige Servlet anspricht. Dieses bearbeitet dann die Anfrage und gibt die Antwort als Response an den Servlet-Container zurück. Der Response wird auch über den Web-Server an den Client gesendet. Dieser Kommunikationsweg ist in Abbildung 5.2 dargestellt.

Die Kommunikation zwischen RapidMiner und dem ROOT-Server wird mittels des HTTP-Protokolls durchgeführt. Das HTTP-Protokoll definiert einen so genannten Header und einen Body. Der Header enthält Informationen, die für die Übertragung benötigt werden. Der Body enthält die eigentlichen Daten. HTTP stellt für die Kommunikation verschiedene Methoden bereit. Die beiden im Projekt verwendeten Methoden sind „GET“ und „POST“. Die Methode GET ist hauptsächlich für das Anfordern von Informationen zuständig, wohingegen die Methode POST für das Übersenden von Informationen vorge-

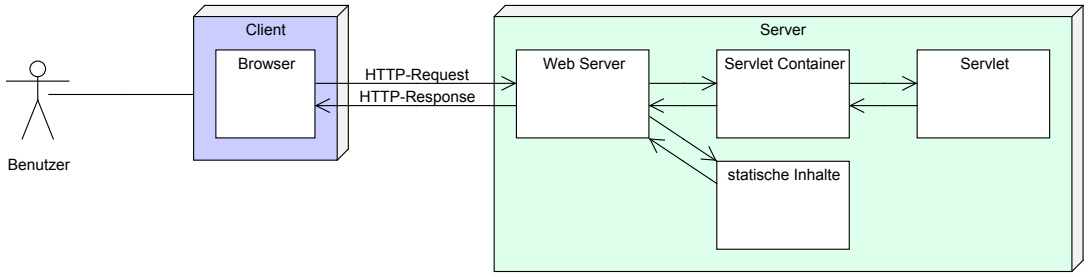



**Abbildung 5.2:** HTTP-Servlet: Kommunikationsweg zwischen Browser und Servlet

sehen ist. Die POST-Methode kann Daten, wie etwa Bilder, versenden. Diese Übertragung ist für den Client unsichtbar.

Für den ROOT-XML-Adapter wurde ein Servlet, das „RootServlet“ implementiert. Es stellt die Funktionalität bereit die für das Konzept der Datenübertragung (Kapitel 4.5) benötigt wird. Die Klasse *RootServlet* erbt von der Klasse *HttpServlet*. HttpServlet stellt verschiedene Service-Methoden zur Verfügung:

- doGet(HttpServletRequest req, HttpServletResponse resp)
- doPost(HttpServletRequest req, HttpServletResponse resp)

Diese werden von der Superklasse überschrieben. Damit können die Clientanfragen von RapidMiner verarbeitet werden. Der Zusammenhang ist in Abbildung 5.3 skizziert.

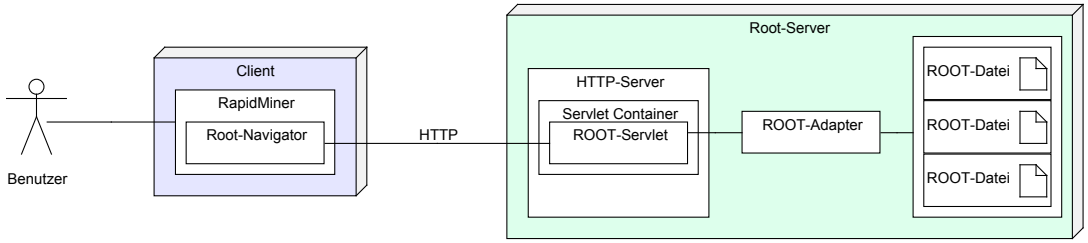



**Abbildung 5.3:** Kommunikationswege zwischen RapidMiner-Client, ROOT-Servlet und Konverter

Anfragen, die vom Client gesendet werden, sind:

- Verzeichnisanfrage
  Die Verzeichnisanfrage wird durch die doGet-Methode bearbeitet. Sie liefert eine Liste von Dateinamen eines Verzeichnisses.

- Anfrage einer Liste vorhandener Datensätze
  Die doPost-Methode liefert als Antwort eine XML-Datei des KeysList-Datensatzes (siehe Kapitel 3.3.3) des als Parameter übergebenden Dateinamens.

- Anfrage einer Liste enthaltender Datenblöcke
  Die doPost-Methode liefert eine Liste von enthaltenden Datenblöcken, so genannten TBaskets innerhalb von TTrees.

- Anfrage der Daten
  Die Daten werden als XML-Datei mittels einer doPost-Methode zurückgegeben. Bei der Anfrage liefert der Client eine Komma-Separierte Liste aller unerwünschten Datenblöcke. Diese werden aus der XML-Datei vor der Übermittlung entfernt.

- Anfrage ausgelagerter Daten
  Die ausgelagerten Daten werden als XML-Dateien mittels einer doPost-Methode zurück gegeben. Als Übergabeparameter liefert der Client die Positionsangabe der ausgelagerten Daten.

## 5.4 Server-Dienst

Der vom Server bereitgestellte Dienst ist der XML-Konverter. Der Dienst kann ROOT-Dateien einlesen und wandelt die enthaltende Daten in das XML-Format um. In diesem Bereich der Arbeit wird der XML-Konverter erläutert. Dazu wird zuerst der Auslesemechanismus und dessen Umsetzung dargestellt. Anschließend wird die beschreibende XML-Datei, die so genannten „Metadaten der ROOT-Metadaten“ erläutert. Der Auslesemechanismus und die Funktion der beschreibenden XML-Datei wird zuletzt an einem Beispiel erläutert.

### 5.4.1 Auslesemechanismus

Der Auslesemechanismus des ROOT-XML-Adapters wird als XML-Konverter bezeichnet. Dieser beruht auf dem Konzept der Datenauslese (Kapitel 4.3). Dabei wird die ROOT-Datei konzeptbedingt byteweise ausgelesen.

In Abbildung 5.4 ist das Grundschema des XML-Konverters skizziert. Eine XML-Datei, die so genannten „Metadaten der Metadaten“ (siehe Abbildung 4.2), beschreibt den Aufbau der ROOT-Datei auf binärer Ebene.

Der Auslesemechanismus wird angestoßen, indem eine Steuerklasse das erste Element der XML-Datei erhält und auf den Verarbeitungsstack legt. Dieses Element wird entsprechend des zu lesenden Bereichs der Datei gewählt. Der genaue Aufbau und die Bedeutung der einzelnen Elemente innerhalb der XML-Datei ist in Kapitel 5.4.4 ausführlich beschrieben.

Es gibt 2 Arten von Elementen. Elemente können entweder Kinderelemente oder Inhalt besitzen, aber nicht beides.

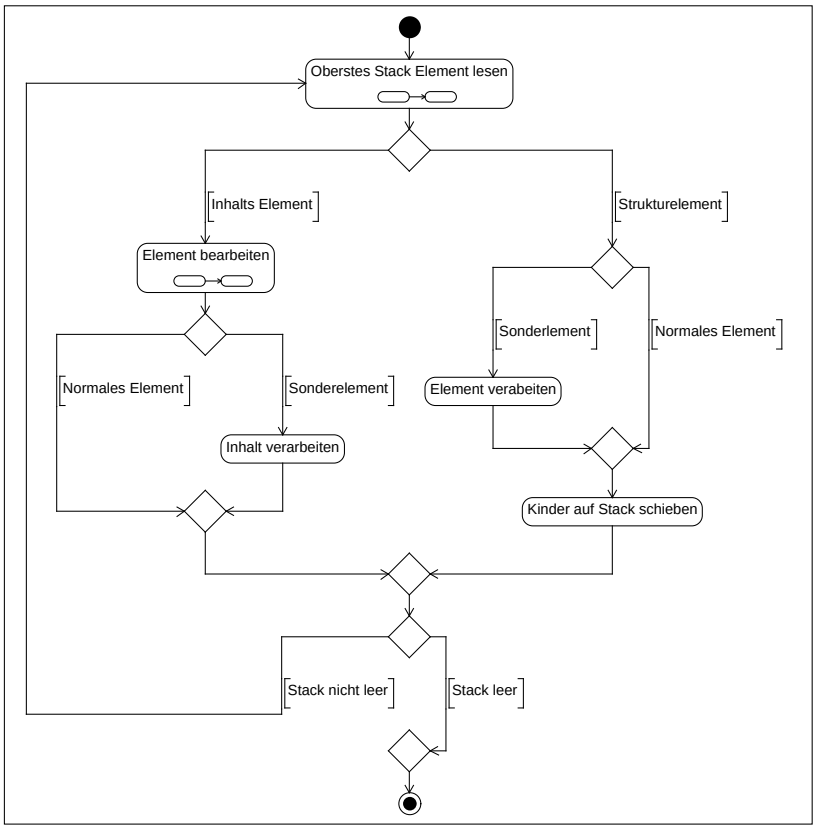



**Abbildung 5.4:** Grundschema des XML-Konverters (Auslesemechanismus)

- Elemente mit Kindern
  Sie besitzen Strukturinformationen über die ROOT-Datei, im weiteren „Strukturelement“ genannt.

- Elemente mit Inhalt
  Sie besitzen Informationen über die zu lesende Anzahl an Bytepositionen, im weiteren „Inhaltselement“ genannt.

Durch diese Struktur der Elemente ergibt sich automatisch das Vorgehen, dass erst die Struktur und dann der jeweilige, zur Struktur gehörende Inhalt beschrieben wird. Elemente werden so lange ausgewertet, bis man zur untersten Ebene geschachtelter Elemente gelangt. Dieses Element besitzt Inhalt, mit dem die ROOT-Datei gelesen werden kann. Dies entspricht einen „pre-order“ Vorgehen in der Baumstruktur der beschreibenden XML-Datei.

Manche dieser Elemente sind so genannte Sonderelemente, sie sind in in der Abbildung 5.4 separat aufgeführt. Ihr Verhalten ist innerhalb der Steuerklasse implementiert und wird durch die Informationen der ROOT-Datei selbst gesteuert. Es kann daher nicht ausschließlich mit Hilfe der beschreibenden XML-Datei abgebildet werden. Sonderlemente

können vom Typ Strukturelement oder Inhaltselement sein. Der Unterschied besteht nur in der Form der Auswertung in der Steuerklasse. Sonderelemente von Typ Struktur werden vor dem Auswertungsvorgang der Strukturelemente behandelt, Inhaltselemente nach dem Auslesevorgang.

Bei normalen Strukturelementen werden alle Kinderelemente aus der beschreibenden XML-Datei gelesen und nacheinander abgearbeitet. Die in Inhaltselementen gespeicherte Anzahl an Bytes wird aus der ROOT-Datei gelesen (siehe Abbildung 5.5). Mit Hilfe der Schema-Datei, den so genannten „Metadaten der XML Repräsentation“ (4.2) werden die gelesenen Bytes in die entsprechenden Typen des Elements umgewandelt. Das Schema enthält für jedes Inhaltselement eine Typdefinition.

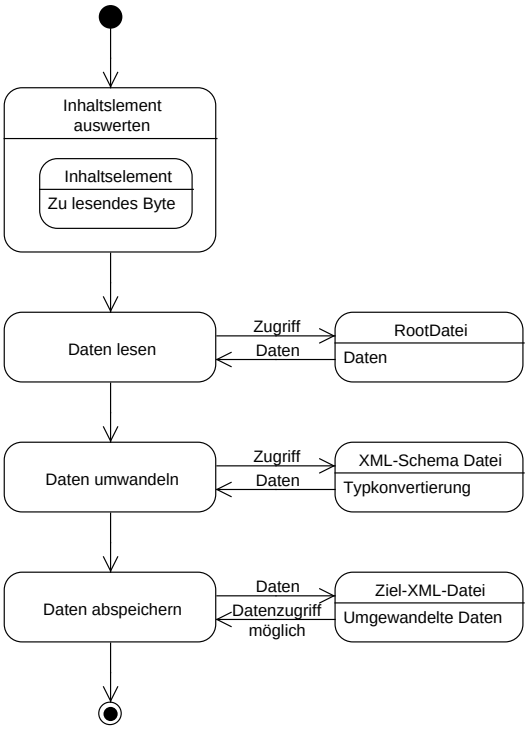



**Abbildung 5.5:** Datenauswertung von ROOT-Daten

Die gelesenen Daten werden in einer XML-Baumstruktur gehalten und der Steuerklasse zur Verfügung gestellt. Dieser Mechanismus wird in Kapitel 5.4.5 an einem Beispiel verdeutlicht.

In diesem Vorgehen finden sich die implementierten Klassen wieder. Sie und ihre Zusammenhänge werden im folgenden beschrieben.

### 5.4.2 Umsetzung

Für die Umsetzung des XML-Konverters zugrunde liegende Datenstruktur sind XML-Bäume. Sie enthalten alle notwendigen Informationen und dienen als Speichermedium der gewonnenen ROOT-Informationen.

Die für den XML-Konverter implementierten Klassen sind in Abbildung 5.6 als Klassendiagramm dargestellt.

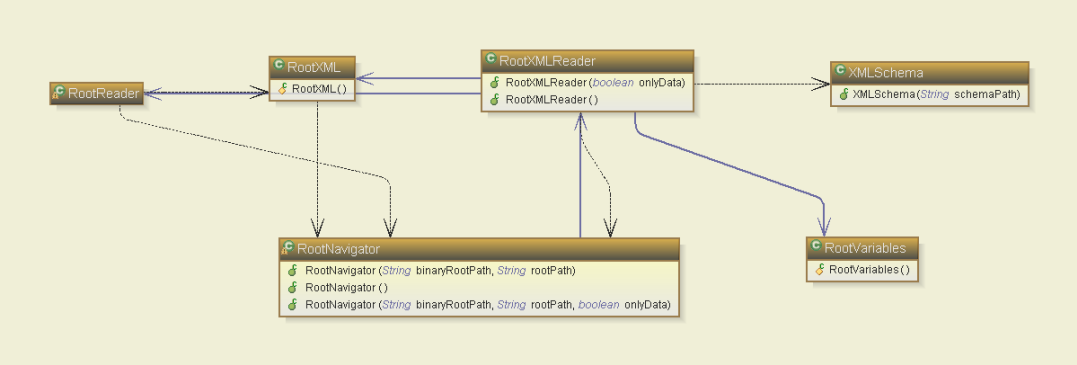



**Abbildung 5.6:** Klassendiagramm Server

Sie werden im folgenden erläutert:

- RootXMLReader
  Einleseklasse der beschreibende XML-Datei und Steuerklasse des Auslesemechanismus. Sie bildet die im in Abbildung 5.4 dargestellte Steuerklasse ab.

- RootReader
  Für das Lesen der ROOT-Datei verantwortliche Klasse.

- XMLSchema
  Klasse für die Bereitstellung der Typen der Elemente des Schemas.

- RootXML
  Für die Klasse, die für die Datenhaltung der ROOT-Daten verantwortlich ist.

- TStreamerInfos
  Für die Verwaltung der TStreamerInfos verantwortliche Klasse.

- RootVariables
  Klasse, welche die zuletzt gelesenen ROOT-Daten in einer Hash-Map zwischenspeichert und zur Verfügung stellt.

- RootNavigator
  Schnittstelle zwischen RootXMLReader und RootServlet.

Für die weiteren Erklärungen der Klassen und dessen Funktionalitäten werden einige Begriffe definiert. Innerhalb der Klassen werden mehrere XML-Bäume benutzt, welche die entsprechenden XML-Dateien darstellen. Da die Inhalte der beschreibenden XML-Datei und die Inhalte der Ziel-XML-Datei abhängig voneinander sind, werden die Knoten des XML-Baumes der beschreibenden XML-Datei „Elemente“ genannt. Diese Elemente sind gleichbedeutend mit den Elementenbegriff aus Kapitel 5.4.1.

### RootXMLReader

Die Klasse *RootXMLReader* setzt den oben beschriebenen Auslesemechanismus um, sie ist die Steuerklasse des Mechanismus. Die Klasse steuert das Anlegen der Knoten in der Ziel-XML-Datei, die beschreibende XML-Datei und das Verarbeiten der Knoten und Elemente.

Für die Umsetzung der Abarbeitung der XML-Knoten und dessen Kinder wurde ein Stack verwendet. Durch das „LIFO“-Prinzip werden die zuletzt eingefügte Knoten des Stacks abgearbeitet. Damit müssen die Kinder eines abzuarbeiteten Knoten in umgekehrter Reihenfolge auf den Stack geschoben werden. Das erste Kind befindet sich oben im Stack und kann bearbeitet werden. Damit wird die korrekte Reihenfolge gewährleistet. Dies entspricht dem Vorgehen im Kapitel 5.4.1. Die Elemente werden ausgewertet, bis die unterste Ebene, ein Blatt, erreicht wird. Die Blätter sind die Inhaltselemente. Da ein aktuelles Element sowohl Kinderelemente, als auch Geschwisterelemente besitzen kann, wurde keine Queue als Datenstruktur verwendet. Die Kinderelemente müssen vor den Geschwisterelemente abgearbeitet werden.

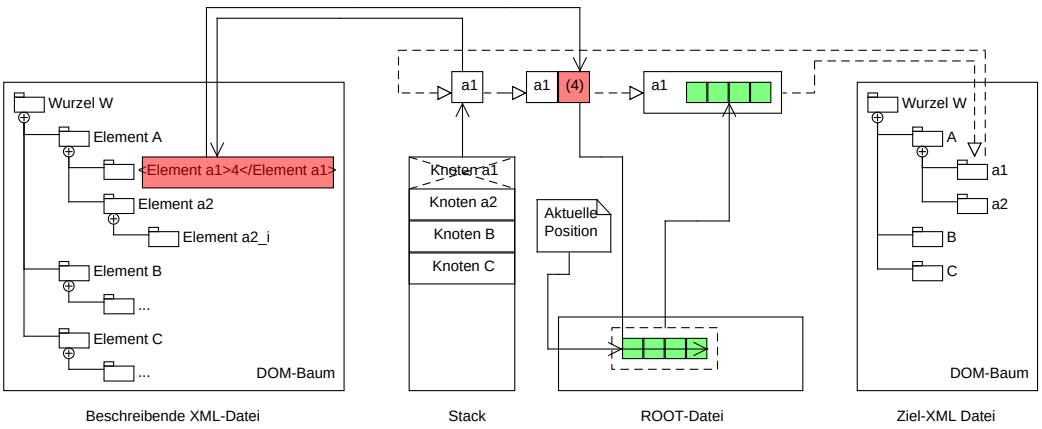



**Abbildung 5.7:** Ablauf der Verarbreitung von Elementen und Knoten

Dieser Mechanismus wird in Abbildung 5.7 veranschaulicht. In dem Beispiel wurde die Wurzel „W“ und anschließend der Knoten „A“ bearbeitet. Durch die Bearbeitung der Wurzel wurde die Beschreibung des Knotens gelesen. Das Elemente für die Wurzel besitzt drei Kinder (A,B,C). Diese wurden in der Ziel-XML-Datei angelegt und dann als Knoten

„A“, „B“ und „C“ auf den Stack gelegt. Der oberste Knoten im Stack war der Knoten „A“, dieser wurde nach dem gleichen Prinzip bearbeitet. Dadurch befinden sich aktuell noch die Geschwister und die Kinder des Knotens „A“ im Stack. In der Abbildung ist die Verarbeitung des Knotens „a1“ zu erkennen. Dieser Knoten wurde, bevor er auf dem Stack gelegt wurde, in der Ziel-XML-Datei angelegt. Für die Verarbeitung wird zuerst das beschreibende Element aus der beschreibenden XML-Datei gelesen. Das Element besitzt den Inhalt „4“. Daher werden 4 Bytes aus der ROOT-Datei gelesen. Der Inhalt wird zum Knoten „a1“ gespeichert. Danach kann der nächste Knoten „a2“ des Stacks bearbeitet werden.

**Steuermethoden des Auslesemechanismus** Die Methode *getBaseRootDaten()* steuert den abzuarbeiteten Knoten. Sie benötigt als Übergabeelement den Anfangsknoten des zu lesenden Datensatzes. Dieser Knoten wurde zuvor im Ziel-XML-Dokument angelegt. Das Ziel-XML-Dokument enthält am Ende die Struktur und die gelesenen Daten der ROOT-Datei. Die einzelnen Elemente der beschreibenden XML-Datei und ihre Bedeutungen sind in Kapitel 5.4.4 beschrieben.

Das oberste Element des Stacks wird zur Bearbeitung an die Hauptmethode der Klasse, die *readXMLInfos()* Methode übergeben. Die *readXMLInfos()* Methode steuert den Lesevorgang der ROOT-Datei. Sie setzt den in Kapitel 5.4.1 beschriebenen Auslesemechanismus um. Für jeden Knoten wird überprüft, ob er eine Versionsinformation enthält. Ist dies der Fall , so wird das Element mit der entsprechenden Versionsnummer in der beschreibenden XML-Datei ausgewählt. Ansonsten wird das erste Vorkommen in der beschreibenden XML-Datei verwendet, welches den Knoten beschreibt. Das Element stellt damit die Beschreibung des Zielknotens dar.

**Sonderelemente** Als erstes werden die Sonderelemente, welche die Struktur beschreiben, betrachtet. Es werden unter anderem die Anfangsposition der „Headers“ gesetzt, die komprimierten Datenbereiche dekomprimiert und das Verhalten von bestimmten Knoten gesteuert. Ein Sonderverhalten bei Knoten kann zum Beispiel ein zusätzlich zu lesender Bereich, unter anderem „TStreamerSTL“ der TStreamerInfos sein (siehe Anhang A.6).

**Strukturelmente** Für Strukturelemente werden die Unterelemente, so genannte Kinder des Elements aus der beschreibenden XML-Datei gelesen. Jedes Kind wird als Kindsknoten des aktuell zu betrachtenden Knotens in der Ziel-XML-Datei angelegt und auf den Stack geschoben.

**Inhaltselement** Ein Inhaltselement wird zuerst auf Attribute überprüft. Der Inhalt der Attribute steuert das weitere Verhalten und wird durch die Methode *getNodeAttributes()* gesteuert. Alle weiteren Inhaltselemente besitzen einen Textinhalt (Aufbau: 5.4.4). Das

Lesen der ROOT-Datei wird durch die Klasse RootReader realisiert. Da Inhaltselemente auch Sonderelemente sein können, werden diese nach dem Lesevorgang der ROOT-Datei noch einmal betrachtet. Die neu gelesenen Werte der ROOT-Datei steuern direkt das weitere Verhalten des Auslesemechanismus. Dazu gehört zum Beispiel das Setzen der Anzahl der Schleifendurchläufe für Arrays.

Die Methode *getNodeAttributes()* wertet die „condition"-Attribute aus. Die Werte des Attributs „condition" können sich in Länge und Bedeutung der Werte unterscheiden. Der Syntax des „conditon" Attributs werden in Kapitel 5.4.4 erläutert. Die Methode liefert je nach Auswertung der Attribute eine entsprechende Anzahl zu lesender Bytes. Dies können im Fall der Sprungadressenauswertung 4 oder 8 Bytes sein. 0 Bytes sind es beim Setzen des Arrays oder beim Setzen von Referenzen. Das ist abhängig von den Werten der Attribute, siehe Kapitel 5.4.4. Eine weitere Methode *conditionString()* hilft den Inhalt des Attributs zu verarbeiten.

**TBasket** Ein TTree-Datensatz enthält die Messdaten in so genannten Baskets. Das Element, welches die Baskets beschreibt, ist ein Sonderelement. Es wird durch die Methode *setTArrayContentInt()* gesteuert. Die Methode setzt die Struktur und die Elemente der Baskets entsprechend der Objekte, die ein Basket enthält. Das Basket selbst beschreibt, welche Objekte es enthält. Die Beschreibung des Typs des Objektes befindet sich in den TStreamerInfos. Mit Hilfe dieser Informationen kann das entsprechende Element in der beschreibenden XML-Datei ausgewählt werden. Die Methode deckt bisher die Primärdaten und grundlegende ROOT-Objekte ab. Neu definierte Typen der Objekte müssen, falls sie nicht automatisch durch die Methode abgedeckt sind, neu mit in die Methode *setTArrayContentInt()* aufgenommen werden.

**Sonstige Methoden** Die Klasse besitzt weitere Methoden, die für das Verarbeiten einer ROOT-Datei benötigt werden.

Die Grunddaten werden mit der *setXMLRoot()* Methode gesetzt. Dazu gehören die Angabe des Dateipfades zu der Schema-Datei die die Typinformationen enthält, die so genannten „Metadaten der XML Repräsentation" (Kapitel 4.3). Die Typdefinitionen werden ausgelesenen und das Wurzelelement in der Ziel-XML-Datei gesetzt.

Viele weitere Methoden werden für das Behandeln der Sonderelemente benötigt. Dazu gehören beispielsweise Methoden wie *setDataNode()* und *setTFile(). setDataNode()* bestimmt den Objekttyp des Datensatzes anhand von Angaben der ROOT-Datei. Die Methode *setTFile()* bestimmt den Typ des TFile Datensatzes. Ein TFile Datensatz kann ein KeysList Datensatz, ein FreeSegment Datensatz oder ein einfacher TFile Datensatz sein, der am Anfang oder am Ende der ROOT-Datei steht.

### RootReader

Die Klasse *RootReader* ist für den eigentlichen Lesevorgang einer ROOT-Datei verantwortlich. Sie stellt verschiedene Methoden für den Umgang mit der Datei und dessen Inhalt bereit. Die Datei kann mittels der Methode *getFile()* gelesen werden.

**Lesen** Die wichtigste Methode ist die *readByte()* Methode. Sie liest eine übergebene Anzahl an Byteposition aus der ROOT-Datei. Dabei wird unterschieden, ob aus einem komprimierten Bereich der Datei gelesen wird oder aus dem unkomprimierten Bereich.

Beim Lesevorgang wird überprüft, ob das zu lesende Element ein so genanntes Tag-Element ist. Dies sind Elemente, die nach einem Tag stehen. Tags sind eine Art spezielle Versionsnummern, sie besitzen ein negatives Vorzeichen. Wenn die Version gleich minus eins ist, muss als nächstes ein String gelesen werden. Die Länge des String ist aber nicht bekannt und damit fehlt die Angabe der zu lesenden Bytepositionen. Der String terminiert mit einer 0. Die Methode *getClassNumber()* liest solange Byte für Byte, bis der Inhalt eine 0 ist. Die gelesenen Bytepositionen werden übergeben und die Anzahl der gelesenen Bytes zurück gesetzt. Die Anzahl der Bytes kann dann in der *readByte()* Methode als String gelesenen werden.

**Speichern der gelesenen Werte** Die gelesenen Bytes werden in der Methode *setByteVariable()* in den passenden Typ konvertiert und gespeichert. Der zu einem Element gehörige Typ wird durch die Klasse *XMLSchema* bereit gestellt. Für das Konvertieren des gelesenen ByteArrays stehen verschiedene Methoden zur Verfügung. Die Daten werden dann in der Ziel-XML-Datei gespeichert. Die Ziel-XML-Datei wird durch die Klasse *RootXML* verwaltet.

**Dekomprimieren** In der Steuermethode der *RootXMLReader* Klasse wird anhand der Objektlänge eines Datenbereichs und seiner tatsächlichen Länge überprüft, ob ein komprimierter Bereich vorliegt. Ist dies der Fall, wird dieser mittels der *setDecompressInput()* Methode dekomprimiert.

**Versionstest** Einige Objekte in ROOT besitzen eine Versionsnummer und einen ByteCount oder auch nur eine Versionsnummer. Ein ByteCount ist eine Zählvariable die von ROOT genutzt wird. Da nicht definiert ist, welche der beiden Möglichkeiten in einem gegebenen Fall vorliegt, wird mit Hilfe einer Maske überprüft, ob eine Versionsnummer vorliegt. Ist dies nicht der Fall, muss es sich um die Zählvariable handeln. Die beiden gelesenen Bytes und zwei weitere Bytes ergeben die Zählvariable, ByteCount. Anschliesend werden zwei weitere Bytes für die Versionsnummer gelesen, die immer hinter einem ByteCount steht. Da für den Auslesevorgang nur die Versionsnummer benötigt wird, wird diese durch die Methode *readVersion()* bereit gestellt.

**Weitere Methoden** Desweiteren stellt die Klasse weitere Methoden bereit, wie etwa *getReadPosition()*, über die Informationen bezüglich des aktuellen Stands des Lesevorgangs abgefragt werden können.

Weitere Methoden ermöglichen das Überspringen von Bytepositionen innerhalb der ROOT-Datei. Dazu gehören Methoden wie *jump()* oder *jumpCompressed()*. Sie ermöglichen das Überspringen von Daten.

### XMLSchema

Die Klasse *XMLSchema* ist für das Verwalten der „Metadaten der XML Repräsentation“ (siehe Kapitel 4.3), die in einer Schema-Datei gespeichert sind, zuständig. Die Klasse kann die Informationen des Schemas einlesen und stellt diese für andere Klassen bereit. Mit Hilfe der *setSchema()*-Methode ist es möglich, die Schema-Datei einzulesen. Die Methode *getGroupTyps* liefert eine HashMap mit Typinformationen für ein angefragtes Element und dessen Unterelemente zurück.

### RootXML

Diese Klasse ist für die Datenhaltung der ROOT-Daten verantwortlich. Die ROOT-Daten werden in einem DOM-Baum gehalten, einer Datenstruktur für XML-Dokumente.

**Datenhaltung** Verschiedene Methoden ermöglichen den Umgang mit den ausgelesenen Daten. Dazu gehört die Methode *setResultDocument()*, sie legt eine neues XML-Dokument an, in dem die Daten gespeichert werden können. Bei dem Dokument handelt es sich um ein DOM. Bei DOM erfolgt der XML-Zugriff über eine Baumstruktur. Die Methoden *addElement()* ermöglicht das Hinzufügen von Knoten in die Baumstruktur. Mit der Methode *addRootElement()* wird das Wurzelelment, das jedes Dokument besitzen muss, angelegt. Knoten können über die Methode *removeNode()*gelöscht und über *replaceNode()* können Knoten gegen andere Knoten ersetzt werden. Die Methode *setTypAttribut()* erlaubt das Hinzufügen des „typ“ Attributs, das innerhalb der Baskets benötigt wird. Mit *setNodeContent()* wird der gelesene Inhalt der ROOT-Datei zum jeweiligen Knoten im Baum gespeichert.

**Informationsbereitstellung** Neben dem Verwalten der Daten stellt die Klassen verschiedene Möglichkeiten bereit, Informationen über die gespeicherten Daten zu erhalten. Das Dokument kann als Ganzes übergeben oder auf einer Festplatte gespeichert werden. Weiterhin kann der zuletzt angelegten Knoten eines Elements einer Ebene des Baumes zurück gegeben werden. Dadurch können Informationen, die in der ROOT-Datei gespeichert sind, zur Verfügung gestellt werden. Für den Auslesevorgang werden verschiedene Infor-

mationen benötigt, die in einer Ebene liegen. TTree-Datensätz besitzen aufgrund ihrer Baumstruktur verschiedene Ebenen für Informationen.

Da ROOT-Dateien eine Vielzahl an Informationen speichern, die sich nicht nur auf die eigentlichen Messdaten beziehen, sondern für die interne Verarbeitung von ROOT benötigt werden, wurden Methoden implementiert, um die in Baskets enthaltenen Messdaten zu erhalten. Dies kann für alle TTree-Datensätze genutzt werden. Die Methode *getTBasket()* kopiert die Baskets aus dem resultDocument in einen eigenen XML-Baum. Dieses Dokument wird von der Methode *getBasketDoc()* und *getBasketXML()* zurück gegeben. *getBasketDoc()* liefert das ganze Dokument, wogegen *getBasketXML()* spezifizierte Baskets zuvor löscht. Durch die Auswahl der Baskets ist es dem Benutzer möglich, eine Vorauswahl der gewünschten Daten zu treffen. Die Anzeige aller enthaltenden Baskets wird mit Hilfe der *getTBasketList()* Methode realisiert. Sie gibt eine Liste aller enthaltenden Basket-Bezeichnungen zurück. Über *removeBasket()* können die Baskets gelöscht werden. Das Basket-XML-Dokument wird durch die Methode *writeTBasketXML()* gespeichert.

#### TStreamerInfos

Die Klasse *TStreamerInfos* verwaltet die TStreamerInfos der ROOT-Datei. Sie ermöglicht das Einlesen und das Suchen von Informationen über beschriebene Objekte innerhalb der TStreamerInfo-XML-Datei. Diese Informationen werden für die weitere Nutzung zur Verfügung gestellt.

**Einlesen** Die Methode initTStreamerInfos() liest die TStreamerInfo XML-Datei. Die Methode isSet() gibt den Statuswert zurück, ob eine TStreamerInfo Datei vorliegt und Typanfragen gestattet sind.

**Enthaltene Informationen** Die TStreamerInfos enthalten Klassenbeschreibungen der verwendeten ROOT-Objekte in einer ROOT-Datei. Die Klassenbeschreibungen bestehen aus den Unterelementen der Klasse. Eine Beschreibung der TStreamerInfos ist in Kapitel 3.3.3 zu finden.

Die Methode *getTyp()* liefert den Typ des übergebenen Elements. Dazu werden weitere Methoden benutzt, die zuerst die Klassenbeschreibung eines Objektes suchen und dann das Element in dieser Klassenbeschreibung. Dies ermöglicht das Navigieren innerhalb der TStreamerInformationen. Wird das gesuchte Element nicht durch sein Oberelement gefunden, so werden alle Elemente der TStreamerInfos durchgegangen. Besitzt das Element keinen Typ, dann wird ein leeres TypObject zurück gegeben. TypObject ist eine innere Klasse, die für das Halten der Typinformationen zuständig ist.

### RootVariables

Die Klasse *RootVariables* speichert ROOT-Daten in einer HashMap. Als Schlüssel der HashMap wird der Elementenname verwendet, die zugehörigen Daten werden als String gespeichert. Da Elemente mehrfach benutzt werden, sind immer nur die zuletzt verwendeten Elemente gespeichert. Elemente mit dem gleichen Namen überschreiben den Eintrag in der HashMap. Die HashMap dient daher nicht für die dauerhafte Speicherung der Daten, sondern als einfacher und schneller Zugriff auf die zuletzt benutzten Informationen.

Die Klasse stellt zwei Methoden für den Umgang mit der HashMap bereit. Die Methode *getRootVariables()* überträgt die HashMap. Mit der Methode *setRootVariables()* kann entweder die gesamte HashMap neu gesetzt werden oder einzelne Einträge neu hinzugefügt werden.

### RootNavigator

Die Klasse *RootNavigator* stellt die Schnittstelle zwischen dem *ROOT-Servlet* und der *RootXMLReader* Klasse bereit. Das Servlet stellt den Dienst bereit, indem es die Anfragen des Clients bearbeitet. Die Klasse *RootXMLReader* steuert den Auslesemechanismus und die Klasse *RootNavigator* stellt Methoden bereit, welche die ROOT-Daten zurück geben oder andere wichtige Informationen, die den Umgang mit einer ROOT-Datei ermöglichen. Solche Informationen können die enthaltenen Basketbezeichnungen eines TTree-Datensatzes sein. Die wichtigsten Methoden sind:

- getKeyList()
  Liest den Header der Datei und danach die KeysList der Datei, diese wird als XML-Dokument zurück gegeben.

- getKey()
  Liest einen Datensatz und gibt ein XML-Dokument zurück.

- getTStreamerInfos()
  Liest die TStreamerInfos einer Datei.

- getBasketXML()
  Liefert ein XML-Dokument das die ausgesuchten Baskets des TTree-Datensatzes beinhaltet.

- getActualTBasket()
  Gibt ein XML-Dokument mit allen Baskets des TTree-Datensatzes.

- getTBaskets()
  Löscht aus einem übergebenden Datensatz-Dokument unerwünschte Baskets und speichert das Dokument ab.

- getTBasketList()
  Liefert eine Liste der Bezeichnungen der enthaltenen Baskets eines TTree-Datensatzes.

- getSpecialData()
  Liest einen bestimmten Datensatz, sortiert die unerwünschten Baskets aus und liefert ein XML-Dokument mit den ausgesuchten Baskets des TTree-Datensatzes zurück.

- getRootFileDescription
  Liefert die Liste aller ROOT-Dateien eines Verzeichnisses.

Die Klasse stellt weitere Methoden für den Umgang mit ROOT-Dateien bereit. Die Klasse enthält eine *main()* Methode, welche das Testen des Auslesevorgangs ermöglicht. Dies bietet sich für das Testen von Erweiterungen der beschreibenden XML-Datei an (siehe Kapitel 5.4.4 und 7.3.3). Die setMaximalElements() Methode beschränkt die zu verarbeiteten Element beim Auslesevorgang. Dies ist hilfreich, um einen Auslesemechanismus zum Testen beschränken zu können. ROOT-Dateien enthalten Array Element, die beim unkorrekten Lesen eine zu große Anzahl an Elementen in den Stack laden könnten.
Die Klasse enthält eine Möglichkeit des Protokollierens, einen so genannten Logger, der Fehler und Informationsausgaben in einer eigenen Logdatei speichert. Dies ist sehr hilfreich, um Erweiterungen zu testen und anzupassen.

### 5.4.3 Grundlagen XML-Konverter

Der XML-Konverter besitzt die Möglichkeit unabhängig von RapidMiner die Funktionen der Klasse *RootNavigator* zu nutzen. Die Klasse kann zum testen und anpassen von neuen Beschreibungen genutzt werden. Dafür sind aber einige Grundlagen zu beachten:

- Eine *main()* Methode im *RootNavigator* stellt eine mögliche Umgebung zum Testen bereit.

- Zuerst muss der Pfad der ROOT-Datei und der beschreibenden XML-Datei mittels der Methode *setPaths()* gesetzt werden.

- Um die TStreamerInfos lesen zu können, muss erst die KeysList gelesen werden.
  Die Methode *getKeyList()* liest nicht nur den KeyList Datensatz, sondern auch den Header der Datei. Dieser enthält die Positionsangabe des TStreamerInfos Datensatzes. Er kann durch die Methode *getTStreamerInfos()* gelesen werden. Die Methoden wurden nicht vereint, da der TStreamerInfo Datensatz nur für ROOT-Dateien gelesen werden muss, die eine unterschiedliche Struktur besitzen. Dann unterscheiden sich die TStreamerInfos, für Dateien des gleichen Typs sind sie identisch.

- Alle anderen Datensätze benötigen eine eigene Instanz des *RootNavigators*.

Für den Umgang mit ROOT-Dateien ist es hilfreich, die Grundlage des Aufbaus zu kennen (siehe Kapitel 4.3). Die Datensätze TFile, TStreamerInfo und TKeysList gehören zu den „Metadaten der Root-Datei“, sie besitzen Informationen über den Aufbau der ROOT-Datei und werden immer für den Umgang mit den Dateien benötigt. Beispielsweise muss die Position der in einer ROOT-Datei enthaltenen Datensätze bekannt sein, um diese auslesen zu können. Diese Information befindet sich in der KeysList der ROOT-Datei.

### 5.4.4 Aufbau der beschreibenden XML-Datei

Die Grundlage für die Funktionalität des Auslesemechanismus des XML-Adapters besteht in der vollständigen Beschreibung der in der jeweiligen ROOT-Datei verwendeten Objekte. Diese sind in der beschreibenden XML-Datei gespeichert.

Die beschreibende XML-Datei enthält sowohl den grundlegenden Aufbau einer ROOT-Datei, als auch andererseits den Aufbau der verschiedenen Objekte in ROOT. Der Aufbau entspricht einer Abbildung der ROOT-Objekte in XML. Er ist im Kapitel 3.3 erläutert. Der Auslesemeachanismus wird durch den jeweiligen Aufbau der beschriebenen ROOT-Objekte gesteuert.

#### Grundgerüst

In Abbildung 5.1 wird das grundlegende Gerüst der beschreibenden XML-Datei dargestellt. Die allgemeine Struktur einer ROOT-Datei lässt sich dabei gut in Abbildung 3.1 erkennen. Diese Struktur findet sich in der Beschreibung der ROOT-Datei in XML, die in Abbildung 5.1 dargestellt ist, wieder. Die ROOT-Datei an sich wird durch das Element *RootFile* dargestellt. *RootFile* enthält die beiden Kinder *RootHeader* und *Root*. Das Grundgerüst der der beschreibenden XML-Datei wird durch die „Metadaten der Meta-Metadaten“ definiert (siehe Abbildung 4.2). Dies ist ein Schema, das den Aufbau dieser Datei beschreibt, es ist im Anhang B dieser Arbeit enthalten.

*RootHeader* bildet den Header der Datei ab und *Root* den Datenbereich. Der Datenbereich kann mehrere Datensätze enthalten. *TKey* bildet einen einzelnen Datenblock ab. Eine ROOT-Datei kann zwar mehrere Datenblöcke enthalten, diese werden in der beschreibenden XML-Datei zusammengefasst durch ein Grundgerüst für Datenblöcke definiert.

Jeder Datenblock besitzt einen Header. Er wird durch das *RecordHeader* Element abgebildet. Zudem gehört zu jedem Datenblock noch der eigentliche Datensatz. Das Element *Data* enthält alle möglichen Beschreibungen der Datensätze und ROOT-Objekte. Erweiterungen von Objekten oder Datensätzen werden ebenfalls unter dem Element *Data* eingetragen. In dieser Abbildung ist obligatorisch nur das *TTree* Element für einen TTree-Datensatz eingetragen.

```
<RootFile xsi:noNamespaceSchemaLocation="RootBinary4.xsd" xmlns:xsi="http://www.w3.
    org/2001/XMLSchema-instance">
<!-- Header der Datei -->
    <RootHeader>
        ...
    </RootHeader>

    <!-- Datenbereich der Datei -->
    <Root>
        <!-- Datenblock -->
        <TKey>
            <!-- Header des Datenblocks -->
            <RecordHeader>
                ...
            </RecordHeader>

            <!-- Datensaetze (z.B TFile, TFree, KeysList, TList oder  TTree )-->
            <Data>
                <!-- z.B. TTree -->
                <TTree>
                ...
                </TTree>
                ...
            </Data>
        </TKey>
    <Root>
</RootFile>
```

**Listing 5.1:** XML Beschreibung einer ROOT-Datei

### Feste Objekte

Eine ROOT-Datei enthält obligatorische Elemente. Zu diesen gehören die beiden Headerarten „RootHeader“ und „RecordHeader“ und bestimmte Datensätze wie etwa „TFile“.

Der grundlegende Aufbau einer Beschreibung für ROOT-Objekte wird anhand des RootHeaders erläutert. In Abbildung 5.2 ist der Eintrag der beschreibenden XML-Datei für den *RootHeader* dargestellt. Er baut auf die Spezifikation des Header-Objektes in ROOT (siehe Tabelle 3.1 ).

Der *RootHeader* besitzt immer 13 Inhaltselemente. Diese Elemente sind: *RootFileIdentifier*, *FVersion*, *FBegin*, *FNbytesFree*, *NFree*, *FNBytesName*, *FUnits* und *FNbytesInfo*. Zwischen dem Anfangs- und dem Endtag steht die Anzahl der zu lesenden Bytepositionen. In Fall des *RootFileIdentifier* beschreibt die Zahl 4 die Bytelänge des zu lesenden Objektes.

Die zu lesenden Bytepositionen eines Inhaltselementes können von der Größe der Datei abhängen. Ab einer bestimmten Größe einer ROOT-Datei benötigen die Byteadressen für referenzierte Objekte in ROOT eine größere Bytezahl. Die Elemente *FEnd*, *FSeekFree*, *FSeekInfo* und *FUuid* sind Objekte, die Byteadressen anderer Objekte innerhalb der ROOT-Datei referenzieren. Die zu lesende Anzahl an Bytepositionen wird durch das Attribut „condition“ abgebildet. Es erlaubt die Unterscheidung zwischen einem Element

und einer Zahl. Wenn der Eintrag des *FVersion* Elements in diesem Beispiel kleiner ist als 1.000.000, dann werden 4 Bytes gelesen, ansonsten 8 Bytes. Elemente, die ein Attribut enthalten, dürfen keinen Inhalt mehr besitzen.

Ein Inhaltselement zeichnet sich entweder durch Textinhalt wie *NFree* oder durch die Attribute „condition“, wie in *FEnd* aus. Neben diesem Attribut gibt es noch ein gemeinsames Paar an Attributen, das es erlaubt Arrays mit Hilfe eines zuvor gelesenen Elements zu definieren. Diese sind „array“ und „arrayTyp“, sie werden im nächsten Absatz erläutert.

Mit diesen Einträgen ist die Beschreibung des RootHeaders abgeschlossen. Der Aufbau des RecordHeaders wird im Kapitel 5.4.5 als Grundlage des Beispiels für den Auslesemechanismus erläutert. Die Beschreibung eines Datensatzes wird im nächsten Abschnitt vorgenommen.

```
<!-- Header der Datei -->
<RootHeader>
    <RootFileIdentifier>4</RootFileIdentifier>
    <FVersion>4</FVersion>
    <FBegin>4</FBegin>
    <FEnd condition="FVersion &lt; 1000000" true="4" false="8"/>
    <FSeekFree condition="FVersion &lt; 1000000" true="4" false="8"/>
    <FNbytesFree>4</FNbytesFree>
    <NFree>4</NFree>
    <FNBytesName>4</FNBytesName>
    <FUnits>1</FUnits>
    <FCompress>4</FCompress>
    <FSeekInfo condition="FVersion &lt; 1000000" true="4" false="8"/>
    <FNbytesInfo>4</FNbytesInfo>
    <FUuid condition="FVersion &lt; 1000000" true="4" false="8"/>
</RootHeader>
```

**Listing 5.2:** XML Beschreibung des RootHeaders

### Datensätze

Der Aufbau eines kompletten Datensatzes wird anhand des KeysList-Datensatzes in Abbildung 5.3 skizziert. Der KeysList-Datensatz wurde gewählt, da er ein einfacher und kurzer Datensatz ist. Er kommt zudem in jeder ROOT-Datei vor und liefert eine Auflistung aller in einer ROOT-Datei enthaltenen Datensätze, abgesehen von den ROOT internen Datensätzen (siehe Kapitel 3.3.3). Der Aufbau des Datensatzes in ROOT ist im Anhang A.4 dargestellt.

```
<!-- KeysList (Auflistung der RecordHeaders der Datenbloecke) -->
<KeysList>
    <NKeysKeysList>4</NKeysKeysList>
    <ListEntrys array="NKeysKeysList" arrayTyp="RecordHeaderList"/>
</KeysList>
```

```
<RecordHeaderList>
    <Nbytes>4</Nbytes>
    <Version>2</Version>
    <ObjLen>4</ObjLen>
    <Datime>4</Datime>
    <KeyLen>2</KeyLen>
    <Cycle>2</Cycle>
    <SeekKey condition="Version &lt; 1000" true="4" false="8"/>
    <SeekPdir condition="Version &lt; 1000" true="4" false="8"/>
    <lnameClass>1</lnameClass>
    <ClassName condition="lnameClass"/>
    <lname>1</lname>
    <Name condition="lname"/>
    <lTitle>1</lTitle>
    <Title condition="lTitle"/>
</RecordHeaderList>
```

**Listing 5.3:** XML Beschreibung des KeysList-Datensatzes

Der Datensatz wird durch das Element *KeysList* angesprochen. *KeysList* besitzt zwei Kinder, einmal *NKeysKeysList* und *ListEntrys*. Das Element *NKeysKeysList* ist ein Inhaltselement.

Das Element *ListEntrys* wird nach dem Auslesevorgang des Datensatzes eine Liste der verfügbaren RecordHeader der ROOT-Datei enthalten (siehe Abbildung 5.4). Die in *NKeysKeysList* enthaltenen Information werden durch das Element *ListEntrys* genutzt, da *NKeysKeysList* die Anzahl der zu lesenden RecordHeader beinhaltet. Die beiden Attribute „array“ und „arrayTyp“ werden benötigt, um die Elemente und dessen Anzahl für *ListEntrys* zu definieren. Das Attribut „array“ gibt das Element an, welches die Anzahl der Kinder von *ListEntrys* bestimmt. In unserem Fall ist das *NKeysKeysList*. Dieses Element enthält im Beispiel die Zahl 7. Das Attribut „arrayTyp“ definiert das Objekt, das in unseren Beispiel 7 mal gelesen werden soll. Dies ist im Fall des *ListEntrys* das Element *RecordHeaderList*. Im Beispiel werden somit 7 mal die Elemente des Typs *RecordHeaderList* als Kinder des Elements *ListEntrys* eingefügt.

```
<TKey>
    <RecordHeader>
        ...
    </RecordHeader>
  <Data>
    <KeysList>
    <NKeysKeysList>7</NKeysKeysList>
     <ListEntrys>
          ...
       <RecordHeaderList>
           <Nbytes>66502</Nbytes>
           <Version>4</Version>
           <ObjLen>171042</ObjLen>
           <Datime>900416463</Datime>
           <KeyLen>69</KeyLen>
```

```
            <Cycle>1</Cycle>
            <SeekKey>75526</SeekKey>
            <SeekPdir>100</SeekPdir>
            <lnameClass>5</lnameClass>
            <ClassName>TTree</ClassName>
            <lname>5</lname>
            <Name>Muons</Name>
            <lTitle>30</lTitle>
            <Title>Tree containing MMuonSearchPar</Title>
        </RecordHeaderList>
        <RecordHeaderList>
            <Nbytes>506803</Nbytes>
            <Version>4</Version>
            <ObjLen>976175</ObjLen>
            <Datime>900416463</Datime>
            <KeyLen>63</KeyLen>
            <Cycle>1</Cycle>
            <SeekKey>142028</SeekKey>
            <SeekPdir>100</SeekPdir>
            <lnameClass>5</lnameClass>
            <ClassName>TTree</ClassName>
            <lname>6</lname>
            <Name>Events</Name>
            <lTitle>23</lTitle>
            <Title>Tree containing MHillas</Title>
        </RecordHeaderList>
        ...
                </ListEntrys>
            </KeysList>
    </Data>
</TKey>
```

**Listing 5.4:** Beispiel eines gelesenen KeysList-Datensatzes

Das Attribut „arrayTyp“ kann beliebige Elemente abbilden. Die Elemente *RecordHeader* und *RecordHeaderList* sind fast gleich aufgebaut. Das liegt an der Tatsache, dass sie die gleiche Struktur an unterschiedlichen Positionen innerhalb der ROOT-Datei beschreiben. Das Attribut „arrayTyp“ könnte in diesem Fall auch das Element *RecordHeader* beschreiben. Wichtig ist nur, dass die zu lesende Struktur definiert wird. Trotzdem wurde die unterschiedliche Bezeichnung *RecordHeaderList* gewählt, um eine klare Unterscheidung zum eigentlichen Header der Datei zu ermöglichen. Dies vereinfacht die Verarbeitung der erzeugten Ziel-XML-Datei.

Der KeysList-Datensatz ist ein einfacher Datensatz, für den es eine feste Beschreibung von ROOT gibt. Objekte in ROOT können in verschiedenen Versionen vorkommen. Ihr Aufbau unterscheidet sich voneinander. Der KeysList-Datensatz dagegen besitzt keine Version, er ist fest definiert.
Der TTree-Datensatz kann dagegen in mehreren Versionen vorkommen. In jeder ROOT-Datei steht vor diesem Objekt die Versionsnummer des nachfolgenden TTree-Objektes.

Diese muss gelesen werden, damit der ROOT-XML-Adapter das korrekte TTree Element aus der beschreibenden XML-Datei ausliest.

Die zugrundeliegende Klasse eines Datensatz-Objektes wird in dem Knoten *ClassName* des RecordHeader definiert. Es beschreibt das ROOT-Objekt, das als Datensatz gelesen werden muss. In Abbildung 5.4 sind 2 Datensatzheader abgebildet, beide Datensatz-Objekte sind von der Klasse TTree.

Um Referenzen auf andere Objekte innerhalb der beschreibenden XML-Datei zu ermöglichen, wird wieder das Attribut „condition“ verwendet. Eine Zusammenfassung der Möglichkeiten dieses Attibutes sind zum besseren Verständnis in Kapitel 5.4.4 noch einmal aufgeführt. Ein Objekt, das auf ein anderes Objekt zugreift, muss in der beschreibenden XML-Datei über das Attribut „condition“ eingebunden werden. Dabei ist zu beachten, dass es drei Arten von Objekten in ROOT-Dateien gibt:

- Objekte mit ByteCount und Version
- Objekte mit Version
- Objekte ohne ByteCount und ohne Version

Ein Element, das eine Referenz auf ein Objekt mit ByteCount und Version oder auf ein Objekt, das nur eine Version enthält, wird über den Eintrag „condition=“Elementenname BV“ vorgenommen werden. Der Elementenname muss gleich dem Namen des zu referenzierten Elements sein. In Abbildung 5.5 ist ein Unterelement von *TTree* das *TNamed* Element, das in Abbildung 5.6 dargestellt ist. Die Bezeichnungen der Referenz und des Elements stimmen überein. Die Angabe „BV“ definiert für den ROOT-XML-Adapter, dass vor dem Objekt der ByteCount und die Version gelesen werden muss oder nur die Version. Diese Unterscheidung übernimmt der Adapter. Der Elementenname und der angegebene Referenzname müssen sich unterscheiden.

Ein Objekt ohne ByteCount oder Version wird durch den Zusatz „PI“ referenziert. Das letzte Element in Abbildung 5.5 stellt solch eine Referenzierung dar. In der Abbildung wird die Einbindung des TTree-Datensatzes und die Referenzierung der eigentlichen Beschreibung dargestellt. Sie unterliegt den zuvor genannten Bedingungen.

```
<!-- TTree (Baumstruktur fuer gespeicherte Messdaten) -->
<TTree>
     <TTreeVP condition="TTreeV BV "/>
</TTree>

<TTreeV version="16">
     <TNamedP condition="TNamed BV "/>
     <TAttLineP condition="TAttLine BV "/>
     <TAttFillP condition="TAttFill BV "/>
     <TAttMarkerP condition="TAttMarker BV "/>
     <fEntries>8</fEntries>
```

```
    ...
    <TObjArrayLeavesPointerP condition="TObjArrayLeavesPointer PI "/>
    ...
</TTree>
```

**Listing 5.5:** Skizzenhafte XML Beschreibung des TTree-Datensatzes

### Allgemeine Objekte

Allgemeine Objekte umfassen alle Objekte, die nicht zu den zuvor definierten Objekten gehören. Sie werden ebenso wie alle zuvor genannten Objekte beschrieben.

Die beschreibenden Elemente der Objekte werden in das Element Data eingefügt, genauso wie die Elemente der Datensätze. Allgemeine Objekte werden unter anderem durch Objekte der Datensätze referenziert. In Abbildung 5.6 wird das Objekt „TNamed“ beschrieben. Es referenziert drei weitere Elemente, einmal *TObject* und zwei mal *TString*.

```
<!-- TNamed (BV) -->
<TNamed version="1">
    <TObjectP condition="TObject BV "/>
    <TTreeNamed condition="TString PI"/>
    <TTreeNamed2 condition="TString PI"/>
</TNamed>
```

**Listing 5.6:** XML Beschreibung des Objektes TNamed

### Zusammenfassung

In diesem Abschnitt der Arbeit werden die wichtigsten Regeln für den Aufbau der beschreibenden XML-Datei noch einmal zusammen gefasst.

- Das durch die „Metadaten der Meta-Metadaten“ beschriebene Grundgerüst der beschreibenden XML-Datei muss beibehalten werden.
- Neue Objektbeschreibungen werden als Kindelement von *Data* eingefügt.
- In der beschreibenden XML-Datei benutzte Bezeichner dürfen nicht umbenannt werden.
- Unterscheidungen an zu lesenden Bytezahlen, die abhängig von einem anderen Element und einem festen Wert sind werden wie folgt definiert:

  ```
  <FEnd condition="FVersion &lt; 1000000" true="4" false="8"/>
  ```

- Einfache Positionsangaben, die abhängig von einem anderen Elementsind, werden entsprechend des angegebenen Elements gesetzt:

  ```
  <lTitle>1</lTitle>
  <Title condition="lTitle"/>
  ```

- Einfache Positionsangaben, abhängig von einem anderen Element für Strings (feste Variante), werden folgendermaßen gesetzt:

```
<!-- TString -->
        <TString version="1">
            <num>1</num>
            <String condition="num"/>
        </TString>
```

- Objekt mit ByteCount und Version werden über folgenden Eintrag referenziert:

```
<TNamedP condition="TNamed BV "/>
```

- Objekt ohne ByteCount und Version können über den Wert „PI“ referenziert werden:

```
<TStreamSpecificP condition="TStreamSpecific PI"/>
```

- Versionangaben von Elementen werden über das Attribut „version“ gesetzt:

```
<TTree version="16">
```

#### Erweiterung

Die Erweiterungen der beschreibenden XML-Datei sind einfach anhand der oben genannten Regeln umzusetzen. Die Veränderung der Strukturen vorhandener Elemente kann bis auf wenige Ausnahmen vorgenommen werden. Bei dem „TBasket“ Element greift der XML-Adapter auf die Strukturen dieses Elements zu, um Informationen über die enthaltenen Messdaten zu erlangen.

Die Schwierigkeiten in der Umsetzung der Beschreibungen bestehen eher im Verständnis der zu lesenden ROOT-Datei als in der Beschreibung. Nur eine korrekt beschriebene Datei kann auch gelesen werden.

### 5.4.5 Beispiel Auslesemechanismus

Der Auslesemechanismus für ROOT-Dateien wird an einem kleinen Beispiel erläutert. In Abbildung 5.7 ist ein Teil der beschreibenden XML-Datei für den Datensatz „TStreamer-Infos“ dargestellt.

```
<!-- Datensatz-->
<TKey>
    <!-- Header des Datenblocks -->
    <RecordHeader>
        <Nbytes>4</Nbytes>
        <Version>2</Version>
        <ObjLen>4</ObjLen>
        <Datime>4</Datime>
        <KeyLen>2</KeyLen>
        <Cycle>2</Cycle>
```

```
        <SeekKey condition="Version &lt; 1000" true="4" false="8"/>
        <SeekPdir condition="Version &lt; 1000" true="4" false="8"/>
        <lnameClass>1</lnameClass>
        <ClassName condition="lnameClass"/>
        <lname>1</lname>
        <Name condition="lname"/>
        <lTitle>1</lTitle>
        <Title condition="lTitle"/>
    </RecordHeader>

    <!-- Datenbereich-->
    <Data>
        <!-- TList (StreamerInfos: Beschreibung der Daten) -->
        <TList>
            ...
        </TList>
    </Data>
</TKey>
```

**Listing 5.7:** Beispiel des Auslesemechanismus (Beschreibende XML-Datei)

Die Steuerklasse erhält das erste Element „TKey“ aus der beschreibenden XML-Datei. Das Element wird als Wurzelelement im Ziel-XML-Dokument angelegt. TKey ist ein Strukturelement, dessen Kinder erst im Ziel-XML-Dokument angelegt und als Knoten dieses Dokuments auf den Stack gelegt werden. In diesem Fall ist dies der Knoten „Data“ und dann der Knoten „RecordHeader“.

Als nächstes wird der Knoten „RecordHeader“ vom Stack gelesen. Sein beschreibendes Element wird aus der beschreibenden XML-Datei gelesen. Hätte RecordHeader eine Versionsnummer gehabt, wäre die beim Anlegen des Knotens als ein Attributwert gespeichert worden. Dann wäre an dieser Stelle auf die Versionnummer zugegriffen worden und das entsprechende Element wäre gewählt worden.

RecordHeader ist wieder ein Strukturelement, dessen Kinder angelegt und auf den Stack geschoben werden. In diesem Beispiel wird direkt ein Datensatz gelesen, die Position des Datensatzes wurde für den Lesevorgang eingegeben.

ROOT-Datei die vom Anfang der Datei gelesen werden, müssen die Position des ersten Datensatzes erst gesetzt und in der Datei besucht werden. Wenn der Auslesemechanismus das RecordHeaders Element erhält, greift er auf die zuvor gelesenen Informationen zurück, die in der Datenhaltung zur Verfügung stehen und die Positionsangaben enthalten. Entsprechend der gelesenen Positionsangabe wird diese als neue Position gewählt. RecordHeader wird dadurch als Sonderelement bezeichnet.

Aktuell befindet sich ganz unten im Stack der Knoten „Data“ und über ihm die Kinderknoten des „RecordHeader“-Knotens. Zuerst wird der Knoten „Nbytes“ abgearbeitet. Sein beschreibendes Element ist ein Inhaltselement, dessen Inhalt ausgewertet wird. In diesem Fall ist es die Zahl 4. Diese 4 Bytes werden ausgelesen und mit Hilfe der Typangaben in der Schema-Datei, die „Metadaten der XML Repräsentation“, in den entsprechenden Typen

umwandelt. Dieser gelesene Wert wird als Inhalt des „Nbytes“-Knotens gespeichert. Beim Umwandlungsprozess wird die Typdefinition eines Elements gelesen und entsprechend des Typs umgeformt. In diesem Fall wird das gelesene Bytearray in ein Integer gewandelt. Diese gelesenen Informationen stehen damit zur Verfügung. In Abbildung 5.8 sind die Typdefinitionen der RecordHeader-Kinderelemente dargestellt.

```
<!-- RecordHeader -->
<xs:element maxOccurs="1" minOccurs="1" name="Nbytes" type="xs:int"/>
<xs:element maxOccurs="1" minOccurs="1" name="Version" type="xs:short"/>
<xs:element maxOccurs="1" minOccurs="1" name="ObjLen" type="xs:int"/>
<xs:element maxOccurs="1" minOccurs="1" name="Datime" type="xs:int"/>
<xs:element maxOccurs="1" minOccurs="1" name="KeyLen" type="xs:short"/>
<xs:element maxOccurs="1" minOccurs="1" name="Cycle" type="xs:short"/>
<xs:element maxOccurs="1" minOccurs="1" name="SeekKey" type="xs:long"/>
<xs:element maxOccurs="1" minOccurs="1" name="SeekPdir" type="xs:long"/>
<xs:element maxOccurs="1" minOccurs="1" name="lnameClass" type="xs:short"/>
<xs:element maxOccurs="1" minOccurs="1" name="ClassName" type="xs:string"/>
<xs:element maxOccurs="1" minOccurs="1" name="lname" type="xs:short"/>
<xs:element maxOccurs="1" minOccurs="1" name="Name" type="xs:string"/>
<xs:element maxOccurs="1" minOccurs="1" name="lTitle" type="xs:short"/>
<xs:element maxOccurs="1" minOccurs="1" name="Title" type="xs:string"/>
```

**Listing 5.8:** Beispiel Typdefinition von RecordHeader

Die Elemente „Version“, „ObjLen“, „Datime“, „KeyLen“ und „Cycle“>2</Cycle> wurden auf identische Art ausgelesen.

Die Elemente „SeekKey“ und „SeekPdir“ sind ebenfalls Inhaltselemente. Hier bestimmt das Attribut „condition“ die zu lesende Byteposition. Wenn der Inhalt des zuvor gelesenen Knotens „Version“ kleiner ist als 1000 dann werden 4 Bytepositionen gelesen ansonsten 8.

Der nächste zu bearbeitende Knoten ist „lnameClass“, er wird genauso wie „Nbytes“ bearbeitet. Seine Informationen werden aber für den Knoten „ClassName“ benötigt. Das beschreibende Element von „ClassName“ ist wieder ein Inhaltselement. Diesmal besitzt das Attribut „condition“ aber nur einen Eintrag. Dieser ist ein Verweis auf den Knoten „lnameClass“. lnameClass ist vom Typ *short*. Dieser Shortwert beinhaltet die Anzahl der zu lesenden Bytepositionen für den Knoten „ClassName“. ClassName ist vom Typ String. In unserem Fall müsste ClassName den Inhalt „TList“ enthalten, um diesen Datensatz zu bearbeiten. Dazu würde in „lnameClass“ die Zahl 5 stehen. Dadurch weist der XML-Adapter, dass in ClassName ein String der Länge 5 steht.

Der Stack enthält zum jetzigen Zeitpunkt nur noch den Knoten „Data“. Dessen Element ist ein Strukturelement des Typs Sonderelement. In Abbildung 5.7 ist als Kind das Element „TList“ zu sehen. TList stellt die Beschreibung für das Auslesen des TStreamerInfos-Datensatzes bereit. Alle Datensätze sind unter dem Element „Data“ angelegt. Wenn das Element Data bearbeitet wird, werden nicht wie sonst alle Kinder des Elements auf den

Stack geschoben, sondern nur nur das Element, das durch den Knoten „ClassName“ spezifiziert wird. Dies kann unter anderem zum Beispiel *TTree* oder *TList* sein.

Entsprechend der Beschreibungen der Datensätze werden diese dann gelesen. Der Auslesevorgang wird beendet, wenn sich keine Knoten mehr auf dem Stack befinden.

## 5.5 Clientarchitektur

RapidMiner stellt eine Schnittstelle bereit, die es ermöglicht, Erweiterungen, so genannte Plugins, zu integrieren. Diese erlauben RapidMiner zusätzliche Funktionalitäten nutzen zu können. Das Root-Plugin ermöglicht das Einlesen von ROOT-Daten. Das Plugin wird mit Hilfe der „build.xml“-Datei erstellt.

### 5.5.1 Umsetzung

Die für die Umsetzung der Clientanwendung wurde ein neuer RapidMiner Operator „Read ROOT“ erstellt. Der Operator steuert die Anfrage an den ROOT-Server und die Bereitstellung der Daten für RapidMiner. Dazu wurde verschiedene Klassen erstellt:

- PluginInitROOTExtension
  Die Klasse initialisiert das Root-Plugin.
- RootFileReader
  Implementation des Operators „Read ROOT“.
- ParameterTypeRootConnection
  Definiert den Parametertyp RootConnection.
- RootExampleSourceConfigurationWizard
  Klasse für den Root-Wizard (Popup)
- RootExampleSourceConfigurationWizardCreator
  Klasse, die für die Erstellung des Wizard verantwortlich ist.

Die Klasse PluginInitROOTExtension initalisiert das Root-Plugin. Dabei wird eine Verbindung zwischen dem Parametertyp *RootConnection* und der Klasse *RootExampleSourceConfigurationWizardCreator* erstellt.

Der Parametertyp *RootConnection* wird als Parameter im Operator „Read ROOT“ verwendet. Er stellt die Funktionalität des Wizards bereit, mit dem der RapidMiner Benutzer durch die Auswahl der ROOT-Daten geleitet wird.

Wie oben erwähnt gibt es eine Verbindung zwischen der Parameterklasse und dem *RootExampleSourceConfigurationWizardCreator*. Dieser implementiert die Verbindung zwischen den Operator „Read ROOT“ und dem *RootExampleSourceConfigurationWizard*. Da-

für wird ein Button bereit gestellt, der durch Aktivierung den Wizard als Popup-Fenster öffnet.

Die Klasse *RootExampleSourceConfigurationWizard* implementiert den Wizard. Dieser besteht aus mehreren graphischen Benutzeroberflächen, die den Benutzer durch Verbindung des ROOT-Servers und der Selektion der gewünschten ROOT-Daten leitet. Der Wizard steuert die Anfragen an den ROOT-Server und übergibt die gelesenen Daten an die Klasse RootFileReader. Der Wizard wird in Kapitel 5.5.2 genauer betrachtet.

Durch die Klasse RootFileReader wird der Operator „Read ROOT“selbst implementiert. Dieser stellt die Daten RapidMiner für weitere Anwendungen zur Verfügung.

Die Zusammenhänge der einzelnen Klassen sind in Abbildung 5.8 noch einmal verdeutlicht.

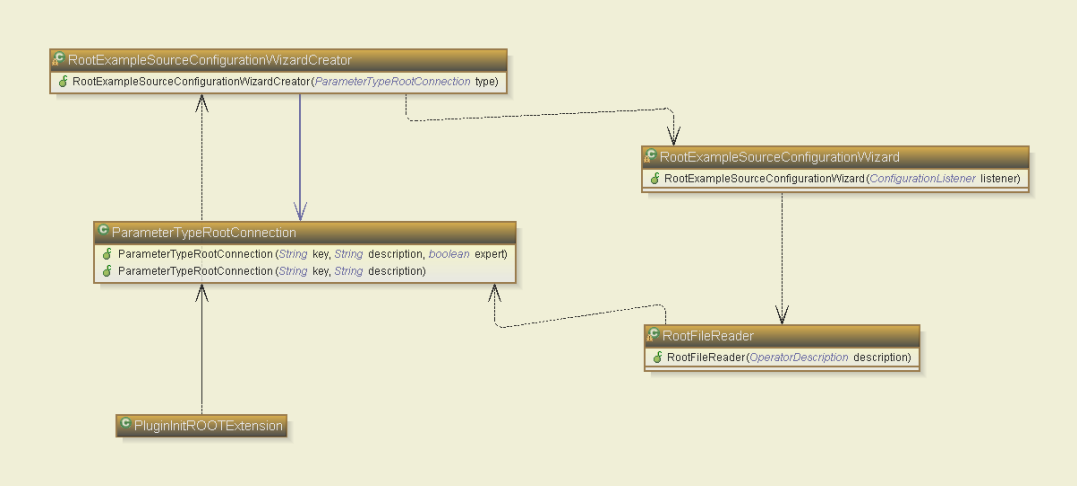



**Abbildung 5.8:** Klassendiagramm RapidMiner Root-Plugin

### 5.5.2 Root Wizard

Die Klasse *RootExampleSourceConfigurationWizard* implementiert den Root-Wizard. Die graphische Benutzeroberfläche wurde mit Swing erstellt. Swing ist eine Grafikbibliothek der Programmiersprache Java. Der Wizard gliedert sich in vier Benutzerschritte mit jeweils eigener Oberfläche.

Jeder Schritt erfüllt eine Aufgabe im Zusammenhang mit der Auswahl der ROOT-Daten. Dabei wird die Kommunikation mit dem ROOT-Server durch jeden Benutzerschritt selbst gehandhabt. Die Kommunikation zwischen Client und Server ist in Abbildung 5.9 dargestellt.

Der erste Schritt im Root-Wizard definiert die Verbindung zwischen Client und Server und ermöglicht dem Benutzer, sich mit dem ROOT-Server zu verbinden. Der Client schickt eine Anfrage aller verfügbaren ROOT-Dateien auf den Server (siehe Abbildung 5.11 a), die vom Server als Liste bereit gestellt wird. Sollte der Benutzer vergessen haben, die Daten

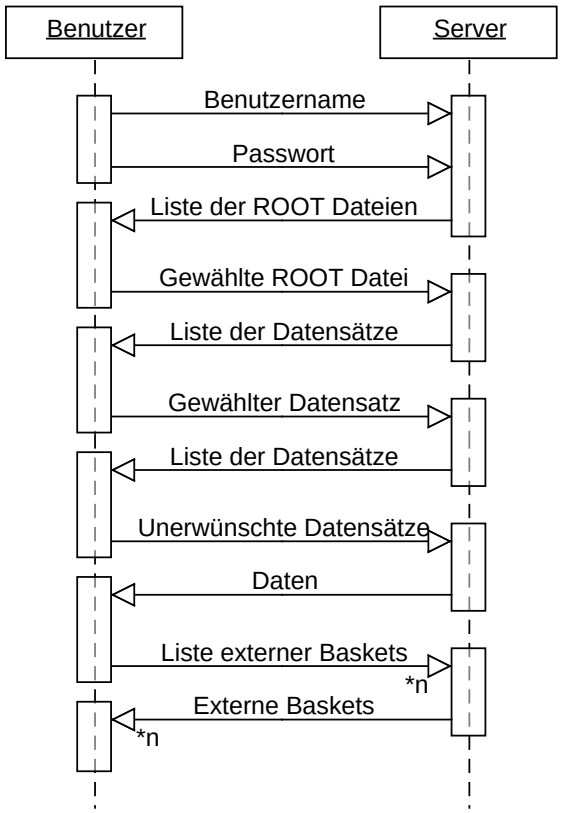



**Abbildung 5.9:** Kommunikation zwischen RapidMiner und dem Server

für die Verbindung einzugeben, oder eine Datei auszuwählen, erscheint eine Fehlermeldung. Der Client übersendet den Dateinamen der ausgewählten Datei und stellt eine Anfrage bezüglich der enthaltenen Datensätze.

Im zweiten Schritt des Root-Wizards (Abbildung 5.11 b)) wird die von Server gelieferte Liste der verfügbaren Datensätze im Form einer XML-Datei dem Benutzer zur Auswahl eines der Datensätze bereit gestellt. Dafür muss die entsprechenden Zeile der Datensatztabelle selektiert werden. Der Client übermittelt mit Hilfe der Datensatzbezeichnung die Anfrage bezüglich der enthaltenen Datengruppen.

Der Server sendet als Antwort eine Liste der enthaltenen Datengruppen, den so genannten Baskets des Datensatzes. Der Benutzer kann nun einen oder mehrere Baskets auswählen (siehe Abbildung 5.11 c)). Zudem besteht in diesem Schritt die Möglichkeit, zwischen einem XSLT-Modus oder einer normalen Umwandlung der Daten zu wählen. Die Bedeutung des XSLT-Modus ist im Kapitel 5.5.3 erläutert. Der Client stellt als nächstes die Anfrage bezüglich der gewünschten Daten. Dabei wird als Übergabeparameter eine Liste aller unerwünschten Datengruppen übermittelt, die der Server nicht mit übertragen

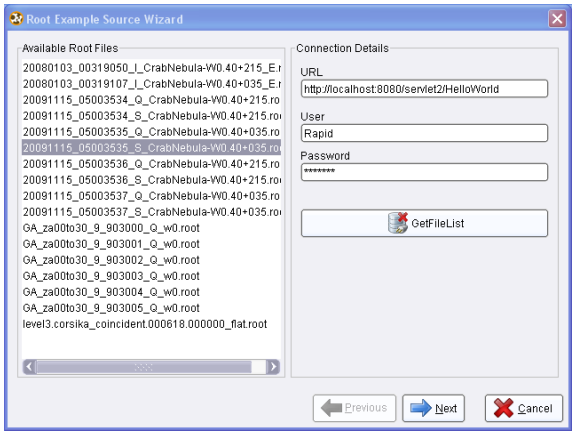



**Abbildung 5.10:** Graphische Oberfläche des Wizards

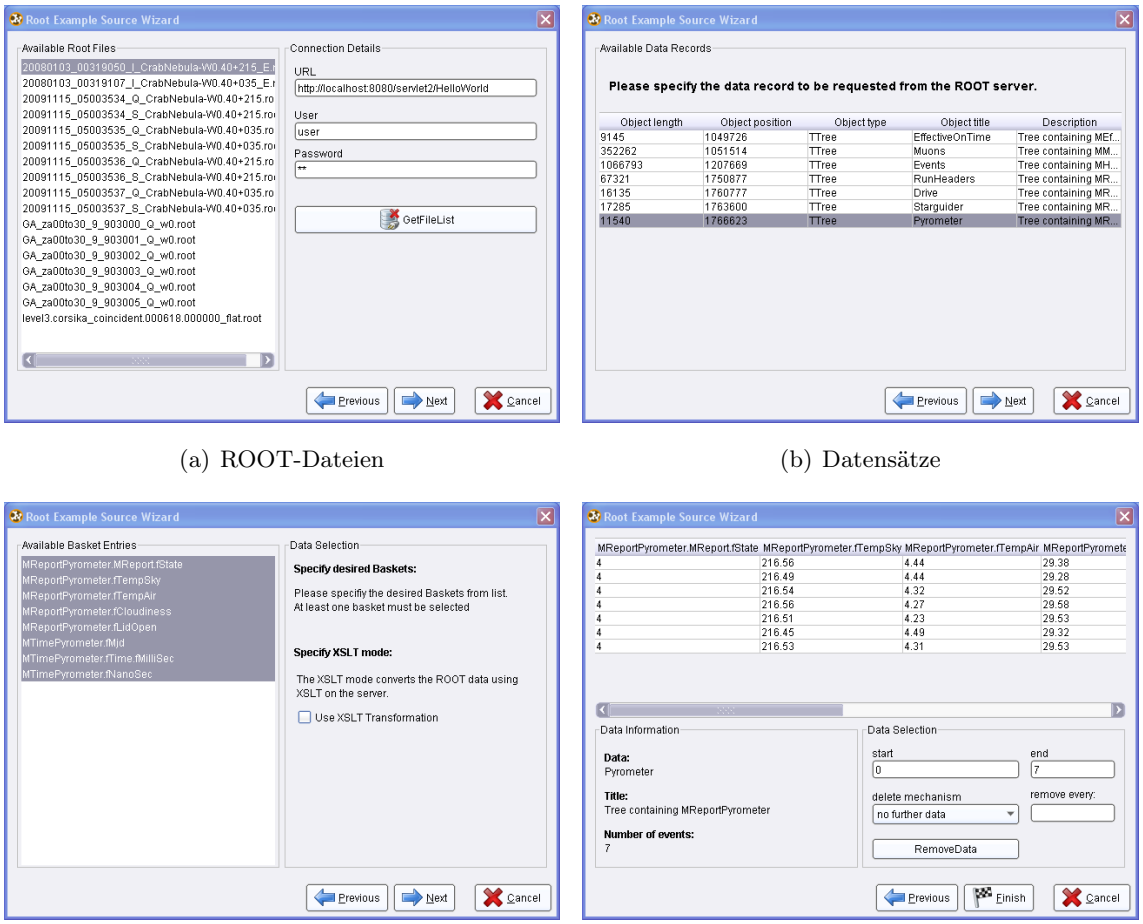
soll. Diese wird aus den nicht selektierten Datengruppen berechnet und vereinfacht das Löschen auf der Seite des Servers.

Im letzten Schritt des Wizards werden die übertragenen Daten zusammengesetzt und angezeigt (siehe 5.11d). Dies erfolgt in mehreren Schritten. Als erstes erhält der Wizard vom Server die Hauptdatei der Daten. Diese werden vom Client aus der übertragenen XML-Datei ausgelesen. ROOT gliedert Daten in so genannte externe Datensätze aus, wenn sie die Größe der im Datensatz enthalten Baskets überschreiten. Die Daten der externen Baskets müssen nachgeladen werden. Das Nachladen der Daten auf der Clientseite wurde gewählt, um die zuvor übertragene Hauptdatei möglichst klein zu halten. Dies verhindert ein große Speicherauslastung bei großen XML-Dokumenten.

Die Hauptdatei enthält für jeden Basket eine Liste aller Positionen der externen Baskets. Für jeden Eintrag in der Liste muss ein eigener Datensatz durch den ROOT-Server gelesen werden und dessen Daten übermittelt werden. Dazu sendet der Client eine Anfrage an den Server und übermittelt als Parameter die Positionsangabe des zu lesenden Datensatzes. Dieser wird als Antwort des Servers in Form einer XML-Datei übermittelt. Dieser Vorgang wird so lange ausgeführt, bis alle Daten nachgeladen wurden.

Danach stellt der Client die Daten im vierten Schritt dem Benutzer zur Ansicht zur Verfügung. Der Benutzer kann an dieser Stelle noch unerwünschte Daten löschen. Dies bezieht sich auf die Ereignisse und nicht die Datenblöcke. Möchte ein Benutzer nachträglich ganze Blöcke löschen, steht ihm die Zurück-Navigation zur Verfügung. Unter Ereignissen werden alle Einträge einer Zeile verstanden. Diese Daten stellen ein zusammengehöriges Ereignis dar.

ROOT erlaubt es zu einem Ereignis nicht nur mehrere Attribute zu speichern, sondern zu einem Attribut mehrere Werte bezogen auf ein Ereignis. Daraus resultieren Datenblöcke mit unterschiedlicher Länge. Der Wizard verteilt diese Werte auf mehrere Datenblöcke, so das ein Ereignis und seine zugehörigen Werte ausschließlich in mehren Blöcken stehen. Dadurch erhält RapidMiner alle zu einem Experiment gehörigen Werte.




(a) ROOT-Dateien

(b) Datensätze

(c) Basket

(d) Daten

**Abbildung 5.11:** Root Wizard

Die Daten werden nach dem Beenden des Wizards an den Operator übergeben und stehen zur weitere Verwendung zur Verfügung. Es wird empfohlen, die eingelesenen Daten mit Hilfe des Store Operators in Repository abzuspeichern. So kann jederzeit mittels des Retrieve Operators auf diese eingelesenen und abgespeicherten Daten zugegriffen werden.

Die für die Auswahl der Daten benutzen Parameter, wie die Dateinamen, der Datensatz, die Basketselektion und der XSLT-Modus, werden gespeichert. Diese werden für die Wiederholung eines Experimentes genutzt. Dadurch wird gewährleistet das ein Experiment in der Kommandozeilenversion in RapidMiner ausgeführt werden kann.

### 5.5.3 XSLT-Modus

Der XSLT-Modus steuert die Datenumwandlung. Das Konzept der Datenumwandlung wurde im Kapitel 4.4 beschrieben. Das Grundproblem, die eine Umwandlung der Daten nötig macht, besteht durch die unterschiedliche Datenhaltung von ROOT und RapidMiner. ROOT speichert die Daten blockweise zu einem bestimmten Attribut zugehörig ab. Ra-

pidMiner dagegen erwartet die Daten in einer so genannten „row-major“ Ordnung. Das bedeutet, dass zu einem Example alle zugehörigen Attribute abgespeichert werden.

Die Umwandlung der Daten wird durch die Klasse RootTransform vorgenommen. Diese stellt Methoden für den Transformationsprozess bereit. Die vom ROOT-Server verwendete Methode transform() erwartet als Übergabewert das zu wandelnde XML-Dokument und den Pfad zum Stylesheet.

Java stellt eine API für das Transformieren der Daten bereit. Die Java API for XML Processing (JAXP) bietet eine Standard-Schnittstelle für verschiedene Prozessoren an. JAXP stellt vier grundlegende Schnittstellen bereit, die DOM Parser-Schnittstelle, die SAX Parser-Schnittstelle, die Streaming API for XML (StAX)-Schnittstelle und die XSLT-Schnittstelle.
Die XSLT- und DOM-Schnittstelle bilden die Grundlage für den Transformationsprozess. Die DOM-Schnittstelle wurde nicht nur für die Umwandlung mit XSLT verwendet, sondern grundsätzlich zum Laden und Speichern aller XML-Dateien.

Das XSLT-Stylesheet in Abbildung 5.9 besitzt ein Haupttemplate, das drei Bereiche der ROOT-Daten umwandelt. Zuerst werden wichtige Informationen für den Benutzer in das Zieldokument integriert. Dazu gehören der Klassenname, der Name und der Titel des ROOT-Datensatzes.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet extension-element-prefixes="saxon" version="2.0" xmlns:xsl="http://
    www.w3.org/1999/XSL/Transform" xmlns:saxon="http://saxon.sf.net/">
     <xsl:output method="xml" media-type="text/xml"/>
     <xsl:template match="/">
     <!-- HEADER -->
          <TKey>
               <RecordHeader>
                    <ClassName>
                         <xsl:value-of select="TKey/RecordHeader/ClassName"/>
                    </ClassName>
                    <Name>
                         <xsl:value-of select="TKey/RecordHeader/Name"/>
                    </Name>
                    <Title>
                         <xsl:value-of select="TKey/RecordHeader/Title"/>
                    </Title>
               </RecordHeader>
               <!-- Typen -->
               <TypListe>
                    <xsl:call-template name="TemplateType"/>
               </TypListe>
               <!-- Ereignisse -->
               <xsl:for-each select=".//DataD/TBasket[1]/TArrayContentInt/*">
                    <Event>
                         <xsl:call-template name="getAttribute"/>
                    </Event>
               </xsl:for-each>
```

```
        </TKey>
    </xsl:template>
</xsl:stylesheet>
```

**Listing 5.9:** XSLT ROOT Stylesheet

Als nächstes werden die Typen der jeweiligen Einträge ausgelesen, dies geschieht mittels eines weiteren Templates (siehe Abbildung 5.10), das in das Haupttemplate integriert ist. Dieses Template geht alle Basketeinträge der XML-Datei durch und liest das Attribut „typ“ aus, welches dann im Zieldokument dargestellt wird.

```
<xsl:template name="TemplateType">
    <xsl:variable name="pos" select="position()"/>
    <xsl:for-each select="//DataD/TBasket">
        <Typ>
            <xsl:attribute name="name">
                <xsl:value-of select="fNameTBasket"/>
            </xsl:attribute><!-- Inhalt -->
            <xsl:value-of select="TArrayContentInt/@typ"/>
        </Typ>
    </xsl:for-each>
</xsl:template>
```

**Listing 5.10:** XSLT Template Type

Die Examples mit den zugehörigen Attributen werden durch das „getAttribute“ Template ausgelesen. In Abbildung 5.11 ist zu sehen, dass mit Hilfe des Templates alle Basketeinträge der Datei durchgegangen werden. Das jeweilige Example-Attribut wird aus der Liste der Kinder des Baskets ausgelesen.

```
<xsl:template name="getAttribute">
    <xsl:variable name="itempos" select="position()"/>
    <xsl:for-each select="//DataD/TBasket">
        <Element>
            <xsl:attribute name="name">
                <xsl:value-of select="fNameTBasket"/>
            </xsl:attribute>
            <xsl:attribute name="testF">
                <xsl:value-of select="$itempos"/>
            </xsl:attribute>
            <!-- Inhalt -->
            <xsl:value-of select="TArrayContentInt/*[$itempos]"/>
        </Element>
    </xsl:for-each>
</xsl:template>
```

**Listing 5.11:** XSLT Template getAttribute

Die Umwandlung mittels XSLT wird auf dem Server vorgenommen. Das verwendete Stylesheet ist nur eine Möglichkeit die ROOT-Daten zu wandeln. Durch XSLT stehen hier viele weitere Möglichkeiten, unter anderem für das Anzeigen der Daten durch XHTML für

Webanwendungen zur Verfügung. Transformationen mit Hilfe von XSLT sind CPU- und Speicher-intensiv. Zudem besitzt das „getAttribute“-Template eine hohe Anzahl an Schleifendurchläufen. Für jedes Example werden alle Baskets durchlaufen, um das passende Eintrag des Attribut zu finden, was ebenfalls zu einer erhöhten CPU- und Speicherauslastung führt. Diese Problematik wird im Kapitel 7.2 betrachtet.

Dem Benutzer des Root-Plugins wurde eine weitere Möglichkeit zur Verfügung gestellt, die Daten ohne die Umwandlung mit XSLT vom Server zu empfangen. Dazu stellt das Root-Plugin dem Benutzer die Auswahl durch die Aktivierung einer Checkbox bereit. Bei größeren Datenmengen wird diese Variante aufgrund der CPU- und Speicherauslastung empfohlen.

Das Anzeigen der Daten mittels einer Tabelle wird mit Hilfe eines TableModels realisiert. Dieser wird verwendet, um die Daten von der in ROOT genutzen „column-major“ Ordnung in die in RapidMiner genutzten „row-major“ Ordnung, zu transformieren. Das Einfügen und Auslesen der Daten der Tabelle geschieht unabhängig vom XSLT-Modus. Nur die Art des Einfügens wird durch den Modus bestimmt.

## 5.6 StreamerInfo Umwandlung

Im Zuge dieser Arbeit wurde eine Zusatzklasse TStreamerInfosModel implementiert. Sie soll den Prozess der Erweiterung der beschreibenden XML-Datei unterstützen.

Dem Konstruktor der Klasse wird der Dateipfad für die TStreamerInfos der zu beschreibenden Datei übergeben, der Pfad der bereits existierenden beschreibenden XML-Datei und der Pfad der Schema Datei, welche die Typbezeichnungen enthält. Die Klasse berechnet aus den TStreamerInfos und den bereits existierenden Beschreibungen alle fehlenden Objekte und beschreibt diese, soweit die Definition der Typen bekannt ist. Die Ergebnisse werden in einer separaten XML-Datei abgespeichert.

Für die Berechnung der fehlenden ROOT-Objekte und ihrer Struktur stellt die Klasse TStreamerInfosModel verschiedene Methoden bereit. Die Hauptmethode ist *translateTStreamer()*, sie steuert den Transformationsprozess. Nur Objekte der TStreamerInfo, die nicht in der beschreibenden XML-Datei enthalten sind oder dessen Version nicht vorhanden ist, werden angelegt. Für das Anlegen der Objekte steht die Klasse *addMainBElement()* bereit. Die Struktur der Objekte wird mittels der *setTStreamObj()* Methode erstellt. Sie steuert das Anlegen der Unterelemente eines jeden Objektes. Die Typen der Unterelemente wurden durch die Klasse *setTypInfo()* gesetzt. Sie ist die wichtigste Methode für den Aufbau der Struktur eines Objektes. Diese Methoden können einfach um noch nicht implementierte Typen erweitert werden. Die Methode *getUnusedName()* steuert die Vergabe von Bezeichnungen eines Elementes.

In Abbildung 5.12 sind zwei von der Klasse TStreamerInfosModel erzeugte Objekte zu sehen.

```
<TNamed version="1">
    <TObjectP condition="TObject BV"/>
    <fName condition="TString PI"/>
    <fTitle condition="TString PI"/>
</TNamed>
<TObject version="1">
    <fUniqueID>4</fUniqueID>
    <fBits>4</fBits>
</TObject>
```

**Listing 5.12:** Erzeugte Beschreibung

Sie sind aus der Beschreibung entstanden, die in Abbildung 5.13 dargestellt ist.Dabei handelt es sich um die beiden Klassenbeschreibungen der Objekte TNamed und TObject. Für die Verarbeitung liegen die TStreamerInfo Daten in XML-Format vor.

```
StreamerInfo for class: TNamed, version=1, checksum=0xfbe93f79
  TObject        BASE            offset=  0 type=66 Basic ROOT object
  TString        fName           offset=  0 type=65 object identifier
  TString        fTitle          offset=  0 type=65 object title

StreamerInfo for class: TObject, version=1, checksum=0x52d96731
  UInt_t         fUniqueID       offset=  0 type=13 object unique identifier
  UInt_t         fBits           offset=  0 type=15 bit field status word
```

**Listing 5.13:** TStreamerInfo darstellung

Die dargestellte Struktur ist kein Garant auf die tatsächliche Struktur der ROOT-Objekte innerhalb der Datei. Sie bildet die Struktur der TStreamerInfos auf der Grundlage der beschreibenden XML-Datei ab. Es hat sich jedoch gezeigt, dass der tatsächliche Aufbau der ROOT-Dateien in Details abweichend ist. Zudem erleichtert sie die Arbeit durch die strukturierten Vorgaben und die einheitlichen Benennung der Elemente.

# Kapitel 6

# System-Bedienung

In diesem Kapitel der Arbeit wird die Bedienung des Systems an Hand von drei Beispielen dargestellt. Sie geben verschiedene, den Funktionsumfang repräsentierende Möglichkeiten des ROOT-Plugins wieder. Die Beispielanwendungen richten sich dabei auch an die verschiedene Anforderungen von möglichen Benutzergruppen.

Als Benutzer wird ein Anwender eines Systems verstanden. Es gibt unterschiedliche Benutzergruppen, welche die Analysemöglichkeit von ROOT-Dateien mittels RapidMiner nutzen möchten. Benutzergruppen können zum Beispiel aus: DataMiner, Physiker, Statistiker und Physikinteressierten bestehen. Die verschiedene Benutzergruppen besitzen unterschiedliche Kenntnisse im Bereich Software, DataMining und physikalische Hintergrundinformationen.

## 6.1 Beispielanwendung

Die Beispielanwendungen benutzen Daten des Physikbereichs der Astroteilchenphysik. Die Daten stammen sowohl aus dem MAGIC-Projekt (siehe Kapitel 2.2.2), als auch aus dem IceCube-Projekt (siehe Kapitel 2.2.3). Da sich die Fähigkeiten der Benutzergruppen überschneiden, sind die Beispiele nicht auf eine Benutzergruppe ausgerichtet.

### 6.1.1 Anlyseverfahren MARS

Die Daten des MAGIC-Projektes werden bisher mit der „Magic Analysis and Reconstruction Software“ (MARS) weiterverarbeitet [5]. MARS ist ein auf ROOT aufbauendes Softwarepaket, welches die von der MAGIC-Kamera aufgenommenen Bilder in einzelnen Schritten verarbeitet. Die wichtigsten Bearbeitungsschritte werden mit Callisto, Star, Ganymed und Sponde [31] bezeichnet.

- Calibrate light signals and time offsets (Callisto)
  Dient der Signalextraktion und der Kalibrierung der Rohdaten. Ausserdem werden Korrekturen bezüglich auftretender Hardware-Fehler ausgeführt.

- Standard analysis and reconstruction (Star)
  Dient der Bildbereinigung und der Bildparameterberechnung. Die Bildbereinigung versucht alle Ereignisse zu filtern, die nicht durch Teilchenschauer entstanden sind. Bei den Bildparametern werden über 30 verschiedene Parameter aus den Rohdaten berechnet.

- Ganymed
  Dient der Klassifizierung der Events. Dabei wird jedes Event der Klasse der Gamma oder Nicht-Gamma zugeordnet. Zur Klasse der Nicht-Gamma gehören alle Hadronen und Events mit zu wenigen Informationen.

- Sponde
  Dient der Berechnung der Energie der Gamma-Teilchen.

### 6.1.2 Beispiel 1

Das Beispiel 1 richtet sich an alle Benutzergruppen. Hier werden ROOT-Daten auf komfortable Art betrachtet. Der in Beispiel 1 benutze Datensatz ist ein so genannter Superstar-Datensatz. Dies bedeutet, die Analyse der Daten eines Teilchenschauers werden für die Teleskope MAGIC I und MAGIC II für die Schritte Calisto und Star der MARS-Analyse für beide Teleskope getrennt durchgeführt. Die Daten von MAGIC I und II werden durch einen weiteren Schritt „Superstar“ eingelesen, die zusammengehörigen Paare der Ereignisse identifiziert und weitere Parameter berechnet. In diesem Beispiel soll gezeigt werden, ob Unterschiede zwischen den Magic I und Magic II Daten vorliegen.

**Datensatz** Grundlagen des verwendete Datensatzes:

- Datei: 20091115_05003535_S_CrabNebula-W0.40+035.root

- Ausgewählte Daten: 1.464.432 (16 Attribute mit je 91.527 Daten)

Die ROOT-Datei beeinhaltet insgesamt 216 Datenblöcke (Attribute) , davon wurden 16 Attribute ausgewählt und 175 externe Datensätze nachgeladen. Dabei handelt es sich um die Hillas Parameter *Length*, *Width*, *Size*, *Delta*, *SinDelta*, *CosDelta*, *MeanX* und *MeanY*. Hillas Parameter sind die Bildparameter, die unter anderem durch den Bearbeitungsschritt Star berechnet wurden. In Abbildung 6.1 sind einige Hillas-Parameter grafisch Dargestellt.

**ROOT-Plugin** Im ersten Schritt des Wizards werden die Daten für die Verbindung zum ROOT-Server eingegeben. Danach kann mit Hilfe des Buttons „GetFileList“ eine Liste der ROOT-Dateien des Server angefragt werden (siehe Abbildung 6.2 a).

Zum nächsten Schritt gelangt man durch das „Next“ Bedienelement. Der Wizard stellt die Anfrage nach allen Datensätze der übermittelten Datei. In diesem Schritt werden die

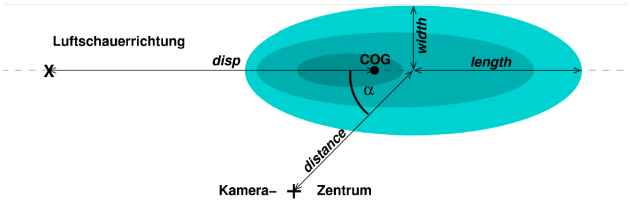
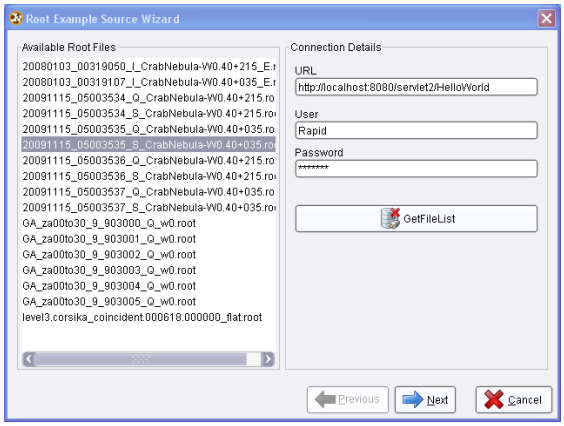
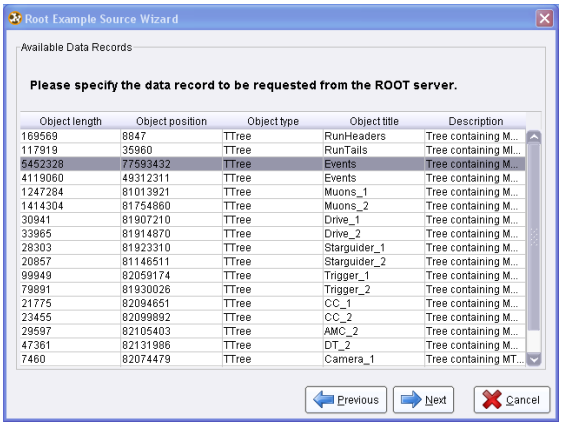



**Abbildung 6.1:** Bedeutung grundlegender Hillas-Parameter [28]

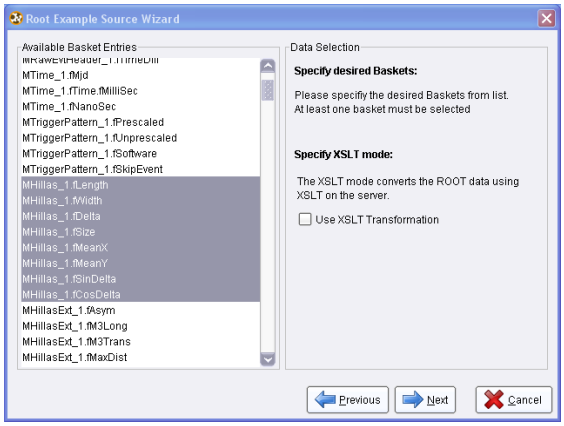
in den ROOT-Dateien enthaltenen Datensätze angezeigt. Mindestens ein Datensatz muss ausgewählt werden (Abbildung 6.2 b).



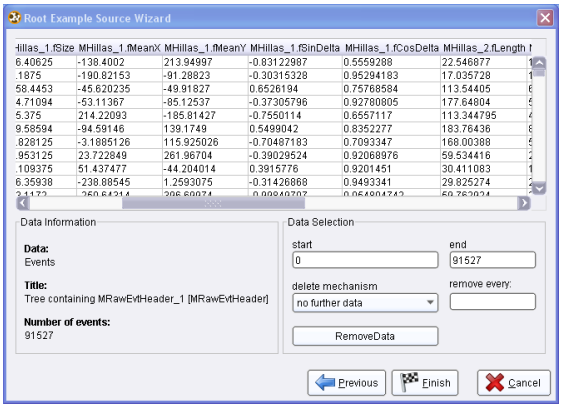

(a) ROOT-Dateien




(b) Datensätze




(c) Basket



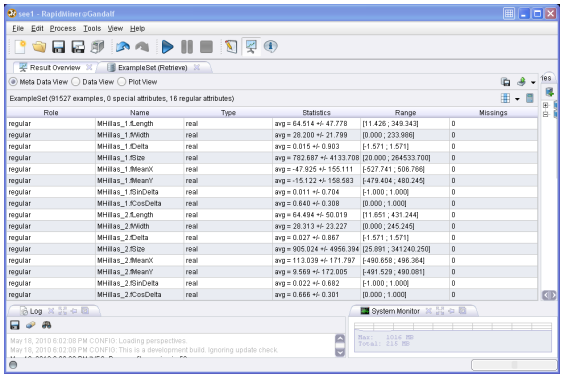

(d) Daten

**Abbildung 6.2:** Root Wizard in Beispiel 1

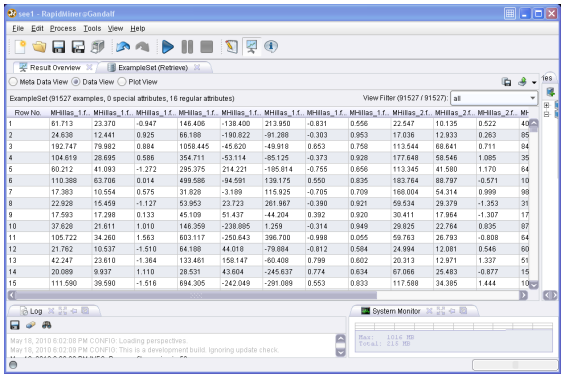
Durch die Aktivierung des nächsten Schritts werden die Daten durch den Server berechnet und eine Liste aller enthaltenen Datenblöcke (Attribute) an den Client geschickt. Der Benutzer kann in diesem Schritt einen oder mehrere Datenblöcke auswählen, die im Datensatz enthalten sind (Abbildung 6.2 c).

Im letzten Schritt erhält der Wizard die vom Server ausgewählten Daten. Der Client integriert im vierten Schritt alle externen Datensätze (Abbildung 6.2 d). Nach Betätigen des „Finish“ Bedienelements werden die Daten für die weitere Verwendung in RapidMiner bereit gestellt.

**Ergebnisse** RapidMiner stellt eine Reihe von Möglichkeiten für die Darstellung von Ergebnissen bereit. Dazu gehören unter anderem die Meta Data Sicht auf die importierten Daten (Abbildung 6.3 a), die Daten Sicht (Abbildung 6.3 b) und die Plot Sicht (Abbildung 6.3 c). Je nach verwendeten Operatoren, wie etwa dem „Decision Tree“, stellt RapidMiner weitere Sichten, Graph- und Textansicht bereit.



(a) Meta Data Sicht



(b) Daten Sicht

(c) Plot Sicht

**Abbildung 6.3:** Ergebnis-Repräsentationen des Beispiel 1

Der Benutzer kann sich Daten direkt nach dem Einlesevorgang in RapidMiner anzeigen lassen. Dazu muss der Ausgang des „Read ROOT“ Operators nur an den Result-Knopf der RapidMiner Oberfläche verbunden werden (siehe Abbildung 6.4).

Zur Veranschaulichung der grafischen Ergebnisdarstellung wurden die Daten als Scatter-Plot dargestellt und als Histogramm.
Die in Abbildung 6.5 dargestellten Scatter zeigen in Abbildung a) die Parameter „Länge“

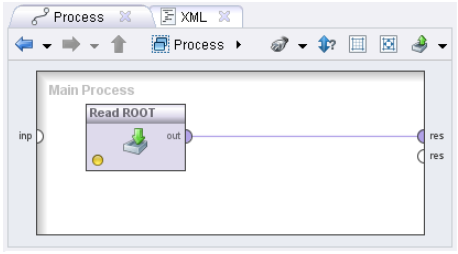
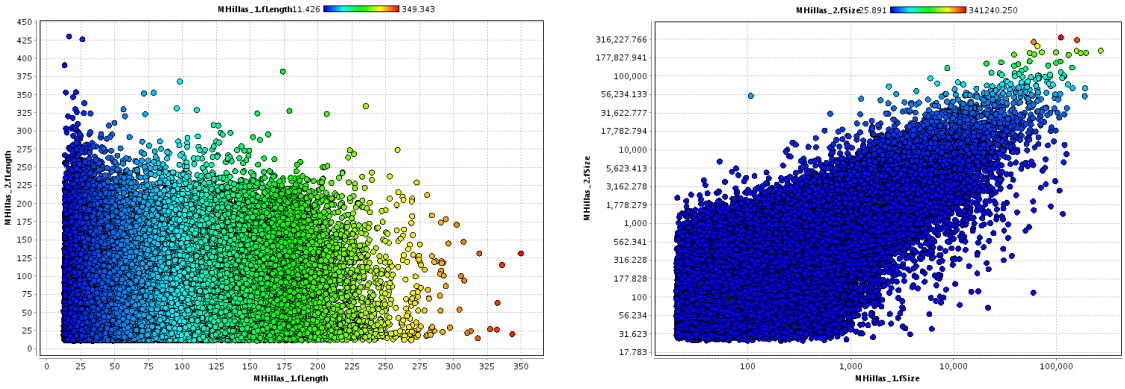



**Abbildung 6.4:** Read ROOT Operator

von Magic I aufgetragen gegen den Parameter Länge von Magic II, in Abbildung b) wird die Verteilung des Hillas-Parameter „Größe“ der beiden Teleskope dargestellt.

Die in Abbildung a) dargestellten Parameter Length_1 und Length_2 sind breit verteilt, daraus wird geschlossen, dass sie unkorreliert sind. Die in der zweiten Abbildung dargestellten Parameter Size_1 und Size_2 wurden im Log Scale Modus betrachtet. Ihre Werte nähern sich etwas mehr an als die Werte des Längen-Parameters. „Die Korrelation ermittelt den Grad der Stärke der Abhängigkeit zwischen zwei Merkmalen“ [12].




(a) Length Parameter

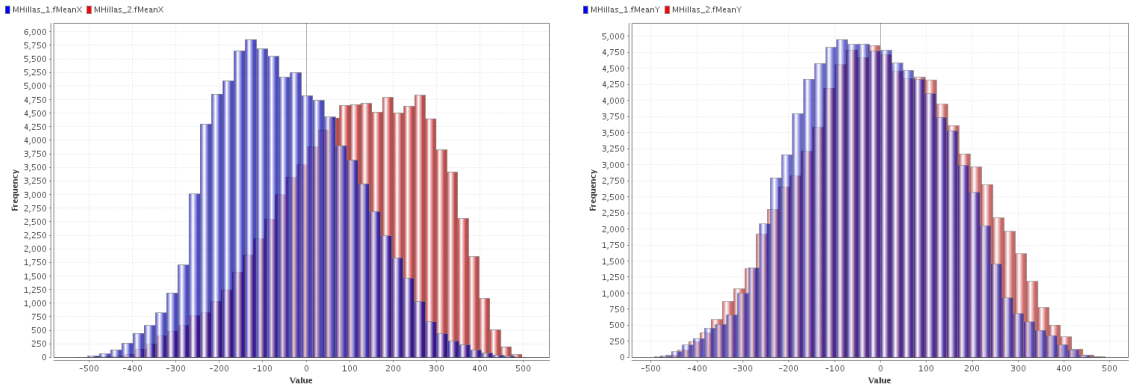
(b) Size Parameter

**Abbildung 6.5:** Beispiel 1 Scatter

Die in Abbildung 6.6 dargestellten Histogramme stellen die Parameter MeanX in Abbildung a) dar und MeanY in Abbildung b). Sie enthalten die X- und Y-Koordinaten des Zentrums der aufgenommenen Ellipse (siehe Kapitel 2.2.2). Das aufgenommene und bereinigte Bild einer Ellipse wird durch die Hillas-Parameter beschrieben (siehe Abbildung 6.1).

### 6.1.3 Beispiel 2

Das Beispiel 2 richtet sich in erster Linie an DataMiner oder Physiker. Es gibt einen kurzen Einblick in die Verarbeitungsmöglichkeit von eingelesenen Daten. RapidMiner stellt eine Vielzahl an Operatoren bereit, welche das Analysieren der Daten erlauben. Beispiel 2 zeigt




(a) ROOT-Dateien (b) Datensätze

**Abbildung 6.6:** Beispiel 1 Histogramm

zudem die Möglichkeit, mehrere ROOT-Dateien mittels des „Read ROOT“ Operators zu lesen.

**Datensatz** Bei dem verwendeten Datensatz handelt es sich um kleinere Datensätze als in Beispiel 1. Die ROOT-Dateien enthalten Daten des MAGIC Teleskop I. Diese wurden mit dem Bearbeitungsschritt STAR verarbeitet.

- Datei 1: 20080103_00319107_I_CrabNebula-W0.40+035_E.root
  Datei 2: 20080103_00319050_I_CrabNebula-W0.40+215_E.root

- Ausgewählte Daten: 862.650 (50 Attribute mit je 17253 Daten)

Es wurden alle möglichen Datenblöcke (Attribute) ausgewählt und 66 externe Datensätze nachgeladen.

**ROOT-Plugin** Die Handhabung des ROOT-Plugins unterscheidet sich nicht vom Beispiel 1. In diesem Beispiel wurden zwei Dateien ausgewählt. Wenn mehrere Dateien ausgewählt werden, müssen diese die gleiche Struktur besitzen und damit die gleichen Attribute enthalten. Der Auslesevorgang wird durch die Erste der ausgewählten ROOT-Dateien gesteuert.

In dem Beispiel wird eine Korrelations-Matrix benutzt, um Zusammenhänge zwischen den einzelnen Parameter betrachten zu können (siehe Abbildung 6.7). Eine Korrelations-Matrix bestimmt die Korrelation zwischen allen Attributen und erzeugt einen Gewichtsvektor basierend auf der Korrelation.

Der „Remove Useless Attributes“ Operators [29] filtert dabei alle für die Korrelation unbrauchbaren Daten aus dem Datensatz des RapidMiners heraus.

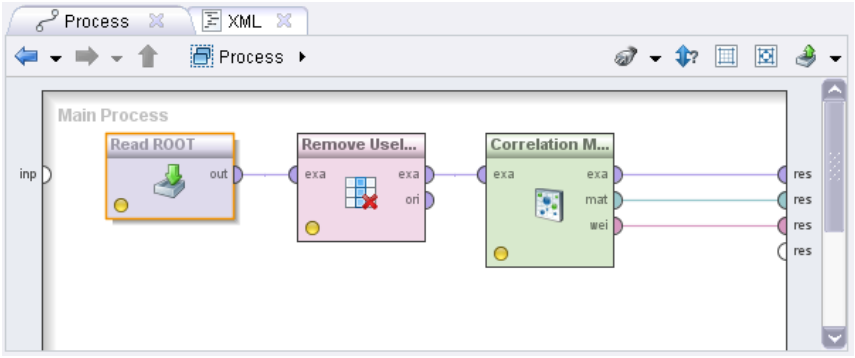



**Abbildung 6.7:** Arbeitsmöglichkeiten mit Hilfe des Read ROOT Operator

**Ergebnisse** In Abbildung 6.8 ist ein Ausschnitt der Korrelations-Matrix zu sehen. In ihr kann man zum Beispiel ablesen, das der MHillas-Parameter Länge nicht mit dem NewImageParameter Dichte korreliert (siehe eingerahmtes Kästchen). Die Zahl 1 steht für eine positive Korrelation, eine 0 für unabhängige Variablen und eine -1 für eine negative Korrelation. Der NewImageParameter „fConc“ stellt die Dichte dar. Sie ist die Summe der höchsten Pixel geteilt durch die Größe.

| Attributes | MHillas.fLen... | MHillas.fWi... | MHillas.fDel... | MHillas.fSize | MHillas.fMe... | MHillas.fMe... | MHillas.fSin... | MHillas.fCo... | MHillasExt.f... | MHillasExt.f... | MHillasExt.f... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MHillas.fLength | 1 | 0.196 | -0.000 | 0.089 | 0.003 | 0.001 | 0.003 | -0.001 | -0.029 | 0.020 | 0.005 |
| MHillas.fWidth | 0.196 | 1 | 0.004 | 0.310 | -0.001 | 0.004 | 0.004 | 0.009 | -0.017 | 0.004 | -0.006 |
| MHillas.fDelta | -0.000 | 0.004 | 1 | 0.002 | 0.007 | 0.000 | 0.260 | -0.007 | 0.001 | 0.017 | -0.003 |
| MHillas.fSize | 0.089 | 0.310 | 0.002 | 1 | -0.008 | 0.007 | 0.005 | -0.002 | -0.040 | -0.015 | -0.024 |
| MHillas.fMeanX | 0.003 | -0.001 | 0.007 | -0.008 | 1 | 0.022 | -0.003 | 0.004 | 0.041 | -0.031 | -0.014 |
| MHillas.fMeanY | 0.001 | 0.004 | 0.000 | 0.007 | 0.022 | 1 | 0.012 | 0.006 | 0.002 | -0.005 | -0.029 |
| MHillas.fSinDelta | 0.003 | 0.004 | 0.260 | 0.005 | -0.003 | 0.012 | 1 | -0.016 | -0.023 | 0.024 | 0.006 |
| MHillas.fCosDelta | -0.001 | 0.009 | -0.007 | -0.002 | 0.004 | 0.006 | -0.016 | 1 | -0.002 | 0.004 | -0.001 |
| MHillasExt.fAsym | -0.029 | -0.017 | 0.001 | -0.040 | 0.041 | 0.002 | -0.023 | -0.002 | 1 | 0.219 | 0.032 |
| MHillasExt.fM3Long | 0.020 | 0.004 | 0.017 | -0.015 | -0.031 | -0.005 | 0.024 | 0.004 | 0.219 | 1 | 0.116 |
| MHillasExt.fM3Trans | 0.005 | -0.006 | -0.003 | -0.024 | -0.014 | -0.029 | 0.006 | -0.001 | 0.032 | 0.116 | 1 |
| MHillasExt.fSlopeLong | -0.012 | -0.006 | -0.001 | -0.023 | -0.009 | -0.005 | -0.003 | -0.005 | 0.033 | 0.043 | 0.067 |
| MHillasExt.fSlopeTrans | 0.009 | -0.004 | -0.004 | -0.000 | 0.011 | 0.009 | -0.005 | -0.006 | 0.002 | -0.003 | 0.018 |
| MHillasSrc.fAlpha | 0.015 | 0.014 | 0.013 | 0.004 | 0.014 | 0.140 | 0.032 | -0.017 | 0.004 | 0.011 | 0.003 |
| MHillasSrc.fDist | 0.163 | 0.078 | -0.002 | 0.058 | 0.002 | -0.028 | 0.003 | 0.005 | -0.024 | 0.012 | 0.001 |
| MHillasSrc.fCosDeltaAlpha | 0.005 | -0.007 | -0.022 | 0.009 | 0.162 | 0.010 | -0.028 | 0.010 | 0.053 | -0.022 | -0.009 |
| MHillasSrc.fDCA | 0.002 | 0.008 | -0.017 | -0.004 | -0.002 | 0.006 | -0.031 | 0.023 | -0.006 | 0.018 | -0.006 |
| MHillasSrc.fDCADelta | -0.006 | -0.004 | -0.068 | 0.001 | -0.004 | -0.111 | -0.072 | -0.032 | 0.003 | 0.007 | 0.050 |
| MImagePar.fNumIslands | 0.168 | 0.160 | 0.005 | 0.036 | 0.001 | -0.005 | -0.011 | 0.002 | 0.002 | 0.001 | 0.006 |
| MImagePar.fNumSinglePixels | 0.115 | 0.120 | 0.004 | 0.042 | 0.006 | -0.001 | -0.022 | 0.000 | -0.007 | -0.005 | -0.007 |
| MImagePar.fSizeSinglePixels | 0.109 | 0.122 | -0.007 | 0.046 | -0.014 | -0.006 | -0.009 | 0.004 | -0.012 | -0.006 | -0.007 |
| MImagePar.fSizeSubIslands | 0.109 | 0.132 | 0.010 | 0.092 | 0.011 | 0.011 | 0.000 | 0.006 | -0.023 | -0.014 | 0.006 |
| MImagePar.fSizeMainIsland | 0.096 | 0.110 | -0.004 | 0.556 | -0.005 | -0.001 | -0.005 | -0.001 | -0.038 | -0.005 | -0.030 |
| MNewImagePar.fConc | -0.746 | -0.182 | -0.003 | -0.062 | -0.013 | 0.003 | -0.002 | -0.009 | 0.017 | -0.019 | -0.009 |
| MNewImagePar.fConc1 | -0.183 | -0.179 | -0.008 | -0.060 | 0.013 | -0.005 | -0.004 | -0.005 | 0.002 | 0.001 | -0.009 |
| MNewImagePar.fConcCOG | -0.197 | -0.167 | -0.008 | -0.047 | 0.013 | -0.005 | -0.003 | -0.003 | 0.001 | -0.001 | -0.013 |
| MNewImagePar.fConcCore | -0.093 | -0.058 | -0.008 | 0.005 | 0.013 | -0.006 | -0.002 | 0.001 | -0.009 | -0.002 | -0.015 |
| MNewImagePar.fUsedArea | 0.176 | 0.232 | 0.002 | 0.298 | -0.004 | -0.001 | 0.005 | -0.000 | -0.028 | -0.009 | -0.031 |
| MNewImagePar.fCoreArea | 0.163 | 0.218 | 0.003 | 0.324 | -0.004 | 0.002 | 0.005 | 0.000 | -0.029 | -0.011 | -0.036 |

**Abbildung 6.8:** Beipiel 2: Korrelations-Matrix

Die aus der Korrelation entstandene Gewichtung wird gleichzeitig in RapidMiner angezeigt. Sie können durch Auswählen des entsprechenden Tab betrachtet und grafisch dargestellt werden (siehe Abbilung 6.9).

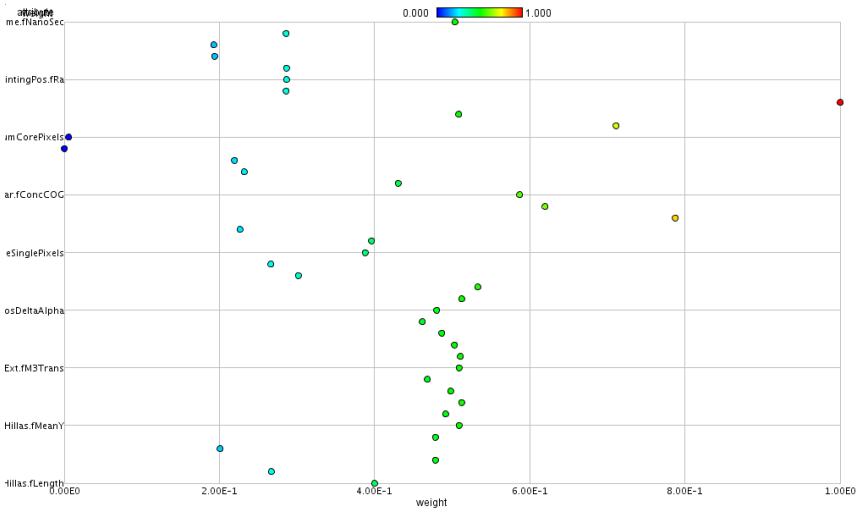



**Abbildung 6.9:** Beipiel 2: Gewichtung

### 6.1.4 Beispiel 3

Das Beispiel 3 stellt eine weitere Verarbeitungsmöglichkeit der eingelesenen ROOT-Daten dar. Es wird eine weitere Funktionalität des Read ROOT Operators dargestellt.

**Datensatz** Der Datensatz besteht aus mehreren IceCube-Dateien. IceCube Datensätze besitzen eine wesentlich kleinere Anzahl an Events, bestehen aber aus mehr Parametern als die MAGIC-Datensätze.

- Datei 1: GA_za00to30_9_903001_Q_w0.root
  Datei 2: GA_za00to30_9_903002_Q_w0.root
  Datei 3: GA_za00to30_9_903003_Q_w0.root
  Datei 4: GA_za00to30_9_903004_Q_w0.root
  Datei 5: GA_za00to30_9_903005_Q_w0.root

- Ausgewählte Daten: 228 Daten mit 88 Attribute

  Es wurden Ausschnitte der Attribute und Ereignisse ausgewählt.

**ROOT-Plugin** Der Read ROOT Operator bietet die Möglichkeit, Ereignisse nach dem Einlesen zu löschen. In Abbildung 6.10 werden die nicht benötigten Daten entfernt. Der Benutzer kann ein Intervall auswählen, alle anderen Daten werden gelöscht. Zudem kann der Benutzer Ereignisse über ihre Id selektieren. Es besteht die Möglichkeit, alle geraden, ungeraden oder jedes X-te Ereignis zu entfernen.

In diesem Beispiel wurde für die Verarbeitung der ROOT-Daten der „Agglomerative Clustering Operator“ und „Flatten Cluster Model Operator“ [29] verwendet (siehe

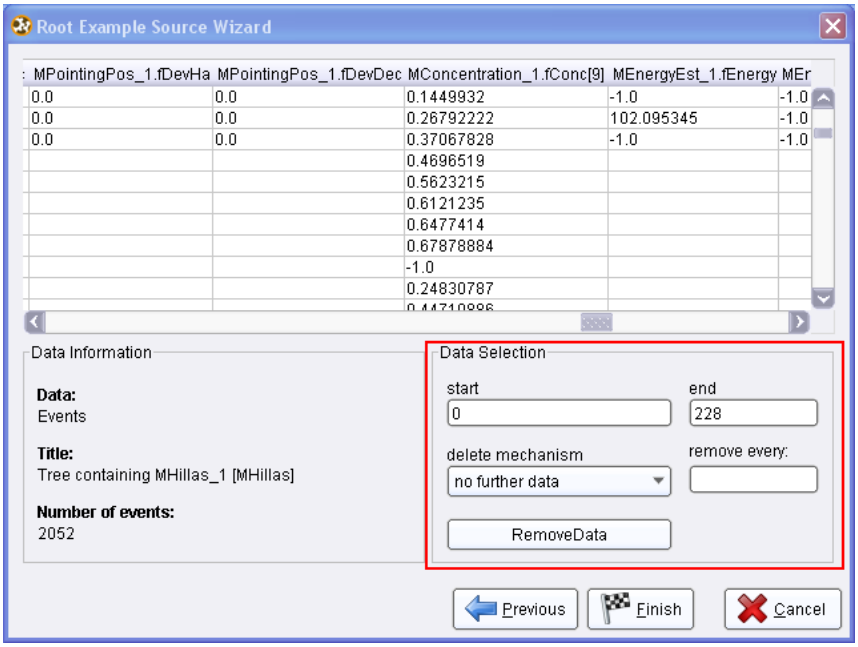



**Abbildung 6.10:** Beipiel 3: Lösch-Funktion

Abbildung 6.11). Das Agglomerative Clustering stellt drei Agglomerative Clusterverfahren bereit: SingleLink, CompleteLink und AverageLink. In dem Experiment wurde das AverageLink-Verfahren eingesetzt. Es bestimmt das durchschnittliche Distanzmaße zwischen zwei Clustern. Als Ausgabe liefert der Operator ein hierarchisches Clustermodell. Der Flatten Cluster Model Operator benutzt das hierarchische Clustermodell und erzeugt ein flaches Clustermodell, indem die Knoten nach ihren Abstände ausgewählt werden, bis die gewünschte Anzahl an Clustern erreicht wird.

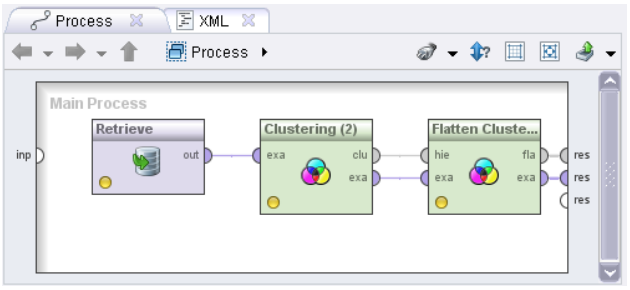



**Abbildung 6.11:** Beipiel 3: Agglomerative Clustering

**Ergebnisse** Als Resultat ergibt sich für das Hierarchische Cluster-Modell eine Anzahl von 455 Clustern und 228 Einheiten. Diese wurden durch das Flatten Cluster Model zu drei Clustern weiter verarbeitet.

Die drei Cluster besitzen folgende Anzahl an Einträgen:

- Cluster 0: 13 Einträge
- Cluster 1: 1 Einträge
- Cluster 2: 214 Einträge

RapidMiner stellt verschiedene Möglichkeiten bereit sich Clusteringergebnisse anzeigen zu lassen. Dazu gehören eine Textdarstellung, die eine Art Übersicht darstellt. Eine Ordner-Ansicht, welche die Cluster als Ordner mit den entsprechenden Einträgen enthält. Zudem besteht die Möglichkeit der grafischen Darstellung. Neben der Dartellung der Cluster als Graphen, besteht die Möglichkeit einen Plot der Parameter in Abhängigkeit vom Cluster erstellen zu lassen. In Abbildung 6.12 sind die beiden Attribute „Size“ und „Energy“ mit der ClusterID als Punktfarbe dargestellt.

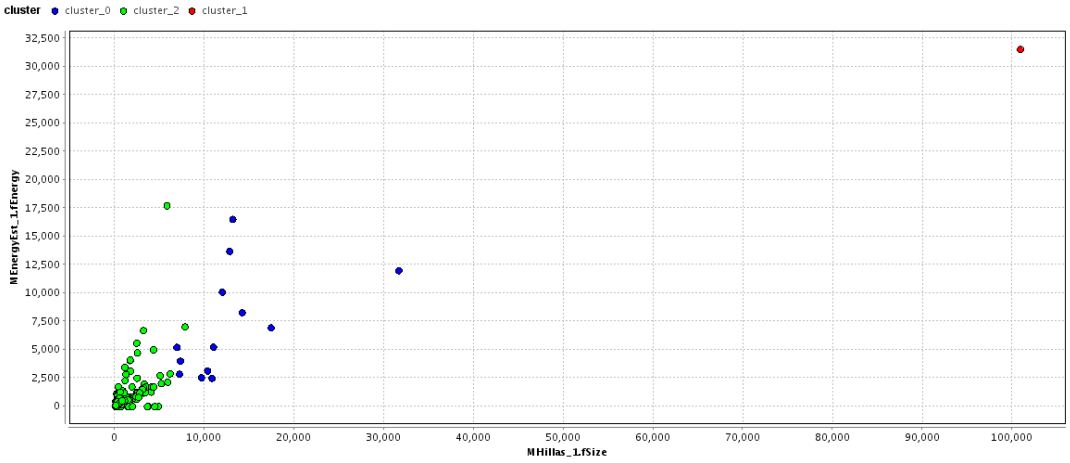



**Abbildung 6.12:** Beipiel 3: Zusammenhänge zweier Attribute bezüglich ihrer Cluster

In der Abbildung 6.12 ist zu erkennen, dass sich die Cluster unterschiedlich zum Ursprung des Koorsinatensystems ausrichten. Die Werte des Cluster 2(grün) befindet sich in der Nähe des Ursprungs. Die Werte des Cluster 0 (blau) heben sich dagegen vom Urspurngsort ab. Der Wert des Clusters 1(rot) befindet sich deutlich zu den beiden anderen Clustern abgesetzt.

# Kapitel 7

# Zusammenfassung

In diesem Kapitel wird die Arbeit in kurzer Form zusammengefasst. Das Ergebnis dieser Arbeit wird betrachtet und diskutiert. Anschließend wird ein Überblick über mögliche Erweiterungen gegeben.

## 7.1 Zusammenfassung der Ergebnisse

Im Rahmen dieser Arbeit wurde ein System entwickelt und implementiert, welches es ermöglicht, das ROOT-Dateiformat mit Hilfe von XML zu beschreiben und zu verarbeiten. Dafür wurde ein Format für das Beschreiben der ROOT-Dateien erstellt und in einer beschreibenden XML-Datei verwendet. Die beschreibende XML-Datei definiert sowohl die Grundlagen der ROOT-Dateien als auch zahlreiche Objekte von ROOT.

Für die Definition der beschreibenden XML-Datei wurde ein Schema erstellt, welches das Grundgerüst und den Inhalt dieser XML-Datei beschreibt.

Für eine Vereinfachung der weiteren Beschreibung von ROOT-Objekten wurde ein Konverter für benutzerdefinierte Objekte (TStreamerInfos) implementiert. Er ermöglicht das Abbilden der zu einer ROOT-Datei gehörigen TStreamerInfos in das beschreibende Format der beschreibenden XML-Datei.

Das Ausgabeformat des XML-Konverters ist XML und bietet dadurch eine Schnittstelle, die eine vielseitige Verarbeitung der Daten ermöglicht. Ausserdem wurde eine XSLT-Schnittstelle bereitgestellt, welche die Umwandlung von XML-Daten in andere Formate erlaubt. Ein XSLT-Stylesheet wurde definiert, das die Umwandlung von kleineren Ziel-XML-Dateien in ein von RapidMiner benötigtes Format ermöglicht.

Für die Umwandlung der gelesenen Bytes der ROOT-Datei in das passende Format wurde ein Schema erarbeitet, das den Aufbau der Typen der in ROOT-Dateien enthaltenen Informationen definiert.

Zudem wurde eine Schnittstelle implementiert, die über einen definierten Webservice die ausgelesenen ROOT-Daten mit Hilfe von XML umwandelt und für die weitere Verarbeitung im benötigten Format des RapidMiner zur Verfügung stellt.

Dem Benutzer von RapidMiner wurde ein „Read ROOT“ Operator für das Lesen von ROOT-Dateien bereit gestellt, der die Funktionalität des XML-Konverters nutzt. Der Operator ermöglicht eine Auswahl an Daten innerhalb der ROOT-Datei sowie eine Datenselektion innerhalb eines Datensatzes und der ausgelesenen Daten über den Webservice.

## 7.2 Disskussion

Der Vorteil des XML-Konverters besteht in der Anpassungsfähigkeit an neue Gegebenheiten bezüglich des ROOT-Formats. Noch nicht in XML beschriebene ROOT-Objekte oder Veränderungen im ROOT-Format können durch die XML-Beschreibung erweitert werden. Somit ist eine hohe Flexibilität des Systems gegeben.

Das Beschreiben der Objekte erfordert, bedingt durch den komplizierten Aufbau der ROOT-Dateien, etwas Übung. Die Schwierigkeiten bestehen aus der sehr verteilten und nicht vollständigen Dokumentation des ROOT-Formats auf der binären Ebene. Der TStreamerInfoModeler (Kapitel 5.6) kann auf Grund vieler Spezialfälle die Arbeit der automatischen Objektdefinition nicht komplett übernehmen, gibt aber durch die Umwandlung der TStreamerInfos ein Grundgerüst wieder, das mit wenig Aufwand angepasst werden kann.

Durch XML als Austauschformat kann die Schnittstelle nicht nur für RapidMiner benutzt werden, sondern erlaubt jedem System, das XML-Daten verarbeiten kann, diese zu nutzen. Zudem können die Daten mit Hilfe des XSLT-Transformationsprozesses in viele weitere Formate umgewandelt werden. Dies kann durch unterschiedliche Stylesheets realisiert werden. Eine Möglichkeit besteht zum Beispiel im Anzeigen von ROOT-Daten als Webseite.

XML und XSLT besitzten neben den Vorteilen auch Nachteile. Bedingt durch den Aufbau des XML-Formats als Human Readable Format benötigt die XML Syntax, d. h. insbersondere die öffnenden und schließenden XML-Tags, einen erhöhten Speicherplatzbedarf. Das implementierte System reduziert die Größe erheblich, indem es nur die Baskets der ROOT-Datei versendet.

Durch die Verarbeitung der XML-Daten mit Hilfe von Document Object Model entsteht auf der Seite des Clients die Problematik des Speicherplatzbedars bei großen Dateien. Die genaue Größe ist dabei abhängig von der Anzahl der enthaltenen Messdaten, der Anzahl der zusammen gelesenen Dateien und die dadurch verbundenen zusätzlichen gespeicherten Informationen. Bei sehr großen Dateien kommt es vor, dass sie den Heap-Speicher der Clientmaschine voll ausschöpfen. Mögliche Auswege der Problematik werden im Kapitel 7.3.1 aufgezeigt.

Ein schon im Kapitel 5.5.3 angesprochener Nachteil von XSLT besteht in der erhöhten CPU- und Speicherauslastung bei Transformationsprozessen. Diese ist abhängig vom Stylesheet und der Größe der umzuwandelnden XML-Datei. Durch das Umwandeln der Daten von Datenblöcken zu ereignisabhängigen Daten kommt es bei dem verwendetem Stylesheet zu einer hohen Anzahl an Durchläufen und damit zu einem starken CPU- und Speicherplatzbedarf. Aus diesem Grund wurde eine Möglichkeit der Verarbeitung ohne XSLT bereit gestellt. Auch wenn das XSLT-Prinzip in diesem speziellen Fall eine schlechte Performance besitzt, bietet es grundsätzlich eine erhöhte Flexibilität und schafft neue Möglichkeiten der Datentransformation.

## 7.3 Ausblick

Im Verlauf dieser Arbeit entstanden verschiedene Ideen hinsichtlich der Verbesserung und Erweiterungen des Systems. Im Folgenden werden einige Ideen vorgestellt und Erweiterungsmöglichkeiten beschrieben.

### 7.3.1 Verbesserungen

Der Hauptansatzpunkt besteht in der Optimierung des Verarbeitungsprozesses der Daten auf der Clientseite. Um der Problematik des Speicherplatzbedarfs bei großen Datenmengen zu begegnen, müssten die Daten auf der Festplatte des Clients zwischengespeichert werden. Diese Daten könnten dann in RapidMiner als Datenquelle benutzt werden. Es ist allerdings zu erwarten, dass die Performance des Systems auf Grund der längeren Zugriffszeiten beeinträchtigt wird.

### 7.3.2 Weitere Möglichkeiten

Nach dem Auslesen der Daten aus der ROOT-Datei werden die Inhalte und die vollständige Struktur der Datei im XML-Format repräsentiert. Dies kann die Grundlage für eine weitere Nutzung des Ergebnisses des Umwandlungsprozesses sein. Durch die Struktur können XML-Dateien wieder in das ROOT-Format zurück gewandelt werden. Es können weitere Daten hinzugefügt werden, dabei müssen die entsprechenden Positionen nicht nur durch die neuen Daten erweitert werden, sondern auch Steuerelemente angepasst werden, die Informationen über den Aufbau der Datei enthalten.

### 7.3.3 Erweiterungen

Es gibt zwei Möglichkeiten, das System zu erweitern. Die Erste besteht in der Erweiterung der Ausgabemöglichkeiten mittels XSLT. Wie im Kapitel 7.2 beschrieben, kann es verschiedene Ausgabeformate wie zum Beispiel XHTML geben.

Die bedeutendere Erweiterungsmöglichkeit ist die Erweiterung der Beschreibung des ROOT-Formats. Durch die Benutzung einer Beschreibung der zu lesenden ROOT-Objekte mit Hilfe der XML-Schnittstelle ist es möglich, das System durch einfaches Definieren von Objekten in XML zu erweitern. Bis dato sind verschiedene Hauptobjekte von ROOT umgesetzt worden, dazu gehören die Datensätze TFile-, TStreamerInfos-, TKeysList-, TBasket- und TTree-Datensätzen. Ausserdem wurden die bisher vorkommenden Unterobjekte definiert. Um Datensätze anderen Typs oder neue Objekte innerhalb der schon beschriebenen Datensätze lesen zu können, müssen diese in der beschreibenden XML-Datei erfasst und definiert sein.

Dabei wird zwischen Datensatzobjekten und normalen ROOT-Objekten unterschieden. Normale Objekte müssen nur in der beschreibenden XML-Datei hinzugefügt werden. Sonderfälle, die vom bisherigen Verhalten abweichen, müssen unter Umständen von der Methode *setTArrayContentInt()* abgebildet werden. Unter abweichendem Verhalten wird ein Verhalten verstanden, das nicht alleine durch eine XML-Beschreibung abgebildet werden kann, sondern stark durch ROOT beeinflusst wird (siehe Kapitel 5.4.1).

Die Definition von neuen Datensatzobjekten wird ebenfalls in der beschreibenden XML-Datei vorgenommen. Sollten diese neuen Datensätze für den Benutzer interessante Daten enthalten, die aus dem Ziel-XML-Dokument extrahiert werden sollen, können Erweiterungen in den entsprechenden Klassen implementiert werden. Hierbei handelt es sich nicht um den Auslesevorgang des XML-Konverters, sondern um Erweiterungen bezüglich der Datenauswertung. Diese Erweiterungen müssen dann auch Clientseitig vorgenommen werden.

Im Rahmen der Diplomarbeit wurden die für die Auslese der MAGIC und IceCube relevanten ROOT-Objekte umgesetzt. Da das ROOT-System dem Benutzer ermöglicht eigene Benutzerobjekte zu definieren, ist das implementierte System explizit auf Flexibilität und Erweiterbarkeit ausgelegt.

# Anhang A

# Aufbau der ROOT-Dateien

Die Definitionen der ROOT-Objekte sind auf der Seite der Conseil Européen pour la Recherche Nucléaire zu finden [7].

## A.1 TKey

```
Format of a TDirectory record in release 3.02.06. It is never compressed.
----------TKey----------
byte 0->3 Nbytes = Number of bytes in compressed record (Tkey+data) | TKey::fNbytes
4->5 Version = TKey class version identifier | TKey::fVersion
6->9 ObjLen = Number of bytes of uncompressed data | TKey::fObjLen
10->13 Datime = Date and time when record was written to file | TKey::fDatime
                |(year-1995)<<26|month<<22|day<<17|hour<<12|minute<<6|second
14->15 KeyLen = Number of bytes in key structure (TKey) |TKey::fKeyLen
16->17 Cycle = Cycle of key | TKey::fCycle
18->21 SeekKey = Byte offset of record itself (consistency check) | TKey::fSeekKey
22->25 SeekPdir = Byte offset of parent directory record | TKey::fSeekPdir
26->26 lname = Number of bytes in the class name (10) | TKey::fClassName
27->.. ClassName = Object Class Name ("TDirectory") | TKey::fClassName
0->0 lname = Number of bytes in the object name | TNamed::fName
1->.. Name = lName bytes with the name of the object <directory name> |
    TNamed::fName
0->0 lTitle = Number of bytes in the object title | TNamed::fTitle
1->.. Title = lTitle bytes with the title of the object <direcory title> |
    TNamed::fTitle
```

**Listing A.1:** ROOT Beschreibung: TKey

## A.2 TDirectory

```
Format of a TDirectory record in release 3.02.06. It is never compressed.

--------DATA--------
0->0 Modified = True if directory has been modified | TDirectory::fModified
1->1 Writeable = True if directory is writeable | TDirectory::fWriteable
2->5 DatimeC = Date and time when directory was created | TDirectory::fDatimeC
          |(year-1995)<<26|month<<22|day<<17|hour<<12|minute<<6|second
```

```
6->9 DatimeM = Date and time when directory was last modified |
    TDirectory::fDatimeM
          |(year-1995)<<26|month<<22|day<<17|hour<<12|minute<<6|second
10->13 NbytesKeys= Number of bytes in the associated KeysList record |
    TDirectory::fNbyteskeys
14->17 NbytesName= Number of bytes in TKey+TNamed at creation |
    TDirectory::fNbytesName
18->21 SeekDir = Byte offset of directory record in file | TDirectory::fSeekDir
22->25 SeekParent= Byte offset of parent directory record in file |
    TDirectory::fSeekParent
26->29 SeekKeys = Byte offset of associated KeysList record in file |
    TDirectory::fSeekKeys
```

**Listing A.2:** ROOT Beschreibung: TDirectory

## A.3 TFile

```
Here is the format of a TFile record for release 3.02.06. It is never compressed.
It is probably not accessed by its key, but from its offset given in the file
    header.

--------DATA----------------
0->0 lname = Number of bytes in the TFile name | TNamed::fName
1->.. Name = lName bytes with the name of the TFile <file name> | TNamed::fName
0->0 lTitle = Number of bytes in the TFile title | TNamed::fTitle
1->.. Title = lTitle bytes with the title of the TFile <file title> |
    TNamed::fTitle
0->0 Modified = True if directory has been modified | TDirectory::fModified
1->1 Writeable = True if directory is writeable | TDirectory::fWriteable
2->5 DatimeC = Date and time when directory was created | TDirectory::fDatimeC
          |(year-1995)<<26|month<<22|day<<17|hour<<12|minute<<6|second
6->9 DatimeM = Date and time when directory was last modified |
    TDirectory::fDatimeM
          |(year-1995)<<26|month<<22|day<<17|hour<<12|minute<<6|second
10->13 NbytesKeys= Number of bytes in the associated KeysList record |
    TDirectory::fNbyteskeys
14->17 NbytesName= Number of bytes in TKey+TNamed at creation |
    TDirectory::fNbytesName
18->21 SeekDir = Byte offset of directory record in file (64) |
    TDirectory::fSeekDir
22->25 SeekParent= Byte offset of parent directory record in file (0) |
    TDirectory::fSeekParent
26->29 SeekKeys = Byte offset of associated KeysList record in file |
    TDirectory::fSeekKeys
```

**Listing A.3:** ROOT Beschreibung: TFile

## A.4 Keys List Record Format

```
Format of KeysList record in release 3.02.06. It is never compressed. There is one
    KeysList record for the main (TFile) directory and one per non-empty
    subdirectory. It is probably not accessed by its key, but from its offset given
     in the directory data.
```

```
----------DATA----------
0->3 NKeys = Number of keys in list (i.e. records in directory (nonrecursive)) |
    Excluded:: The directory itself, KeysList, StreamerInfo, and FreeSegments
4->.. TKey = Sequentially for each record in directory, the entire TKey portion of
    each record is replicated. Note that SeekKey locates the record.
```

**Listing A.4:** ROOT Beschreibung: KeysList

## A.5 FreeSegments Record Format

```
Format of FreeSegments record, release 3.02.06. It is never compressed. It is
    probably not accessed by its key, but from its offset given in the file header.

----------DATA----------
0->1  Version   = TFree class version identifier | TFree::ClassVersion()
2->5  fFirst    = First free byte of first free segment |   TFree::fFirst
6->9  fLast     = Byte after last free byte of first free segment |   TFree::fLast
     Sequentially, Version, fFirst and fLast of additional free segments. There is
         always one free segment beginning at file end and ending before
         2000000000.
```

**Listing A.5:** ROOT Beschreibung: FreeSegments

## A.6 StreamerInfo Record Format

```
Format of StreamerInfo record in release 3.02.06. It is probably not accessed by
    its key, but from its offset given in the file header. The StreamerInfo record
    DATA consists of a TList (list) object containing elements of class
    TStreamerInfo.

----------TList-(always compressed at level 1 (even if compression level 0))-----
The DATA is a TList collection object containing TStreamerInfo objects. Below is
    the format of this TList data.
Here is the format of a TList object in Release 3.02.06. Comments and offsets refer
     specifically to its use in the StreamerInfo record.

0->3  ByteCount = Number of remaining bytes in TList object (uncompressed) OR'd
    with kByteCountMask (0x40000000)
4->5  Version   = Version of TList Class
6->15           = TObject object (a base class of TList) (see tobject.txt). Objects
     in StreamerInfo record are not referenced. Would be two bytes longer (6->17)
    if object were referenced.
16->16 fName    = Number of bytes in name of TList object, followed by the name
    itself. (TCollection::fName). The TList object in StreamerInfo record is
    unnamed, so byte contains 0.
17->20 nObjects = Number of objects in list.
21->.. objects  = Sequentially, TStreamerInfo Objects in the list. In the
    StreamerInfo record, the objects in the list are TStreamerInfo objects. There
    will be one TStreamerInfo object for every class used in data records other
    than core records and the the StreamerInfo record itself.

----------TStreamerInfo object----------
Here is the format of a TStreamerInfo object in Release 3.02.06.
Note: Although TStreamerInfo does not use the default streamer, it has the same
    format as if it did. (compare with dobject.txt)
```

```
0->3  ByteCount = Number of remaining bytes in TStreamerInfo object (uncompressed)
    OR'd with kByteCountMask (0x40000000)
4->.. ClassInfo = Information about TStreamerInfo class. If this is the first
    occurrence of a TStreamerInfo object in the record
4->7  -1        = New class tag (constant kNewClassTag = 0xffffffff)
8->21 Classname = Object Class Name "`TStreamerInfo"' (null terminated)
Otherwise
4->7 clIdx      = Byte offset of new class tag in record, plus 2. OR'd with
    kClassMask (0x80000000)
0->3  ByteCount = Number of remaining bytes in TStreamerInfo object (uncompressed)
    OR'd with kByteCountMask (0x40000000)
4->5  Version   = Version of TStreamerInfo Class. Begin TNamed object (Base class
    of TStreamerInfo)
6->9  ByteCount = Number of remaining bytes in TNamed object OR'd with
    kByteCountMask (0x40000000)
10->11 Version  = Version of TNamed Class
12->21          = TObject object (Base class of TNamed) (see tobject.txt). Objects
     in StreamerInfo record are not referenced. Would be two bytes longer (12->23)
    if object were referenced.
22->.. fName    = Number of bytes in name of class that this TStreamerInfo object
    describes, followed by the class name itself. (TNamed::fName).
0->.. fTitle    = Number of bytes in title of class that this TStreamerInfo object
    describes, followed by the class title itself. (TNamed::fTitle). (Class title
    may be zero length) -End TNamed object
0->3  fCheckSum = Check sum for class that this TStreamerInfo object describes.
    This checksum is over all base classes and all persistent non-static data
    members. It is computed by TClass::GetCheckSum(). (TStreamerInfo::fCheckSum)
4->7  fClassVersion = Version of class that this TStreamerInfo object describes. (
    TStreamerInfo::fClassVersion). Begin TObjArray object (Data member of
    TStreamerInfo)
0->3  ByteCount = Number of remaining bytes in TObjArray object (uncompressed) OR'd
     with kByteCountMask (0x40000000)
4->.. ClassInfo = Information about TObjArray class. If this is the first
    occurrence of a TObjArray object in the record
4->7  -1        = New class tag (constant kNewClassTag = 0xffffffff)
8->17 Classname = Object Class Name "`TObjArray"' (null terminated)
Otherwise
4->7 clIdx      = Byte offset of new class tag in record, plus 2. OR'd with
    kClassMask (0x80000000)
0->3  ByteCount = Number of remaining bytes in TObjArray object (uncompressed) OR'd
     with kByteCountMask (0x40000000)
4->5  Version   = Version of TObjArray Class
6->15           = TObject object (a base class of TObjArray) (see tobject.txt).
    Objects in StreamerInfo record are not referenced. Would be two bytes longer
    (6->17) if object were referenced.
16->16 fName    = Number of bytes in name of TObjArray object, followed by the
    name itself. (TCollection::fName). TObjArray objects in StreamerInfo record
    are unnamed, so byte contains 0.
17->20 nObjects = Number of objects (derived from TStreamerElement) in array.
21->24 fLowerBound = Lower bound of array. Will always be 0 in StreamerInfo record
    .
25->.. objects  = Sequentially, TStreamerElement objects in the array. In a
    TStreamerInfo object, the objects in the TObjArray are of various types (
    described below), all of which inherit directly from TStreamerElement objects.
     There will be one such object for every base class of the class that the
```

```
    TStreamerInfo object describes, and also one such object for each persistent
    non-static data member of the class that the TStreamerInfo object describes.
    End TObjArray object and TStreamerInfo object

The objects stored in the TObjectArray in TStreamerInfo are of various classes,
    each of which inherits directly from the TStreamerElement class. The possible
    classes (which we refer to collectively as TStreamer<XXX>) are:
TStreamerBase:          Used for a base class. All others below used for data
    members.
TStreamerBasicType:     For a basic type
TStreamerString:        For type TString
TStreamerBasicPointer:  For pointer to array of basic types
TStreamerObject:        For an object derived from TObject
TStreamerObjectPointer: For pointer to an object derived from TObject
TStreamerLoop:          For pointer to an array of objects
TStreamerObjectAny:     For an object not derived from TObject
TStreamerSTL:           For an STL container (not yet used??)
TStreamerSTLString:     For an STL string (not yet used??)

Here is the format of a TStreamer<XXX> object in Release 3.02.06. In description
    below,
0->3  ByteCount = Number of remaining bytes in TStreamer<XXX> object (uncompressed)
     OR'd with kByteCountMask (0x40000000)
4->.. ClassInfo = Information about the specific TStreamer<XXX> class. If this is
    the first occurrence of a TStreamerXXX object in the record
4->7  -1        = New class tag (constant kNewClassTag = 0xffffffff)
8->.. Classname = Object Class Name "`TStreamer<XXX>"' (null terminated)
Otherwise
4->7 clIdx      = Byte offset of new class tag in record, plus 2. OR'd with
    kClassMask (0x80000000)
0->3  ByteCount = Number of remaining bytes in TStreamer<XXX> object (uncompressed)
    . OR'd with kByteCountMask (0x40000000)
4->5  Version   = Version of TStreamer<XXX> Class. Begin TStreamerElement object (
    Base class of TStreamerXXX)
0->3  ByteCount = Number of remaining bytes in TStreamerElement object (
    uncompressed) OR'd with kByteCountMask (0x40000000)
4->5  Version   = Version of TStreamerElement Class. Begin TNamed object (Base
    class of TStreamerElement)
6->9  ByteCount = Number of remaining bytes in TNamed object OR'd with
    kByteCountMask (0x40000000)
10->11 Version  = Version of TNamed Class
12->21          = TObject object (Base class of TNamed) (see tobject.txt). Objects
     in StreamerInfo record are not referenced. Would be two bytes longer (12->23)
    if object were referenced.
22->.. fName    = Number of bytes in class name of base class or member name of
    data member that this TStreamerElement object describes, followed by the name
    itself. (TNamed::fName).
0->.. fTitle    = Number of bytes in title of base class or data member that this
    TStreamerElement object describes, followed by the title itself. (
    TNamed::fTitle). End TNamed object
0->3  fType     = Type of data described by this TStreamerElement. (
    TStreamerElement::fType)
     |   Built in types:
```

```
     |   1:char, 2:short, 3:int, 4:long, 5:float, 8:double, 11, 12, 13, 14:unsigned
          char, short, int, long respectively, 6: an array dimension (counter), 15:
          bit mask (used for fBits field)

     |   Pointers to built in types:
     |   40 + fType of built in type (e.g. 43: pointer to int)

     |   Objects:
     |   65:TString, 66:TObject, 67:TNamed
     |   0: base class (other than TObject or TNamed)
     |   61: object data member derived from TObject (other than TObject or TNamed)
     |   62: object data member not derived from TObject
     |   63: pointer to object derived from TObject (pointer can't be null)
     |   64: pointer to object derived from TObject (pointer may be null)
     |   501: pointer to an array of objects
     |   500: an STL string or container

     |   Arrays:
     |   20 + fType of array element (e.g. 23: array of int)

4->7  fSize     = Size of built in type or of pointer to built in type. 0 otherwise
    . (TStreamerElement::fSize).
8->11 fArrayLength = Size of array (0 if not array). (
    TStreamerElement::fArrayLength).
12->15 fArrayDim = Number of dimensions of array (0 if not an array). (
    TStreamerElement::fArrayDim).
16->35 fMaxIndex = Five integers giving the array dimensions (0 if not applicable).
     (TStreamerElement::fMaxIndex).
36->.. fTypeName = Number of bytes in name of the data type of the data member that
     the TStreamerElement object describes, followed by the name itself.  If this
    TStreamerElement object defines a base class rather than a data member, the
    name used is 'BASE'. (TStreamerElement::fTypeName). End TStreamerElement object
     The remaining data is specific to the type of TStreamer<XXX> class.\\

For TStreamerInfoBase:
0->3  fBaseVersion   = Version of base class that this TStreamerElement describes.
For TStreamerBasicType:
     No specific data For TStreamerString:
     No specific data For TStreamerBasicPointer:\
0->3  fCountVersion = Version of class with the count (array dimension)
4->.. fCountName= Number of bytes in the name of the data member holding the count,
     followed by the name itself.
0->.. fCountName= Number of bytes in the name of the class holding the count,
    followed by the name itself.

For TStreamerObject:
     No specific data For TStreamerObjectPointer:
     No specific data For TStreamerLoop:
0->3  fCountVersion = Version of class with the count (array dimension)
4->.. fCountName= Number of bytes in the name of the data member holding the count,
     followed by the name itself.
0->.. fCountClass= Number of bytes in the name of the class holding the count,
    followed by the name itself.

For TStreamerObjectAny:
```

```
     No specific data
     For TStreamerSTL:
0->3  fSTLtype  = Type of STL container:
     1:vector, 2:list, 3:deque, 4:map, 5:set, 6:multimap, 7:multiset
4->7  fCType    = Type contained in STL container: Same values as for fType above,
    with one addition: 365:STL string

For TStreamerSTLString:
     No specific data
```

**Listing A.6:** ROOT Beschreibung: StreamerInfo

## A.7 TProcessID Record Format

```
Format of TProcessID record in release 3.02.06. Will be present if there are any
    referenced objects.

----------DATA----------
0->3 ByteCount = Number of remaining bytes in TProcessID object (uncompressed). OR'
    d with kByteCountMask (0x40000000)
4->5 Version   = Version of TProcessID Class. Begin TNamed object (Base class of
    TProcessID)
6->9 ByteCount = Number of remaining bytes in TNamed object (uncompressed). OR'd
    with kByteCountMask (0x40000000)
10->11 Version  = Version of TNamed Class
12->21          = TObject object (Base class of TNamed) (see tobject.txt). The
    TProcessID object is not itself referenced.
22->22 lname    = Number of bytes in the object name | TNamed::fName
23->.. Name     = lName bytes with the name of the object | TNamed::fName
     The name will be "`ProcessID"' concatenated with a decimal integer, or "`pidf"
         '.
0->0  lTitle   = Number of bytes in the object title | TNamed::fTitle
1->.. Title    = lTitle bytes with the title of the object | TNamed::fTitle
     (Identifies processor, time stamp, etc.) See detailed explanation below. End
         TNamed object

----------Explanation of the title of a TProcessID object----------
The title of a TProcessID object is a globally unique identifier of the ROOTIO
    process that created it.  It is derived from the following quantities.

1) The creation time ("`fTime"') of the TProcessID record.  This is a 60 bit time
    in 100ns ticks since Oct. 15, 1582.

2) A 16 bit random unsigned integer ("`clockeq"') generated from a seed that is the
     job's process ID.  The highest two bits are not used.

3) A six byte unsigned quantity ("`fNode"') identifying the machine.  If the
    machine has a valid network address, the first four bytes are set to that
    address, and the last two bytes are stuffed with 0xbe and 0xef respectively.
    Otherwise a six byte quantity is generated from the time and random machine
    statistics. In this case, the high order bit of the first byte is set to 1, to
    distinguish it from a network ID, where the bytes can be no larger than 255.

We the define the following quantities (class TUUID):
     UInt_t    fTimeLow;               // 60 bit time, lowest 32 bits
```

```
    UShort_t  fTimeMid;                // 60 bit time, middle 16 bits
    UShort_t  fTimeHiAndVersion;       // 60 bit time, highest 12 time bits (low 12
        bits)
                                      // + 4 UUID version bits (high 4 bits)
                                          version is 1 if machine has valid network
                                           address and 3 otherwise.
    UChar_t   fClockSeqHiAndReserved; // high 6 clockseq bits (low 6 bits) + 2
       high bits reserved (currently set to binary 10)
    UChar_t   fClockSeqLow;            // low 8 clockseq bits
    UChar_t   fNode[6];                // 6 node (machine) id bytes

Then the following sprintf() call defines the format of the title string:
    sprintf(Title, "`%08x-%04x-%04x-%02x%02x-%02x%02x%02x%02x%02x%02x"',
                   fTimeLow, fTimeMid, fTimeHiAndVersion, fClockSeqHiAndReserved,
                   fClockSeqLow, fNode[0], fNode[1], fNode[2], fNode[3], fNode[4],
        fNode[5]);
Since the title written to disk is preceded by its byte count, the delimiting null
   is not written.
```

**Listing A.7:** ROOT Beschreibung: TProcessID

## A.8 TRef Format

```
Here is the format of the DATA for a TRef object in Release 3.02.06.

0->1  Version   = Version of TObject Class (base class of TRef)
2->5  fUniqueID = Unique ID of referenced object.  Typically, every referenced
    object has an ID that is a positive integer set to a counter of the number of
    referenced objects in the file, beginning at 1. fUniqueID in the TRef object
    matches fUniqueID in the referenced object.
6->9  fBits     = A 32 bit mask containing status bits for the TRef object. The
    bits relevant to ROOTIO are:
     | 0x00000008 - Other objects may need to be deleted when this one is.\\
     | 0x00000010 - Object is referenced by pointer to persistent object.
     | 0x01000000 - Object is on Heap.
     | 0x02000000 - Object has not been deleted.
10->11 pidf     = An identifier of the TProcessID record for the process that wrote
     the referenced object. This identifier is an unsigned short.  The relevant
    record has a name that is the string "`ProcessID"' concatenated with the ASCII
    decimal representation of "`pidf"' (no leading zeros). 0 is a valid pidf.
```

**Listing A.8:** ROOT Beschreibung: TRef

## A.9 TRefArray Format

```
Here is the format of the DATA for a TRefArray object in Release 3.02.06.

0->3  ByteCount = Number of remaining bytes in TRefArray object (uncompressed) OR'd
     with kByteCountMask (0x40000000)
4->5  Version   = Version of TRefArray Class
6->15           = TObject object (Base class of TRefArray) (see tobject.txt). Will
    be two bytes longer (6->17) if TRefArray object is itself referenced (unlikely)
    .
```

```
16->.. fName      = Number of bytes in name of TRefArray object, followed by the
    name itself. (TCollection::fName). Currently, TRefArrays are not named, so
    this is a single byte containing 0.
0->3  nObjects  | Number of object references (fUIDs) in this TRefArray.
4->7  fLowerBound= Lower bound of array. Typically 0.
8->9  pidf      = An identifier of the TProcessID record for the process that wrote
     the referenced objects. This identifier is an unsigned short. The relevant
    record has a name that is the string "`ProcessID"' concatenated with the ASCII
    decimal representation of "`pid"' (no leading zeros). 0 is a valid pidf.
10->.. fUIDs      = Sequentially, object Unique ID's. Each Unique ID is a four byte
    unsigned integer. If non-zero, it matches the Unique ID in the referenced
    object. If zero, it is an unused element in the array. The fUIDs are written
    out only up to the last used element, so the last fUID will always be non-zero.
```

**Listing A.9:** ROOT Beschreibung: TRefArray

## A.10 TClonesArray

```
Here is the format (release 3.02.06) of the DATA for a TClonesArray object in a
    ROOTIO file.
0->3  ByteCount = Number of remaining bytes in TClonesArray object (uncompressed)
    OR'd with kByteCountMask (0x40000000)
4->.. ClassInfo = Information about TClonesArray class. If this is the first
    occurrence of a TClonesArray object in the record
4->7  -1        = New class tag (constant kNewClassTag = 0xffffffff)
8->17 Classname = Object Class Name "TClonesArray" (null terminated)
Otherwise
4->7 clIdx      = Byte offset of new class tag in record, plus 2. OR'd with
    kClassMask (0x80000000)
0->3  ByteCount = Number of remaining bytes in TClonesArray object (uncompressed).
    OR'd with kByteCountMask (0x40000000)
4->5  Version   = Version of TClonesArray Class
6->15           = TObject object (a base class of TClonesArray) (see tobject.txt).
    Would be two bytes longer (6->17) if object were referenced.
16->.. fName     = Number of bytes in name of TClonesArray object, followed by the
    name itself. (TCollection::fName). This name will be the class name of the
    cloned object, appended with an 's' (e.g. "TXxxs")
0->..           = Number of bytes in name and version of the cloned class, followed
     by the name and version themselves (e.g. "TXxx;1")
0->3  nObjects  = Number of objects in clones array.
4->7  fLowerBound= Lower bound of clones array.
8->.. objects   = Sequentially, objects in the clones array. However, the data
    ordering depends on whether or not kBypassStreamer (0x1000) is set in
    TObject::fBits. By default, it is set. If it is not set, the objects are
    streamed sequentially using the streamer of the cloned class (e.g.
    TXxx::Streamer()).

If it is set, the cloned class is split into its base classes and persistent data
    members, and those streamers are used. So, if the base classes and persistent
    data members of class TXxx are TXxxbase, TXxxdata0, TXxxdata1, etc., all the
    TXxxbase data from the entire clones array is streamed first, followed by all
    the TXxxdata0 data, etc. This breakdown is not recursive, in that the member
    objects are not again split. End TClonesArray object
```

**Listing A.10:** ROOT Beschreibung: TClonesArray

# Anhang B

# Metadaten der Meta-Metadaten

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:annotation>
  <xs:documentation>Allgemeines Schema fuer ROOT Dateien. Beschreibt den Aufbau von
      ROOT-Dateien.</xs:documentation>
 </xs:annotation>

<!-- allgemeine Definition RootFile -->
 <xs:element name="RootFile" type="RootFileTyp"/>
  <xs:complexType name="RootFileTyp">
   <xs:sequence>
   <!-- Der ROOTFileHeader muss genau 1 mal vorkommen -->
    <xs:element maxOccurs="1" minOccurs="1" name="RootHeader" type="RootHeaderTyp"/
        >
    <!-- Der ROOTBody kann vorkommen darf aber nur 1 mal vorkommen-->
    <xs:element maxOccurs="1" name="Root" type="RootTyp"/>
  </xs:sequence>
 </xs:complexType>

<!--Condition Spezifikation fuer Sprungadressen -->
 <xs:attributeGroup name="ByteCondition">
  <xs:attribute name="condition" type="xs:string"/>
  <xs:attribute fixed="4" name="true" type="xs:int"/>
  <xs:attribute fixed="8" name="false" type="xs:int"/>
 </xs:attributeGroup>

<!-- Definition ROOT Header -->
 <xs:complexType name="RootHeaderTyp">
 <!-- müssen in der reihenfolge vorkommen (sequence)-->
  <xs:sequence>
   <xs:element fixed="4" maxOccurs="1" minOccurs="1" name="RootFileIdentifier" type
       ="xs:int"/>
   <xs:element fixed="4" maxOccurs="1" minOccurs="1" name="FVersion" type="xs:int"/
       >
   <xs:element fixed="4" maxOccurs="1" minOccurs="1" name="FBegin" type="xs:int"/>
   <!-- Elemente ist entweder 4 oder 8 Bytes groß -->
   <xs:element maxOccurs="1" minOccurs="1" name="FEnd">
    <xs:complexType>
```

```
    <xs:attributeGroup ref="ByteCondition"/>
   </xs:complexType>
  </xs:element>
  <xs:element maxOccurs="1" minOccurs="1" name="FSeekFree">
   <xs:complexType>
    <xs:attributeGroup ref="ByteCondition"/>
   </xs:complexType>
  </xs:element>
  <xs:element fixed="4" maxOccurs="1" minOccurs="1" name="FNbytesFree" type="
      xs:int"/>
  <xs:element fixed="4" maxOccurs="1" minOccurs="1" name="NFree" type="xs:int"/>
  <xs:element fixed="4" maxOccurs="1" minOccurs="1" name="FNBytesName" type="
      xs:int"/>
  <xs:element fixed="1" maxOccurs="1" minOccurs="1" name="FUnits" type="xs:int"/>
  <xs:element fixed="4" maxOccurs="1" minOccurs="1" name="FCompress" type="xs:int
      "/>
  <xs:element maxOccurs="1" minOccurs="1" name="FSeekInfo">
   <xs:complexType>
    <xs:attributeGroup ref="ByteCondition"/>
   </xs:complexType>
  </xs:element>
  <xs:element fixed="4" maxOccurs="1" minOccurs="1" name="FNbytesInfo" type="
      xs:int"/>
  <xs:element maxOccurs="1" minOccurs="1" name="FUuid">
   <xs:complexType>
    <xs:attributeGroup ref="ByteCondition"/>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>

<!-- Definition von Root -->
<xs:complexType name="RootTyp">
 <xs:sequence>
  <!-- Tkye kann vorkommen, muss aber nicht, TKey sind die einzelnen Baume-->
  <xs:element maxOccurs="1" minOccurs="0" name="TKey" type="TKeyTyp" />
 </xs:sequence>
</xs:complexType>

<!-- Definition TKey, Baueme und Co -->
<xs:complexType name="TKeyTyp">
 <xs:sequence>
  <!-- TKye besteht aus einem RecordHeader und den Daten-->
  <xs:element maxOccurs="1" minOccurs="1" name="RecordHeader" type="
      RecordHeaderTyp" />
  <xs:element maxOccurs="1" minOccurs="1" name="Data" />
 </xs:sequence>
</xs:complexType>

<!-- Definition des  RecordHeader-->
<xs:complexType name="RecordHeaderTyp">
 <xs:sequence>
  <!-- TKye besteht aus einem RecordHeader und den Daten-->
  <xs:element fixed="4" maxOccurs="1" minOccurs="1" name="Nbytes" type="xs:int"/>
  <xs:element fixed="2" maxOccurs="1" minOccurs="1" name="Version" type="xs:int"/>
```

```
   <xs:element fixed="4" maxOccurs="1" minOccurs="1" name="ObjLen" type="xs:int"/>
   <xs:element fixed="4" maxOccurs="1" minOccurs="1" name="Datime" type="xs:int"/>
   <xs:element fixed="2" maxOccurs="1" minOccurs="1" name="KeyLen" type="xs:int"/>
   <xs:element fixed="2" maxOccurs="1" minOccurs="1" name="Cycle" type="xs:int"/>
   <xs:element maxOccurs="1" minOccurs="1" name="SeekKey">
    <xs:complexType>
     <xs:attributeGroup ref="ByteCondition"/>
    </xs:complexType>
   </xs:element>
   <xs:element maxOccurs="1" minOccurs="1" name="SeekPdir">
    <xs:complexType>
     <xs:attributeGroup ref="ByteCondition"/>
    </xs:complexType>
   </xs:element>
   <xs:element fixed="1" maxOccurs="1" minOccurs="1" name="lnameClass" type="xs:int
       "/>
   <xs:element maxOccurs="1" minOccurs="1" name="ClassName">
    <xs:complexType>
     <xs:attribute name="condition" fixed="lnameClass" type="xs:string"/>
    </xs:complexType>
   </xs:element>
   <xs:element fixed="1" maxOccurs="1" minOccurs="1" name="lname" type="xs:int"/>
   <xs:element maxOccurs="1" minOccurs="1" name="Name">
    <xs:complexType>
     <xs:attribute name="condition" fixed="lname" type="xs:string"/>
    </xs:complexType>
   </xs:element>
   <xs:element fixed="1" maxOccurs="1" minOccurs="1" name="lTitle" type="xs:int"/>
   <xs:element maxOccurs="1" minOccurs="1" name="Title">
    <xs:complexType>
     <xs:attribute name="condition" fixed="lTitle" type="xs:string"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

**Listing B.1:** Metadaten der Meta-Metadaten (Schema)

# Quellenverzeichnis


[1] *Astroteilchenphysik - Kosmische Strahlung.* http://www.astroteilchenphysik.de.

[2] *Document Type Definition.* http://www.stefanheymann.de/xml/dtdxml.htm.

[3] *GNU Affero General Public License.* http://rapid-i.com/content/view/29/82/lang,en/.

[4] *MAGIC II.* http://www.weltderphysik.de/de/7087.php.

[5] *Mars - MAGIC Analysis and Reconstruction Software.* http://magic.astro.uni-wuerzburg.de/mars.

[6] *RapidMiner Operatoren.* http://rapid-i.com/content/view/12/69/lang,de/.

[7] *ROOT - A Data Analysis Framework.* http://root.cern.ch/drupal/.

[8] *ROOT Einleitung.* http://root.cern.ch/drupal/content/about.

[9] *SAX und DOM mit Java.* http://www.torsten-horn.de/techdocs/java-xml.htm.

[10] *Tscherenkow-Strahlung.* http://psi.physik.kit.edu/107.php?PHPSESSID=5b17fd48a3eaa394a5b5d056f9771a47.

[11] Astroteilchenphysik, Experimentelle Physik Vb:. http://app.uni-dortmund.de/.

[12] Berger, Dipl.Ing. Klaus: *Korrelation.* http://www.mathe-online.at/materialien/klaus.berger/files/regression/korrelation.pdf.

[13] Künstliche Intelligenz, Lehrstuhl für. http://www-ai.cs.uni-dortmund.de/allgemein.html.

[14] Redelbach, Dr. Andreas: *Aktuelle Probleme der experimentellen Teilchenphysik.* http://www.physik.uni-wuerzburg.de/fileadmin/11010700/TeilchenVorlesung_WS0910/ROOT_examples_WS0910.pdf.

[15] Redelbach, Dr. Andreas: *ROOT - eine beispielorientierte Einführung.* http://www.physik.uni-wuerzburg.de/fileadmin/11010700/TeilchenVorlesung_WS0910/ROOT_examples_WS0910.pdf, 10 2009.

[16] SAX-Spezifikation. http://www.saxproject.org/.

# Literaturverzeichnis


[17] *Das Gammastrahlen-Teleskop MAGIC.* http://www.astronomie-heute.de/artikel/894883\&_z=798889.

[18] *Document Object Model (DOM) Level 1 Specification.* http://www.w3.org/TR/REC-DOM-Level-1/.

[19] *Extensible Markup Language (XML) 1.0 (Fifth Edition).* http://www.w3.org/TR/REC-xml/.

[20] *IceCube.* http://www.icecube.wisc.edu/info/.

[21] *Kosmische Strahlung und die energiereichsten Himmelskörper.* http://www.weltderphysik.de/de/5070.php.

[22] *MAGIC Picture Gallery.* http://wwwmagic.mppmu.mpg.de/gallery/pictures/.

[23] *The MAGIC Telescope.* http://wwwmagic.mppmu.mpg.de/.

[24] *XML Path Language (XPath).* http://www.w3.org/TR/xpath/.

[25] *XML Schema Part 0: Primer.* http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

[26] *XSL Transformations (XSLT).* http://www.w3.org/TR/xslt.

[27] *ROOT-based Object Persistency.* http://www.uscms.org/s&c/reviews/doe-nsf/2002-06/docs/rootio.pdf, 2002.

[28] *Hochenergetische Gammaund NeutrinoAstronomie - Hillas Parameter.* http://wwwiexp.desy.de/studium/lehre/astroteilchen/ws0809/Vorlesung_Astroteilchenphysik_20081209.pdf, 2008.

[29] *RapidMiner 4.3 User Guide.* http://sourceforge.net/projects/yale/files/1.%20RapidMiner/4.3/rapidminer-4.3-tutorial.pdf/download, 2008.

[30] *ROOT Users Guide 5.21.* ftp://root.cern.ch/root/doc/Users_Guide_5_21.pdf, 2008.

[31] *Genetische Merkmalsselektion für die Gamma-Hadron-Separation im MAGIC-Experiment.* http://www.cs.uni-dortmund.de/nps/de/Forschung/Publikationen/Graue_Reihe1/Ver__ffentlichungen_2009/828.pdf, 2009.

[32] Collider, The Large Hadron. http://public.web.cern.ch/public/en/LHC/LHC-en.html.

[33] Erik T. Ray, Übersetzung Olaf Brodacki: *Einführung in XML.* O'Reilly Verlag GmbH & Co. KG, ISBN 3-89721-286-2, 2001.

[34] Evan Lenz, Übersetzung Lars Schulten: *XSLT 1.0.* O'Reilly Verlag GmbH & Co. KG, 2006.

[35] Helf, Marius: *Studienarbeit: Genetische Merkmalsselektion für die Gamma-Hadron-Separation im MAGIC-Experiment.* http://www.physik.tu-dortmund.de/images/stories/Diplomarbeiten/E5b/studienarbeit_marius_helf.pdf, 9 2009.

[36] Kappes, Alexander: *Das Universum im Neutrinolicht: Neues von IceCube und ANTARES.* http://www.pi1.physik.uni-erlangen.de/~kappes/vortraege/2009-03-12_DPG-Munich/antares_icecube.pdf, 2009.

[37] Knobloch, Manfred und Matthias Kopp: *Web-Design mit XML.* Dpunkt Verlag, 2001.

[38] Krüger, Guido: *Handbuch der Java-Programmierung.* Addison Wesley Verlag, 4. A. Auflage, 12 2004.

[39] Käser, Daniel: *Java Enterprise Edition (J2EE: Servlets / Servlet Container / Web Applications.* wwwi2.informatik.uni-wuerzburg.de/mitarbeiter/fischer/teaching/seminar/j2ee/2005-WS/Servlets-Ausarbeitung.pdf, 6 2005.

[40] McLaughlin, Brett: *Java und XML, Übersetzung Matthias Kalle Dalheimer.* O'Reilly Verlag GmbH & Co. KG, 2001.

[41] RapidMiner: *Report the Future.* http://rapid-i.com/content/view/181/190/.

[42] Simon St.Laurent & Michael Fitzgerald, Übersetzung Nikolaus Schüler: *XML kurz & gut.* O'Reilly Verlag GmbH & Co. KG, ISBN 3-89721-516-0, 3. Auflage, 2006.

[43] Wagner, Wolfgang: *TWRDaq - Ein Datennahme- und Triggersystem für das AMANDA* Neutrinoteleskop.* http://www.astroteilchenschule.physik.uni-erlangen.de/schule2005/teilnehmer_vortraege/Wagner-Wolfgang.pdf, 2005.

# Abkürzungsverzeichnis

**MAGIC** Major Atmospheric Gammaray Imaging Cherenkov
**AGN** Active Galatic Nuclei
**GRB** Gamma Ray Burst
**XML** Extensible Markup Language
**SGML** Standard Generalized Markup Language
**MARS** „Magic Analysis and Reconstruction Software“
**XSLT** Extensible Stylesheet Language Transformations
**XHTML** Extensible HyperText Markup Language
**HTML** Hypertext Markup Language
**DTD** Document Type Definition
**XSD** XML Schema Definition
**LHC** Large Hadron Collider
**GUI** graphische Benutzeroberfläche
**CERN** Conseil Européen pour la Recherche Nucléaire
**W3C** World Wide Web Consortium
**DOM** Document Object Model
**SAX** Simple API for XML
**AGPL** GNU Affero General Public Lizenz
**XPath** XML Path Language
**JSP** Java Server Pages
**JAXP** Java API for XML Processing
**StAX** Streaming API for XML
**JVM** Java Virtuelle Maschine
**JMX** Java Management Extension
**Callisto** Calibrate light signals and time offsets
**Star** Standard analysis and reconstruction

# Danksagung

Während des dieser Arbeit wurde ich von vielen Menschen unterstützt, dafür möchte ich allen danken.

Zuerst möchte ich mich besonders bei Prof. Dr. Katharina Morik für die Möglichkeit diese Arbeit durchzuführen, so wie für die Unterstützung während der Diplomarbeitszeit, bedanken. Mein Dank gilt ebenfalls meinem Betreuer, Dipl.-Inf. Benjamin Schowe, der mir jederzeit mit gutem Rat zur Seite stand.

Desweiteren gilt mein herzlicher Dank für die freundliche Hilfestellung im Bereich der Physik Prof. Dr. Dr. Wolfgang Rhode welcher mir jederzeit für Fragen zur Verfügung stand so wie den Mitarbeitern des Lehrstuhls E5b „Astroteilchenphysik“ für ihre Hilfsbereitschaft.

Für die Unterstützung im Bereich der ROOT-Software und für die gute Zusammenarbeit bei der Präsentation des Themengebietes möchte ich Marius Helf danken.

An dieser Stelle gilt mein besonderer Dank meinen Eltern und meinem Bruder, die mir mein Studium durch Ermutigungen und tatkräftige Unterstützung ermöglicht haben.

Einen ganz besonderen Dank gebührt meinem Freund Gerrit Schünemann der mich während des Studiums und der Diplomarbeit immer unterstützt hat und mir mit Geduld und Verständnis beiseite stand.

Zu guter Letzt möchte ich mich für die gesamte Unterstützung während des Studiums und der Arbeit bei Familie Schünemann bedanken.