Technische Universität Darmstadt

Fachbereich Elektrotechnik und Informationstechnik

Institut für Datentechnik

Fachgebiet Echtzeitsysteme

Prof. Dr. Andy Schürr

Bachelor-Thesis

Entwicklung einer automatisierten Dokumentation von LabVIEW Quellcode für das Rahmenwerk CS

Bearbeitung: Martin Feldmann

Betreuung: Dr. Holger Brand

Beginn: 01.06.2007

Abgabe: 01.10.2007

Erklärung zur erstellten Arbeit

Hiermit versichere ich, die vorliegende Arbeit selbständig und ohne Hilfe Dritter angefertigt zu haben. Gedanken und Zitate, die ich aus fremden Quellen direkt oder indirekt übernommen habe sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen und wurde bisher nicht veröffentlicht.

Ich erkläre mich damit einverstanden, dass die Arbeit durch das Fachgebiet Echtzeitsysteme und der GSI der Öffentlichkeit zugänglich gemacht werden kann.

Darmstadt, den 01. Oktober 2007

_____

Unterschrift

Zusammenfassung

Die zunehmende Komplexität von Softwaresystemen in Industrie und Forschung, sowie die Anforderung der Wiederverwendbarkeit an Softwarekomponenten, machen die Dokumentation von Software zu den Pflichten jedes Entwicklers.
Die Bachelor-Thesis beschäftigt sich mit der Entwicklung eines Dokumentationswerkzeuges für ein objektorientiertes Rahmenwerk, das auf der graphischen Programmiersprache LabVIEW aufsetzt, und speziell zur Entwicklung von Kontrollsystemen großer Experimente an Forschungseinrichtungen weltweit entwickelt wurde.
Dabei wird diskutiert, welche Voraussetzungen für bestimmte Dokumentationen erfüllt sein müssen und auf welchem Weg eine Dokumentation erzeugt werden soll. Hierzu wird die Verwendung des OMG-Standards XMI der Erzeugung von Quellcode einer gängigen Programmiersprache als Zwischenschritt gegenübergestellt.
Als Ergebnis wird die Implementierung eines Prototyps beschrieben, der aus dem LabVIEW-Quellcode als Zwischenschritt Java-Quellcode generiert, um anschließend daraus eine Dokumentation in Form eines UML-Klassendiagrammes automatisch mit einem CASE-Tool erzeugen zu können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Die GSI[1]

Gegründet wurde die Gesellschaft für Schwerionenforschung (GSI) 1969 als Großforschungseinrichtung für Grundlagenforschung. Gesellschafter sind das Land Hessen zu 10% und die Bundesrepublik Deutschland zu 90%. Das Personal an der GSI besteht aus 1050 Mitarbeitern, davon sind ca. 300 Wissenschaftler und Ingenieure. Hinzu kommen an die 1200 Forscher von Hochschulen und Forschungsinstituten aus der ganzen Welt.
Die GSI betreibt eine weltweit einmalige Beschleunigeranlage für Ionenstrahlen. Diese besteht aus dem UNILAC[2], dem SIS[3] und dem ESR[4]. Der UNILAC ist ein 120m langer Linearbeschleuniger, der die Ionen auf 20% der Lichtgeschwindigkeit bringt. Darauf folgend werden die Ionen im SIS nach mehreren hunderttausend Umläufen auf 90% Lichtgeschwindigkeit weiter beschleunigt. Der ESR dient der Speicherung der beschleunigten Ionen, die darin bei konstanter Geschwindigkeit einige Millionen bis Milliarden Umläufe vollführen.
Die Grundlagenforschung an der GSI widmet sich unter anderem dem Erzeugen und Untersuchen neuer Elemente. Dabei werden Ionen, also geladene Atome, mit Hilfe von magnetischen Feldern, in einen Strahl gebündelt und auf hohe Geschwindigkeiten beschleunigt. Diese treffen anschließend auf Atome in einer Materialprobe. Bei der Reaktion können die Atomkerne miteinander verschmelzen und zerfallen danach in ein stabiles Element. Die wohl bekanntesten Resultate in diesem Bereich sind die Entdeckung von sechs neuen chemischen Elementen mit den Ordnungszahlen 107 bis 112 (Darmstadtium, Hassium, Meitnerium, Roentgenium Bohrium).
Neben Grundlagenforschung im Bereich der Kern- und Atomphysik, wird auch in Forschungsbereichen an der GSI gearbeitet, deren Nutzen direkt Vor Ort zur Anwendung kommt. Hierzu zählen die Materialforschung sowie Biophysik und Strahlenmedizin. Vor Allem die Entwicklung einer neuartigen Tumortherapie mit Ionenstrahlen, ist weltweit einzigartig. Dabei werden Tumore im Kopfbereich mit Kohlenstoff Ionen bestrahlt.
Seit 1997 bis heute (Stand 2006) konnten mit dieser neuen Therapiemethode über 300 Patienten erfolgreich behandelt werden. Die bisher in diesem Gebiet gewonnen Erkenntnisse werden nun genutzt, um ein Therapiezentrum in Heidelberg zu bauen, an dem bis zu 1000 Patienten pro Jahr eine Behandlungsmöglichkeiten geboten werden soll.
Darüber hinaus ist ein zentraler Bestandteil der Forschungsaktivität an der GSI die Wei-

[1] Gesellschaft für Schwerionenforschung, siehe [GrLP06], [GMPR06]
[2] Universal Linear Accelerator
[3] Schwerionensynchrotron
[4] Experimentierspeicherring

terentwicklung und Verbesserung der Forschungsinstrumente selbst. D.h. Beschleuniger und Detektoren, die der Grundlagenforschung dienen, werden an der GSI ständig weiterentwickelt, um die Experimentiermöglichkeiten zu verbessern und den Forschern eine bessere Grundlage für ihre Experimente zu bieten.
Weitere Angebote der GSI sind diverse Ausstellungen und Vorträge sowie Bildungseinrichtungen, wie z.B. das Schülerlabor oder die Saturday Morning Physics, welche eine Kommunikation zwischen Schülern, Studenten, Lehrern und Forschern ermöglichen soll.
Die Zukunftspläne der GSI sehen eine Erweiterung der bestehenden Beschleunigeranlage vor. Diese trägt den Namen FAIR[5] Projekt und soll voraussichtlich bis 2012 fertig gestellt und in Betrieb genommen werden. Hierdurch verspricht sich die GSI, durch erhöhte Strahlintensität und verbesserte Strahlqualität, im Bereich der Grundlagenforschung neue Entdeckungen und Erkenntnisse gewinnen zu können. Die folgende Graphik soll einen Eindruck des momentanen Zustandes der Beschleunigeranlage und deren geplanten Ausbau vermitteln.

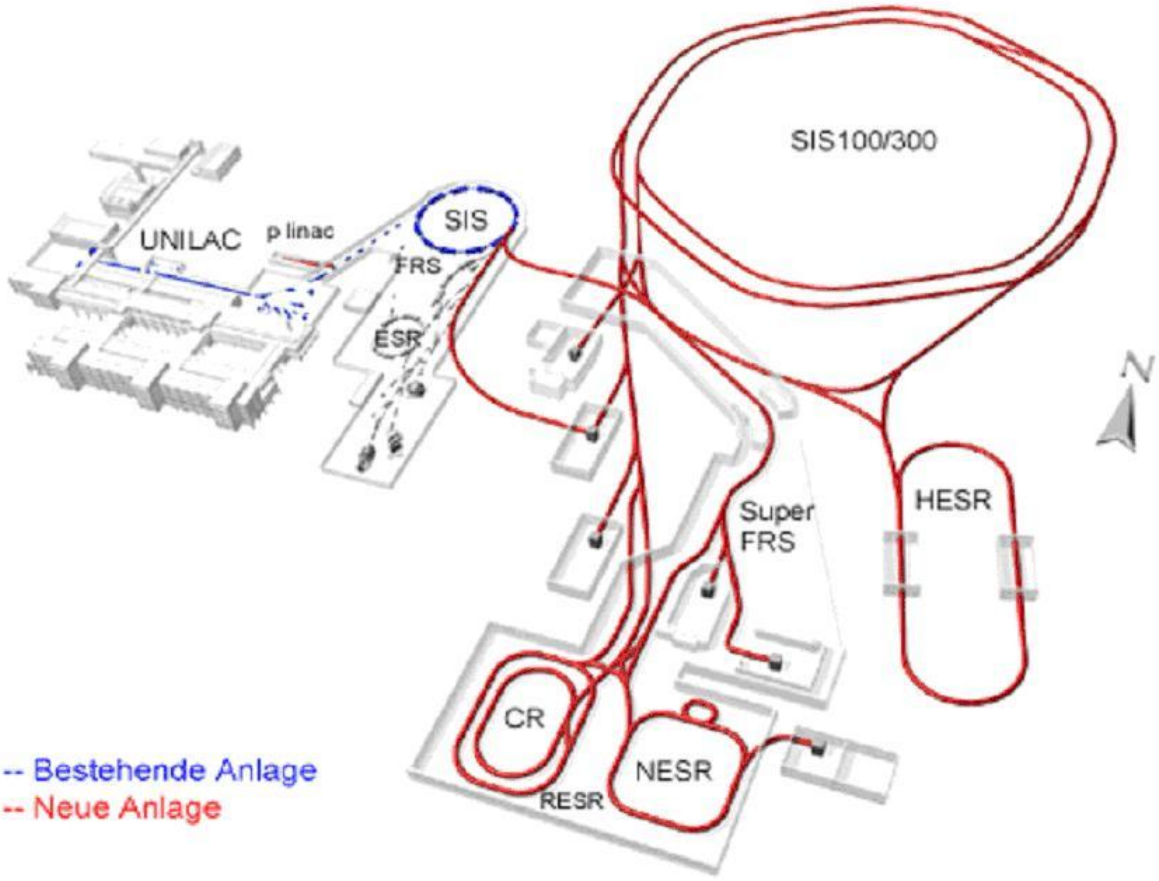


Abbildung 1: GSI Anlage und deren Ausbau

[5]Facility for Antiproton and Ion Research

Detaillierte Informationen zur GSI, ihrer Infrastruktur und Angeboten sind im Internet auf den folgenden Seiten zu finden:
www.gsi.de (Homepage der GSI)
www.gsi.de/portrait (Überblick über die GSI)
www.gsi.de/zukunftsprojekte (Homepage des FAIR Projektes)

Die Bachelor-Thesis wurde in der Abteilung Experiment-Elektronik der GSI durchgeführt. Diese unterstützt die Experimente mit speziell angefertigter Hardware, inklusive der zur Steuerung nötigen Software. In den letzten Jahren wurde hier ein objektorientiertes Rahmenwerk zur Entwicklung von Kontrollsystemen für Experimente entwickelt, das so genannte *CS*-Framework[6]. Die Arbeit beschäftigt sich mit der Entwicklung eines Dokumentationswerkzeuges für Kontrollsysteme, die mit *CS* entwickelt wurden und für das *CS*-Framework selbst. Das folgende Kapitel erläutert kurz die Entwicklungsumgebung LabVIEW[7] und das darauf aufgesetzte Rahmenwerk *CS*.

1.2 Einführung in die Entwicklungsumgebung

Im folgenden Kapitel wird eine kurze Einleitung zur Entwicklungsumgebung LabVIEW, sowie dem darauf basierenden *CS*-Framework gegeben.
LabVIEW ist die Grundlage dieser Arbeit und *CS* das zu dokumentierende System.
Detaillierte Erläuterungen über LabVIEW und *CS* befinden sich in Kapitel 2, Entwicklungsumgebung.

1.2.1 Einführung in LabVIEW

LabVIEW ist eine graphische Programmiersprache von National Instruments, die sich vorwiegend in Forschungseinrichtungen, Industrie und Lehreinrichtungen verbreitet hat. LabVIEW ist ein leistungsstarkes und vielseitiges Software-System zur Analyse und Instrumentierung und wird standardmäßig für Kontrollsoftware von Instrumenten und zur Datenerfassung verwendet. Dabei weicht LabVIEW von der sequentiellen Abarbeitung von Befehlen traditioneller Programmiersprachen ab, und stellt eine einfache und intuitive graphische Entwicklungsumgebung bereit, inklusive aller Werkzeuge, die für Datenerfassung, Datenanalyse und Darstellung der Ergebnisse benötigt werden [Harm98].
In LabVIEW wird die graphische Programmiersprache G[8] verwendet. Die Programmierung wird anhand einer graphischen Repräsentation des Datenflusses eines Programms,

[6] Control System Framework
[7] Laboratory Virtual Instruments Engineering Workbench
[8] siehe Glossar

des so genannten Blockdiagramms durchgeführt, während die Präsentation der Ergebnisse, also die Datenquellen und -senken eines Programms, auf dem so genannten Frontpanel dargestellt werden. Hier erscheint die Präsentation der Daten in genau der gewünschten Form, z.B. als Diagramm, Graph oder benutzerdefinierter Graphik.
LabVIEW ist eine Compilersprache[9] und daher mit der Performance von C/ C++ vergleichbar. Es ist jedoch nicht quelloffen. Ein LabVIEW-Programm wird auch VI[10] genannt [LVUG05].
Weiterhin bietet LabVIEW mehr Flexibilität als Standard-Laborinstrumente, da es softwarebasiert ist. Der Benutzer kann exakt den gewünschten Typ eines virtuellen Instruments erzeugen, und spart damit Zeit und Geld. Wenn sich die Anforderungen an ein virtuelles Instrument ändern, kann es in kurzer Zeit modifiziert werden.
LabVIEW unterstützt beim Programmieren vieler nebenläufiger Prozesse mit Hilfe der threadsicheren[11] Runtime Engine, die notwendige Programmierstrukturen für Ereignissteuerung und Synchronisierung bereitstellt, so dass der Entwickler sich nicht mit Zeigern, Speicherallokation und anderen Problemen beschäftigen muss, die man in vielen gängigen Programmiersprachen hat.
LabVIEW besitzt viele Module die ähnliche Funktionalitäten wie MATLAB und Simulink[12] in den Bereichen der Analyse und Gestaltung von Kontrollsystemen, Signalverarbeitung, Mathematik und Simulation bereitstellen. Zusätzlich hat LabVIEW eingebaute Unterstützung für die große Auswahl an mess- und automatisierungstechnischer Hardware, die von National Instruments hergestellt wird. Durch vielfältige Treiber und die Unterstützung der Kommunikationsstandards OPC[13], Modbus, GPIB[14] und vielen mehr wird aber auch die Hardware von Drittanbietern unterstützt.

1.2.2 Einführung in das *CS*-Rahmenwerk

Das allgemeine Kontrollsystem *CS* wird seit 2001 an der GSI entwickelt und ist seit 2002 an einigen Experimenten in verschiedenen Forschungseinrichtungen im Einsatz. Es wurde aus der Anforderung heraus entwickelt, eine gemeinsame Basis zur Entwicklung spezieller Kontrollsysteme zu bilden. *CS* kann durch einige experimentspezifische Erweiterungen an nahezu beliebige Experimente angepasst werden. Es wird von einer zentralen Gruppe gewartet und entwickelt, erhöht die Wiederverwendbarkeit der bereits entwickelten Software

[9] siehe Glossar
[10] Virtual Instrument
[11] siehe Glossar
[12] siehe Glossar
[13] Object Linking and Embedding for Process Control
[14] General Purpose Interface Bus

und spart Entwicklungszeit neuer Kontrollsysteme für Experimente.
Hauptmerkmale von *CS* sind Objektorientierung, ereignisgesteuerte Kommunikation von Objekten, das Fehlen intrinsischer Engpässe[15], die Fähigkeit zur Verteilung auf Knoten in Netzwerken, und die Verwendung von LabVIEW, was eine steile Lernkurve, sowie eine sehr gute Hardwareanbindung garantiert. Der Schwerpunkt von *CS* liegt in der Unterstützung vieler verschiedener Gerätetypen, wobei sowohl SCADA[16] Eigenschaften, als auch Schnittstellen zu Feldbussen wie CAN[17] oder Profibus[18] und OPC[19]-Server benötigt werden.
CS basiert auf dem Einsatz objektorientierter Techniken in LabVIEW, wobei jeder Typ eines Laborgerätes typischerweise durch eine eigene Klasse repräsentiert wird. Zur Laufzeit wird dann dynamisch für jedes Gerät eines Typs ein Objekt instanziiert. Einzelne, verteilte Objekte sind für Unteraufgaben wie Benutzerschnittstelle, Sequenzer, Datenakquisition oder Gerätetreiber verantwortlich. Diese Objekte sind aktiv und haben mindestens zwei unabhängige Threads[20]. Ein Thread dient zum Empfangen und Verarbeiten von Ereignissen, da die Objekte ereignisgesteuert kommunizieren können. Im zweiten Thread können periodische Aktionen, wie das Auslesen von Ist-Werten oder Regelschleifen realisiert werden. Optional kann in einem dritten Thread eine Zustandsmaschine implemetiert werden.
Neben zahlreichen Klassen für Laborgeräte verfügt das *CS* über Basisklassen für graphische Benutzeroberflächen, Ablaufsteuerung, Pufferung und Speicherung von Daten, Objektverwaltung und Überwachung. Es können viele Objekte auf ein Netzwerk von mehreren PCs verteilt werden, wobei die Objekte auch über Rechnergrenzen hinweg miteinander kommunizieren können. SCADA-Funktionalität wie Alarm-Meldung und Ereignis-Protokollierung wird durch das Datalogging und Supervisory Control Modul bereitgestellt. Außerdem beinhaltet *CS* auch Werkzeuge, um neue Klassen durch Vererbung zu erzeugen und zu testen.
Das *CS*-Framework für Kontrollsysteme kann mit bis zu einigen 10000 Prozessvariablen eingesetzt werden [Bran05].

[15] Engpässe, die im System durch dessen Architektur entstehen, siehe auch Kapitel 2.2.4
[16] Supervisory Control and Data Acquisition, siehe Glossar
[17] Controller Area Network
[18] Process Field Bus
[19] Object Linking and Embedding for Process Control
[20] siehe Glossar

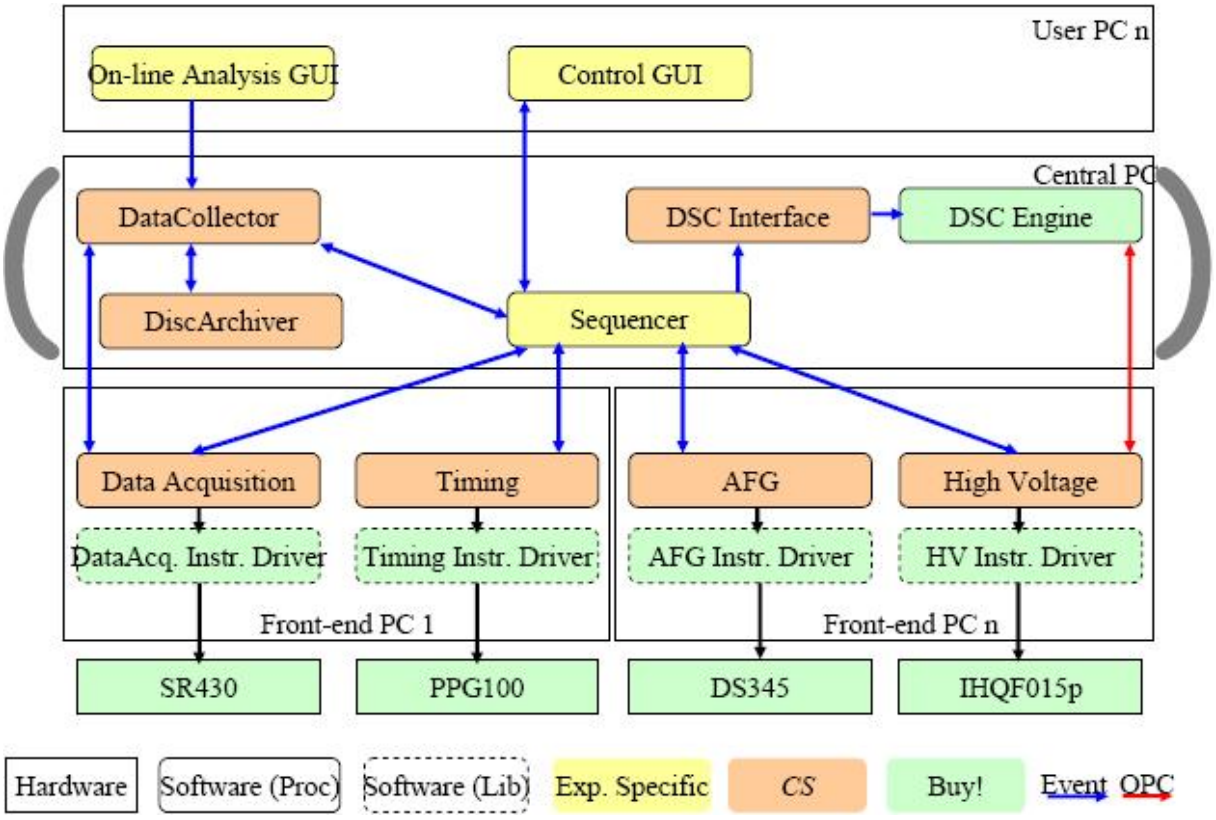


Abbildung 2: Beispiel für ein Kontrollsystem [Beck05]

Da das *CS*-Framework bei immer mehr Experimenten an verschiedenen Forschungseinrichtungen zum Einsatz kommt, steigt auch das Bedürfnis nach einer einfachen Art der Dokumentation der Systeme mit der Zahl der eingesetzten Entwickler. Während *CS*-Entwickler im Moment eine LabVIEW interne Dokumentation ihres Quellcodes manuell erstellen müssen, besteht die Nachfrage nach einem Werkzeug, das aus dieser Dokumentation automatisch eine UML[21] Darstellung in Form eines Klassendiagramms erzeugt. In diesem Kontext ist die Idee zur Bachelor-Thesis entstanden.

1.3 Motivation, Themenstellung und Überblick

Motivation: Die zunehmende Komplexität von Kontrollsystemen großer Experimente an Forschungseinrichtungen weltweit stellt immer höhere Anforderungen an die dort verwendete Hardware, und an die Software die diese steuert. Die Anforderung der Wiederverwendbarkeit von Softwarekomponenten, deren Wartung sowie deren Einsatz durch den Benutzer oder Kunden macht die Dokumentation eines Softwaresystems zur Aufgabe des Entwicklers.

Da die technischen Anforderungen für Experimente im Bereich der Kernphysik einzigartigartig und in keinem anderen Bereich der Industrie anzutreffen sind, werden die Kompo-

[21] Unified Modeling Language, siehe Glossar

nenten für den Aufbau eines Versuchs normalerweise vor Ort maßgeschneidert entwickelt und gebaut, weshalb keine kommerziellen Gesamtlösungen hierfür existieren. Trotzdem sind die Anforderungen an Software für Kontrollsysteme zum Teil immer ähnlich gelagert, so dass zur Verbesserung der Wiederverwendbarkeit und Zusammenarbeit von Arbeitsgruppen und Forschungseinrichtungen das *CS*-Framework entwickelt wurde.
Da *CS* eine einheitliche Plattform für viele Entwickler von Kontrollsystemen darstellt, lohnt sich auch die Entwicklung eines Dokumentationswerkzeuges für *CS*-Kontrollsysteme. Damit ist der Entwickler nicht mehr gezwungen, seine individuelle Dokumentation manuell anzufertigen, wobei Zeit und Arbeitsaufwand eingespart wird. Dem Entwickler wird hier eine einfache und komfortable Möglichkeit gegeben, seine Dokumentation anzufertigen. Dadurch wird der Zugang zur Dokumentation verbessert und die Akzeptanz erhöht, überhaupt eine Dokumentation anzufertigen, da diese Tätigkeit im Alltag des Entwicklers meist an letzter Stelle steht.
Einer der wesentlichen Gründe, die zur Entscheidung beigetragen haben, das *CS*-Framework in LabVIEW zu realisieren, ist die intuitive Anwendung von LabVIEW als graphische Programmiersprache und deren steile Lernkurve, da die Entwicklung von Kontrollsystemen meist von Doktoranden und Studenten übernommen wird, die keine oder nur wenig Erfahrung auf dem Gebiet der Softwareentwicklung haben. Eine Dokumentation ist einerseits nötig, da diese Mitarbeiter naturgemäß nicht für die komplette Zeit eines Experiments verfügbar sind, und andererseits die Anforderung einer schnellen Einarbeitung in bestehende Systeme haben [BeBH03].
Weiterhin bietet eine Dokumentation einen besseren Austausch über Neuentwicklungen, sowie eine Verbesserung von Lehrveranstaltungen und Trainingskursen zum *CS*-Framework.

Themenstellung: Thema der Arbeit ist die Entwicklung eines Dokumentationswerkzeuges für das *CS*-Framework. Im Rahmen dieser Entwicklung wird erläutert, welche Formen der Dokumentation in Frage kommen, und welche Voraussetzungen vorhanden sind oder geschaffen werden müssen, um eine bestimmte Form der Dokumentation zu erzeugen.
Da das *CS*-Framework objektorientiertes Programmieren ermöglicht, sind als Ziele die automatische Erzeugung von UML Dokumentationen, insbesondere UML-Klassendiagramme, zu nennen. Klassendiagramme haben einen großen Bekanntheitsgrad, sind einfach zu erlernen und zu verstehen, und geben einem Entwickler in kurzer Zeit einen guten Überblick über Software Systeme. Sie sind zusammen mit Sequenz und Aktivitätsdiagrammen im Moment schon an der GSI im Einsatz, müssen aber manuell erstellt werden.

Weiterhin ist die Frage nach einer geeigneten Umsetzung hinsichtlich der Implementierung einerseits, und der späteren Verwendung des Werkzeuges andererseits zu klären. Da eine komplette Implementierung eines eigenen Dokumentationswerkzeuges über den Rahmen dieser Arbeit hinausgehen würde, wird auf die Funktionalität eines CASE-Tools[22] bezüglich der Darstellung verschiedener Diagramme zurückgegriffen. Es werden verschiedene CASE-Tools analysiert, die in Frage kommen, um die graphische oder schriftliche Dokumentation später darzustellen. CASE-Tools greifen in der Regel auf Metadaten eines Software-Systems zurück, um eine Darstellung verschiedener Diagramme zu generieren, die die Funktionsweise des Systems dokumentieren. Damit stellt sich die Frage nach einem geeigneten Weg, Metadaten aus *CS* zu sammeln und diese für ein oder mehrere CASE-Tools zur Verarbeitung aufzubereiten.
Ziel der Arbeit ist eine Aussage über die Machbarkeit einer solchen automatisierten Dokumentation zu treffen, die geeignete Methodik dazu zu finden oder zu entwickeln, und anhand eines Prototypen deren Umsetzung zu zeigen.

Überblick: Die Bachelor-Thesis ist in acht Kapitel eingeteilt. Die Gliederung ist an einen Softwareentwicklungsprozess, das Wasserfallmodell, angelehnt.
Das erste Kapitel besteht aus einer Einleitung, in der erklärt wird, worum es in der Arbeit geht. Es wird ein kurzer Überblick über die technische Umgebung gegeben und die Forschungseinrichtung wird vorgestellt.
Kapitel zwei behandelt die technischen Voraussetzungen und beschreibt die Entwicklungsumgebung LabVIEW und das *CS*-Framework im Detail.
Im dritten Kapitel werden verwandten Arbeiten beschrieben und mit der Bachelor-Thesis verglichen.
Im vierten Kapitel werden die Anforderungen an die Arbeit definiert. Es wird festgelegt was das zu entwickelnde System leisten soll.
Das fünfte Kapitel ist der Systementwurf. Es ist ein Grobentwurf. der bestimmt, wie die Funktionen der Software zu realisieren sind.
In Kapitel sechs werden die konkrete Realisierung der einzelnen Module der Software anhand eines Walkthroughs sowie Integration der Module, Test und Benutzerschnittstelle beschrieben.
Kapitel acht beinhaltet eine Schlussbetrachtung, sowie den Ausblick auf die Zukunft des *CS*-Frameworks und des Dokumentationswerkzeuges.

[22] Computer-Aided Software Engineering Werkzeug, siehe Glossar

2 Entwicklungsumgebung

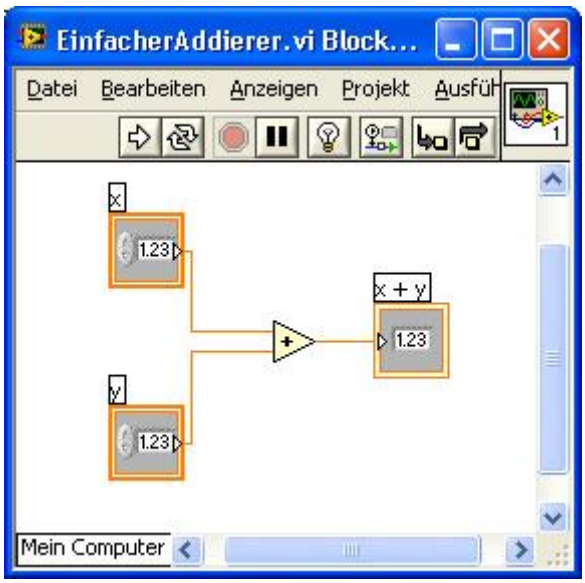
In diesem Kapitel wird die Entwicklungsumgebung erklärt, auf der diese Arbeit aufgebaut ist. Hier werden LabVIEW und *CS* im Detail beschrieben.

2.1 LabVIEW

Im folgenden Kapitel wird LabVIEW detailliert beschrieben. Es wird eine Einführung in die Datenflussprogrammierung mit Blockdiagramm und gleichzeitiger Erstellung einer Benutzerschnittstelle, dem Frontpanel gegeben. Weiterhin enthält das Kapitel Abschnitte über modulare Programmierung, unterstützte Betriebsysteme und Schnittstellen, sowie eine Beurteilung der Vor- und Nachteile von LabVIEW.

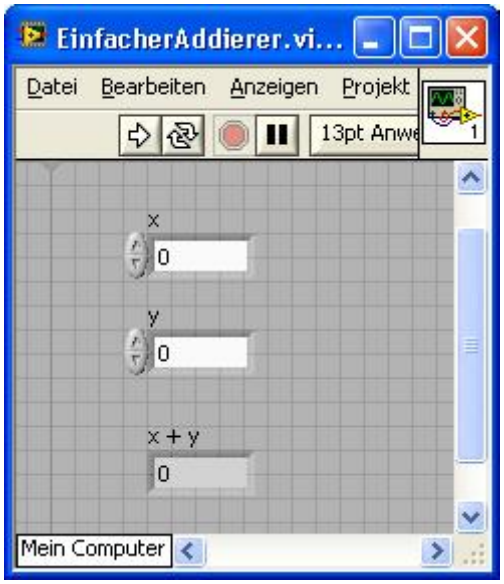
2.1.1 Überblick

LabVIEW ist eine Kombination aus graphischer Programmiersprache und Entwicklungsumgebung, die auf den Einsatz in einer Laborumgebung zugeschnitten ist. Darin unterscheidet sie sich maßgeblich von konventionellen Programmiersprachen wie C oder Basic. Die Parallele zum Labor ist das Virtuelle Instrument (VI). Genau wie ein reales Instrument hat ein LabVIEW Programm eine Steuerung die die benutzerdefinierten Eingangsparameter darstellt, Anzeigeelemente, die die vom Programm produzierten Ausgangsparameter in einer benutzerdefinierten Form abbilden, und eine Logik, die die Beziehung zwischen Ein- und Ausgangsparametern herstellt.
Logik und Benutzerschnittstelle sind zwei verschiedene Funktionseinheiten in einem LabVIEW Programm. Die Benutzerschnittstelle heißt Frontpanel, und die Programmlogik wird Blockdiagramm genannt [Holl05].



Die Linien die den Datenfluss anzeigen, werden Drähte genannt, analog zu realen Laborgeräten. Eingänge werden Kontrollen und Ausgänge Indikatoren genannt. Der eigentliche Addierer, das dreieckige Symbol mit dem Plus-Zeichen, wird SubVI genannt, und entspricht einer Subroutine gewöhnlicher Programmiersprachen.

Abbildung 3: Programmlogik eines einfachen Addierers in Form des Blockdiagramms



Kontrollen sind hier die Bedienelemente x und y, Indikator ist das Anzeigelement x+y.

Abbildung 4: Benutzerschnittstelle in Form des Frontpanels

Obwohl sich LabVIEW mittlerweile zur Universal-Sprache weiter entwickelt hat, wurde LabVIEW in erster Linie als Programm zur Erfassung und Verarbeitung von Messwerten entwickelt. Dazu greift es auf applikationsspezifische Bibliotheken für die Steuerung von Einsteckkarten, sowie externe Mess- und Automatisierungssysteme, wie PCI[23], PXI[24], RS232, RS485, Firewire, USB oder Ethernet zurück. Zur Vervollständigung der Entwick-

[23]Peripheral Component Interconnect
[24]PCI eXtensions for Instrumentation

lungsumgebung hat LabVIEW weitere Bibliotheken Datenrepräsentation, Datenanalyse, Dateiverwaltung und eine Reihe konventioneller Entwicklungs- und Programmierhilfsmittel.

2.1.2 Blockdiagramm/ Frontpanel

Frontpanel: Wie bei den Frontplatten von realen Labormessgeräten, deren Bedien- und Anzeigeelemente als Schnittstelle zwischen Anwender und Gerät dienen, ist das Frontpanel die Bedienoberfläche eines LabVIEW Programms, das die Interaktion des Benutzers mit dem Programm ermöglicht. Hier werden die Ereignisse der Anwendungen visualisiert. Das Frontpanel kann Felder für Zeichen, Schieberegler, Graphiken und Graphen, Dreh- und Drückschalter, kombiniert zu Cluster, Matrizen, Arrays, Listen oder Tabellen, als Ein- und Ausgabeelemente haben [Pich06].

Blockdiagramm: Die Funktionalität eines Laborgerätes hingegen ist in Form von elektronischen Komponenten realisiert, die durch elektrische Schaltkreise verbunden sind. Das entspricht dem so genannten Blockdiagramm eines LabVIEW Programms. Es ähnelt einem Schaltplan, in dem graphische Symbole mit softwarebasierten Datenleitungen in Form von 'Drähten' verbunden werden können, um eine bestimmte Funktion zu erfüllen.
Dieses Blockdiagramm stellt den eigentlichen Programmcode dar. Dieser Quellcode ist, im Gegensatz zum konventionellen, textbasierten Quellcode anderer Sprachen wie C oder Pascal, graphisch. Er wird als Blockschaltbild von einem Entwickler gezeichnet. Bestandteile können grundlegende Funktionen wie Addierer, als auch komplexe Funktionen wie Fourier-Transformationen oder selbst geschriebene Unterprogramme sein. Diese Elemente werden auch SubVI genannt. Die Schnittstellen von SubVIs zu ihrem aufrufenden VI heißen Terminals, die als Datenquellen oder Senken ausgelegt sind.
Grundsätzlich werden in einem Blockdiagramm drei Gruppen von Elementen verwendet:

- Knoten sind Programmausführungselemente, wie etwa der oben genannte Addierer. Diese sind mit Operatoren, Funktionen oder Subroutinen textbasierter Programmiersprachen vergleichbar.

- Terminals sind Ports, die eine Kommunikation des Blockdiagramms mit dem Frontpanel gewährleisten. Alle Ausführungselemente besitzen Terminals, die mit Drähten untereinander verbunden sind. Terminals bilden Schnittstellen von Methoden, Ein- und Ausgangsparameter konventioneller Programmiersprachen nach. In LabVIEW gibt es Kontroll- und Anzeigeterminals, Konstanten und spezielle Terminals von Strukturen.

- Datenflußverbindungsdrähte dienen zur Verbindung sämtlicher Elemente eines Blockdiagramms und entsprechen Variablen textbasierter Programmiersprachen. Es ist nicht möglich mehrere Quellen an ein Eingangsterminal eines SubVI, oder Zyklen durch Drähte anzulegen.

Weiterhin bietet LabVIEW dieselben Standardkonstrukte wie textbasierte Sprachen: Schleifen, bedingte Ausführung, (Natur-) Konstanten, mathematische, Konvertierungs-, String-, Array-, Cluster-, Datei I/O-, Dialogfunktionen und Bibliotheken. In Kontrast zu textbasierten Programmiersprachen sind noch Sequenzstrukturen zu erwähnen, mit deren Hilfe der Datenfluss an kritischen Stellen durch eine genau festgelegte Ausführungsreihenfolge zu kontrollieren ist. Sollte die Funktionalität von LabVIEW nicht ausreichen für ein spezielles Problem, so ist es möglich, durch Schnittstellenknoten auch C oder Matlab Code auszuführen.

2.1.3 Datenfluss

LabVIEW verwendet die graphische Programmiersprache G, um den Quellcode eines Programms im Blockdiagramm zu beschreiben. Diese eliminiert viele syntaktische Details, was es dem Benutzer ermöglicht, sich ganz auf den Datenfluss einer Anwendung zu konzentrieren. Für Wissenschaftler und Ingenieure ist dies oft von Vorteil, da LabVIEW Symbole, Terminologien und Ideen verwendet, die ihnen bereits bekannt sind. Das Datenflussprinzip erleichtert ihnen den Einstieg in die Sprache, auch wenn sie vorher keine Programmierkenntnisse hatten.
Das Datenflussprinzip unterscheidet sich jedoch fundamental vom Kontrollfluss einer konventionellen Programmiersprache, vor allem in Hinblick auf die Ausführungsreihenfolge der Befehle eines Programms. Während in textbasierten Sprachen die Ausführungsreihenfolge der Textstruktur folgt, ist die Anordnung der Elemente eines LabVIEW Programms völlig irrelevant für deren Ausführung.
Die Ausführung eines Befehls in LabVIEW wird durch die Bedingung gesteuert, ob alle Eingangswerte an einem Element verfügbar sind oder nicht.
LabVIEW ist außerdem in der Lage, mehrere Befehle für die die Ausführungsbedingungen erfüllt sind, parallel abzuarbeiten. Dazu stellt die Datenflussmaschine von LabVIEW Parallelisierungs- und Multitaskingkonstrukte zur Verfügung. Durch parallelisierende Strukturen innerhalb einzelner Prozessoren und dem Einsatz von FPGAs[25] oder Multiprozessorsystemen bei leistungsfähigen Computerarchitekturen hat der Datenflussansatz Vorteile. LabVIEW ermöglicht das Konzept der Parallelisierung von Virtuellen

[25] Field Programmable Gate Array

Instrumenten auf Multiprozessorsysteme auszudehnen. Hierzu wird Multithreading verwendet, um die optimale Aufteilung von einzelnen Aufgaben innerhalb einer Anwendung zu verteilen.
Ist die Ausführungsreihenfolge für die Anwendung von Bedeutung, so bietet LabVIEW mit der Sequenzstruktur eine Möglichkeit, diese aktiv zu steuern.

2.1.4 SubVI

SubVIs entsprechen Subroutinen oder Methoden in textbasierten Programmiersprachen. Strukturierte Programmierung zeichnet sich durch einen durchdachten Einsatz von Unterprogrammen aus:

- bessere Wiederverwendbarkeit
- mehr Übersicht im Programmcode
- Modularität
- leichtere Wartbarkeit

Ein wohl definierter Abstraktionsmechanismus hinsichtlich der Datenübergabe zwischen Modulen wird durch die Korrespondenz des Blockdiagramms mit einem Symbol erfüllt, indem die Datenquellen und Senken den Terminals des SubVIs zugewiesen werden.
Ein SubVI wird also erzeugt, indem die Terminals eines VI festgelegt werden, um das SubVI später in ein größeres Programm mit Hilfe von Drähten zu integrieren. Dazu werden Frontpanelelemente als Terminals markiert. Um später die Übersicht zu behalten, ist es allerdings notwendig, ein kleines Icon dafür zu entwerfen. LabVIEW hat zu diesem Zweck einen einfachen Iconeditor. Weiterhin ist es möglich, automatisch ein SubVI aus einem Teil des Blockdiagramms extrahieren zu lassen. Zu erwähnen ist, dass der Benutzer eines Programms später aber nur Zugriff auf das Frontpanel des Hauptprogramms haben wird und nur die Argumente an SubVIs weitergegeben werden, solange kein Dialog dafür konfiguriert worden ist oder das Frontpanel nicht programmatisch aufgeschaltet wird.
Auf ein SubVI kann von übergeordneten Programmen aus beliebig oft zugegriffen werden. Eine Begrenzung bezüglich der Schachtelungstiefe in Programmen ist nicht vorhanden.
In Analogie an reale Laborgeräte entspricht ein SubVI in etwa einem integrierten Schaltkreis, der über verschiedene Ein- und Ausgänge beschaltet werden kann.

2.1.5 Plattform

Trotz der weiten Verbreitung von Windows ist es grade für eine Entwicklungsumgebung wie LabVIEW wichtig, eine gewisse Plattformunabhängigkeit zu haben. Da LabVIEW

für den Einsatz in Industrie und Labor gedacht ist, wo zum Teil sehr viele verschiedene Betriebsysteme benutzt werden, ist dies ein großer Vorteil. LabVIEW unterstützt daher auch die gängigsten Betriebsysteme, Windows, MacOS und Linux.
Eine explizite Portierung von LabVIEW Programmen ist nicht nötig, da man einfach den Quellcode auf das gewünschte System kopieren und starten kann. Portierungen zwischen Betriebsystemen sind nur bei der Verwendung von externen Bibliotheken notwendig, die mit bestimmten Betriebsystemen verlinkt sind.

2.1.6 Graphischer Compiler

Im Gegensatz zu den meisten graphischen Werkzeugen verwendet LabVIEW keinen Interpreter[26], sondern den Compiler von G. Das bringt signifikante Performancevorteile, da ein Compiler den Quellcode in optimierten Maschinencode umsetzt, und ein komplettes, ausführbares Programm erzeugt, anstatt das Programm direkt aus dem Quellcode heraus zu starten, wobei jede Anweisung einzeln übersetzt und ausgeführt werden muss. Dabei überprüft LabVIEW schon während des Programmierens, ob eine Übersetzung überhaupt zu möglich ist. Dadurch werden Fehler im Programmcode sofort angezeigt, und nicht erst während des Übersetzungvorgangs.

2.1.7 Unterstützte Schnittstellen

LabVIEW unterstützt die meisten gängigen Soft- und Hardwareschnittstellen, wie ActiveX, DDE[27], DLL[28], TCP/IP[29], RS232, RS422, RS485, GPIB (IEEE 488), PCI, AT-Bus, PCMCIA (PCCARD)[30], VXI, PXI, USB, IrDA[31], Fire Wire (IEEE1394), CAN, Profibus, PXI/Compact PCI, ISA/EISA[32], Multi-I/O, A/D[33], D/A[34], CTR, Dig I/O, DSP[35], und viele mehr.

2.1.8 Verteilte Anwendungen

LabVIEW besitzt einige Mechanismen, die geeignet sind, verteilte Anwendungen zu entwickeln:

[26] siehe Glossar
[27] Dynamic Data Exchange
[28] Dynamic Link Library
[29] Transport Protocol/ Internet Protocol
[30] Personal Computer Memory Card International Association
[31] Infrared Data Association
[32] (Extended) Industry Standard Architecture
[33] Analog-Digital-Wandler
[34] Digital-Analog-Wandler
[35] Digitaler Signalprozessor

- TCP/IP Unterstützung
- VI-Server
- Data Socket
- Shared Variable

Shared Variables werden zur gemeinsamen Nutzung von Daten zwischen VIs, die auf verschiedenen Knoten im Netzwerk operieren, verwendet. Sie ermöglichen VIs in einem Netzwerk oder Projekt das Lesen und Schreiben von gemeinsamen Daten. Um VIs die gemeinsame Nutzung von Daten lokal zu ermöglichen, werden single-process Shared Variables eingesetzt, während eine gemeinsame Verwendung auf verschiedenen Targets im Netzwerk durch Veröffentlichung der Daten erreicht wird.
Das Echtzeitmodul von LabVIEW kann die Shared Variable mit einer Echtzeit FIFO[36] ausstatten. Durch die Aktivierung der Echtzeit FIFO ist eine deterministische, gemeinsame Nutzung von Variablen möglich, ohne den Determinismus von VIs zu beeinflussen, die auf dem Echtzeit Target operieren. Dabei kommt eine Single- oder Multi-Element FIFO zum Einsatz. Die Single-Element FIFO erlaubt eine gemeinsame Nutzung des letzten bekannten Wertes. Dieser wird von der Shared Variable bei Bekanntgabe eines neuen Wertes überschrieben, was manchmal zu Datenverlust führen kann. Die Multi-Element FIFO hingegen puffert die gemeinsam genutzten Daten der Shared Variable. Der Puffer kann dabei vom Benutzer konfiguriert werden.
Wird eine im Netzwerk veröffentlichte Shared Variable auf einem Target erzeugt, so wird sie von der Shared Variable Engine (SVE) des Targets, auf dem sie erzeugt wurde, gehostet. Die SVE verschickt die gemeinsam genutzten Daten dann an andere Targets im Netzwerk, wobei die Kommunikation hier nicht deterministisch ist.
Um dennoch eine deterministische Kommunikation zu ermöglichen, gibt es auch zeitgesteuerte Shared Variables. Diese übertragen Daten deterministisch über ein geschlossenes, privates oder zeitgesteuertes Netzwerk. Andere Echtzeit Targets im zeitgesteuerten Netzwerk können dann auf die Daten zugreifen. Zeitgesteuerte Kommunikation mit Hilfe des LabVIEW Echtzeit Moduls benötigt eine benutzerdefinierte Konfiguration des Echtzeit Targets für zeitgesteuerte Kommunikation, der zeitgesteuerten Shared Variables und des zeitgesteuerten Netzwerkes.

Data-Socket ermöglicht einen Datenaustausch zwischen verteilten Anwendungen, indem es eine API[37] zur Verfügung stellt, die Low Level Funktionen wie Datei I/O, TCP/IP

[36] First In First Out
[37] Application Programming Interface

und FTP/HTTP[38] Kommunikationsanforderungen kapselt. Damit ist eine Verwendung verschiedener Datenstrukturen, Kontroll- und Kommunikationsmechanismen auf einer höheren Ebene, ohne tiefere Kenntnis dieser, möglich. Data Socket erkennt dabei automatisch die jeweiligen Protokolle und Formate, ist aber aufgrund des dadurch produzierten Overheads nicht so performant wie etwa VI-Server Kommunikation.
Data Socket unterstützt Methoden zum Abonnieren und Publizieren von Daten, was durch Ereignisse gesteuert, bzw. ausgelöst wird. Dabei ist die Übertragung von Daten beliebigen Typs möglich, wobei Objekte gleichzeitig Abonnent als auch Publizist sein können. Die Kommunikation ist ereignisgesteuert und asynchron.
Data Socket eignet sich für Broadcasting, wenn die Information eines Servers vielen Clients zur Verfügung gestellt werden soll.

VI-Server basiert auf TCP/IP, und ermöglicht Remote Procedure Calls[39]. Es ist die Grundlage für Methodenaufrufe auf entfernte Objekte. Neben der höheren Kommunikationsgeschwindigkeit bietet VI-Server den Vorteil der Multiplattformfähigkeit, aufgrund der Verwendung des TCP/IP Protokolls, das von allen Betriebsystemen unterstützt wird. In Verbindung mit Objekt Technologie Toolset (OTT) Komponenten ist dies die Basis zur Entwicklung hochperformanter, intelligenter, verteilter Applikationen in heterogenen LAN[40] und WAN[41] Umgebungen.

2.1.9 Vorteile/ Nachteile

Vorteile: Mit LabVIEW ist der Entwickler in der Lage, qualitativ hochwertige Programme zu schreiben.
Einzelne Module sind vor dem Integrationsschritt sehr gut zu testen. Da auch die Modularität an sich sehr hoch ist, kann ein einmal getestetes Modul immer wieder verwendet werden.
Verbesserungen und Weiterentwicklungen von LabVIEW finden zum großteil unsichtbar für den Entwickler statt. Das bedeutet, es gibt keine neue API in die sich der Entwickler einarbeiten muss. Dadurch wird die Wiederverwendbarkeit und Wartbarkeit von LabVIEW Programmen weiter erhöht.
Da sich der Entwickler nicht mit Implementierungsdetails wie Speicherverwaltung oder Zeiger auseinandersetzen muss, kann er seine ganze Aufmerksamkeit der Funktionalität des Programms, der Qualität und Zuverlässigkeit widmen. Auch die Wartbarkeit wird

[38] File Transfer Protocol/Hypertext Transfer Protocol
[39] siehe Glossar
[40] Local Area Network
[41] Wide Area Network

dadurch weiter verbessert.
Durch die einfache, intuitive Entwicklungsumgebung ist ein ausgeprägtes Rapid Prototyping[42] möglich. Der Entwickler kann die Sprache dadurch sehr schnell lernen, viele Möglichkeiten der Realisierung eines Programms ausprobieren, und so schnell eine effiziente Lösung finden.
Fehler sind leicht durch Error Cluster zu finden. Ein Error Cluster ist ein standardisierter, zusammengesetzter Datentyp, der in jedem SubVI ein Ein- und Ausgangsterminal belegen sollte. So kann man Error Cluster durch das gesamte Programm verdrahten, und Fehler sind sehr leicht zu finden.
Weiterhin ist ein Debuggingwerkzeug verfügbar, mit dem man Breakpoints und Probes setzen, sowie eine schrittweise Analyse durchführen kann. Es existiert eine Highlightfunktion, mit der sich der Datenfluss während der Ausführung verfolgen und beobachten lässt. Außerdem steht ein integriertes Profilerwerkzeug zur Verfügung, das Performanceanalysen ermöglicht.
Durch eine integrierte Dokumentationshilfe kann der Entwickler seine eigene Dokumentation während der Arbeit mit LabVIEW als Soforthilfe verwenden.
Außerdem existieren sehr viele VIs für die unterschiedlichsten Anwendungen auf dem Markt, die sich ein Entwickler zu nutze machen kann. Sie funktionieren out-of-the-box, kapseln schwieriges Spezialwissen, das sich der Entwickler nicht selbst aneignen muss, und erhöhen weiter die Zuverlässigkeit und Wartbarkeit eines Programms.
Insgesamt ist LabVIEW eine sehr gute Entwicklungsumgebung für Entwickler mit wenig Erfahrung im Bereich der Softwareentwicklung.

Nachteile: Bis Version 8.0 gab es in LabVIEW gar keine Objektorientierung. Bis heute gibt es immer noch keinen objektorientierten Ansatz inklusive Mehrfachvererbung und einem Ereignisflussmodel. LabVIEW beruhte auf dem traditionellen hierarchisch und modular aufgebauten Datenflussmodell. Erste Versuche waren ein rudimentär implementierter GOOP-Ansatz[43].
Die Entwicklung von LabVIEW Programmen basiert meist auf dem veralteten Top-Down Designansatz, wobei die graphische Benutzeroberfläche zentraler Bestandteil des Top-Level VIs ist, während den SubVIs keine strategische Rolle innerhalb der Problemdomäne zukommt. Die Applikationsschicht ist in herkömmlichen Ansätzen kaum berücksichtigt.
Ein Problem ist die Skalierbarkeit traditioneller LabVIEW Programme. Übersteigen mo-

[42] siehe Glossar
[43] Graphical Object Oriented Programming, siehe Glossar

nolithische[44] VIs eine gewisse Größenordnung, sind oft massive Performanceeinbrüche die Folge. Der Grund hierfür ist, das viele dieser Programme aus parallelen Zustandsautomaten mit einer Vielzahl von Schleifen und Schieberegistern bestehen. Die Parallele Abarbeitung erfordert zunehmend Resourcen. Um diese Problematik zu umgehen ist der großflächige Einsatz von Okkurenzen[45], Queues[46], Notifications[47], Referenzknoten, Semaphoren[48] und Rendezvous[49] nötig. Da aber viele LabVIEW Entwickler sehr wenig Erfahrung im Bereich der Softwareentwicklung haben, kann dies ein größeres Problem darstellen. Der Einsatz von Status- und Parameterinformationen, die als globale Variablen realisiert sind, erschwert dies zusätzlich, da eine vollständige Applikationskapselung damit unmöglich ist.
Dadurch wird die Wartung von großen Programmen erschwert, was in Industrie und Forschung aufgrund der langen Laufzeit von leicht 10 oder mehr Jahren allerdings nötig ist.

2.2 *CS*-Framework

In diesem Kapitel wird das *CS*-Framework detailliert beschrieben. Es enthält Abschnitte über die Entstehung von *CS* , die Umsetzung des objektorientierten Ansatzes, eine Zusammenfassung über die Basisklassen und den Ereignismechanismus.

2.2.1 Überblick

Motivation: Die Motivation zur Entwicklung eines Softwaresystems, das zur Entwicklung von experimentspezifischen Kontrollsystemen dient, entsteht aus der stark begrenzten Wiederverwendbarkeit bisher existierender Systeme, und dem daraus entstehenden Arbeitsaufwand, jedes Kontrollsystem von Grund auf neu zu entwickeln. Obwohl sich die Umsetzung einzelner Experimente stark voneinander unterscheiden, sind die Anforderungen von Kontrollsystemen im allgemeinen immer wieder sehr ähnlich, so dass sich auf dieser gemeinsamen Basis ein universelles Softwaresystem entwickeln lässt. Konkrete Anforderungen für ein solches System entstanden im Jahr 2001, als ein neues Kontrollsystem für die Experimente SHIPTRAP[50], ISOLTRAP[51], PHELIX[52] und LEBIT[53] benötigt

[44] siehe Glossar
[45] siehe Glossar
[46] siehe Glossar
[47] siehe Glossar
[48] siehe Glossar
[49] siehe Glossar
[50] Das Experiment an der GSI führt Massemessungen an schweren Ionen durch.
[51] Mehr Informationen http://rextrap.web.cern.ch/rextrap/ .
[52] Petawatt High-Energy Laser for Heavy Ion Experiments
[53] Low Energy Beam and Ion Trap

wurde.

Zwar kommen bei Experimenten häufig ähnliche oder sogar gleiche Geräte zum Einsatz, trotzdem beschränkt sich die Wiederverwendbarkeit von Softwarekomponenten meist auf die Treiber. Regelmechanismen, die eine Kommunikation unter den verwendeten Geräten voraussetzen, müssen immer wieder neu implementiert werden. Daher muss ein modulares System geschaffen werden, das durch einheitliche Schnittstellen die Kombination von Softwaremodulen je nach Bedarf des Experiments zulässt, ohne Abhängigkeit von den tatsächlich verwendeten Geräten [BeBH03].

Aus diesen Anforderungen entsteht die Idee einer Software im Format des Lego-Stecksystems. Das Konzept des *CS*-Frameworks macht eine Kombination von generischen Softwaremodulen mit experimentspezifischen Erweiterungen möglich, um ein funktionierendes Kontrollsystem zu konstruieren.

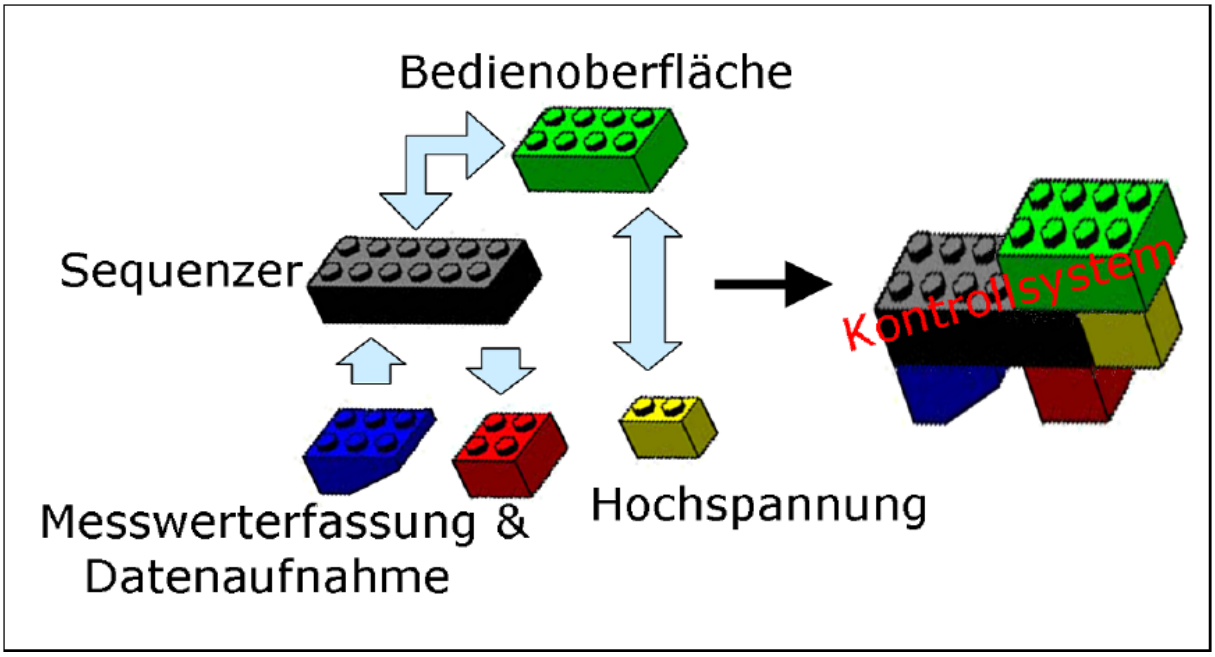


Abbildung 5: Legobaukastensystem von *CS* [Beck05]

Auch die Verbesserung der Wartbarkeit, der Dokumentation und der Erweiterung von Kontrollsystemen ist Ziel des *CS*-Frameworks. Diese Aufgabe wird von der Kontrollsystemgruppe der GSI übernommen, während die Anpassung des *CS*-Frameworks an konkrete Experimente die Aufgabe der Experimentatoren ist. Für die Lösung bestimmter Aufgaben, wie die Implementierung gerätespezifischer Software oder einer graphischen Benutzeroberfläche, stellt *CS* bereits Entwurfsmuster zur Verfügung. Auf der anderen Seite können experimentspezifische Entwicklungen, wie etwa gerätespezifische Software, die bei anderen Experimenten wieder verwendet werden kann, direkt in das *CS*-Framework integriert werden.

Da auch die in einem Experiment benutzten Komponenten selbst Teil der Forschung darstellen, kann man als weiteres Ziel des *CS*-Frameworks die Untersuchung der Erweiterungsmöglichkeiten bezüglich der Skalierbarkeit eines auf LabVIEW basierten Systems angeben.

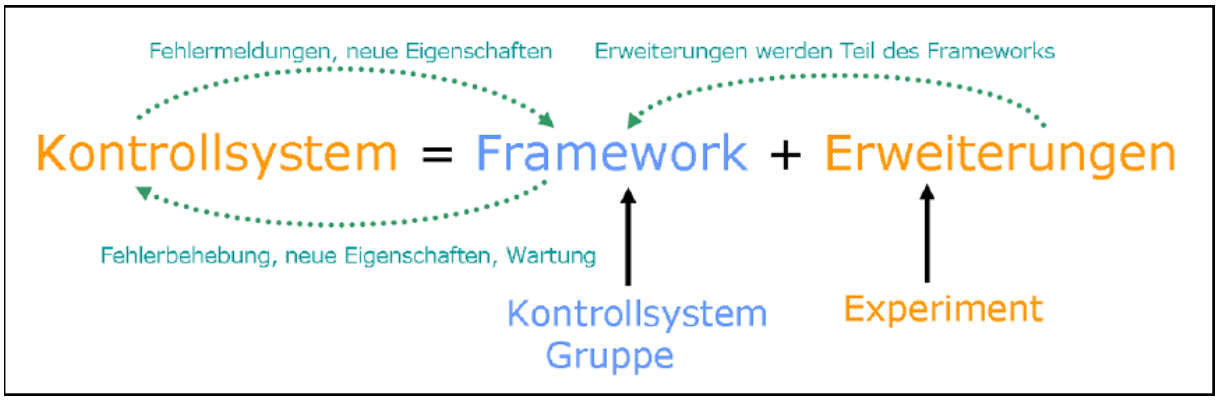


Abbildung 6: Integration von *CS* und experimentspezifischen Erweiterungen zu einem Kontrollsystem.

Anforderungen: Die Anforderungen an das *CS*-Framework sind freie Skalierbarkeit, leichte Erlern- und Beherrschbarkeit der Entwicklungsumgebung, Archivierbarkeit der Mess- und Konfigurationsdaten und die Verteilbarkeit der Softwarekomponenten über ein Netzwerk von Rechnern.

Ein Experimentaufbau befindet sich im ständigen Wandel. Geräte und Konfiguration ändern sich je nach Ziel des Experiments, wobei verschiedene Geräteschnittstellen und Protokolle angesprochen werden. Geräte werden erst auf Erfüllung ihrer Anforderungen getestet, um später dann in großer Stückzahl zum Einsatz zu kommen. Daher muss die Software möglichst frei skalierbar sein.

Viele der beteiligten Entwickler eines Kontrollsystems sind Studenten und Doktoranden, die dem Experiment nur eine beschränkte Zeit zur Verfügung stehen. Daher muss der Umgang mit der Entwicklungsumgebung möglichst einfach und schnell zu erlernen sein. An dieser Stelle ist die Dokumentation der Software von besonderem Interesse, da das Wissen über die entwickelten Kontrollsysteme der zentralen Gruppe eines Experiments verfügbar gemacht werden muss. Sonst droht der Verlust dieses Wissens, wenn ein Entwickler das Experiment verlässt.

Um nach Durchführung des Experiments Analysen anfertigen zu können, ist die Archivierung der Mess- und Konfigurationsdaten nötig. Die Überwachung von Sensordaten durch Alarmfunktionalität ist zentraler Bestandteil eines jeden Kontrollsystems. Daher müssen SCADA Funktionalitäten bereits im Kern des Kontrollsystems unterstützt werden.

Aus Sicherheitsgründen ist die Fernsteuerung der an einem Experiment beteiligten Hardware nötig, da ein Zugang zum Experiment durch die starke Strahlenbelastung nicht möglich ist. Ein verteiltes Kontrollsystem zur Steuerung der Hardware ist daher unerlässlich. Auch die Skalierbarkeit wird dadurch weiter erhöht, da diese nun nicht von der Rechenleistung eines einzelnen Rechners beschränkt wird.

Umsetzung: Das *CS*-Framework basiert bis auf die Netzwerkkommunikation vollständig auf LabVIEW, da dies die oben genannten Anforderungen sehr gut erfüllt. LabVIEW dient zur Entwicklung von Anwendungen im Bereich der Mess- und Automatisierungstechnik und stellt daher eine große Auswahl an Treibern und unterstützten Schnittstellen zur Verfügung. Die graphische Datenflussprogrammierung ist intuitiv und leicht zu erlernen. SCADA Funktionalität wird vom DSC Modul unterstützt. In Kombination mit DIM[54] ist ein Mechanismus realisiert, über den *CS*-Objekte rechnerübergreifend miteinander kommunizieren können, so dass auch die Anforderung eines verteilten Systems erfüllt ist [BBGH05].

Status: In der aktuellen Version von *CS* sind die durch Semaphoren geschützte Verwaltung von Objektattributen und die Einbettung des rechnerübergreifenden DIM-Ereignismechanismus beinhaltet.
Es existieren ca. 60 Geräteklassen für verschiedene Gerätetypen unterschiedlicher Hersteller. Diese erlauben den freien Zugang zu Funktionengeneratoren, Delay-Gate Generatoren, Oszilloskopen, Multikanalskalierer, Schrittmotorkontrollen, Hochspannungsgeräte, und Hardware wie ADCs[55], DACs[56] und DIOs[57]. Der Quellcode ist GPL[58] lizenziert.
CS wurde unter Windows entwickelt und erfolgreich nach Linux portiert.
Kontrollsysteme, die mit *CS* entwickelt wurden, sind heute bei Experimenten der GSI, des CERN[59] und der MSU[60] zu finden.

2.2.2 Objektorientierung

Um von den Vorteilen objekorientierter Techniken zu profitieren, sollten diese zusammen mit LabVIEW zur Anwendung kommen. Als Konsequenz wurde ein neuer objektorien-

[54]Distributed Information Management System, Kapitel 2.2.4
[55]Analog-to-Digital-Converter
[56]Digital-to-Analog-Converter
[57]Digital Input Output
[58]General Public License
[59]Europäische Organisation für Kernforschung, siehe Glossar
[60]Michigan State University

tierter Ansatz in *CS* implementiert. Dieser erlaubt die Konstruktion und Zerstörung von Objekten zur Laufzeit, wobei ein Werkzeug zur Klassennavigation und zur Vererbung hinzugefügt wurden.
Der Kern des *CS*-Frameworks besteht aus Basisklassen, die z.B. klassenspezifische Attribute, Methoden mit Zugang zu diesen Attributen und das Sperren um deren Integrität zu schützen, unterstützen. Klassen, die spezielle Gerätetypen repräsentieren, erben dann von diesen Basisklassen, wobei jedes einzelne Gerät durch eine Instanz seiner Klasse dargestellt wird. Bei Instanzen einer *CS* Klasse handelt es sich nicht um passive Datencontainer, die Methoden zur Datenmanipulation darstellen, als vielmehr um aktive Prozesse, die miteinander über Ereignisse interagieren können. Aktive *CS*-Objekte sind Threads, deren generisches Verhalten in Form von Basisklassen des *CS*-Frameworks bereitgestellt wird.
Zu beachten ist allerdings, dass es sich bei *CS* nicht um Objektorientierung im Sinne von Java oder C++ handelt. In diesem Zusammenhang spricht man von einem objektorientierten Ansatz, bei dem objektorientierte Techniken zum Einsatz kommen. Da der Compiler von LabVIEW keine Objektorientierung unterstützt, liegt es am Entwickler, sich an Konventionen des *CS*-Frameworks zu halten. Dabei wird er von einigen Entwurfsvorlagen, anhand derer er seine Klassen aufbauen kann, unterstützt.

Klassen und Instanziierung: Eine *CS*-Klasse besteht aus Attributen, Konstruktor, Destruktor, und Methoden, die auf den Attributen operieren. Zusätzlich hat jede *CS*-Basisklasse ein VI Template[61], das den aktiven Code der Klasse enthält. Im wesentliche ist eine *CS*-Klasse eine Sammlung von VIs, die die Methoden der Klasse bilden, sowie einer Control, die die Attribute der Klasse darstellt, und einer LabVIEW Bibliothek, die alle zugehörigen Dateien referenziert und deren Sichtbarkeit speichert. Diese Dateien sind im selben Verzeichnis gespeichert, welches den Namen der Klasse trägt.
Wird eine Klasse instanziiert, so wird der Konstruktor der Klasse aufgerufen, der zur Laufzeit eine Instanz des VI Templates in den Speicher lädt, und eine eindeutige Referenz unter Verwendung der VI-Server Funktionen von LabVIEW erzeugt.

[61] siehe Glossar

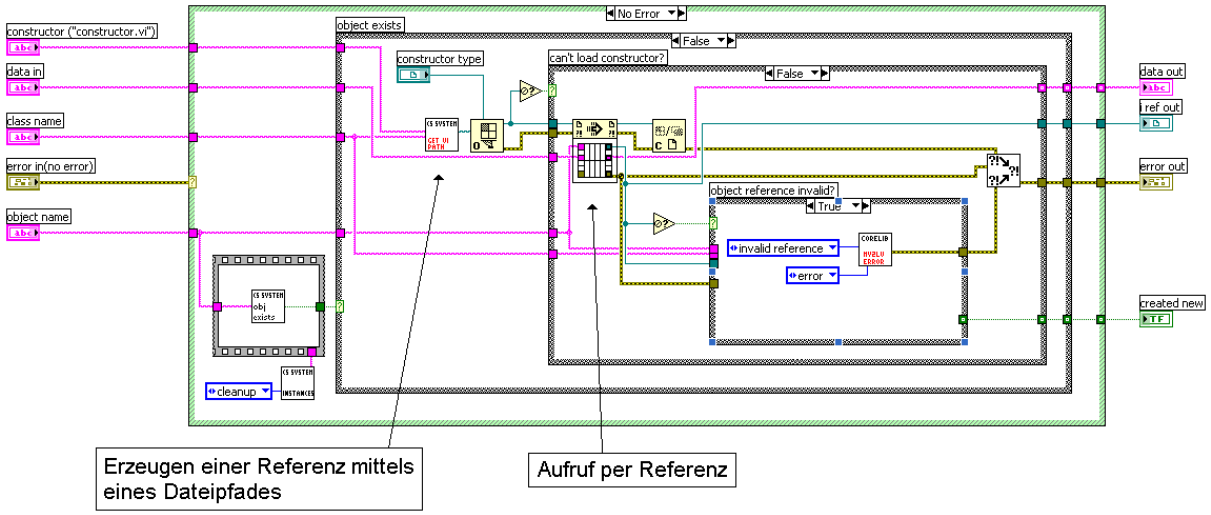


Abbildung 7: *CSSystem._new.vi* dient dem dynamischen Aufruf des Konstruktors einer Klasse.

Die Referenz auf die Klasse wird in deren Attributen gespeichert, um sie im Destruktor wieder schließen und den allokierten Speicher frei geben zu können. Zusätzlich wird sie als FGV[62] gespeichert, da LabVIEW diese sonst automatisch wieder aus dem Speicher löscht.
Methoden einer Klasse werden als abgeschlossene VI im Verzeichnis der Klasse gespeichert. Auch Konstruktor und Destruktor sind, genau wie normale Methoden, eigene VIs. Hier existiert folgende Namenskonvention: "KLASSENNAME.METHODENNAME.vi". Das *CS*-Framework verwaltet die Verzeichnisstruktur der *CS*-Klassenbibliothek, sowie die Dateipfade der VIs, die zum Aufruf der Methoden benötigt werden.

Objekt und Attributverwaltung: Das *CS*-System speichert alle Referenzen auf Objekte zusammen mit ihren Objekt- und Klassennamen in einem Shiftregister, das beim Erzeugen oder Zerstören einer Instanz aktualisiert wird.
Zur Verwaltung der Attribute werden auch Shiftregister verwendet, wobei hier pro Klasse ein VI existiert, das die Attribute aller Instanzen dieser Klasse verwaltet.

[62] Funktionale Globale Variable

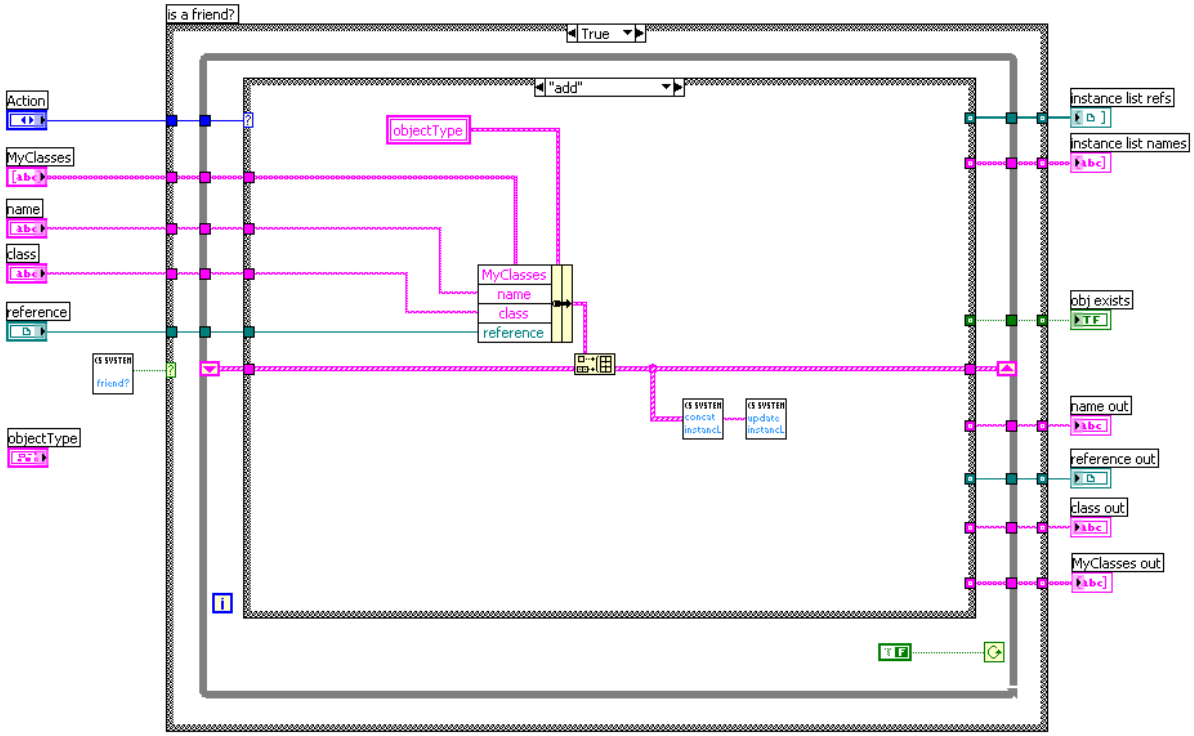


Abbildung 8: *CSSystemLib.instances.vi* verwaltet die Objekte und Referenzen innerhalb eines lokalen CS-Systems.

Weiterhin benötigt man noch einen geschützten Zugriff auf Attribute, da *CS*-Objekte aus mehreren Threads bestehen können, die asynchron nebeneinander laufen. Durch die Verwendung der von LabVIEW bereitgestellten Semaphoren wird der parallele Zugriff auf Attribute vermieden und deren Konsistenz sichergestellt.

Assoziationen: Assoziationen können in LabVIEW auf verschiedene Arten realisiert werden.
Ein direkter Aufruf der Methode einer anderen Klasse als assoziative Beziehung ist möglich, wird aber aufgrund der verteilten Applikationen im *CS*-Framework selten genutzt.
Assoziative Beziehungen werden in *CS*-Klassen nicht in Form eines Attributs des Typs der assoziierten Klasse gespeichert. In LabVIEW ist nur die Speicherung von einfachen Datentypen, die von LabVIEW unterstützt werden, möglich. Zwar können diese frei zu Struct ähnlichen Clustern oder als Arrays zusammengefasst werden, bleiben aber Datencontainer ohne programmatische Referenz auf die Assoziation, die sie abbilden.
Üblich ist die Realisierung einer Assoziation durch den Ereignismechanismus. Dabei werden Informationen über das Netzwerk verschickt, die in den primitiven oder zusammengesetzten Datencontainer der assoziierten Klasse gespeichert werden. Der Ereignismecha-

nismus implementiert sowohl das Observer Pattern[63] als auch das Command Pattern[64]. Definiert werden allerdings nur Ereignisse, die empfangen werden können, wohingegen jedes Objekt an jedes andere beliebige Kommandos schicken kann. Erst zur Laufzeit werden diese Assoziationen im Zuge der Konfiguration einer *CS*-Anwendung aktiv umgesetzt. Dies ermöglicht eine große Flexibilität von verteilten Anwendungen einerseits, andererseits erschwert es die Dokumentation eines *CS*-Systems, das nur aus Quellcode besteht, ohne Wissen über das zugrunde liegende Modell des Entwicklers. Daher ist die nachträgliche Bestimmung einer Assoziation aus dem Quellcode einer *CS*-Klasse nicht möglich.

Vererbung: Vererbung wird über den geschachtelten Aufruf von Konstruktoren realisiert. Ruft ein Konstruktor einen oder mehrere andere Konstruktoren auf, so erbt die aufrufende Klasse von den Aufgerufenen. Kommt es durch Mehrfachvererbung zu doppelten Konstruktoraufrufen derselben Oberklasse, so wird dieser nur einmal ausgeführt.
Im Gegensatz dazu wird eine neue Instanz einer Klasse, in oder außerhalb eines Konstruktors, durch den Aufruf des new Operators erzeugt. Dies stellt eine Möglichkeit dar, Assoziationen in einer *CS*-Klasse zu realisieren. Im Falle einer Komposition muss diese allerdings auch wieder durch Benutzung des delete Operators gelöscht werden.
Da das manuelle Vererben einen relativ großen Zeitaufwand benötigt, beinhaltet das *CS*-Framework ein Werkzeug, *CSClassUtilities.InheritClass.vi*, mit dessen Hilfe Vererbungsbeziehungen automatisch erstellt werden können. Hierzu wählt man eine Basisklasse aus, gibt den Namen der neuen Klasse an, und dann erzeugt das VI die neue Klasse komplett mit Ordnerstruktur, und allen geerbten Methoden.

2.2.3 Klassenbibiliothek

MVC: Die Architektur des *CS*-Framework entspricht der in der Softwaretechnik üblichen MVC-Architektur: Model, View, Control. Hier werden die Aufgaben einer Software in drei verschiedene Pakete eingeteilt. Durch die Unterteilung in diese drei Pakete sind Änderungen im Programm leichter durchzuführen, da Änderungen einzelner Pakete die Funktion der anderen Pakete nicht beeinflussen.

- Im Modell befinden sich Klassen, die Hauptsächlich der Datenhaltung dienen. Sie bilden das Modell, das die Software abzubilden versucht.

  Im *CS*-Framework ist das Modell eine Abbildung der verwendeten Geräte. Hier wird die Treibersoftware in Geräteklassen gekapselt, um einheitliche Schnittstellen für

[63] siehe Glossar
[64] siehe Glossar

den Zugriff im *CS*-Framework zu schaffen. Eine Instanz einer solchen Geräteklasse repräsentiert eine Hardwarekomponente, wobei es zur Laufzeit möglich ist, durch Instanziierung weiterer Objekte das Kontrollsystem an die tatsächliche Hardware anzupassen.

- Unter View versteht man die Benutzeroberfläche. Es ist die Schnittstelle zum Anwender.

- Das Control-Paket besteht aus Klassen, die die Interaktion anderer Klassen steuern, selbst aber fast keine Daten speichern. Im *CS*-Framework steuern Klassen dieses Paketes unter Verwendung von Geräteinstanzen deren Funktionalität. Hier werden Gerätefunktionen in einen für den Anwender sinnvollen Zusammenhang gesetzt.

Systemklassen: Das *CS*-Framework stellt einige Basisklassen bereit, auf denen der Entwickler seine eigenen Klassen aufbauen kann. Diese sind abstrakt und haben virtuelle Methoden, die erst in den Unterklassen implementiert werden. Abstrakt bedeutet in diesem Kontext allerdings nur, dass eine abstrakte Klasse nicht instanziiert werden kann. Es gibt von Seiten der abstrakten Klasse selbst keinen Hinweis darauf, dass die Klasse abstrakt ist. Vielmehr muss man die entsprechende Kenntnis des *CS*-Systems soweit mitbringen. Eine virtuelle Methode ist in diesem Zusammenhang eine zur Vererbung konstruierte, inhaltsleere Methode deren Schnittstellen, also deren Ein- und Ausgangsparameter, definiert sind. Diese Methoden befinden sich in einem separaten Unterordner der Klasse, mit dem Namen “inheritance”.

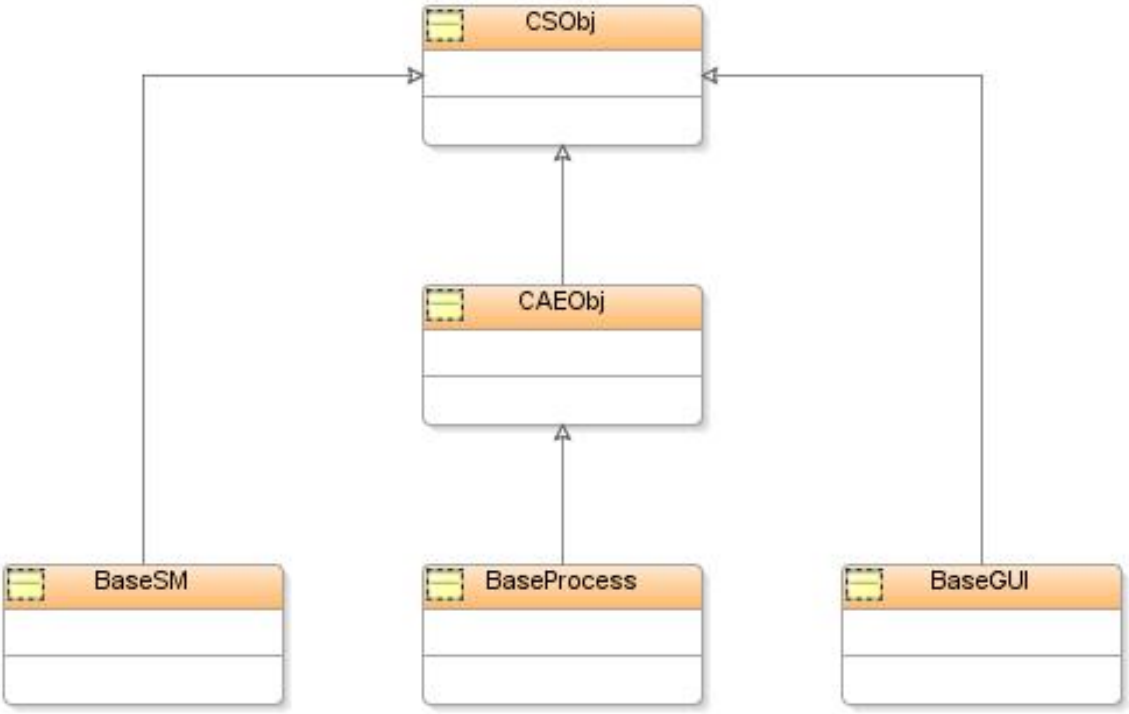


Abbildung 9: Basisklassen des *CS*-Framework

- CSObj ist die Basisklasse aller *CS*-Klassen. Sie erzeugt in ihrem Konstruktor eine Referenz auf die Instanz ihres VI-Templates, die an die abgeleiteten Objekte zur eindeutigen Identifizierung zurückgegeben werden.

  Weiterhin stellt CSObj Zugriffsmethoden bereit, um Objekt-Attribute zu lesen oder zu schreiben, wahlweise geschützt durch Semaphore oder nicht.

  CSObj ist keine aktive Klasse, da sie nur der Erzeugung einer Objektreferenz dient, selbst aber keinen weiteren Code beinhaltet.

- CAEObj ist eine Unterklasse von CSObj. Sie ist die Basisklasse aller aktiven Klassen und unterstützt den Ereignismechanismus aktiver Threads. Unterstützt werden das Erzeugen, Empfangen und Senden von Nachrichten. Unterstützte Nachrichten sind gepufferte Messages, die via LabVIEW-Message-Queues versendet werden, und ungepufferte Notifications, versendet durch LabVIEW-Notifikationen. Andere Typen von Nachrichten können allerdings vom Entwickler hinzugefügt werden.

  Diese Klasse kapselt den größten Teil des DIM-Ereignismechanismus durch Verwendung von C-Bibliotheken.

- Die BaseProcess-Klasse ist die Basisklasse für alle Geräte-Klassen. Jede Instanz der BaseProcess-Klasse registriert sich beim so genannten Super. Der Super-Prozess

selbst ist auch eine BaseProcess-Klasse, die zum *CS*-System gehört und alle anderen BaseProcess-Klassen verwaltet.

Als Unterklasse von CAEObj verfügt die BaseProcess-Klasse über Methoden zur Ereignissteuerung.

Sie hat zwei Threads, die beide über eine Watchdog-Funktionalität[65] verfügen:

- Die periodische Schleife: Sie dient zur Ausführung periodischer Aktionen, wie z.B. das Auslesen eines Wertes in festen Intervallen. Sie wird vom BaseProcess selbst überwacht. Bei jeder Iteration wird die Alarmzeit aktualisiert, wobei ein Alarm ausgelöst wird, wenn die aktuelle Zeit größer als die Alarmzeit ist, andernfalls nicht. Die virtuelle Methode, die von der Schleife aufgerufen wird, heißt ProcPeriodic. Sie muss von der Unterklasse implementiert werden, um das gewünschte Verhalten der periodischen Aktion zu bestimmen.
- Die Ereignisschleife: Sie ermöglicht die Kommunikation zwischen Objekten. Sie dient als Listener, indem sie auf ankommende Ereignisse wartet. Überprüft wird sie direkt vom Super. Wie auch bei der periodischen Schleife gibt es virtuelle Methoden, die im Falle eines eintreffenden Ereignisses aufgerufen werden, und daher von den Unterklassen implementiert werden müssen. In der Methode ProcEvents werden die verfügbaren Ereignisse, auf die die Klasse reagiert, definiert, und in ProcCases wird ein entsprechendes Verhalten der Klasse für den Ereignisfall implementiert.

Außerdem kann der BaseProcess beim Erzeugen von Instanzen eine Datenbank auslesen, in der Datensätze für Prozesse gespeichert sind. Da BaseProcess die Oberklasse zu allen Geräten ist, enthält die Datenbank meist Konfigurationsdaten dieser Geräte, wie z.B. Busadressen.

- BaseSM ist die Oberklasse aller Zustandsmaschinen in *CS*. Sie ist Unterklasse von CSObj. BaseSM-Klassen werden als Hilfsklassen zur Sicherheit von Abläufen durch klar definierte Zustandsübergänge verwendet.

  BaseSM ähnelt der der Klasse BaseProcess, da es auch hier eine Schleife gibt, die auf Ereignisse wartet, und dann die virtuelle Methode ProcState aufruft, die eine Unterklasse implementieren muss, um ein entsprechendes Verhalten zu gewährleisten.

[65] siehe Glossar

- Die BaseGUI-Klasse ist Unterklasse von CSObj und Oberklasse aller *CS*-Benutzeroberflächen. Die Benutzeroberfläche kann sich frühestens nach Aufruf des Konstruktors öffnen, und zerstört sich selbst nach Beenden der Schleife, die die Interaktion mit dem Benutzer regelt.

  Eine BaseGUI-Klasse beinhaltet einen Thread für das GUI, sowie Methoden um auf Parameter des GUIs[66], wie z.B. Größe und Position, zugreifen zu können.

2.2.4 Ereignismechanismus

Eine der grundlegenden Ideen des *CS*-Frameworks ist es, den objektorientierten Ansatz mit einem ereignisgesteuerten Kommunikationsmechanismus zu verknüpfen. Objekte kommunizieren durch das Verschicken und Empfangen von Ereignissen, und nur sehr selten durch den direkten Aufruf der Methode einer anderen Klasse.
Kindklassen von CSObj sind in der Lage, Ereignisse zu verwalten. Erbt eine Klasse von BaseProcess, beinhaltet sie sogar schon eine vordefinierte Struktur zur Ereignisbehandlung. Außerdem unterstützt die Klasse BaseProcess drei Arten von Ereignissen:

- Einfache Ereignisse werden von Sender zum Empfänger geschickt.
- Beim Verschicken von synchronen Ereignissen wartet der Sender auf eine Antwort vom Empfänger, bis sein Thread fortgesetzt wird.
- Wird ein asynchrones Ereignis verschickt, so sendet der Empfänger nach Empfang des Ereignisses eine Antwort an ein drittes Objekt, das vom Sender vorher spezifiziert wurde, wobei dies natürlich auch wieder der Sender selbst sein kann.

Ereignisse enthalten Informationen über den Empfänger, wie den Namen des angesprochenen Objektes und den Namen der aufzurufenden Methode, einen Timeout, und die zu übertragenden Parameter als Bytearray.
Die Verwendung von Ereignissen bietet Vorteile. Während der Implementierungsphase muss nicht entschieden werden, welche Methoden von welchen anderen Klassen aufgerufen werden sollen, da Ereignisse von einem beliebigen Objekt an alle anderen Objekte gesendet werden können. Dieses System bietet eine enorme Flexibilität, bis hin zur kompletten Umkonfiguration eines Kontrollsystems zur Laufzeit. Weiterhin ist es möglich, Ereignisse über ein Netzwerk zu verschicken, was ein verteiltes Kontrollsystem möglich macht. Daher spielt es keine Rolle auf welchen Knoten Objekte erzeugt werden oder wo ein Gerät tatsächlich angeschlossen ist.

[66] Graphical User Interface

Durch die Möglichkeit, Ereignisse direkt zwischen beliebigen Objekten zu verschicken, entsteht ein gewisser Overhead bei den Objekten, da der Empfänger immer die Gültigkeit eines Ereignisses, die Parameterkonsistenz und Datentypen prüfen muss. Auf der anderen Seite entsteht dadurch kein Engpass in der Anwendung, da auf eine zentrale Ereignisverwaltung verzichtet werden kann.

DIM: DIM ist ein Kommunikationssystem für verteilte Systeme und inhomogene Umgebungen, das eine netzwerktransparente Interprozesskommunikationsebene bereitstellt. Es wurde am CERN entwickelt und basiert auf einer Client-Server-Architektur.
Der grundsätzliche Ansatz von DIM ist das Konzept des Service (also der Leistung, des Dienstes). Server bieten ihren Clients Services an, die normalerweise aus einem Datensatz von beliebigem Datentyp und Größe bestehen der durch einen Namen identifiziert werden kann - ein so genannter Named Service. Die Namensgebung eines Services ist dabei uneingeschränkt.
Um eine möglichst hohe Transparenz, sowie eine einfache Wiederherstellung nach Abstürzen oder Umzügen von Servern zu gewährleisten, existiert in DIM der so genannte Nameserver (DNS). Server veröffentlichen ihre angebotenen Services, indem sie sie auf dem Nameserver registrieren. Clients können dann einen Service abonnieren, indem sie den Nameserver nach dem Server fragen, der den angeforderten Service anbietet, und diesen Server dann direkt mit dem Typ des Services und den Aktualisierungsparametern kontaktieren. Der Nameserver hält sein Verzeichnis aller Server und Services die im System verfügbar sind immer aktuell.

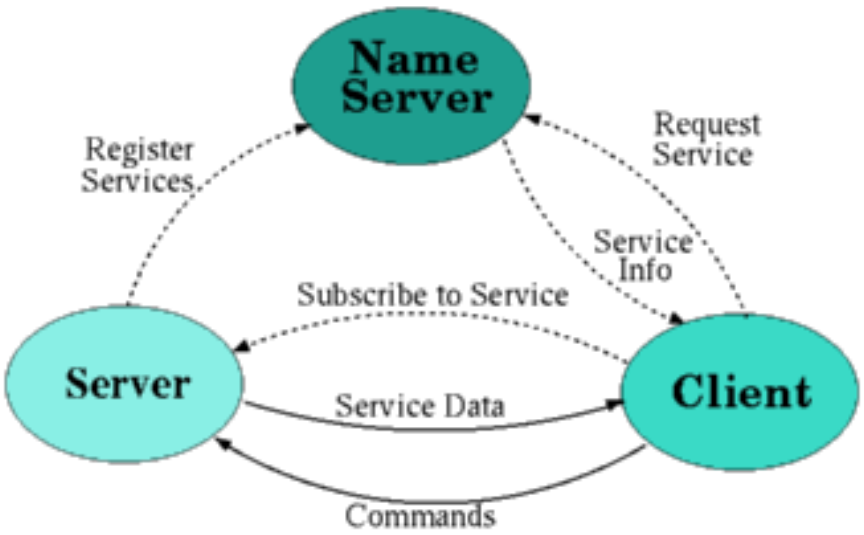


Abbildung 10: Hier sieht man wie die DIM Komponenten Server, Client und Nameserver interagieren [Gasp06].

Stürzt ein Server oder ein Nameserver im System ab oder stirbt, werden alle verbundenen Prozesse darüber benachrichtigt, können sich aber später wieder verbinden, sobald die beendeten Prozesse wieder gestartet wurden. Dieses Feature erlaubt eine einfache Wiederherstellung, einen einfachen Umzug von Servern, indem man den Prozess auf einem Rechner beendet und ihn auf einem anderen wieder startet, und die Möglichkeit, die Prozessorlast zwischen verschiedenen Workstations auszubalancieren.
DIM ist ereignisgesteuert, und wurde für die Betriebssysteme Windows, Linux, VMS, Unix und die Echtzeitplattformen OS9, LynxOS und VxWorks implementiert. DIM hat Schnittstellen für die Programmiersprachen C, C++, Java und Fortran. Die hohe Performance und die Fähigkeit eine Vielzahl verschiedener Plattformen zu verbinden machen DIM zu einem idealen Werkzeug, um LabVIEW mit anderen Anwendungen zu verknüpfen. In LabVIEW wird dies durch eine Wrapper-Bibliothek der C-Schnittstelle von DIM instrumentalisiert.

Leistungsfähigkeit: Mit DIM ist es möglich die volle Bandbreite der zur Verfügung stehenden Netzwerkverbindung auszunutzen. Die Performance wird nicht vom Protokoll limitiert, sondern nur durch die verwendete Verbindung. Da keine eindeutigen Computernamen, sondern nur eindeutige Servicenamen benötigt werden, ist es möglich, mehrere *CS*-Instanzen pro Computer zu starten.

3 Verwandte Arbeiten

Im folgenden Kapitel werden zwei verwandte Arbeiten beschrieben.
Die erste Arbeit ist direkt verwandt. Hier wird das Konzept der Bachelor-Thesis für das GOOP Toolkit von Endevo realisiert, das ähnlich dem *CS*-Framework einen Ansatz zur Objektorientierung in LabVIEW bietet.
Die zweite Arbeit ist nicht direkt mit der Bachelor-Thesis verwandt. Es wird allerdings ein Lösungsansatz beschrieben, der auch für die Bachelor-Thesis diskutiert wird. Hier werden schon im Vorfeld positive sowie negative Aspekte dieser Lösung deutlich.

3.1 Endevo UML Modeller

3.1.1 Überblick

Der UML Modeller von Endevo ist ein visuelles Designwerkzeug. Es ist in LabVIEW zur Modellierung von Systemen integriert, und existiert ab LabVIEW Version 7.1. Zur Codegenerierung wird der Endevo GOOP Wizard 3 zusätzlich benötigt. Der Endevo UML Modeller ermöglicht Roundtrip-Engineering nur für GOOP konformen Code. Das bedeutet er kann nicht mit dem *CS*-Framework zusammen verwendet werden, da sich die Ansätze zur Realisierung der Objektorientierung hier voneinander unterscheiden.
Entwickler bekommen mit dem UML Modeller eine werkzeuggestützte Notation, um ihre Test- und Messsysteme zu beschreiben und Design sowie Architektur zu diskutieren. Dadurch wird eine Effizienzsteigerung beim Modellieren gegenüber herkömmlichen Methoden mit Hilfe von White Boards oder Mehrzweckzeichenprogrammen erreicht.
Endevo UML Modeller unterstützt UML Use Case, Sequenz-, Klassen-, Paketdiagramme und Zustandsautomaten.
Codegenerierung ist aus einem Klassendiagramm heraus möglich, wobei der GOOP Wizard von Endevo zur Generierung von Klassen und Reverse-Engineering verwendet wird. Damit können nur GOOP konforme Klassen generiert werden. Der Endevo UML Modeller kann daher nicht zusammen mit dem *CS*-Framework eingesetzt werden.
Die Unterstützung der Codegenerierung, des Reverse Engineering sowie der Synchronisation stellen die Konsistenz der Systembeschreibung zum Code während der Entwicklung sicher.

3.1.2 Eigenschaften

Endevo UML Modeller unterstützt Use Case, Sequenz-, Klassen-, Paketdiagramme und Zustandsautomaten.

Dadurch entsteht der Vorteil, die Architektur eines Systems vor der Implementierung in einem einzigen Werkzeug zu erstellen, verschiedene Darstellungen auf verschiedenen Ebenen zu skizzieren, und diese während der Entwicklung anzupassen. Allerdings bleiben Use Case, Sequenz-, Paketdiagramme und Zustandsautomaten auf der Ebene eines Zeichenprogramms. Hier besteht nicht die Möglichkeit, über eine reine Spezifikation hinaus Code zu generieren oder andere Implementierungshilfen zu bieten.
Aus dem vom UML Modeller unterstützten Klassendiagramm hingegen kann das Werkzeug mit Hilfe des GOOP Wizards GOOP konformen Code, sowie aus bestehendem GOOP-Code ein Diagramm generieren. Weiterhin können Modell und Code synchronisiert werden.

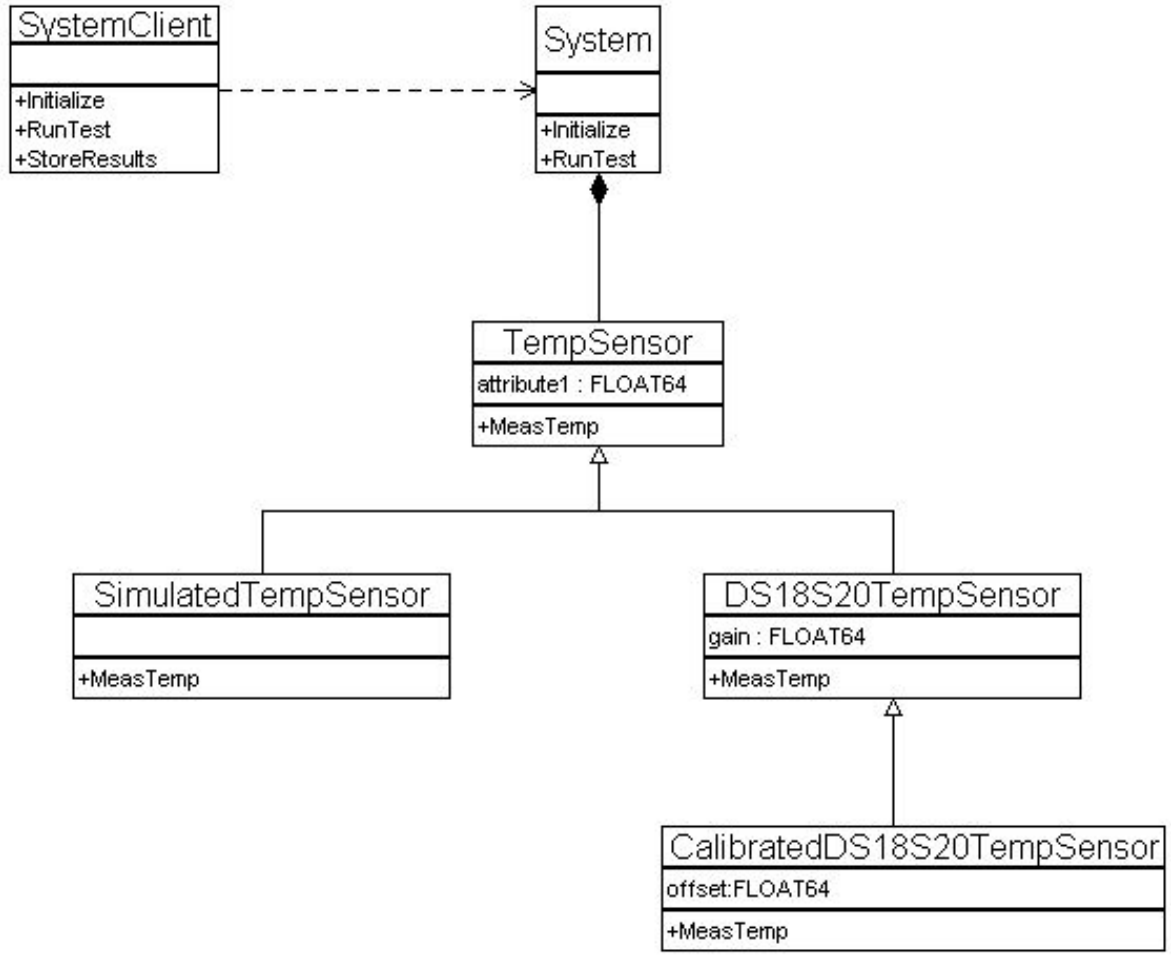


Abbildung 11: Klassendiagramm des Endevo UML Modellers

Im Klassendiagramm werden die Elemente Klasse, Interface, Paket, Abhängigkeit, Attribut, Methode, Assoziation, Vererbung und Notizen unterstützt.
Zwar werden Elemente beim Erstellen eines Diagramms aus bestehendem Code automatisch platziert, ein Layout Manager existiert jedoch nicht, so dass der Entwickler die Diagrammelemente von Hand platzieren muss.

Mit dem GOOP Wizard von Endevo ist der UML Modeller in der Lage, GOOP konformen Code zu generieren. Dazu werden automatisch Klassen, Methoden, Attribute und Vererbungsbeziehungen erzeugt. Ähnlich wie beim *CS*-Framework besteht eine so generierte Klasse aus einer Sammlung von VIs, die zusammen in einem Verzeichnis gespeichert sind, wobei Elemente zur Unterscheidung ihrer Sichtbarkeit in entsprechenden Unterordnern gespeichert sind. Automatisch generiert werden die Methoden *new*, *delete*, sowie *get* und *set* Methoden passend zu den modellierten Attributen. Ein- und Ausgangsparameter von Methoden können allerdings weder modelliert noch erzeugt werden. Assoziationen werden durch einen Mechanismus realisiert, der eine Referenz auf die entsprechende Klasse erzeugt, die im weitern Programm dann verwendet wird. Hier wird also lokal eine neue Instanz der assoziierten Klasse erschaffen.
Außerdem ist es möglich, aus bereits existierenden Systemen, die GOOP-konform sind, Klassendiagramme zu erzeugen. Dies beinhaltet auch Systeme, die nicht von Beginn an mit dem UML Modeller erarbeitet worden sind. Weiterhin gibt es dadurch die Möglichkeit zur Synchronisierung des Diagramms mit dem Code. Da auch der Code mit dem Diagramm synchronisiert werden kann, ist das Werkzeug zum Roundtrip-Engineering fähig. Zwar bietet das Werkzeug keine automatisierten Hinweise auf nicht synchronisierte Strukturen in Diagramm oder Code an, beinhaltet dafür aber eine eigene Ansicht zur Synchronisation, ähnlich wie die Synchronisationsansicht von eclipse bei CVS Projekten.

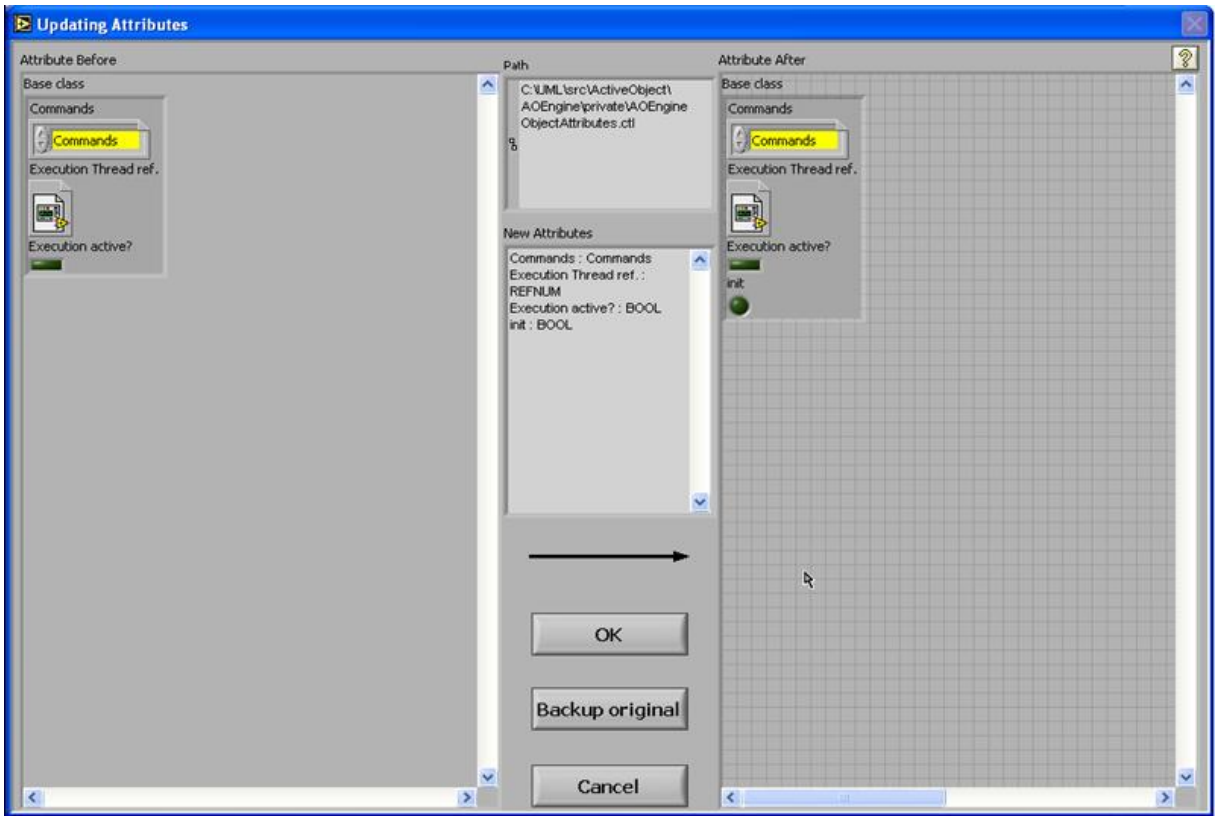


Abbildung 12: Synchronisation von Code und Modell

Der UML Modeller bietet auch die Möglichkeit Klassen direkt aus dem Werkzeug heraus zu starten, sowie Diagramme zu drucken und im jpg-Format oder png-Format zu exportieren.
Weiterhin benutzt der UML Modeller die Kontexthilfe von LabVIEW. Diese wird noch mit zusätzlichen Informationen über das Werkzeug selbst und über die von den Diagrammen benutzten Notationen ausgestattet.

3.1.3 Vergleich zur Arbeit der Bachelor-Thesis

Der Endevo UML Modeller ist das einzige Programm, das man direkt mit der Arbeit der Bachelor-Thesis vergleichen kann. Als einziges auf dem Markt erhältliches Programm bietet es die Möglichkeit der automatisierten Dokumentation und sogar des Roundtrip-Engineerings für objektorientierte LabVIEW-Systeme. Diese Fähigkeit beruht weitestgehend auf der Konvention, GOOP zur objektorientierten Entwicklung von Systemen mit LabVIEW einzusetzen. Daher wird auch der GOOP Wizard von Endevo zur Codegenerierung und automatisierten Darstellung des Klassendiagramms benötigt. Da das *CS*-Framework hingegen auf völlig anderen Konventionen und Konzepten zur Realisierung objektorientierter Programmierung basiert, ist der Endevo UML Modeller hier nicht verwendbar.
Der UML Modeller ist völlig in LabVIEW integriert. Das heißt, er ist selbst ein LabVIEW-Programm, und lässt sich nach der Installation als Plug-in starten. Damit bietet er die Möglichkeit, gleichzeitig am Code und am Modell in derselben Entwicklungsumgebung arbeiten zu können.
Die Motivation ist ähnlich gelagert wie bei der Bachelor-Thesis. Zur Dokumentation, aber auch zur Steigerung der Effizienz durch gutes Design und der Möglichkeit der Codegenerierung soll dem Entwickler objektorientierter LabVIEW-Systeme ein Werkzeug zur Verfügung gestellt werden.
Zwar werden vom UML Modeller mehrer Diagramme unterstützt, doch bis auf das Klassendiagramm kommen sie nicht über die Funktionalität eines Designwerkzeuges hinaus. Diese Funktionalität wird allerdings auch von den meisten anderen Case-Tools unterstützt, so dass es sich hier nur um eine Ergänzung zum Konzept des Werkzeuges handelt. Klassendiagramme hingegen lassen sich aus vorhandenem Code generieren und Code lässt sich aus den Klassendiagrammen generieren. Die Fähigkeit zur Synchronisation des Diagramms und des Codes macht Roundtrip Engineering möglich. Demgegenüber führt die Voraussetzung einer möglichst großen Flexibilität an das *CS*-Framework zu sehr wenigen Konventionen bei der Entwicklung von Kontrollsystemen mit *CS*. Dies beschränkt allerdings die Möglichkeiten im Bereich der Codegenerierung und des Reverse-Engineerings

für *CS* auf Methodenhülsen mit Ein- und Ausgangsparametern. Assoziationen können nicht generiert werden. Weiterhin ist in LabVIEW nicht vorgesehen, Elemente generisch zu generieren. In einer textbasierten Programmiersprache ist es immer möglich, durch Zugriff auf Dateien neue Textdateien zu erzeugen, die ihrerseits wieder als Quellcode benutzt werden können. Nach dem Stand der Bachelor-Thesis ist diese Art der Codegenerierung in LabVIEW nicht möglich. Einen Weg zur automatischen Codegenerierung in LabVIEW zu finden ist weiterhin nicht Teil dieser Arbeit.
Die Druckfunktion und der Export von Diagrammen ins jpg- und png-Format sind Standardfunktionen der meisten Case-Tools.
Die Kontexthilfe ist eigentlich eine Eigenschaft von LabVIEW, die im UML Modeller erweitert wird. Das Wissen um die Notation der verschiedenen Diagramme wird von den meisten Case-Tools allerdings vorausgesetzt, und daher nicht weiter adressiert.

3.2 Integration von Werkzeugen mit Hilfe des Metadata Interchange Formats XMI[67]

Die hier beschriebene Arbeit untersucht zwei Ansätze, verschiedene Case-Tools mit einem Werkzeug Knight, zu integrieren. Ein Lösungsansatz basiert auf dem Austausch von Daten durch das gemeinsame Austauschformat XMI.
Die Arbeit unterscheidet sich durch die Voraussetzungen bezüglich der Entwicklungsumgebung und dem vorgegebenen Ziel von der Bachelor-Thesis. Verwandt ist jedoch der Ansatz, verschiedene Werkzeuge im Softwareentwicklungsprozess durch ein gemeinsames Format zum Austausch von Daten zu verwenden.
Dieser Ansatz ist mit einem möglichen Lösungsansatz der Bachelor-Thesis vergleichbar, da auch hier die Überlegungen in Richtung eines gemeinsamen Austauschformats zur größeren Flexibilität bei der Auswahl von Case-Tools gehen. Da es im Rahmen der Bachelor-Thesis nicht möglich ist, das angestrebte UML-Klassendiagramm direkt aus dem LabVIEW-Quellcode zu erzeugen, müssen in einem Zwischenschritt zuerst Metadaten aus dem zu dokumentierenden Softwaresystem extrahiert werden, um diese dann von einem Case-Tool darstellen lassen zu können. Die beschriebene Arbeit beschäftigt sich genau mit der dabei auftretenden Problematik. Da so schon im Vorfeld positive sowie negative Aspekte dieses Lösungsansatzes deutlich werden, wird diese Arbeit hier kurz beschrieben.

[67] [DHTT00]

3.2.1 Zusammenfassung

Da es nicht möglich ist, alle Aktivitäten bei der Entwicklung von Software in einem Werkzeug vereint zu bearbeiten, spielt die Integration von verschiedenen Werkzeugen im Softwarenentwicklungsprozess eine große Rolle, was im Idealfall zur freien Kombination verschiedener Werkzeuge im Entwicklungsprozess führt.
Um dieser Lösung näher zu kommen, wird der Ansatz zum Austausch von Daten durch das gemeinsame Austauschformat XMI diskutiert. Voraussetzung ist die Operation auf Daten, die auf einem UML Metamodell basieren, durch die zu integrierenden Werkzeugkomponenten.
Als Beispiel zur Integration von Werkzeugen wird Knight benutzt. Knight erweitert bekannte Case-Tool Funktionalität um eine alternative Benutzerschnittstelle, die ein berührungsempfindliches Whiteboard beinhaltet, auf dem Modelle direkt mit einem geeigneten Stift skizziert werden können. Um einen Austausch von Daten zu ermöglichen, wurde Knight um die Unterstützung des XMI Formats erweitert, wobei die UML DTD[68] des IBM[69] XMI Toolkits zum Einsatz kam, da diese den Austausch von Modellen mit nicht vollständig spezifizierten Elementen erlaubt.

3.2.2 Erfahrungen mit dem Austausch durch XMI

Obwohl das Konzept eines einheitlichen Austauschformats und dessen Implementierung einfach aussieht, entstehen doch eine Reihe von Problemen damit:

- Da die UML DTD direkt auf dem UML Metamodell basiert, kann über die reine Modellinformation hinaus nichts spezifiziert werden. Da die Werkzeuge aber jeweils Diagramme erzeugen, die ein solches Modell darstellen, können keine Angaben zur Präsentation der Diagramme mitgeliefert werden.

- XMI erlaubt allerdings, Erweiterungen zu den beinhalteten Elementen anzugeben. Dies wurde in dieser Arbeit aufgegriffen, um fehlende Informationen zur Präsentation des Modells als Diagramm zu geben. Da der XMI Standard allerdings keine Struktur für diese Erweiterungen definiert, werden sie von verschiedenen Werkzeugen auch verschieden verarbeitet. Als Konsequenz können Case-Tools nur Modelle, aber keine Diagramme austauschen.

- Weiterhin ist es verschiedenen Werkzeugen erlaubt, eigene Erweiterungen zu Elementen beim Export eines Diagramms hinzuzufügen, aber nicht Erweiterungen an-

[68] Document Type Definition, siehe Glossar
[69] International Business Machines Corporation

derer Werkzeuge beim Import zu entfernen oder zu bearbeiten. Dies führt zu Inkonsistenzen bezüglich des Zustandes eins Diagramms beim Roundtrip Engineering.

- Außerdem existieren verschiedene Versionen der XMI und der UML Spezifikation, was den Austausch von Daten durch Kompatibilitätsprobleme verschiedener Werkzeuge weiter erschwert.

3.2.3 Vergleich zur Arbeit der Bachelor-Thesis

Das XMI Format macht eine asynchrone Integration der verwendeten Werkzeuge durch gemeinsam verwendete Daten möglich, koordiniert durch die beteiligten Entwickler. Die UML DTD definiert die Syntax der Kommunikation zwischen den Werkzeugen, die für verschiedene XMI Erweiterungen offen ist. Ein standardisiertes Austauschformat ist wichtig um verschiedenen Werkzeugen die Kollaboration bezüglich gemeinsamer Daten zu ermöglichen. Mit XMI können Modelle standardisiert gespeichert werden.
Andererseits sind nicht alle Bedürfnisse von Case-Tools mit diesem Standard abgedeckt. Case-Tools werden häufig Erweiterungen hinzufügen müssen, was die volle Integration verschiedener Werkzeuge erschwert. Zwar wurde ein standardisiertes Austauschformat benutzt, doch sind nur ein Teil der notwendig auszutauschenden Informationen darin standardisiert.
Es ist festzustellen, dass weitergehende Standardisierungsbestrebungen nötig sind auf dem Weg zu flexiblen, kompatiblen und integrierbaren Werkzeugen.

4 Anforderungsanalyse

In diesem Kapitel wird eine Anforderungsanalyse durchgeführt, die definiert, was genau entwickelt werden soll.
Es werden Ziele der Entwicklung, primäre technische Anforderungen sowie Anforderungen der Benutzer ausgearbeitet. Darauf folgt eine erste Übersicht über das Produkt, die die zu realisierenden Funktionen, eine Modellierung der Datenhaltung und Qualitätsanforderungen beschreibt. Am Ende wird ein grober Lösungsansatz vorgeschlagen, der die Überleitung zum nächsten Kapitel, dem Systementwurf bildet.

4.1 Zielbestimmung

Für die objektorientierte Virtual Instrument Bibliothek *CS* der graphischen Programmiersprache LabVIEW, Version 8.2, soll eine Analyse bezüglich möglicher Werkzeuge zur automatischen Erzeugung von UML-Dokumentationen, insbesondere UML-Klassendiagramme, des LabVIEW Quellcodes von Programmen durchgeführt werden.
Unbedingtes Ziel der Analyse ist die Machbarkeit einer solchen automatisierten Dokumentation zu ermitteln, die geeignete Methodik dazu zu finden oder zu entwickeln, und anhand eines Prototypen deren Umsetzung zu zeigen. Unerlässliches Zielkriterium ist die Analyse der vorliegenden Themen selbst. Hier sollen Wege zur Realisierung gezeigt werden, die einerseits aufgrund von Anforderungen der Experimente und den daran beteiligten *CS*-Entwicklern entstehen, und andererseits technische Anforderungen zur Dokumentation von Softwaresystemen mit einbeziehen.
Darüber hinaus sind eine Aussage über Machbarkeit, Eignung und Umsetzung verschiedener Lösungsansätze bezüglich der geeigneten Methodik sowie ein möglichst funktionsfähiger und einsatzbereiter Prototyp wünschenswerte Ziele. Dazu ist zunächst eine Identifikation verschiedener Lösungsansätze nötig, deren Eignung bezüglich der vorliegenden Anforderungen geprüft werden muss. Weiterhin wird die Umsetzung der Lösungsansätze soweit wie möglich analysiert und durchgeführt, wobei als Ergebnis die Implementierung mindestens eines Ansatzes durch einen Prototyp entstehen soll.

4.2 Primäre technische Anforderungen und Lösungsvorschlag

Primäre Anforderungen sind die Verfügbarkeit von LabVIEW Quellen, die alle nötigen Informationen zur Durchführung der automatisierten Dokumentation enthalten. Weiterhin wird der Zugriff auf VI Attribute benötigt. Außer den LabVIEW Quellen sollen andere verwendete Werkzeuge sowie der produzierte Code in anderen Sprachen quelloffen sein.

Um ein Klassendiagramm generieren zu können, werden verschiedene Informationen von den darzustellenden *CS*-Klassen benötigt. Diese Informationen müssen im Vorfeld gesammelt und für die anschließende Darstellung verarbeitet werden. Da im Rahmen dieser Arbeit kein in LabVIEW selbst entwickelter Diagrammgenerator zur Darstellung der Daten als Klassendiagramm geschrieben werden kann, müssen die gewonnenen Daten für einen bereits existierenden aufbereitet werden. Es findet also ein Export von Informationen aus *CS*, sowie ein anschließender Import der Informationen in eine zur Darstellung geeignete Umgebung statt.

Die Umgebung zur Darstellung kann ein CASE-Tool sein, da sich dessen Einsatzgebiet genau auf objektorientierte Entwicklungsprozesse bezieht. CASE-Tools werden schon heute von Entwicklern an der GSI zu Entwicklungs- und Dokumentationszwecken verwendet. Sie unterstützen die gängigsten UML-Diagramme, inklusive dem angestrebten Klassendiagramm.

Um den Im- und Export der Daten zu gewährleisten, muss eine geeignete Transformation gefunden werden. Jedoch ist die Transformation der Daten für das CASE-Tool eng an das verwendete CASE-Tool selbst geknüpft, und bietet Raum für einige Lösungsalternativen diesbezüglich. Hier muss iterativ gearbeitet werden, um neue Erkenntnisse im Umgang mit verschiedenen CASE-Tools umsetzen und die Transformation immer weiter vervollständigen zu können.

Da die meisten CASE-Tools zum Import von Dateien verschiedener Formate fähig sind, ist eine gezielte Analyse in Hinblick auf die größtmögliche Flexibilität zur Auswahl eines CASE-Tools nötig. In die engere Auswahl kommen daher die Erzeugung von Quellcode in einer gängigen Programmiersprache wie Java oder C++, sowie die Erzeugung von Modellbeschreibungen im XMI Austauschformat. Diese Ansätze werden von den meisten CASE-Tools per Import-Funktion unterstützt.

Als weitere Form der automatisch erzeugbaren Dokumentation sollen HTML Seiten mit schriftlichen Beschreibungen der jeweiligen Strukturen, Klassen, Methoden und Vererbungen des Programmcodes, ähnlich der Java API Spezifikation, in Frage kommen.

Als Lösungsvorschlag wird ein Weg gesucht, die relevanten Daten aus den LabVIEW VIs zu extrahieren, und weiter zu verarbeiten. Die Analyse stellt dabei fest, mit welchen Mitteln die für einzelne Dokumentationskomponenten wichtigen Daten gefunden und wie diese dann wieder zu einer zusammenhängenden Dokumentation zusammengefasst werden können.

4.3 Anwendungsbereiche des Dokumentationswerkzeuges

Das Dokumentationswerkzeug wird von *CS*-Entwicklern für Kontrollsysteme von Experimenten verwendet. Durch die LabVIEW Bibliothek *CS* sind Entwickler von Kontrollsystemen in der Lage, objektorientierte Techniken in LabVIEW zur Steuerung von Geräten zu verwenden, die bei Experimenten zum Einsatz kommen. Im Moment wird *CS* bei PHELIX, SHIPTRAP, Motion CaveA, RISING[70] und FOPI[71] an der GSI, bei REXTRAP und ISOLTRAP[72] am CERN und LEBIT am MSU eingesetzt. Diesen Einsatzbereich soll das Dokumentationswerkzeug abdecken.
Als Aufgabenfeld ist die Dokumentation bestehender Software zu nennen, sowie die Fähigkeit, die Dokumentation einfach portieren und bearbeiten zu können, um bei Bedarf über den Dokumentationszweck als solchen hinaus von nutzen zu sein.
Zielgruppe des Werkzeugs sind die Entwickler von Kontrollsystemen für Experimente, die das *CS*-Framework verwenden. Diese Aufgabe wird oft von Doktoranden und Studenten wahrgenommen, die keine unmittelbare Ausbildung als Softwareingenieur haben, und daher auch nicht über tiefes Fachwissen auf diesem Gebiet verfügen.
Die Betriebsbedingungen beschränken sich auf den Einsatz im Büro, wobei das Programm auf Windows und eventuell Linux Betriebsystemen verwendet wird. Zur Darstellung und Präsentation der Dokumentation ist auch der Einsatz auf mobilen Laptop PCs bei Vorträgen oder Meetings vorgesehen.

4.4 Anforderungen der Benutzer

Das UML-Klassendiagramm hat einen großen Bekanntheitsgrad, ist einfach zu erlernen und zu verstehen, gibt in kurzer Zeit einen guten Überblick über Software-Systeme und gilt im Allgemeinen als geeignete Dokumentation. Es deckt daher die Bedürfnisse der *CS*-Entwickler gut ab.
Weiterhin soll der Dokumentationsprozess möglichst automatisiert sein, um dem Entwickler Zeit und manuelle Arbeit beim Erstellen der Dokumentation einzusparen sowie seine Akzeptanz zur Erstellung von Dokumentation zu steigern.
Darüber hinaus beschränken sich die Anforderungen aus Sicht der Benutzer stark auf die Bedienung des Dokumentationswerkzeuges. Da die Tätigkeit der Dokumentation für den Entwickler eine sehr wichtige Aufgabe zusätzlich zu seinen Entwicklungsaufgaben ist, muss diese vor allem einfach sein.

[70] Rare Isotope Spectroscopic Investigation at GSI

[71] Steht für 4 Pi und spielt auf die Fähigkeit des gleichnamigen Detektors an, den kompletten Raumwinkel erfassen zu können (siehe http://www.gsi.de/forschung/kp/kp1/experimente/fopi/index.html/).

[72] siehe http://rextrap.web.cern.ch/rextrap/

Für die LabVIEW Seite des Dokumentationswerkzeuges bedeutet dies eine übersichtliche und intuitiv zu bedienende Benutzeroberfläche.
Da ein CASE-Tool zur Darstellung und Bearbeitung des Diagramms verwendet wird, bestehen hier verschiedene Anforderungen. Zum einen sollte es über eine automatische Layoutfunktion verfügen, um größere Diagramme nicht komplett manuell gestalten zu müssen. Eine Updatefunktionalität ist in diesem Kontext wünschenswert, die geänderte Teile des Modells in das Diagramm übernehmen kann, ohne das restliche Diagramm zu zerstören. Weiterhin ist die Fähigkeit zur Bearbeitung der Diagramme sehr wichtig. Hier sind das Editieren, Hinzufügen oder Löschen von Elementen wie Klassen, Assoziationen, Attributen oder Vererbungsbeziehungen zu nennen, sowie eine aktive Unterstützung des CASE-Tools bei den genannten Tätigkeiten. Das CASE-Tool muss die generierten Diagramme speichern und als Bilddateien exportieren können. Wünschenswert ist auch eine Exportfunktion in Austauschformate wie XMI. Auch eine Unterstützung von komplementären Diagrammen wie Sequenz-, Zustands-, Anwendungsfall- und Ablaufdiagramm ist von Vorteil. Auch wenn das Werkzeug diese nicht automatisch generiert, sollte es dem Benutzer möglich sein, verschiedene Diagramme in einem einzigen CASE-Tool zu erstellen. Zum anderen soll trotzdem eine große Flexibilität in der Auswahl des verwendeten CASE-Tools bestehen, da der Entwickler hier in seinen eigenen Vorlieben nicht eingeschränkt werden soll. Schließlich arbeitet er dann effizient, wenn er ein CASE-Tool verwenden kann, dass seinen Bedürfnissen am meisten entspricht.

4.5 Produktübersicht

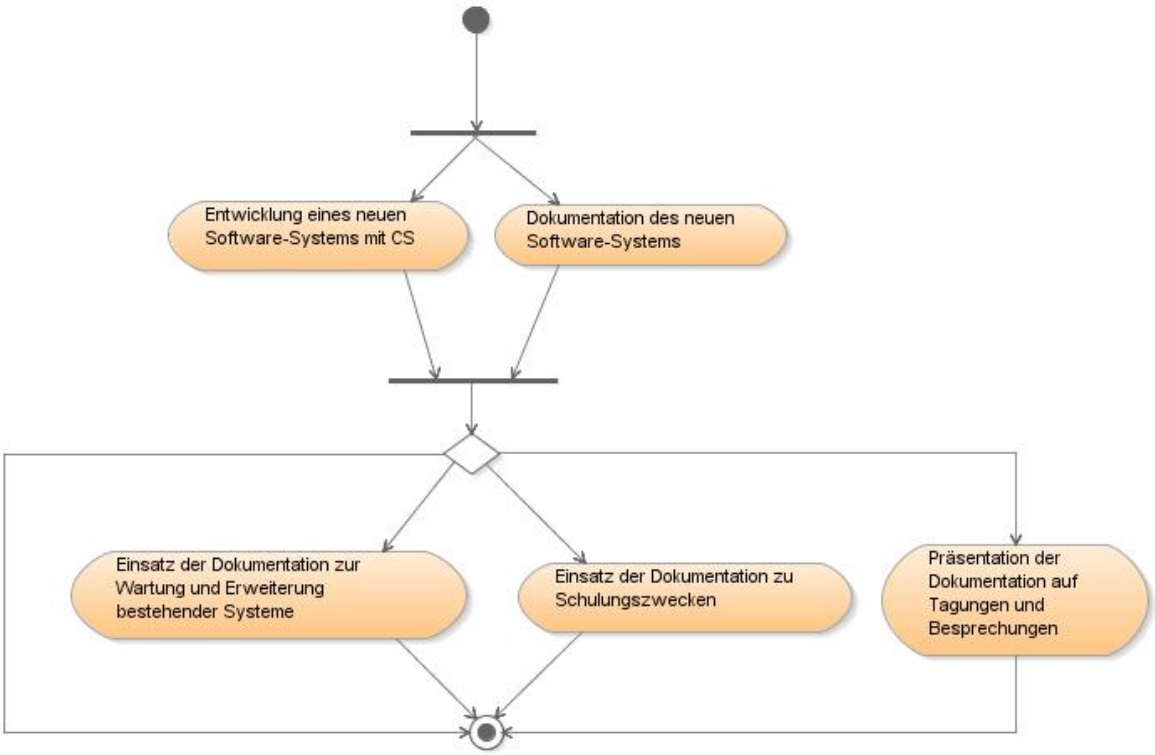


Abbildung 13: Geschäftsprozesse

Entwickelt wird ein Werkzeug zur Dokumentation von Software-Systemen in Form von UML-Klassendiagrammen. Die gezeigten Geschäftsprozesse geben einen Überblick über das Werkzeug und seine Einsatzbereiche.
Hauptsächlich wird er es zur Dokumentation von *CS*-Systemen benutzt. Weitere Anwendungsbereiche finden sich im Rahmen von Schulungen und Weiterbildung, zur Präsentation auf Tagungen und Besprechungen, sowie zur Wartung und Weiterentwicklung von Systemen.
Das folgende Bild gibt einen Überblick über die Anwendungsfälle, die dabei auftreten.

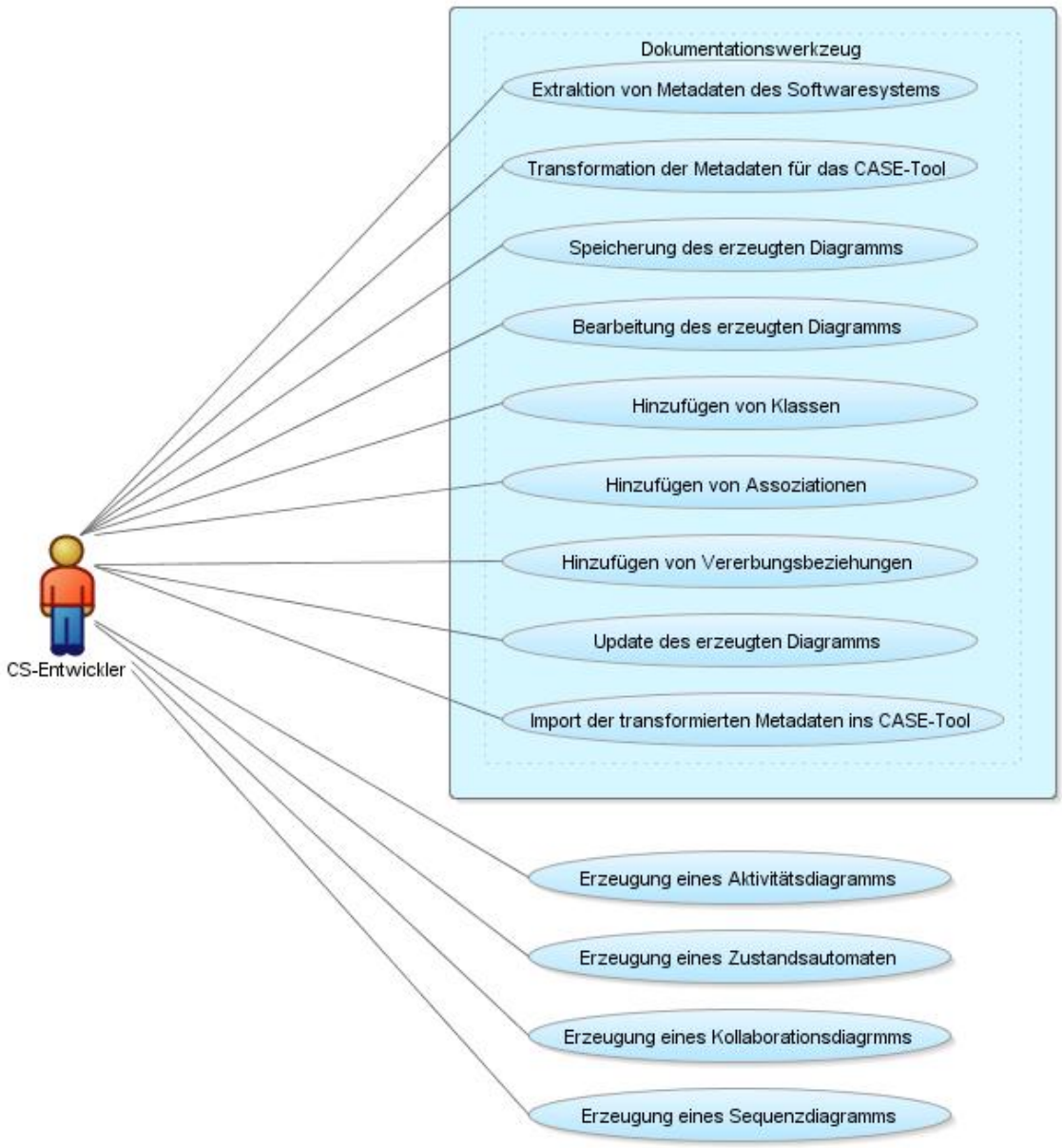


Abbildung 14: Systemgrenze mit Akteuren

Der einzige auftretende Akteur ist der *CS*-Entwickler. Er ist der Adressat für den das Werkzeug entwickelt wird.

Weiterhin ist hier die Systemgrenze zu sehen. Diagramme wie Aktivitäts-, Kollaborations- oder Sequenzdiagramme sind zwar nicht im Umfang eines automatisierten Dokumentationswerkzeuges für UML-Klassendiagramme enthalten, aber trotzdem vom Entwickler erwünscht. Die vom Dokumentationswerkzeug erzeugten Klassendiagramme helfen ihm allerdings dabei, auch diese Diagramme einfacher erstellen zu können.

Im nächsten Bild ist eine erste Gliederung des Dokumentationswerkzeuges zu sehen.

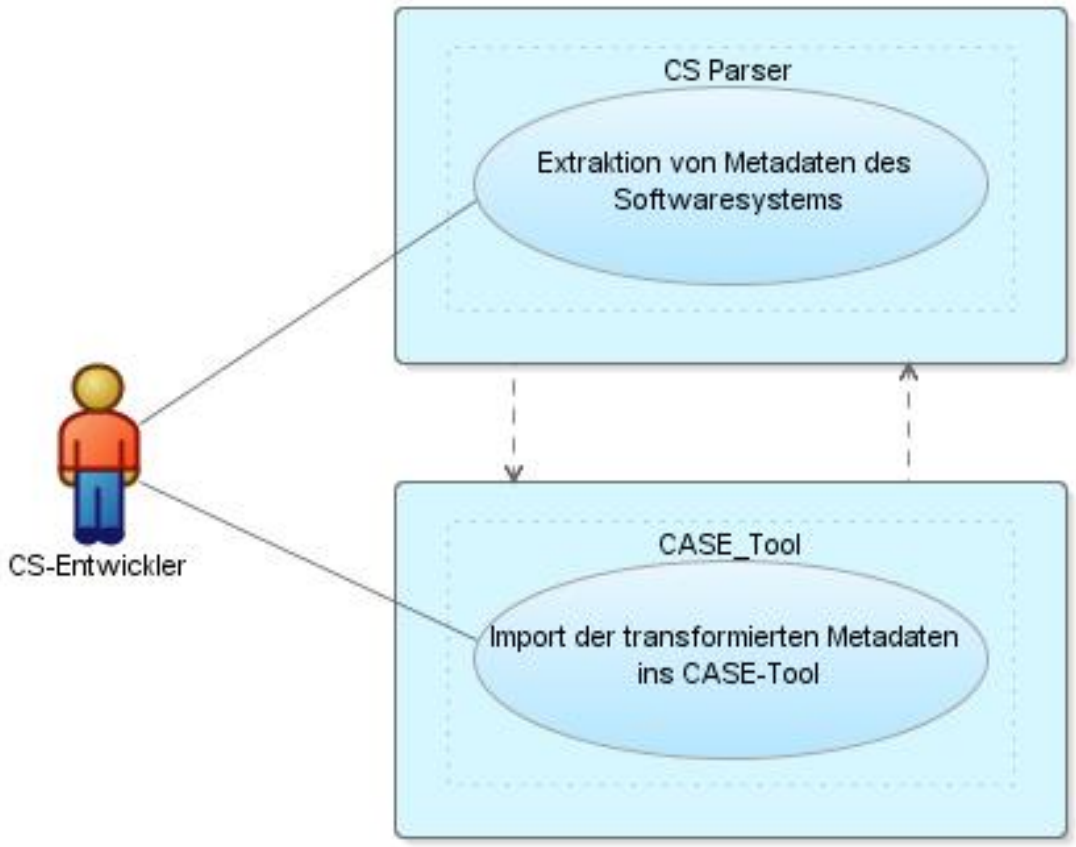


Abbildung 15: Systemkontextdiagramm

Es wird deutlich, das das Werkzeug eigentlich aus zwei voneinander abhängigen Teilen besteht, die separat vom *CS*-Entwickler bedient werden. Einerseits ist ein Parser notwendig, der die Metadaten eines *CS*-Systems extrahieren und transformieren kann. Andererseits ist zur Darstellung und Bearbeitung eines Modells ein CASE-Tool notwendig, das die gewonnenen Metadaten auch importieren kann.
Unter Metadaten ist in diesem Kontext die Abstraktion konkreter Daten zur Beschreibung einer Klasse zu verstehen. Dazu gehören etwa der Name einer Klasse oder deren Attribute, wobei “Attribut” und “Klassenname” die Metadaten sind, “int i” und “MyCSClass” dagegen die konkreten Inhalte.

4.5.1 Produktfunktionen

Die folgenden zwei Bilder zeigen eine Ablaufmodellierung der aufgeführten Anwendungsfälle, die verdeutlicht, wie der *CS*-Entwickler die Funktionen des Werkzeuges verwenden wird, sowie eine verfeinerte Gliederung der zwei Teile des Werkzeuges mit den darin enthaltenen Anwendungsfällen.

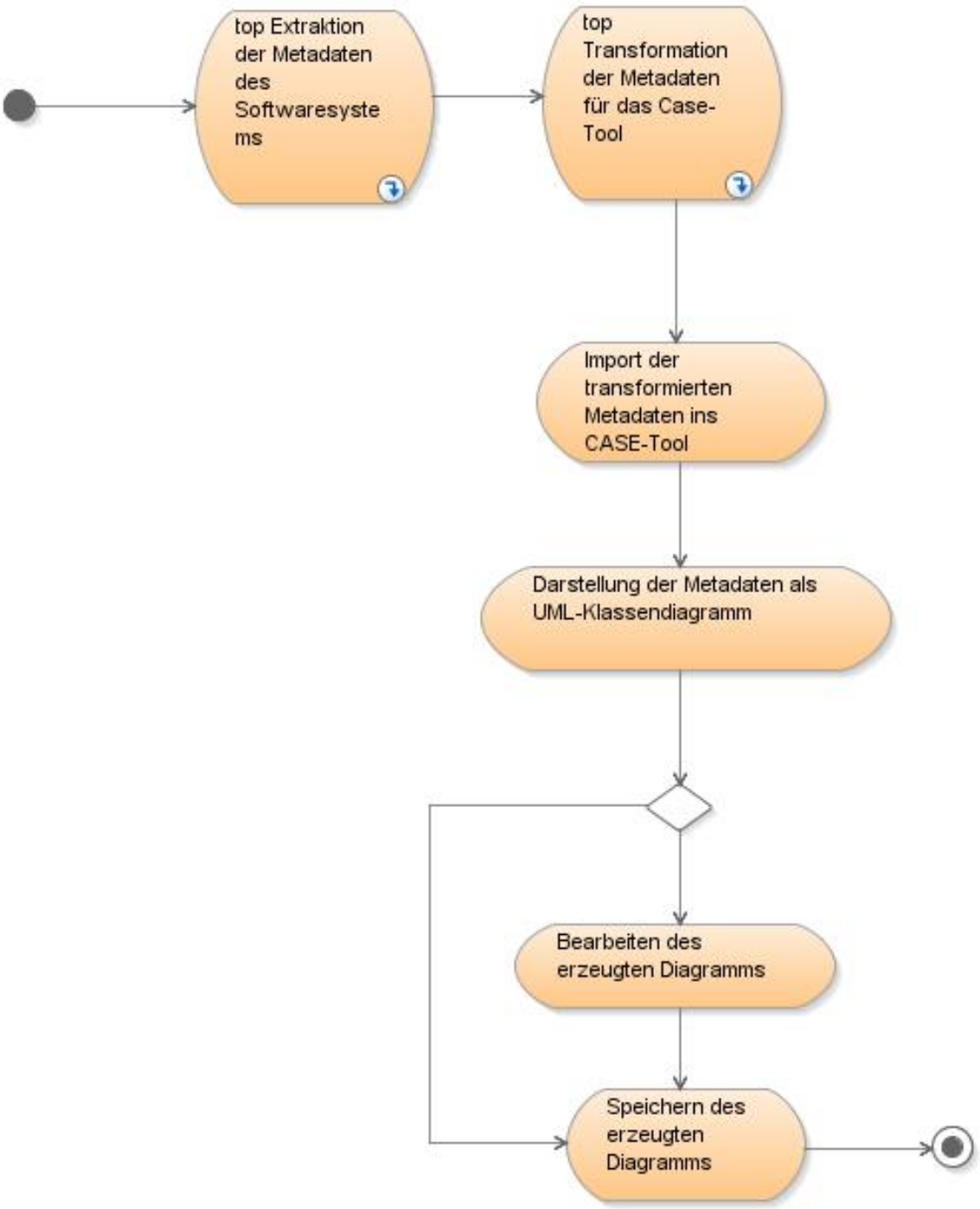


Abbildung 16: Ablaufmodellierung

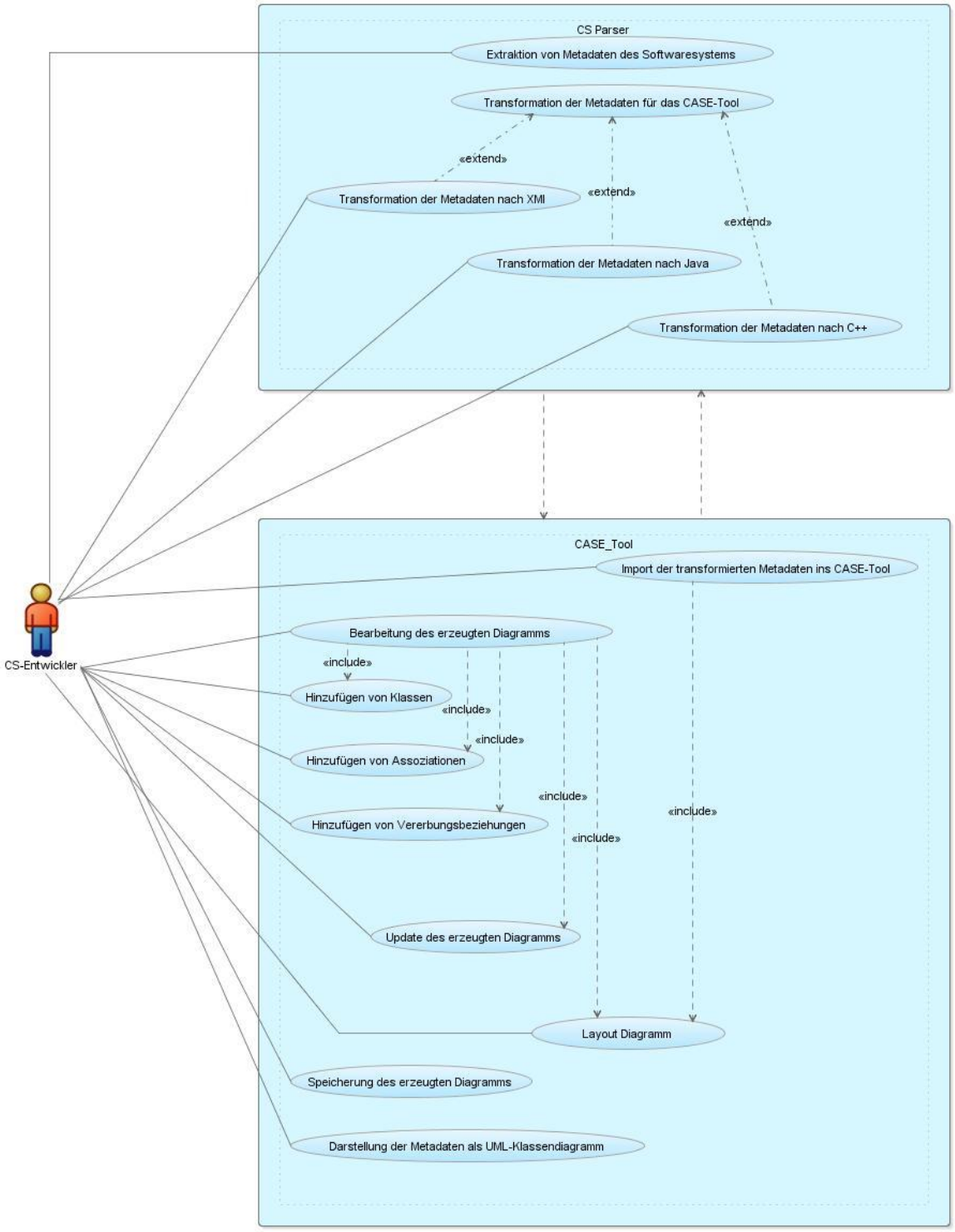


Abbildung 17: Zerlegung in Teilsysteme

In einem ersten Schritt, der zwar vom *CS*-Entwickler angestoßen aber nicht selbst ausgeführt wird, werden die Metadaten des *CS*-Systems extrahiert. Dieser Vorgang ist zwingend notwendig, um die Grundlage einer späteren Darstellung als Diagramm zu gewinnen. Der *CS*-Entwickler kann danach entscheiden, auf welche Art die gewonnen rohen Metadaten transformiert werden sollen. Im Hinblick auf den Lösungsweg sind drei alternative

Austauschformate sinnvoll:

- Java Quellcode
- C++ Quellcode
- das Austauschformat XMI

Eine solche Transformation ist notwendig, da das CASE-Tool eine Grundlage für die Darstellung als Diagramm benötigt.
Dies impliziert eine weitere Funktion des CASE-Tools, den Import der transformierten Metadaten. Das Case-Tool muss fähig sein, mindestens eine der transformierten Metadatensätze zu importieren, um eine Darstellung zu erzeugen.
Weitere Funktionen des CASE-Tools sind die Speicherung und Bearbeitung des dargestellten Diagramms. Es soll die Fähigkeit zur detaillierten Bearbeitung einzelner Elemente des Klassendiagramms besitzen.
Um nicht das ganze Diagramm manuell anordnen zu müssen, ist eine Layout-Funktion notwendig, die die einzelnen Elemente des Diagramms möglichst sinnvoll und übersichtlich anordnet.
Außerdem ist ein Update-Mechanismus wünschenswert. Wenn der Quellcode des *CS*-Systems geändert wurde, soll das CASE-Tool die Änderungen in das Diagramm übernehmen können, ohne das bestehende Layout zu zerstören.

4.5.2 Produktdaten

Im Folgenden wird die Datenmodellierung des Dokumentationswerkzeugs beschrieben. Dabei wird vor allem auf die Speicherung der Daten des Parsers eingegangen, da die Datenspeicherung im CASE-Tool gekapselt ist und daher keiner weiteren Behandlung bedarf. Das folgende Klassendiagramm gibt einen kurzen Überblick.

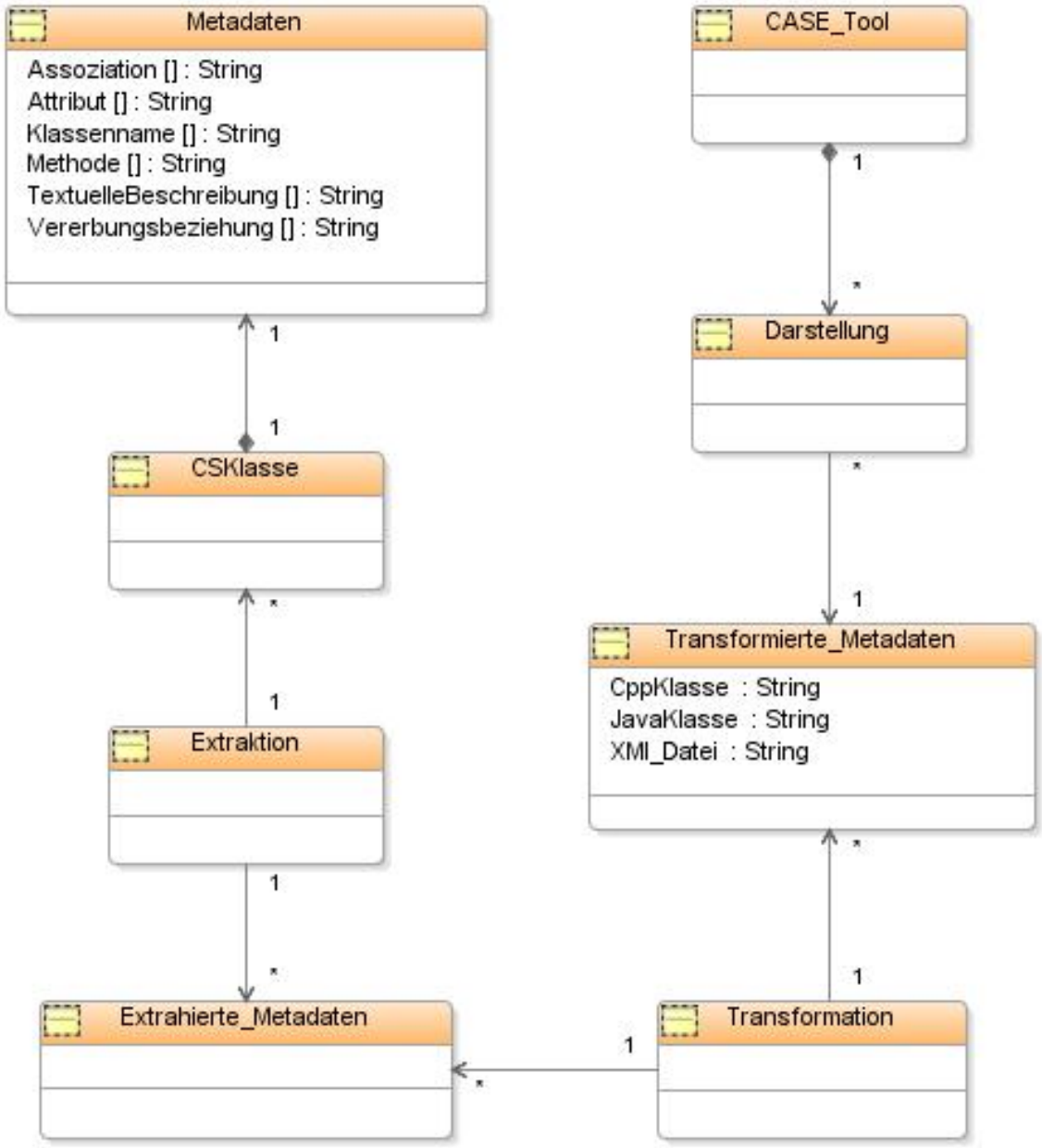


Abbildung 18: Produktdatenmodellierung

Um die wesentlichen Beziehungen zwischen Klassen in einem Diagramm zu modellieren, müssen die Namen von Klassen, Vererbungsbeziehungen, Attributen, Methoden und Assoziationen, sowie die dazugehörigen Datentypen bekannt sein und gespeichert werden. Da eine LabVIEW interne Dokumentation existiert, müssen weiterhin die dort gespeicherten textuellen Beschreibungen von Elementen, soweit vorhanden, gespeichert werden. Ein solches Set von Metadaten bildet dann eine *CS*-Klasse ab. Die Klassen CSKlasse und Metadaten sind dabei nur zur Modellierung eingeführt. Sie bilden die nötigen Informationen ab, die über eine Klasse bekannt sein müssen um sie in einem Klassendiagramm darstellen zu können. CSKlasse und Metadaten gehören dabei komplett zur Seite des zu

dokumentierenden *CS*-Systems, und sind nicht Teil des eigentlichen Dokuemtationswerkzeuges.
Auf den Metadaten operiert die Extraktion, die ein eigenes Metadatenset, Extrahierte_Metadaten, als Zwischenschritt generiert. Dieser Zwischenschritt ist notwendig, um die Metadaten in das Dokumentationswerkzeug zu überführen und diese für die Transformation zu normalisieren. Das bedeutet, die Metadaten werden aus dem tatsächlichen *CS*-System extrahiert und auf ein einheitliches Format gebracht, von dem aus sie im nächsten Schritt einheitlich transformiert werden können.
Im nächsten Schritt werden die extrahierten Daten in die Austauschformate C++, Java oder XMI transformiert. Die Transformation operiert dabei auf der Klasse Extrahierte_Metadaten und erzeugt ein transformiertes Set dieser Metadaten. Die Metadaten, die ursprünglich von einer CSKlasse kommen, werden hier auf einem Austauschformat abgebildet. Dazu wird allerdings die Klasse Extrahierte_Metadaten benötigt, die der Klasse Transformation ein einheitliches Format aller zu dokumentierender *CS*-Klassen zur Verfügung stellt.
Auf den in die Austauschformate C++, Java oder XMI transformierten Metadaten, operiert dann die Darstellung des CASE-Tools. Durch die Import-Funktion eines CASE-Tools werden die Daten geladen und danach vom CASE-Tool dargestellt. Auch die Klassen Darstellung und CASE_Tool werden nur zur Modellierung eingeführt. Sie stellen die Informationen über *CS*-Klassen mit Hilfe der generierten Austauschformate als Klassediagramm dar, sind aber selbst nicht Bestandteil des Dokumentationswerkzeuges.

4.5.3 Qualitätsanforderungen

Das Dokumentationswerkzeug soll zur Erzeugung von Klassendiagrammen aus *CS*-Quellcode fähig sein. Dabei soll auch eventuell auftretende Mehrfachvererbung dargestellt werden. Weiterhin soll der als Zwischenschritt generierte Quellcode in Java, C++ oder XMI so korrekt wie möglich sein.
Allgemeine Qualitätsanforderungen, wie die Fehlerfreiheit des Codes, die Einhaltung von Richtlinien und Konventionen beim Programmieren, die Einrichtung einer Fehlerbehandlung und eine gute Benutzerfreundlichkeit müssen erfüllt werden. Weiterhin sollen Windows und Linux als Plattformen für das Werkzeug eingerichtet werden können.
Oberste Priorität werden den Funktionen der Extraktion und Transformation der Metadaten eingeräumt. Diese Funktionen sind maßgebliche Voraussetzungen um überhaupt ein Diagramm generieren zu können.
Mittlere Priorität wird der Layout-Funktion des CASE-Tools eingeräumt, sowie der Qualität, mit der das CASE-Tool ein Diagramm darstellt. Dies sind zwar wichtige Punkte, die

die Grundlage der Bearbeitung eines Diagramms bilden, bekommen aber aufgrund ihrer Abhängigkeit vom CASE-Tool nur mittlere Priorität zugeordnet, da bei der Auswahl eines CASE-Tools immer Kompromisse eingegangen werden müssen.
Nur geringe Priorität werden der Qualität der Bearbeitungsmöglichkeiten und dem Vorhandensein einer Update-Funktion zugeordnet. Wichtig ist hier, dass eine Möglichkeit zur Bearbeitung des Diagramms besteht, gleichgültig wie gut diese umgesetzt ist. Ein Update-Mechanismus muss auch nicht explizit vorhanden sein. Im Falle einer Änderung des Quellcodes ist es bei den meisten CASE-Tools möglich, die geänderten Klassen einfach in das Arbeitsverzeichnis des CASE-Tools zu kopieren, wobei die alten Dateien überschrieben werden, und das Diagramm automatisch aktualisiert wird.

4.6 Lösungswege

Wie in den vorherigen Diagrammen dargestellt besteht das Dokumentationswerkzeug aus zwei Teilen.
Der erste Teil extrahiert Informationen aus den *CS*-Klassen und transformiert diese in einer geeigneten Weise. Der zweite Teil greift die so transformierte Information auf und stellt diese als UML-Klassendiagramm dar.
Zur Umsetzung des ersten Teils ist es sinnvoll, ein in LabVIEW selbst geschriebenes Programm zu entwickeln, da dieses sehr einfach mit dem CS-Rahmenwerk interagieren kann.
Der zweite Teil wird durch die Verwendung eines CASE-Tools realisiert. In Anbetracht der Lösungsalternativen, in einem Zwischenschritt aus LabVIEW gewonnene Metadaten zu C++ oder Java Code zu transformieren, wird ein entsprechender Codegenerator benötigt. Als Alternativen bei der Umsetzung der Transformation können die Verwendung eines existierenden Codegenerators, sowie die Eigenentwicklung eines Codegenerators angeführt werden. Die durch das Programm gesammelten Metadaten müssen dann für den Codegenerator aufbereitet und auf den generierten C++/ Java Quellen ein Reverse-Engineering durchgeführt werden, um UML-Klassendiagramme zu erzeugen.
Die Vorgehensweise ist ein Ausbaustufenmodell. Das Projekt wird in verschiedenen Stufen realisiert. Die erste Stufe soll die Daten aus einfachen VIs extrahieren. In der zweiten Stufe werden diese transformiert und mit verschiedenen CASE-Tools ausprobiert. Darauf folgt die Vervollständigung der Extraktion von ganzen CS-Klassen in der nächsten Stufe. Hier werden die Metadaten um spezielle Semantik von CS-Klassen ergänzt, die nicht alleine im Code enthalten ist. Die letzte Stufe sieht eine Extraktion und Darstellung der Ereignisse von CS-Klassen vor.

5 Systementwurf

In diesem Kapitel wird der Entwurf der Software ausgearbeitet. Zu diesem Zweck werden verschiedene Architektursichten auf die Software beschrieben.
Um ein konkretes Modell der Daten des Programms zu erstellen, werden hier zu erst die Anwendungsfälle der Anforderungsanalyse detailliert mit Hilfe von Aktivitätsdiagrammen beschrieben. Daraus werden die zu implementierenden Operationen und die dazu nötige Datenhaltung abgeleitet.
Die Ablaufmodellierung (siehe Abb. 16) und die Zerlegung in Teilsysteme (siehe Abb. 17) aus der Anforderungsanalyse sind die Grundlage der hier gezeigten Diagramme. Daraus werden feinere Aktivitätsdiagramme abgeleitet, um die nötigen Methoden zu identifizieren. Aus diesen Aktivitätsdiagrammen entsteht dann ein Klassendiagramm, das die einzelnen Operationen und Attribute detailliert beschreibt.
Das Dokumentationswerkzeug besteht aus einem LabVIEW-Programm, das die Daten aus den *CS*-Klassen extrahiert und sie für verschiedene CASE-Tools weiterverarbeitet. Als erstes wird also die Akquisition der Daten näher beschrieben, dann die Weiterverarbeitung der Daten, und am Ende des Kapitels folgt eine Beschreibung verschiedener CASE-Tools.

Zum besseren Verständnis des folgenden Kapitels wird hier noch einmal kurz der grobe Ablauf einer Anwendung mit Hilfe zweier Diagramme beschrieben.

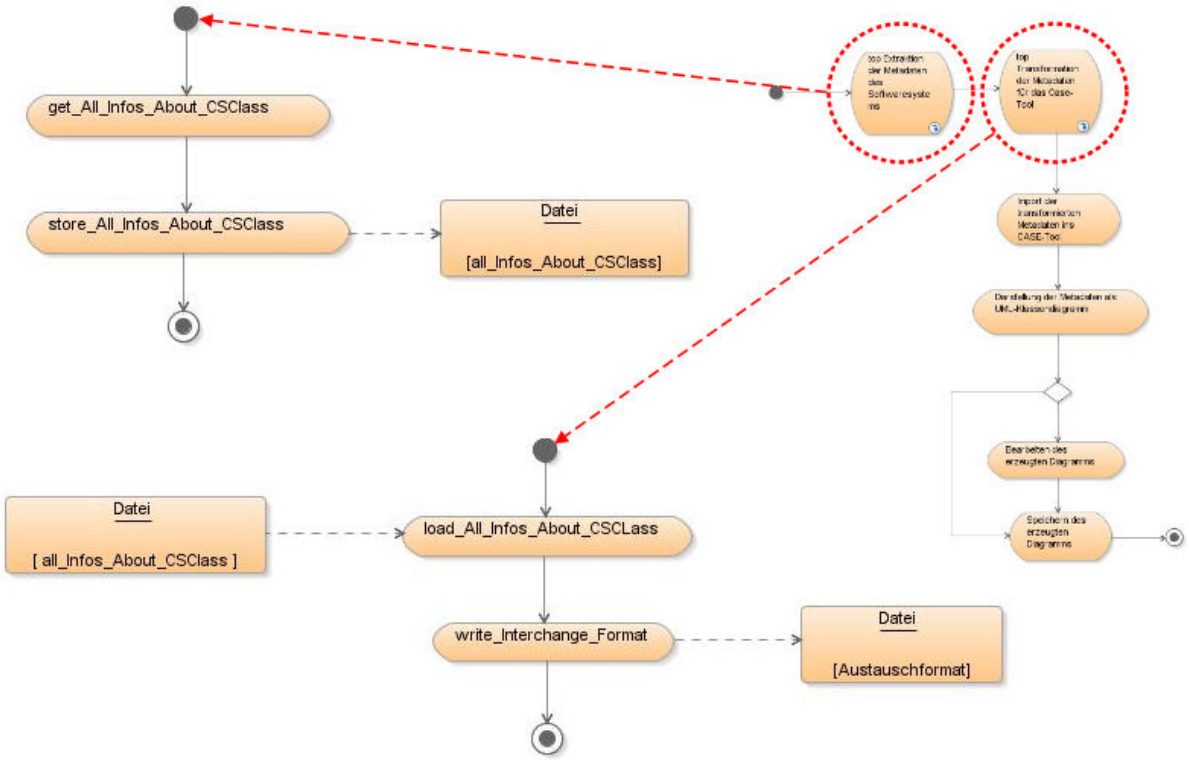


Abbildung 19: Grober Ablauf einer Anwendung 1

In Abb. 19 wird ein grober Überblick über die Phasen der Datenakquisition und des Transformationsprozess gegeben, basierend auf den in Kapitel 4.5.1 spezifizierten Ablaufmodellierung (Abb. 16).
Darauf wird ein kurzes Beispiel eingeführt (Abb. 20), dass die eben gezeigte Ablaufmodellierung mit Hilfe eines Sequenzdiagrammes konkretisiert.

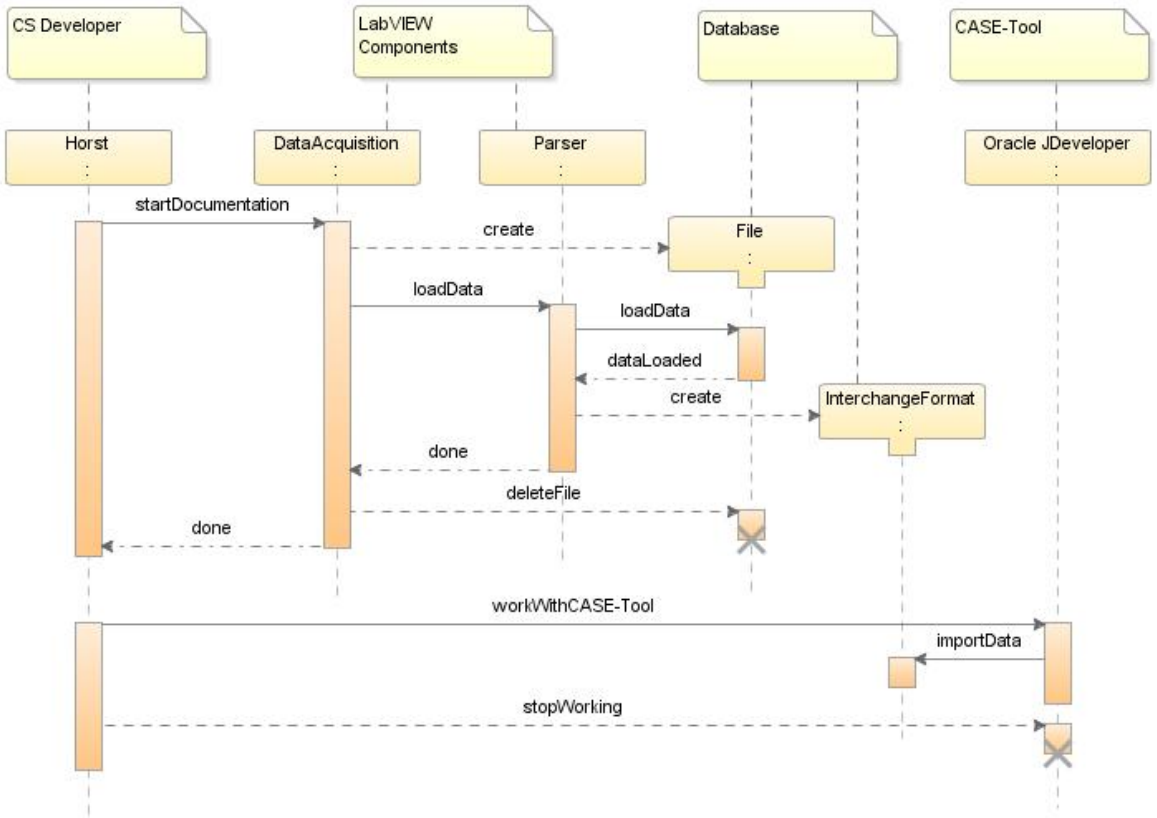


Abbildung 20: Grober Ablauf einer Anwendung 2

Zu sehen ist der Benutzer des Dokumentationswerkzeuges, der die in Abb. 19 dargestellten Abläufe ausführt, indem er die Dokumentation seines Systems startet (siehe Abb. 20). Dabei ist die Interaktion der Komponenten des LabVIEW-Programms zu bemerken. Nach der Ausführung des LabVIEW-Programms bearbeitet der Benutzer die erzeugten Quellen noch mit Hilfe eines CASE-Tools, um seine Dokumentation in Form eines UML-Klassendiagramms zu vollenden.

5.1 Beschreibung der Datenakquisition

Ziel der Datenakquisition ist die Zusammenstellung von Beschreibungen der Elemente einer *CS*-Klasse. Um Zugriff auf die Metadaten einer *CS*-Klasse zu bekommen, die in den VI-Attributen enthalten sind, werden entweder VI-Server-Methoden programmatisch genutzt oder direkt entsprechende SubVI-Aufrufe der VIs selbst getätigt. Weiterhin muss

jede *CS*-Klasse einmal instanziiert werden, um Zugriff auf die Ereignisse der Klasse zu bekommen. Die Ereignisse werden in der Methode ProcEvents der Klassen, die von Base-Process geerbt haben, deklariert, können aber erst zur Laufzeit programmatisch abgefragt werden.
Eine *CS*-Klasse besteht aus einer Verzeichnisstruktur, in der VIs, Controls und eine Library abgelegt sind.
Assoziationen werden im *CS*-Framework im Allgemeinen über den Ereignismechanismus realisiert. Eine Assoziation entspricht also einem DIM-Ereignis[73].
Attribute einer *CS*-Klasse werden durch eine Control realisiert. Controls sind spezielle VI mit der Dateiendung '.ctl', die den Structs aus C/C++ ähnlich sind. Sie werden für die Definition von zusammengesetzten Datentypen verwendet. Trotzdem definiert eine Control einzelne Attribute als primitive Datentypen. Daher ist der Typ eines einzelnen Attributs ein von LabVIEW unterstützter primitiver Datentyp, und keine Control.
Der Klassenname ist auch als Metaattribut ein einfacher String, der sich als Name des Verzeichnisses der *CS*-Klasse und als Präfix im Namen der VI der Klasse wieder findet.
Methoden entsprechen einem LabVIEW VI.
Durch die LabVIEW interne Dokumentation existiert zu allen Elementen eine textuelle Beschreibung, sofern der Entwickler diese ausgefüllt hat.
Vererbungsbeziehungen werden durch geschachtelte Aufrufe von Konstruktoren realisiert (siehe Kapitel 2.2.2, Vererbung). Da ein Konstruktor auch durch ein VI realisiert wird, wird eine Vererbungsbeziehung durch die Existenz des Aufrufs eines anderen Konstruktors in diesem VI modelliert.
Die Sichtbarkeiten aller aufgeführten Metaattribute sind privat, außer dem Klassennamen, da dieser als Präfix aller VI und als Name des Verzeichnisses der Klasse zu sehen ist. Alle Multiplizitäten sind offene Intervalle [0..n], außer die des Klassennamens, der genau einmal existiert. Weiterhin sind alle Attribute statisch, da sie als Metaattribute für *CS*-Klassen desselben Typs gleich deklariert und definiert sind.

Daraus ergeben sich die Operationen, die nötig sind, um diese Metadaten zu sammeln. Das nächste Aktivitätsdiagramm zeigt den Ablauf einer Datenakquisition. Zu erwähnen ist, dass alle gesammelten Daten als Strings in Dateien gespeichert werden.
Der erste Schritt ist die zu bearbeitende *CS*-Klasse zu erfassen.
Um die Sichtbarkeit aller Elemente einer *CS*-Klasse herauszufinden, wird die Library-Datei der *CS*-Klasse benötigt (Dateiendung '.lvlib'). Dies ist eine LabVIEW-Library, die mit dem Projektexplorer von LabVIEW geöffnet werden kann. Da die Sichtbarkeit von

[73] siehe Kapitel 2.2.4 und 2.2.2, Assoziationen

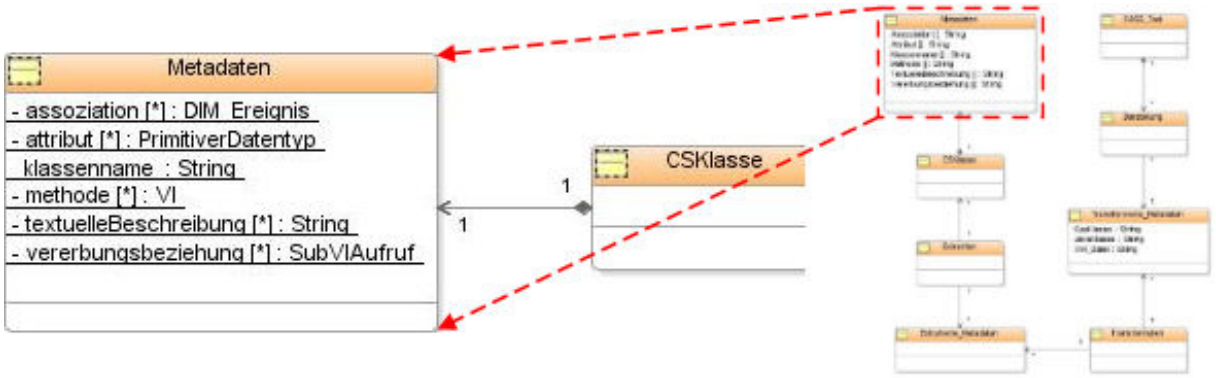


Abbildung 21: Verfeinerte Klasse Metadaten, Ausschnitt aus 18

Elementen einer *CS*-Klasse nur in Form von Zugehörigkeiten zu Unterverzeichnissen realisiert ist, wird der Zugriff auf die Library benötigt. Die Sichtbarkeit wird dann zusammen mit den restlichen Informationen zu einem VI abgespeichert.
Parallel dazu werden die restlichen Informationen über VIs gesammelt. Außer dem Namen sind dies die Beschreibung der LabVIEW-Dokumentation, sowie die Ein- und Ausgangsparameter eines VI.
Da die Konstruktoren von *CS*-Klassen per Konvention den Namen "Klassenname.constructor.vi" haben, wird überprüft, ob das gerade bearbeitete VI ein Konstruktor ist, um dann gegebenenfalls die Vererbungshierarchie herauszufinden. Diese wird allerdings separat zu den VI-Informationen gespeichert.
Die Attribute einer Klasse sind per Konvention als Control mit dem Namen "Klassenname.i attribute.ctl" gespeichert. Passend dazu existiert ein VI mit dem Namen "Klassenname.i attribute.vi", das als Eingangsparameter genau die Attribute der Klasse hat. Dadurch können die Attribute erstmal ohne Extrabehandlung zusammen mit den Methodeninformationen abgespeichert werden.
Aufgrund der Realisierung von Assoziationen über den Ereignismechanismus ist es an dieser Stelle nicht möglich, Informationen darüber zu erlangen.

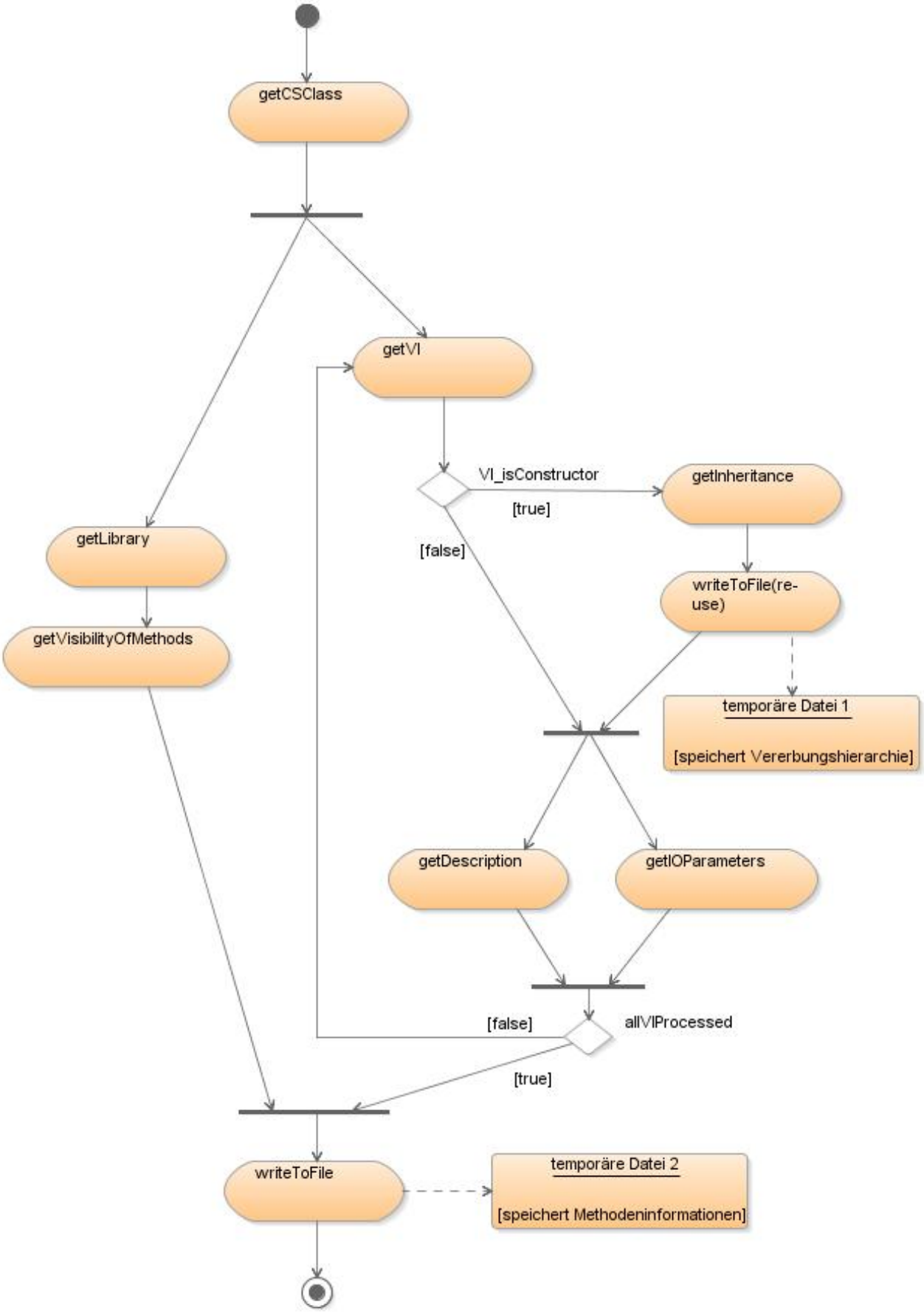


Abbildung 22: Extraktion der Metadaten des Softwaresystems

Aus den Details der Metadaten ergeben sich die Parameter der Methoden der Datenakquisition.
Die erste Methode setzt eine Interaktion mit dem Benutzer voraus, woraus sich ein Teil der Benutzerschnittstelle ableiten lässt. Der Benutzer muss zum Start der Datenakquisition mindestens den Namen der zu analysierenden Klasse angeben. Das Frontpanel der Steuerung wird also wenigstens ein Textfeld beinhalten, in das der Benutzer den Namen der *CS*-Klasse eingeben kann. Damit die folgenden Methoden weiterarbeiten können, liefert *getCSClass* den Pfad zur angefragten Klasse zurück. Da alle *CS*-Klassen im lokalen *CS*-System registriert sind, kann *getCSClass* eine *CS*-Systemmethode benutzen, die den Pfad zu einer mit Namen bekannten *CS*-Klasse zurückgibt.
Da jetzt der Pfad zur Klasse bekannt ist, kann *getLibrary* diesen benutzen um die Library der Klasse herauszusuchen, die dann an *getVisibilityOfMethods* weitergegeben wird.
getVisibiltyOfMethods speichert dann die Methoden mit ihren Sichtbarkeiten in einer Collection ab und gibt diese zurück.
Der bekannte Dateipfad wird parallel dazu an *getVI* weitergegeben, das in einer Schleife alle VIs der Klasse weitergibt.
getInheritance wird aufgerufen, falls das gerade zu bearbeitende VI ein Konstruktor ist. Die Vererbungshierarchie muss nur einmal pro Klasse festgestellt werden, und wird in einem Stringformat zurückgegeben und separat zu den Methodeninformationen von *writeToFile* in einer eigenen Datei gespeichert.
getDescription und *getIOParameters* bekommen jeweils das aktuelle VI als Eingabe und liefern die textuelle Beschreibung der LabVIEW Dokumentation sowie eine Beschreibung der Ein- und Ausgangsparameter als Strings zurück.
Diese Informationen werden danach zusammen von *writeToFile* in einer Datei gespeichert.

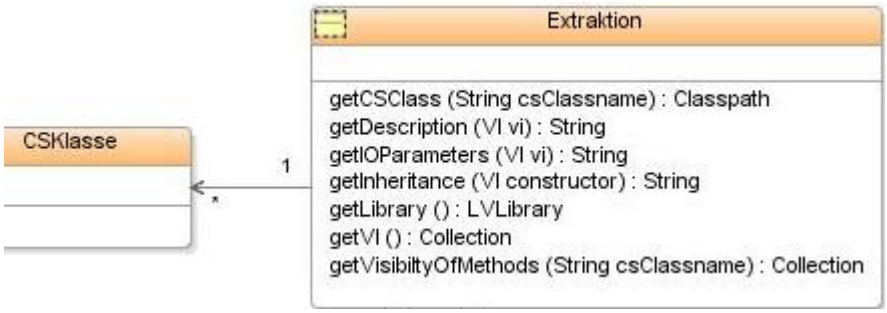


Abbildung 23: Extraktion

Die Methode *writeToFile* wird einer zusätzlichen Klasse Tools zugeordnet. Hier werden universell einsetzbare Funktionen realisiert. *writeToFile* soll einen beliebigen String in

einer Datei speichern. Dazu werden sowohl der String als auch der komplette Pfad mit Dateinamen als Eingangsparameter übergeben.
Die Methoden der Datenakquisition speichern alle gesammelten Informationen als Strings in Dateien. Da die Sammlung der Sichtbarkeiten durch den Zugriff auf die Library parallel zur restlichen Datenakquisition verläuft, werden die Sichtbarkeiten zuerst in einer Collection, später aber zusammen mit den restlichen Daten in einer Datei gespeichert. Die Datentypen der Attribute der Klasse Extrahierte_Metadaten werden hier trotzdem als Strings angegeben, da die Informationen zum einen tatsächlich als einfache Strings in den Dateien gespeichert sind, und später auch als Strings weiterverarbeitet werden. Die Attribute haben alle private Sichtbarkeit, und dieselben Multiplizitäten wie ihre Gegenstücke in der Klasse Metadaten.

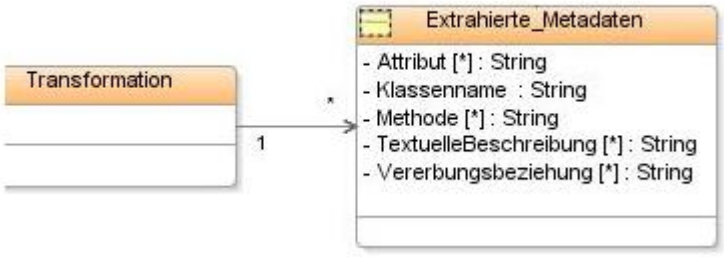


Abbildung 24: Extrahierte Metadaten

5.2 Beschreibung der Transformation

Nachdem die zur Erstellung eines Klassendiagramms notwenigen Metadaten der *CS*-Klassen gesammelt wurden, werden diese weiterverarbeitet. Das Ziel ist, die gesammelten Informationen in ein Austauschformat umzuwandeln. Unabhängig vom Format, das die Transformation erzeugen soll, sind dem Konzept nach immer dieselben Schritte durchzuführen (siehe auch 6.2.3).
Als erstes muss eine Vorlage in Form eines Grundgerüstes der in Frage kommenden Austauschformate erstellt werden. Dies dient der in Module gegliederten Strukturierung des Programms. Weiterhin wird das Gründgerüst so konstruiert, dass die folgenden Methoden Orientierungspunkte vorfinden, um das Gerüst zu vervollständigen. Die Methode *parseTemplate* erstellt das Gerüst eines Formates, sowie Markierungen als Kommentare. An diesen Markierungen können sich dann die folgenden Methoden orientieren.
Anhand der erstellten Vorlage werden dann die Vererbungshierarchie, die Attribute und Methodendeklarationen geparst, um die Vorlage zu vervollständigen und das Austauschformat zu erzeugen.

Die Ein- und Ausgangsparameter der Methoden werden eingefügt, wenn die Methodendeklaration abgeschlossen ist.
Da die Methoden des Parsers als Eingänge im Wesentlichen nur die Informationen der Datenakquisition und die Datei haben, in die geschrieben werden soll, würde LabVIEW diese parallel abarbeiten (siehe Kapitel 2.1.3). Da aber alle Methoden in dieselbe Datei schreiben werden, entstehen bei paralleler Abarbeitung Inkonsistenzen. Daher muss der Ablauf der Methoden des Parsers streng sequentiell sein. Dies wird in LabVIEW durch die Verwendung einer Sequenzstruktur realisiert.

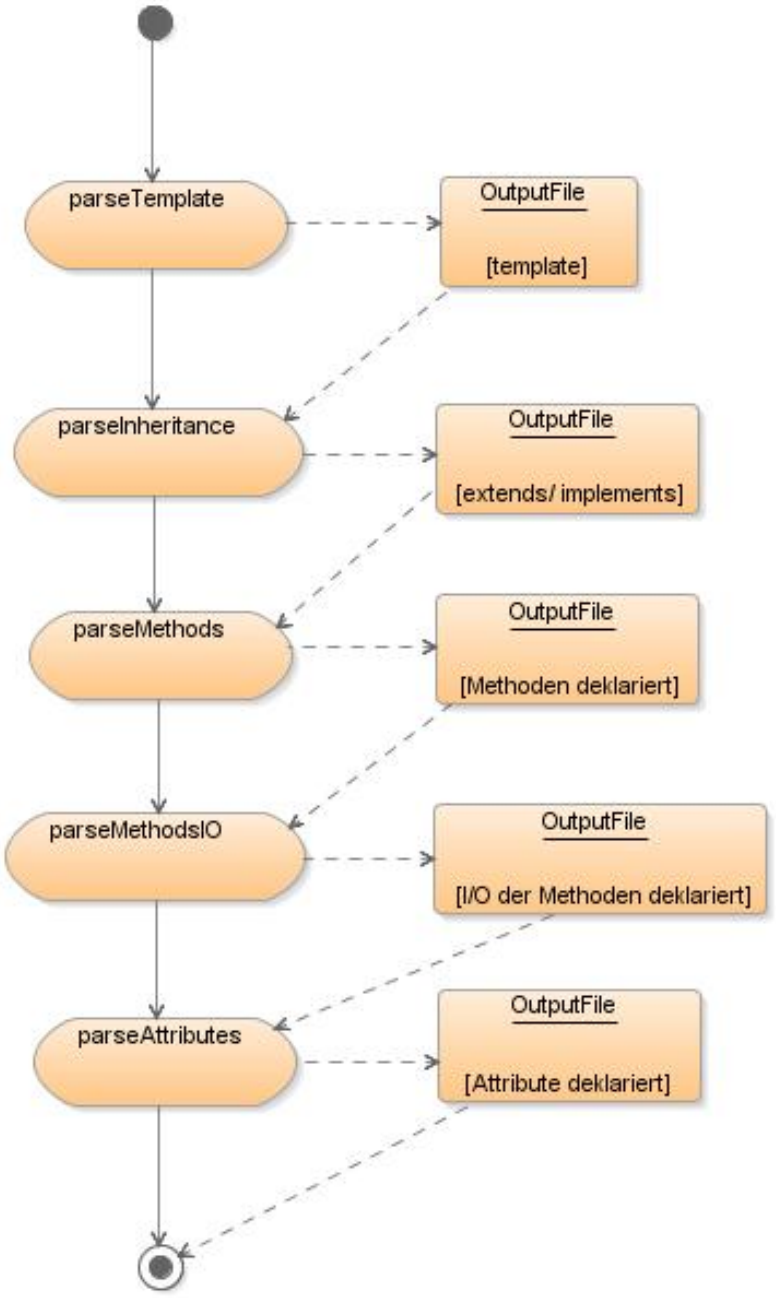


Abbildung 25: Funktionsweise des Parsers

Vom Parser sind eine oder mehrere Dateien zu erzeugen, die entweder Modellinformationen über die analysierte Klassenstruktur enthalten oder selbst ein Abbild des Modells darstellen. Das genaue Format der Dateien ist noch unklar und wird in Wechselwirkung mit dem CASE-Tool erarbeitet.
Die zu erzeugenden Dateien wirken sich auch auf die Benutzerschnittstelle aus, da Felder für eventuelle Optionen angeboten werden müssen, je nachdem welches Format erzeugt werden soll.

Aus dem gezeigten Aktivitätsdiagramm ergeben sich die Parameter der Methoden der Transformation.
Unabhängig vom erzeugten Format wird im Rahmen der Optionen aber mindestes ein Feld zur Angabe des Zielverzeichnisses der erzeugten Dateien, sowie ein Feld für den Namen der aktuell zu transformierenden Klasse bereitgestellt. Sollen mehrere Formate erzeugt werden, wird auch hierfür ein Feld eingeführt. Diese Informationen werden von allen aufgeführten Methoden benötigt. Daher scheint es sinnvoll, diese als Attribute der Klasse Transformation zu modellieren, anstatt sie in jeder Methode wieder als Eingangsparameter aufzuführen.
Die Methode *parseTemplate* benötigt nur das gewünschte Format, die Optionen und den Namen der abzubildenden Klasse als Eingangsparameter. Da diese als Attribute der Klasse modelliert werden, sind sie nicht mehr in der Parameterliste aufgeführt.
Die Methode *parseInheritance* benötigt als Eingangsparameter die Datei, in der die Vererbungshierarchie von der Datenakquisition gespeichert wurde. Da die Information in der Datei als String abgespeichert wurde, wird der Eingangsparameter hier als String modelliert.
Zum parsen der Methoden und deren Parameter benötigen die Methoden *parseMethods* und *parseMethodsIO* das Attribut methode der Klasse Extrahierte_Metadaten, das als String übergeben wird. Eigentlich ist das Attribut methode eine Liste aller Methoden einer Klasse und deren Parameter, die hier wieder als String modelliert wird.
parseAttributes benötigt das Attribut attribut der extrahierten Metadaten als Eingangsparameter. attribute ist ein String, daher ist der entsprechende Eingangsparameter auch ein String.

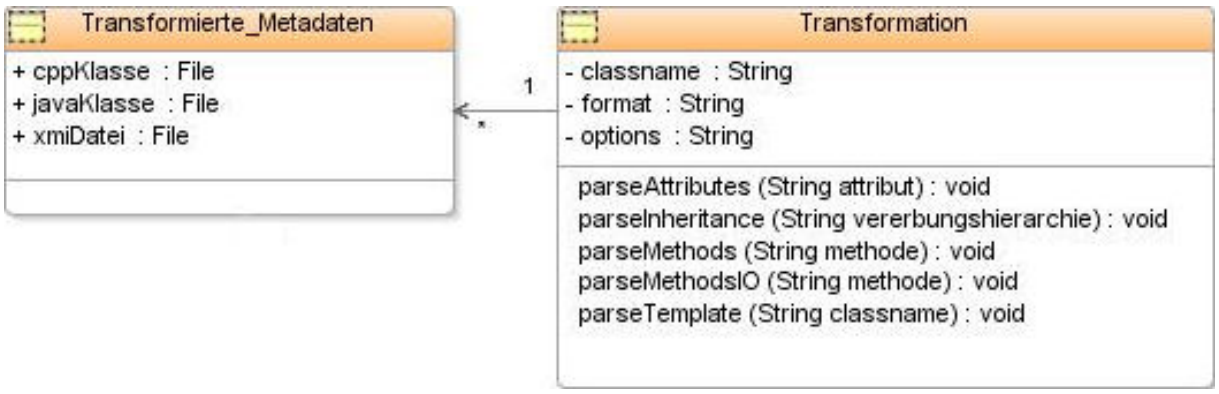


Abbildung 26: Transformation

Zu erzeugen sind Dateien in den alternativen Formaten Java, C++ oder XMI. Diese werden als public modelliert, da sie direkt vom CASE-Tool importiert werden.

Daraus ergibt sich das vollständige Klassendiagramm:

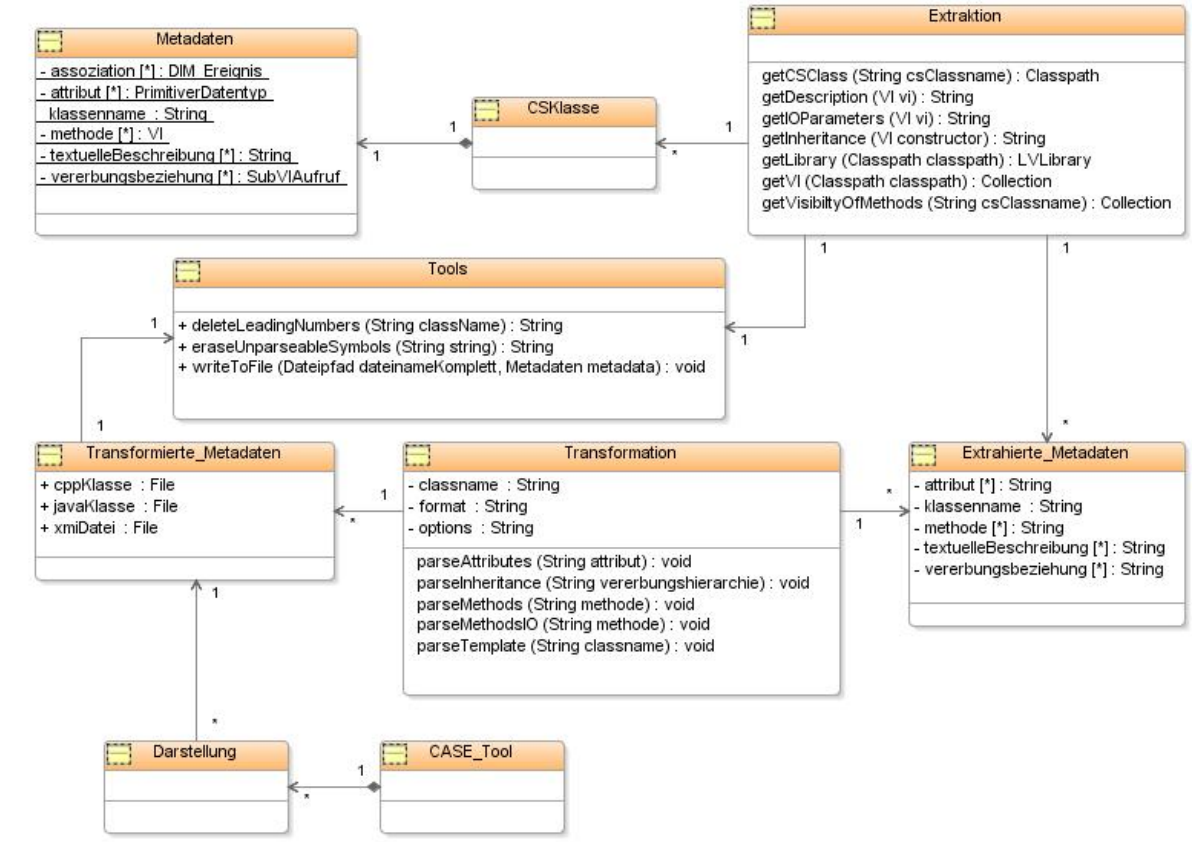


Abbildung 27: Produktdatenmodellierung

Zu sehen ist die Klasse Tools, die noch zwei zusätzliche Methoden aufführt:

- *deleteLeadingNumbers* ist eine speziell für *CS*-Klassen entwickelte Methode. Da es im *CS*-System möglich ist, Klassennamen mit vorangestellten Ziffern zu erzeugen,

müssen diese umbenannt werden, bevor ein CASE-Tool diese Information verarbeiten kann. Da es keinen programmatischen Weg gibt, dieses Problem zu lösen, öffnet die Methode einen Dialog, der dem Benutzer den Namen der fraglichen Klassen zeigt, und ihn bittet, einen neuen Namen für die Erzeugung der Klasse des Modells einzugeben.

- *eraseUnparseableSymbols* ändert oder löscht bestimmte andere Zeichen in Strings. LabVIEW lässt Zeichen wie Leerzeichen oder ,/,[,(,],),, ´,‘,*, ,+,’,# etc. in Namen für Methoden oder Variablen zu. Da diese aber für die normale Modellierung in einem CASE-Tool illegal sind, werden sie hier programmatisch entfernt und gegebenenfalls durch legale Zeichen ersetzt.

5.3 Aufteilung in Pakete und Zuordnung zur Architekturschicht

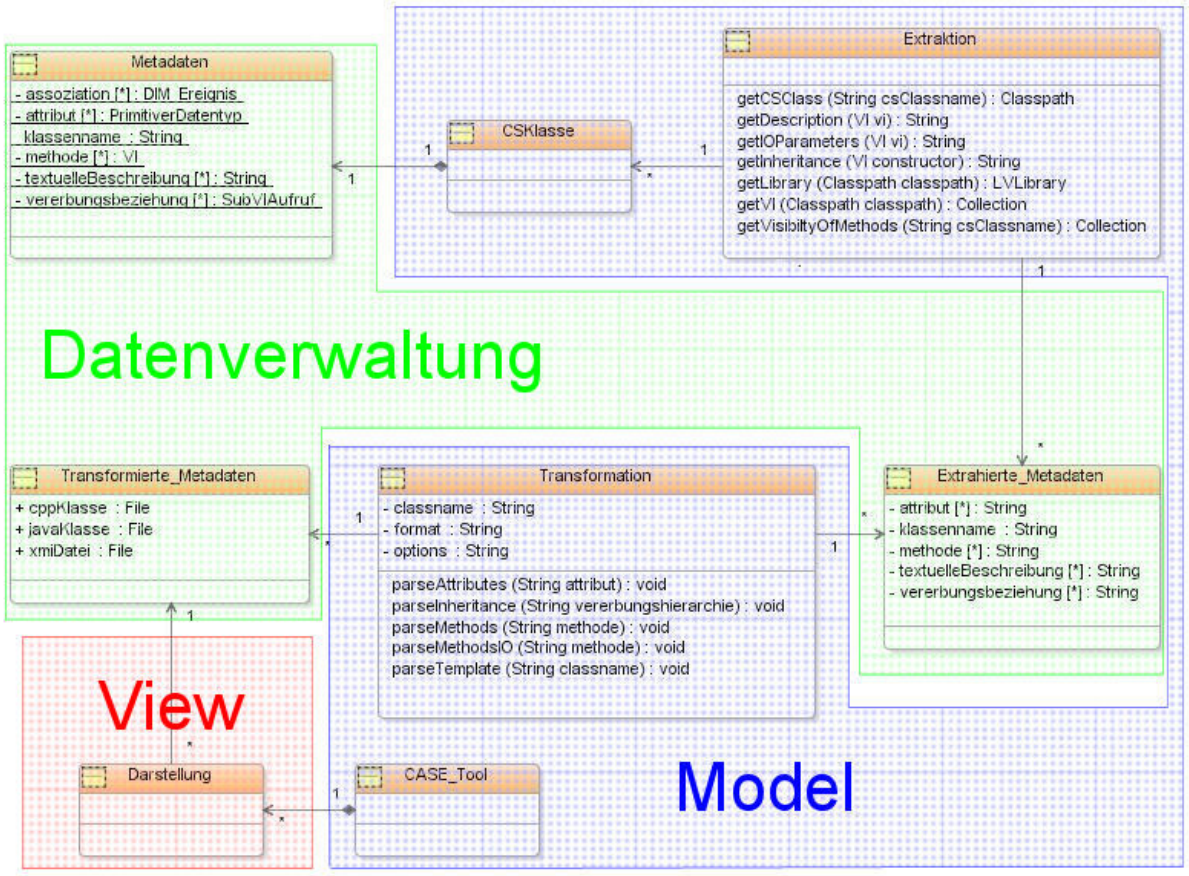


Abbildung 28: Aufteilung der Komponenten

Hier ist die Aufteilung der Komponenten in die bekannte MVC-Architektur zu sehen. Es fällt auf, dass sich “View” nur auf die Darstellung des CASE-Tools bezieht. Dies ist tatsächlich nicht der Fall. Der Rest der Benutzerschnittstelle wird durch die Frontpanels von LabVIEW, speziell das Frontpanel der Steuerung der Software, abgedeckt. Weiterhin

ist auch keine “Control” zu sehen, da sich die Einheit der Steuerung des Programms auf ein einziges VI, ähnlich einer Main-Methode erstreckt. Diese Steuerungseinheit ist Teil der “Control” und des “View”. Da sich der “Control”-Teil der Steuerungseinheit auf den Aufruf der Datenakquisition und des Parsers beschränkt, wird sie hier nicht extra dargestellt. Weiterhin erzeugt LabVIEW die Benutzerschnittstelle, also den “View”-Teil der Steuerungseinheit, automatisch aus den Ein- und Ausgängen des Blockdiagramms dieses Steuerungs-VI in Form des Frontpanels. Es bedarf nur kleineren Änderungen, wie z.B. der Positionierung der automatisch erzeugten Elemente, um Einsatzfähig zu sein. Daher hat eine detaillierte Modellierung der Benutzerschnittstelle in diesem Fall keine Auswirkungen auf die tatsächliche Codierung. Da hier nur ein Prototyp ohne spezielle Anforderungen an die Benutzerschnittstelle implementiert wird, kann auf die Modellierung der Benutzerschnittstelle verzichtet werden.
Weiterhin ist hier eine zusätzliche Schicht zu sehen, die Datenverwaltung. Dieser Teil wurde eingeführt, um mehr Struktur in die Architektur zu bringen. Es wird sich jedoch zeigen, dass es im Rahmen der Programmierung mit LabVIEW einfacher ist, diesen Teil nicht als einzelne Komponente zu implementieren. Die Datenverwaltung fällt dann weg, und wird durch die zugehörigen Modellteile inkludiert. Dies resultiert allerdings in einer anderen Komponentenaufteilung. Die letztlich als Komponenten implementierten Teile sind:

- Datenakquisition, bestehend aus Extrahierte_Metadaten und Extraktion,
- Transformation, bestehend aus Transformierte_Metadaten und Transformation,
- sowie die Anwendungssteuerung, die die Benutzerschnittstelle enthält.

Daher zeigen die folgenden zwei Diagramme den Ablauf einer kompletten Benutzung des Dokumentationswerkzeuges, inklusive Benutzer und Steuerung. Die einzelnen Objekte sind auch hier wieder der MVC-Architektur, inklusive Datenverwaltung zugeordnet.

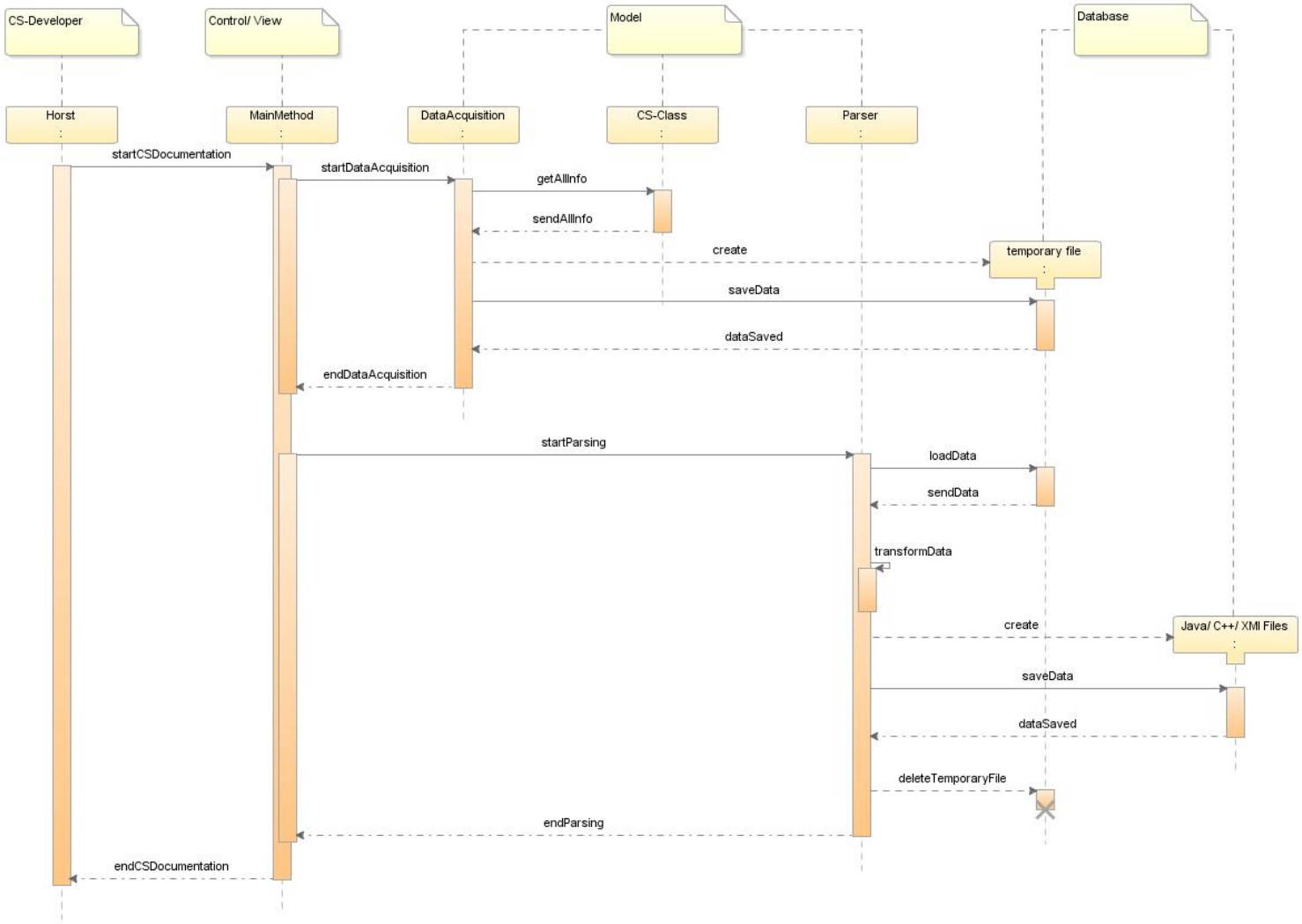


Abbildung 29: Sequenzdiagramm einer normalen Anwendung, Teil 1

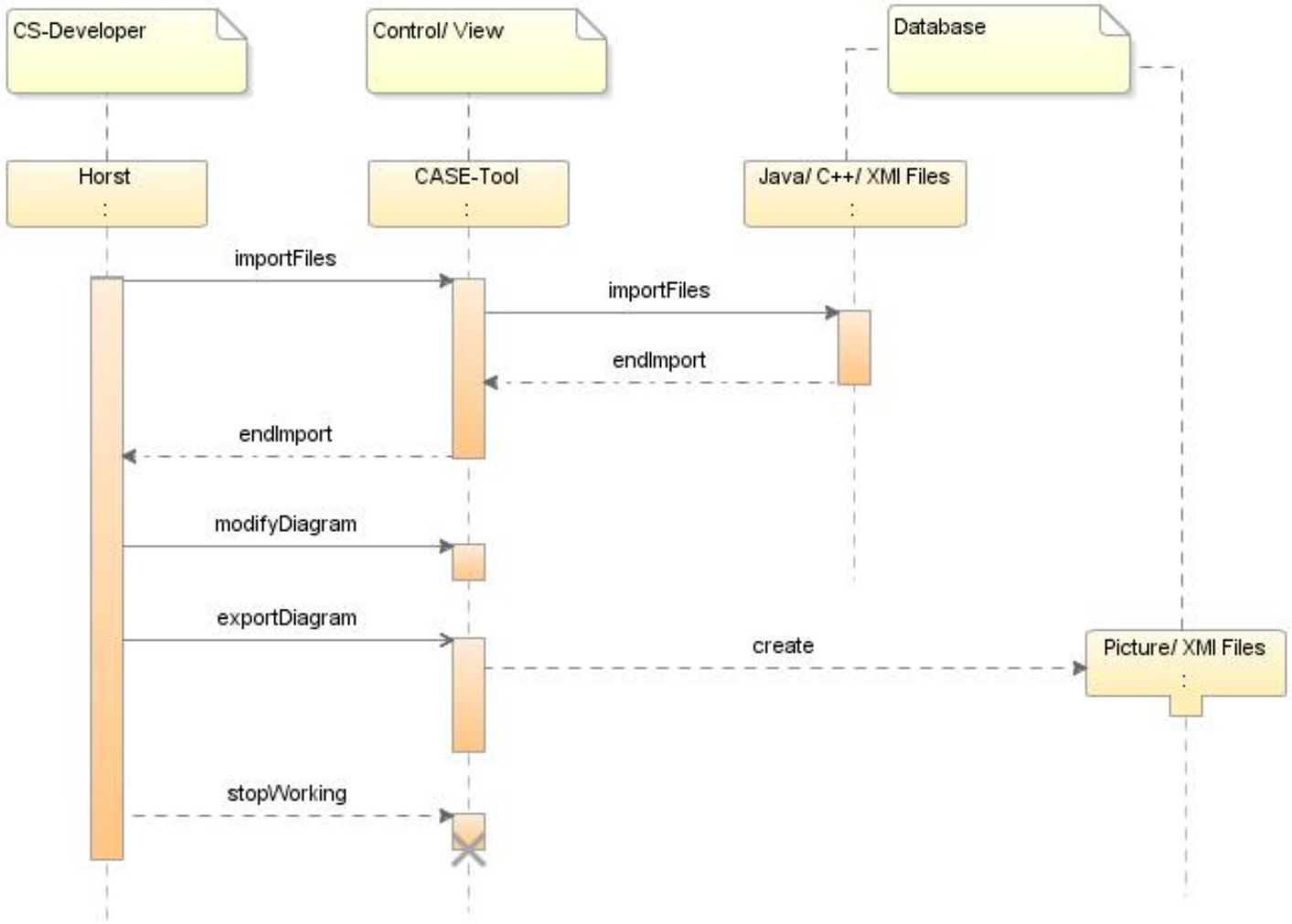


Abbildung 30: Sequenzdiagramm einer normalen Anwendung, Teil 2

Im Sequenzdiagramm ist zu sehen, wie der Benutzer Horst den Dokumentationsprozess startet, und wie er danach die erzeugten Quellen mit Hilfe des CASE-Tools bearbeitet.
Der erste Schritt ist der Start des LabVIEW-Dokumentationswerkzeuges durch den Benutzer Horst. Er interagiert dabei mit der Steuerung des LabVIEW-Programms. Die Steuerung übernimmt von da an den restlichen Prozess.
Sie startet die Modellkomponenten der Datenakquisition, die sich die Metadaten der *CS*-Klasse holt und diese durch die Erzeugung einer neuen temporären Datei abspeichert. Ist die Datenakquisition fertig, meldet sie sich bei der Steuerung zurück, die darauf den Prozess des Parsers startet.
Die Modellkomponente Parser lädt die Informationen der temporären Datei und erzeugt eine neue Datei im Java, C++, oder XMI Format. Ist der Parser fertig, meldet er sich bei der Steuerung zurück, die dem Benutzer das Ende des Programms signalisiert.
Der Benutzer startet daraufhin das CASE-Tool und importiert die erzeugten Quellen. Er modifiziert diese mit Hilfe des vom CASE-Tool erzeugten Diagrammes, speichert und exportiert sie anschließend noch ins XMI-Format. Danach ist er fertig mit seinem Anwendungsbeispiel und schließt das CASE-Tool.

5.4 Beschreibung des CASE-Tools

Der Entwurf des CASE-Tools beschäftigt sich mit der Erfassung und Bewertung potentieller Kandidaten. Es wird eine Liste der gefundenen Kandidaten mit deren Eigenschaften erstellt. Der Funktionsumfang und die Adaptierbarkeit an die Erfordernisse werden überprüft, sowie die notwendige Anpassung der vom Parser erzeugten Quellen ermittelt.
Das CASE-Tool muss folgenden Anforderungen genügen:

- es muss kostenlos sein
- eine automatische Layoutfunktion für Klassendiagramme haben
- Updatefunktionaliät unterstützen. Für den Fall einer Änderung des Quellcodes muss es eine Möglichkeit geben, den geänderten neuen Teil in das alte Diagramm zu übernehmen, ohne den Rest des Diagramms zu zerstören.
- mindestens Editieren, Hinzufügen oder Löschen von Elementen im Klassendiagramm unterstützen
- Klassendiagramme speichern und als Bilddateien exportieren
- Import verschiedener Quellen unterstützen

- Mehrfachvererbung darstellen

Nicht notwendig aber wünschenswert sind:

- Exportfunktion in Austauschformate wie XMI
- Unterstützung von komplementären Diagrammen wie Sequenz-, Zustands-, Anwendungsfall- und Aktivitätsdiagramm

Die folgenden Tabellen zeigen alle CASE-Tools, die als nähere Kandidaten in Frage kommen:

Tool	**Layout**	**Update**	**Export**	**Import**	**UC**	**CL**	**ST**	**AC**	**CS**	**IOD**	**S**	**COL**	**COM**	**D**	**T**
ArgoUML	x	x	XMI1.0, Image	XMI, Java, C++	x	x	x	x	x?	x?	x	x	x?	x	
Poseidon	x	x	XMI, Image	Java	?	x	x	x	x			x	x	x	x
StarUML	x	x	XMI1.3, Model Fragment, emf, wmf, Image	.net, MFC6.0, Java1.4, Model Fragment, Rational Rose, XMI, Java, C++, C#	x	x	x	x	x	x?	x	x	x	x	
Fujaba	x	x	XMI, GXL, Java, Image	GXL, Java	x	x	x	x				x			
BOUML			Java, C++, Idl, HTML Doc, XMI1.2, XMI2.1	C++, Java, Rational Rose, XMI2.1	x	x	x	x			x		x	x?	

Tabelle 1: CASE-Tools und ihre Eigenschaften (1)

Tool	**Layout**	**Update**	**Export**	**Import**	**UC**	**CL**	**ST**	**AC**	**CS**	**IOD**	**S**	**COL**	**COM**	**D**	**T**
ESS Model	x	x	XMI, HTML Doc, Image	Java		x									
Oracle JDeveloper	x	x	XMI, Image	XMI, Java	x	x		x			x				
Omondo eclipse	x	x	UML, Image,	Java, UML	x	x	x	x			x	x	x	x	
Doxygen	x	x	HTML Doc, LaTeX, rtf, Man, XMI, DEF, PerlMod	C++, Java		x									

Tabelle 2: CASE-Tools und ihre Eigenschaften (2)

UC	Use Case	CS	Composite Structure	COM	Component
CL	Class	IOD	Interaction Overview	D	Deployment
ST	State Machine	S	Sequenece	T	Timing
AC	Activity	COL	Collaboration		

Als erstes werden die aufgeführten Programme auf ihre Kernfunktionalität getestet, um den Funktionsumfang zu überprüfen. Weiterhin werden sie auf Komfort und Bedienbarkeit getestet.
Dazu werden verschiedene Tests durchgeführt:

- Um einen Eindruck über das Programm, seine Funktionen und Bedienbarkeit zu erlangen, werden einfache Klassendiagramme erstellt. Dazu werden alle Grundfunktionen des Editierens von Modellelementen benutzt. Die Diagramme werden abgespeichert und nach Neustart des Programms wieder geladen.

  Während dieses Tests wird die Layoutfunktion benutzt, da nicht jedes Programm über einen zufrieden stellenden Layoutalgorithmus verfügt.

- Die Updatefunktionalität wird getestet. Da normalerweise nur Programme die Roundtrip-Engineering fähig sind über einen expliziten Synchronisationsmechanismus verfügen, wird hier nur getestet, wie praktikabel ein Update von den erzeugten Quellen zum Diagramm ist. Das heißt, wie leicht es ist, eine geänderte Quelle erneut in das Diagramm zu importieren und neu auszurichten. Dies ist stark an die Layoutfunktion gekoppelt.

- In einem weiteren Test muss die Belastungsfähigkeit und Stabilität des Programms geprüft werden. Dazu wird das Modell eines Systems in der Größenordnung des *CS*-Systems mit ca. 70 Klassen verwendet.

Im nächsten Schritt werden die zur Auswahl stehenden Programme nach ihrer Import/ Export Fähigkeit unterteilt. Da noch unklar ist, welche Art von Quellen erzeugt werden sollen, werden hier die praktischen Ansätze dazu ausprobiert. Aufgrund der wünschenswerten Flexibilität bei der Auswahl der CASE-Tools für den späteren Benutzer werden die Robustheit zweier verschiedener Ansätze getestet: Die Erzeugung von Quellcode einer Programmiersprache wird der Erzeugung von Quellen im XMI-Format in Bezug auf die Handhabung im CASE-Tool gegenübergestellt.
Die erste Versuchsreihe testet die Reverse-Engineering-Fähigkeiten der Programme. Diese sind in der Liste durch den Eintrag einer Programmiersprache in der Import Spalte gekennzeichnet. Zu erkennen ist, dass alle zur Auswahl stehenden Programme zum Reverse Engineering auf Java-Quellen fähig sind. C++ wird allerdings auch von vielen CASE-Tools unterstützt. Die Unterstützung anderer Sprachen außer Java oder C++ ist selten und daher nicht mit der Anforderung der Flexibilität bei der Auswahl von CASE-Tools vereinbar.

Zu diesem Zweck werden mit Hilfe einer Entwicklungsumgebung einfache Java und C++ Quellen erzeugt. Diese enthalten die wesentlichen Elemente Klasse mit Attributen und Methoden, (Mehrfach-) Vererbungsbeziehung und Assoziation. Aus diesen Quellen erzeugt das CASE-Tool dann ein Klassendiagramm durch Reverse Engineering. Dabei ist festzustellen, dass alle Programme zum Reverse Engineering auf Java-Quellen fähig sind und die Unterschiede hier lediglich in Bedienung und Benutzerfreundlichkeit des einzelnen Programms liegen. Beim Reverse Engineering auf C++ Quellen melden einige Programme Fehler, vermutlich aufgrund eines Formats, das zwar für C++ zugelassen ist, aber nicht vollständig vom Reverse-Engineering-Mechanismus der betreffenden Programme unterstützt wird. Trotzdem generieren auch diese Programme Diagramme, die jedoch unterschiedlich vollständig ausfallen.
Abschließend ist festzustellen, das die Benutzung von Java oder C++ Quellen von fast allen CASE-Tools unterstützt wird und zum Großteil auch gut funktioniert. Dabei ist Java aufgrund besserer Kompatibilität die bessere Alternative. Problematisch ist allerdings die inhärente Unfähigkeit von Java zur Darstellung der direkten Mehrfachvererbung auf Klassen.

Die zweite Versuchsreihe testet die Erzeugung von Diagrammen mit Hilfe des Austauschformats XMI. Die betreffenden Programme sind anhand des Eintrages “XMI” in der Import-Spalte der Tabelle zu erkennen. Dazu wird dasselbe Modell wie beim Testen des Reverse Engineering verwendet. Aus einem Klassendiagramm werden XMI-Quellen mit Hilfe der entsprechenden Exportfunktion erzeugt. Diese XMI-Quellen werden zuerst von dem erzeugenden Programm selbst, und dann von anderen Programmen auf der Liste wieder importiert, wobei jedes Programm mindestens einmal der Erzeuger von XMI-Quellen sein sollte und Importeur von möglichst vielen fremderzeugten XMI-Quellen. Dabei treten folgende Probleme auf:

- Aus unersichtlichen Gründen können manche Programme ihre selbst exportierten Daten nicht wieder korrekt importieren, was diese Programme direkt aus der Auswahl ausschließt.

- Obwohl XMI ein Austauschformat sein soll, funktioniert der Import von Daten, die mit anderen Programmen als dem Importierenden erzeugt wurden, für fast keine Paarung von Programmen.

- Es existieren mindestens drei gängige Versionen des XMI-Formats, die von verschiedenen CASE-Tools unterstützt werden. Diese Versionen sind untereinander nicht oder nur stark begrenzt kompatibel.

- Da XMI nur ein Austauschformat für Modelle ist, gehen Diagramminformationen wie die Position und Größe von Elementen verloren. Dies ist beim Update eines Diagramms ungünstig, da der Benutzer dann das komplette Diagramm wieder neu gestalten muss. Zwar geht auch beim Reverse Engineering von Quellcode diese Information verloren, jedoch ist ein Teilimport der betreffenden Elemente möglich, was nur eine Neugestaltung der veränderten Elemente nach sich zieht, der Rest des Diagramms aber erhalten bleibt.

Diese Erkenntnisse entsprechen denen, die bereits in einer der verwandten Arbeiten gemacht wurden, siehe Kapitel 3.2, [DHTT00]. Zwar sind die Voraussetzungen zum Einsatz von XMI als Austauschformat sehr günstig, doch die Umsetzung ist aufgrund fehlender Standardisierung in Bezug auf von CASE-Tools erzeugte Diagramme nicht praktikabel. Daraus folgt, dass die Benutzung des gemeinsamen Austauschformats ausgeschlossen werden kann.

Diskussion Lösungsansatz: Als Lösungsansatz bleibt die Erzeugung von Quellcode in den Programmiersprachen Java oder C++. Hier steht eine Diskussion der Lösungsansätze zur Darstellung von Mehrfachvererbung in Java aus. Diese ist gegen den Kompatibilitätsvorteil von Java und die größere Flexibilität bei der Auswahl eines CASE-Tools abzuwägen.
In Java wird Mehrfachvererbung als Konzept nicht unterstützt. Eine Klasse darf nur von einer Oberklasse erben. Es ist lediglich möglich, Interfaces zu definieren, von denen eine Klasse mehrere implementieren kann. Es gibt also keine Lösung, die Mehrfachvererbung in Java korrekt und ohne Verfälschung der zugrunde liegenden Semantik der *CS*-Klassen und deren Abbildung auf UML-Klassen, darstellt.
Werden trotzdem Interfaces zur Darstellung der Mehrfachvererbung eingesetzt, stellt sich die Frage, welche *CS*-Klassen durch Interfaces repräsentiert werden sollen, und welche durch normale Klassen. Dies müsste der LabVIEW Parser intelligent zur Laufzeit entscheiden. Da sich die *CS*-Klassenstruktur aber weiterentwickelt, müssten Regeln erstellt werden, nach denen der Parser entscheidet, wer Interface und wer Klasse wird. Dies führt nicht nur zu einem sehr komplizierten Code, sondern auch zu Inkonsistenzen bei der Erzeugung von Diagrammen verschiedener *CS*-Versionen. Damit ist diese Variante auszuschließen.
Als Alternative bleibt die Möglichkeit, alle *CS*-Klassen als Interfaces abzubilden, was zumindest konsistent und konsequent ist. Da Roundtrip Engineering aufgrund der in LabVIEW geschriebenen *CS*-Quellen nicht möglich ist, wird das Argument für die se-

mantische Korrektheit der Abbildung von *CS*-Klassen jedoch abgeschwächt, weil die so erzeugten Diagramme nicht für eine weitere automatisierte Erzeugung von *CS*-Quellen in Frage kommen.
Im Rahmen einer Diskussion mit dem Auftraggeber wurde präzisiert, welche Prioritäten die Argumente der verschiedenen Lösungsansätze für den Benutzer haben. Dabei liegt die größte Priorität auf der Flexibilität bei der Auswahl von CASE-Tools, sowie auf der Darstellung von Mehrfachvererbung im Klassendiagramm, gegenüber der semantisch korrekten Darstellung von *CS*-Klassen durch Interfaces oder Klassen. Daher wird zuerst ein Lösungsansatz gewählt, der Java-Quellen benutzt, wobei dem Benutzer eine Option gegeben wird, alle *CS*-Klassen als Interfaces oder als Klassen zu generieren. So kann der Benutzer entscheiden, welche Darstellung gerade für seinen Zweck geeignet ist und die passenden Quellen dazu erzeugen. In einem zweiten Schritt wird dann auch die Erzeugung von C++ Quellen realisiert, sofern die Implementierung in den Zeitrahmen der Bachelor-Thesis passt. Der Benutzer kann dann frei zwischen der Erzeugung verschiedener Quellen wählen.

Empfehlung zur Verwendung eines CASE-Tools: Auf der Basis dieser Diskussion wird der JDeveloper von Oracle für die weitere Arbeit verwendet. Das Programm setzt auf eclipse als Grundlage auf. Es ist daher mit Editoren für Quellcode und andere Textdateien ausgestattet, sowie einem Interpreter und vielen weiteren Werkzeugen zur Modellierung und Entwicklung von Entwicklungsprozessen und Quellen. Das Programm läuft stabil und stürzt auch bei größeren Diagrammen mit mehr als 50 Klassen nicht ab. Es verfügt über einen guten Layoutalgorithmus, kann ein Reverse Engineering auf Java Quellen ausführen sowie XMI Quellen und viele andere importieren und wird zum offiziellen Vorschlag zur Erzeugung von Diagrammen als Dokumentation von *CS*-Systemen gemacht.
Weiterhin ist die Verwendung des Omondo Plugins für eclipse zu empfehlen. Die Entwicklungsumgebung eclipse ist beliebt und bekannt und bietet viele Möglichkeiten der Softwareentwicklung und Modellierung.
Für die Erzeugung von HTML-Dokumenten inklusive Klassendiagrammen ist die Verwendung von Doxygen sehr zu empfehlen. Das Programm bietet umfassende Optionen zum Prozess der Erzeugung von HTML-Dokumenten, sowie LaTeX, XML, RTF und Man Pages. Obwohl es in der Tabelle der zur Auswahl stehenden Programme verzeichnet ist, kann es nur statische Dokumentationen erzeugen, und erfüllt somit nicht die Kriterien der Bearbeitbarkeit von Diagrammen an das zu verwendende CASE-Tool.
Natürlich steht es dem Benutzer frei, sich für ein anderes CASE-Tool seiner Wahl zu entscheiden, schließlich ist die Flexibilität bei der Auswahl eines CASE-Tools ein integraler

Bestandteil der Anforderungen an das Dokumentationswerkzeug. Dies soll lediglich ein Leitfaden zur Verwendung von CASE-Tools sein und verdeutlichen, welches zur Entwicklung des Dokumentationswerkzeuges verwendet wird.

6 Codierung

6.1 Einleitung

In diesem Kapitel wird die Codierung anhand eines Walkthroughs erklärt. Das bedeutet, es werden die im Entwurf beschriebenen Schritte einmal beispielhaft durchgespielt, wobei Ein- und Ausgangswerte präsentiert werden. Dabei wird auch auf die Unterschiede zwischen Entwurf und Implementierung eingegangen.
Darauf folgen eine Erklärung zu den durchgeführten Tests, ein Abschnitt zur Benutzerschnittstelle und der Bedienung des Dokumentationswerkzeuges, sowie ein Abschnitt zur Integration der Module und dem Gesamtsystemtest.
Da das Dokumentationswerkzeug in LabVIEW geschrieben wird, ist es nicht zwangsläufig erforderlich, das objektorientierte Design in allen Einzelheiten umzusetzen, da LabVIEW an sich keine objektorientierte Sprache ist. Um Design und Codierung konsistent zu halten, und Struktur in die Programmierung zu bringen, werden die umgesetzten Klassen des Designs in Form von VI durch zusammengehörige Verzeichnisse repräsentiert. Dabei ist keine Zuordnung von Sichtbarkeiten für die erstellten VI vorgesehen. Aufgrund der Unterschiede von objektorientiertem Design und datenflussgetriebener Programmiersprache weichen die tatsächlichen Ein- und Ausgangsparameter von den modellierten an Stellen ab, an denen die Objektorientierung nicht konsistent auf den Datenfluss übertragbar ist. Weiterhin wird die Klasse ExtrahierteMetadaten durch eine Control realisiert, “ExtrahierteMetadaten.ctl”.
Im Folgenden wird die Softwarekomponente, die im Anschluss an die Datenakquisition die Dateien mit Quellcode erzeugt, “Parser” genannt. Dieselbe Namensgebung trifft auch auf die von der Komponente implementierten Methoden zu (z.B. *parseTemplate*). Diese Namensgebung entstand während der iterativen Entwicklung des Werkzeuges. Dabei ist die Funktionalität des Lesens und neu Ordnens von Informationen aus “ExtrahierteMetadaten.ctl” (siehe Kapitel 6.2.2, Die Klasse “Extrahierte_Metadaten”), sowie des Bereitstellens von Informationen für nachgeschaltete Software maßgeblich in die hier verwendete Terminologie eingeflossen.

6.2 Walkthrough am Beispiel der Klasse 4WinsServer

In diesem Abschnitt wird die konkrete Codierung einmal beispielhaft für die Klasse 4WinsServer durchgeführt. Als erstes folgt eine kurze Beschreibung der Klasse und ihrer Eigenschaften. Danach werden die verschiedenen Schritte der Datenakquisition und des Parsers durchlaufen, und am Ende wird die erzeugte Datei durch das CASE-Tool importiert und

dargestellt.
Dabei werden die einzelnen in Abb. 22 und Abb. 25 beschriebenen Schritte durchgeführt. Zur Orientierung wird an die Bilder dieses Kapitels eine verkleinerte Version des jeweiligen Diagramms mit einer Markierung versehen hinzugefügt.

6.2.1 Beschreibung der Klasse 4WinsServer

Die Klasse 4WinsServer des *CS*-Systems ist Teil des "Vier Gewinnt" Spiels. Dabei benutzt ein menschlicher Vier Gewinnt Spieler die Klasse 4WinsClient. Die Klasse 4WinsServer verwaltet und verteilt Informationen über das laufende Spiel. "Sie nimmt Registrierungen, Zustandsänderungen und Abmeldungen von Clients entgegen und versendet nach jeder Änderung eine aktualisierte Spielerliste an alle registrierten Clients." (Vgl. [Bran05], S.121) Das Spiel selbst läuft jedoch nicht über den Server, sondern erfolgt im Peer-To-Peer Verfahren zwischen den Clients.
Im folgenden Bild (Abb. 31) ist die Projektansicht der Klasse 4WinsServer zu sehen, mit ihren Methoden, deren Sichtbarkeiten durch Verzeichniszugehörigkeit realisiert ist.

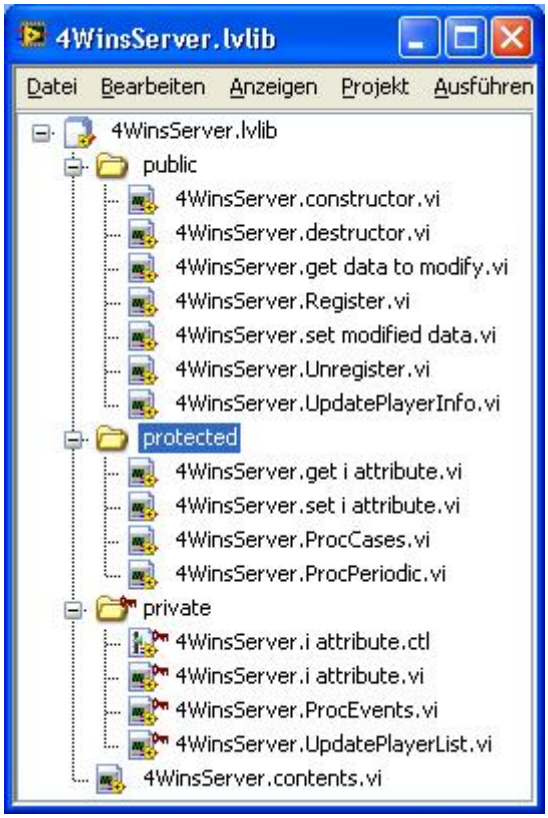


Abbildung 31: Projektansicht der 4WinsServer Library

Ein- und Ausgangsparameter sind in der Projektansicht nicht zu sehen. Daher folgt ein Ausschnitt aus der Kontexthilfe (Abb. 32), die die Ein- und Ausgangsparameter beispielhaft am Symbol des VI *get data to modify* mit Namen darstellt.

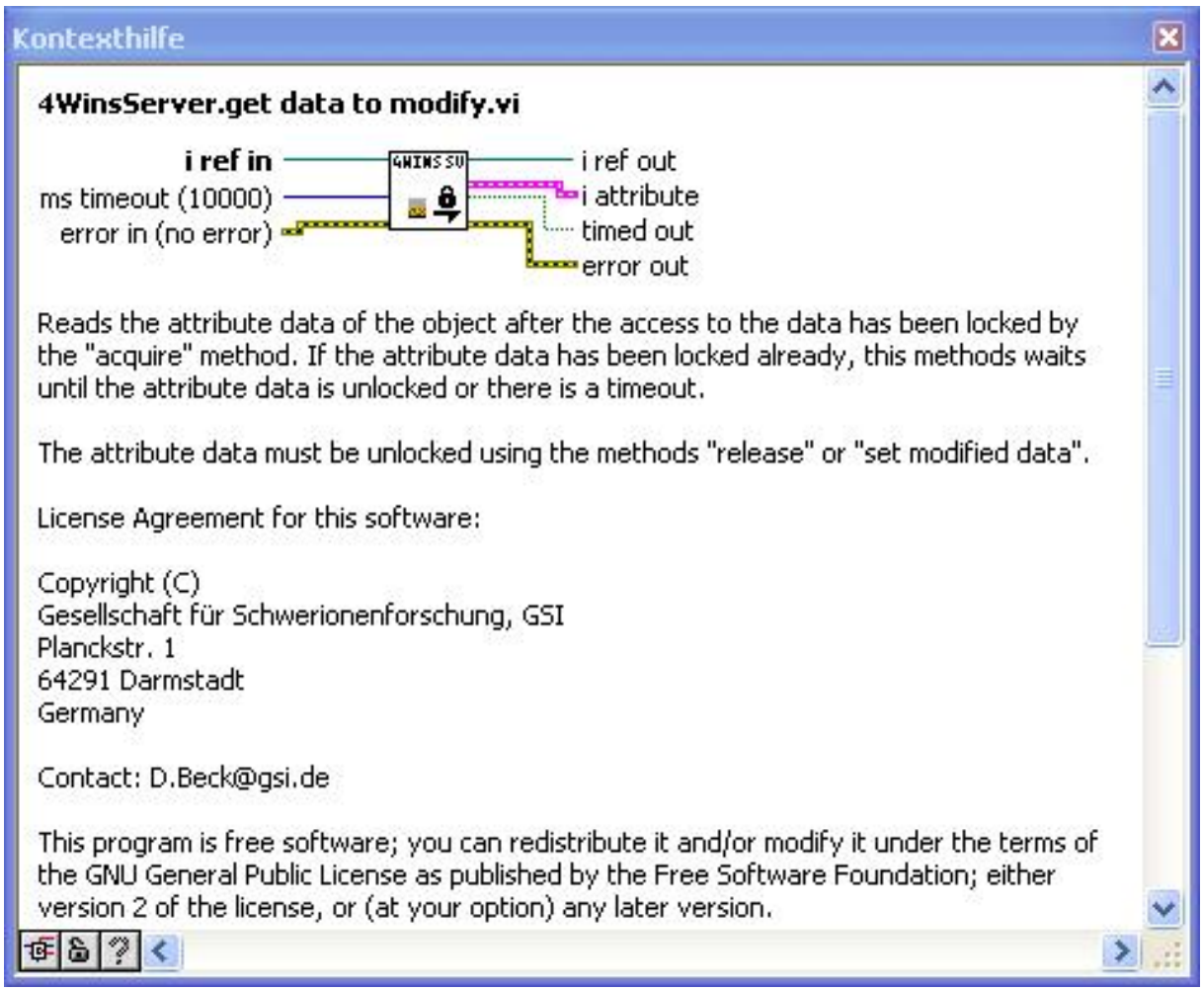


Abbildung 32: Kontexthilfe des VI *get data to modify*

Hier eine Zuordnung der entsprechenden Datentypen:

- 'i ref in'/ 'i ref out' ist vom Datentyp Referenz
- 'ms timeout' ist vom Datentyp Integer
- 'timed out' ist vom Typ Boolean
- 'error in'/ 'error out' ist vom Typ Error Cluster
- 'i attribute' ist ein Cluster vom Typ der Control der Klasse, der ein komplettes Set der Attribute der Klasse beinhaltet.

Das folgende Bild (Abb. 33) stellt die Attribute der Klasse dar. Zu sehen ist die Frontpanelansicht der Control "i attribute.ctl", die jede *CS*-Klasse per Konvention haben muss, um ihre Attribute zu speichern.

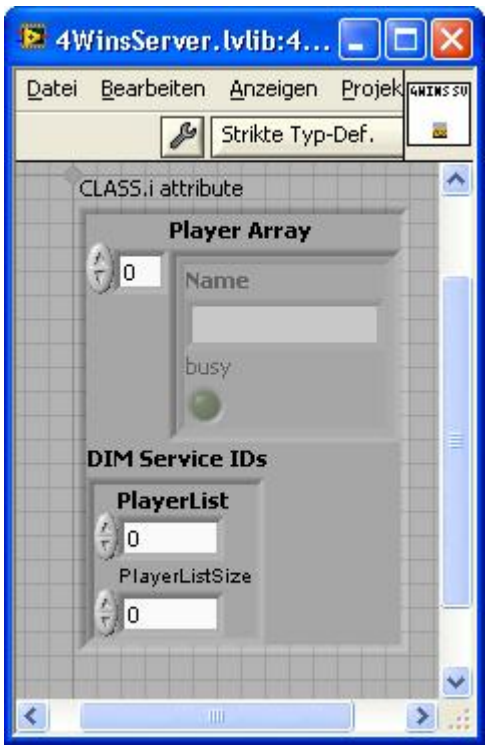


Abbildung 33: "i attribute.ctl" der Klasse 4WinsServer

Die Control besteht aus einem Array, "Player Array" und einem Cluster, "DIM Service IDs".
Das Array besteht wiederum aus Clustern, die aus einem String, "Name", und einem Boolean, "busy", zusammengesetzt sind.
Der Cluster hingegen besteht aus zwei Integer, "PlayerList" und "PlayerListSize".

Die Klasse 4WinsServer erbt von der Klasse BaseProcess. Dies ist am markierten Konstruktoraufruf der Klasse Base Process im Konstruktor von 4WinsServer zu erkennen (Abb. 34).

6.2.2 Beschreibung der Datenakquisition

In diesem Abschnitt wird die Klasse Extrahierte_Metadaten beschrieben. Aufgrund der Datenflussprogrammierung ohne Objektorientierung wird die Klasse von einer Control repräsentiert. Diese ist einem Struct aus C++ ähnlich und eignet sich daher zur Speicherung von Daten, vor allem da die Klasse Extrahierte_Metadaten selbst keine Operationen definiert. Anschließend wird der Walkthrough fortgeführt und die Unterschiede zwischen Entwurf und Codierung werden erläutert. Die Bilder zum Walkthrough wurden alle während eines Step-by-Step Debuggings erzeugt.

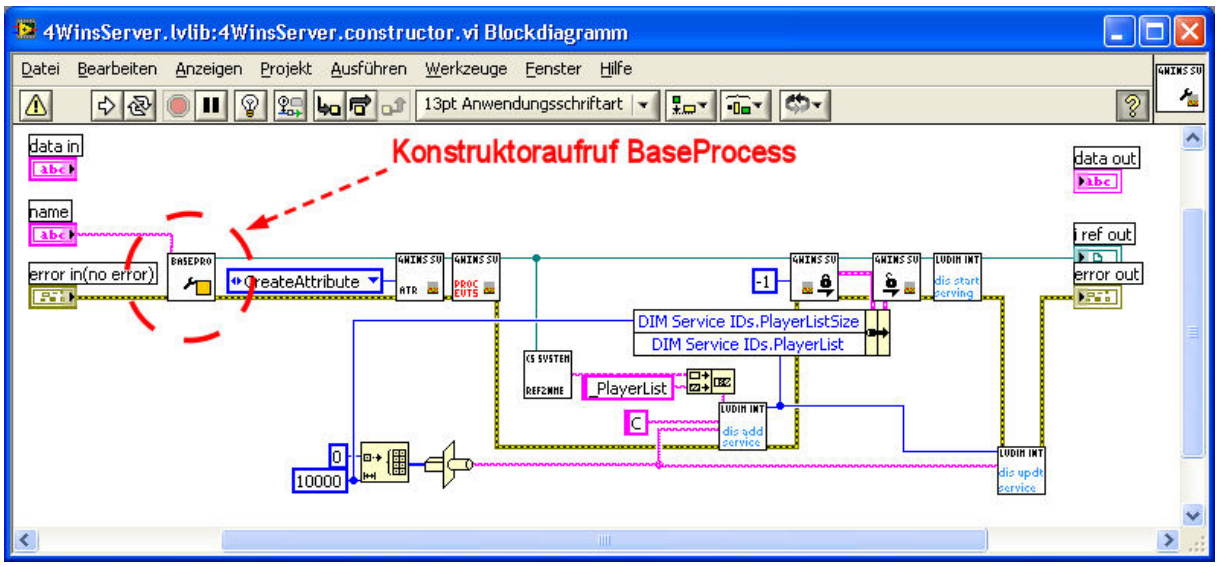


Abbildung 34: Konstruktor der Klasse 4WinsServer

Die Klasse Exrahierte_Metadaten: Zu Beginn der Datenakquisition wird eine Control eingerichtet, "ExtrahierteMetadaten.ctl", die die Klasse Extrahierte_Metadaten repräsentiert (Abb. 35). Sie speichert die Werte der Datenakquisition und reicht sie an die SubVI weiter. Die Control besteht aus einem Array "Methoden" das wiederum aus einem Cluster besteht, der alle relevanten Informationen zu den analysierten VI speichert. Jeder Eintrag im Array entspricht also einer Methode. In diesem Cluster sind der Name der Methode, die Sichtbarkeit, textuelle Beschreibung, sowie Ein- und Ausgangsparameter gespeichert. Ein- und Ausgangsparameter werden wieder als Arrays gespeichert, wobei hier Angaben über Name, Datentyp und textuelle Beschreibung als ein einziger String gespeichert sind. Das Array der Methoden stellt also eine Klasse dar, da auch die Attribute einer *CS*-Klasse als Eingangsparameter des VI "i attribute.vi" gespeichert sind. Alle bei der Datenakquisition gefundenen Werte werden ohne weitere Manipulation als Strings gespeichert.

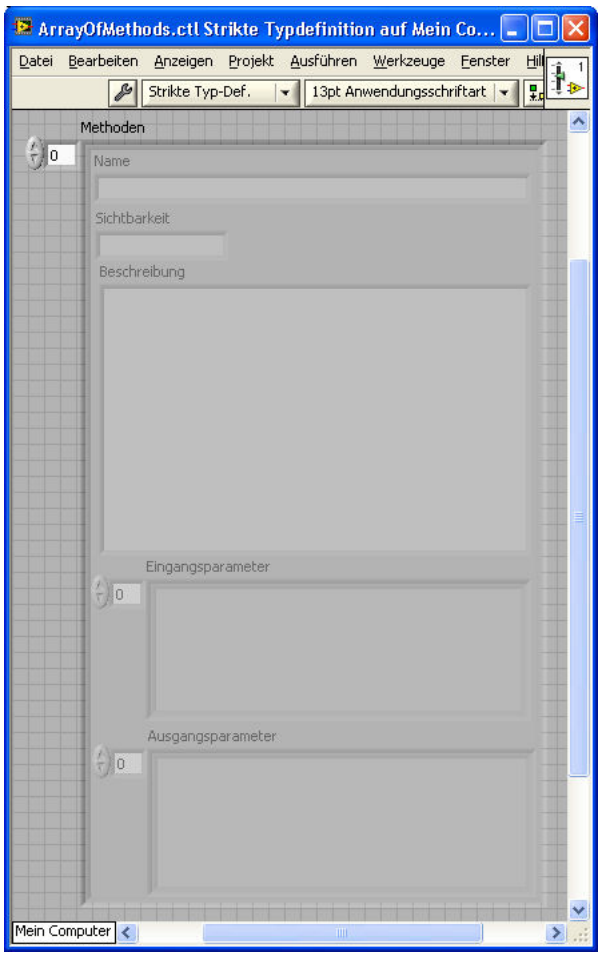


Abbildung 35: Die leere Control "ExtrahierteMetadaten"

Walkthrough: Als erstes werden die Namen der Methoden einer *CS*-Klasse mit ihren Sichtbarkeiten zusammen in der Control abgespeichert. Die in Abb. 36 gezeigte Methode, *get data to modify* steht an 3. Stelle im Array.

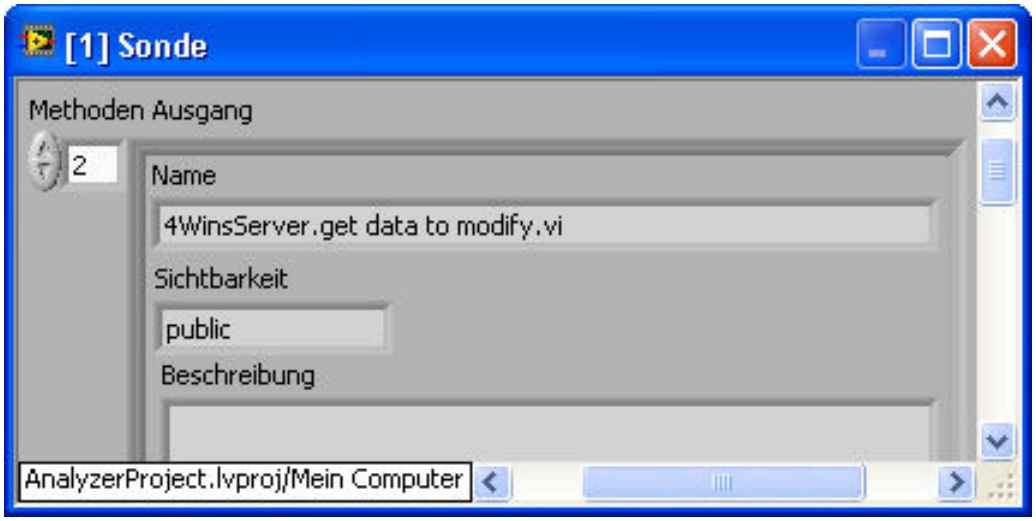


Abbildung 36: Sichtbarkeit der Methoden der Klasse 4WinsServer

Darauf folgt die Suche nach der textuellen Beschreibung der LabVIEW Dokumentation. Abb. 37 zeigt einen Ausschnitt der Control nach Speicherung der Beschreibung der Methode *get data to modify*.

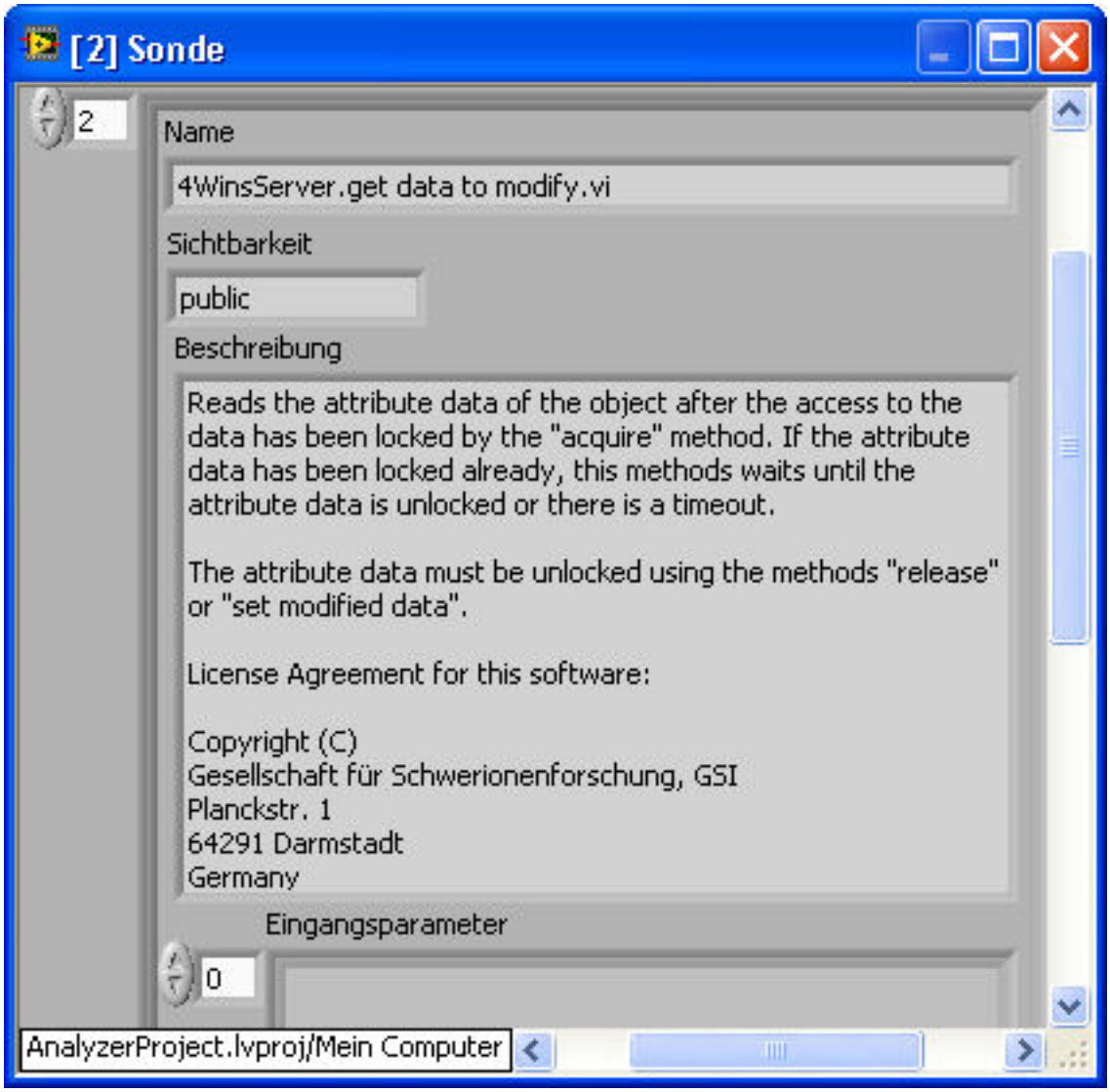


Abbildung 37: Beschreibung der Methoden in der Control ExtrahierteMetadaten

Am Ende werden die Ein- und Ausgangsparameter hinzugefügt (siehe Abb. 38). In diesem Beispiel werden jeweils ein Ein- und ein Ausgangsparameter der Methode *get data to modify* gezeigt (siehe auch Abb. 32).

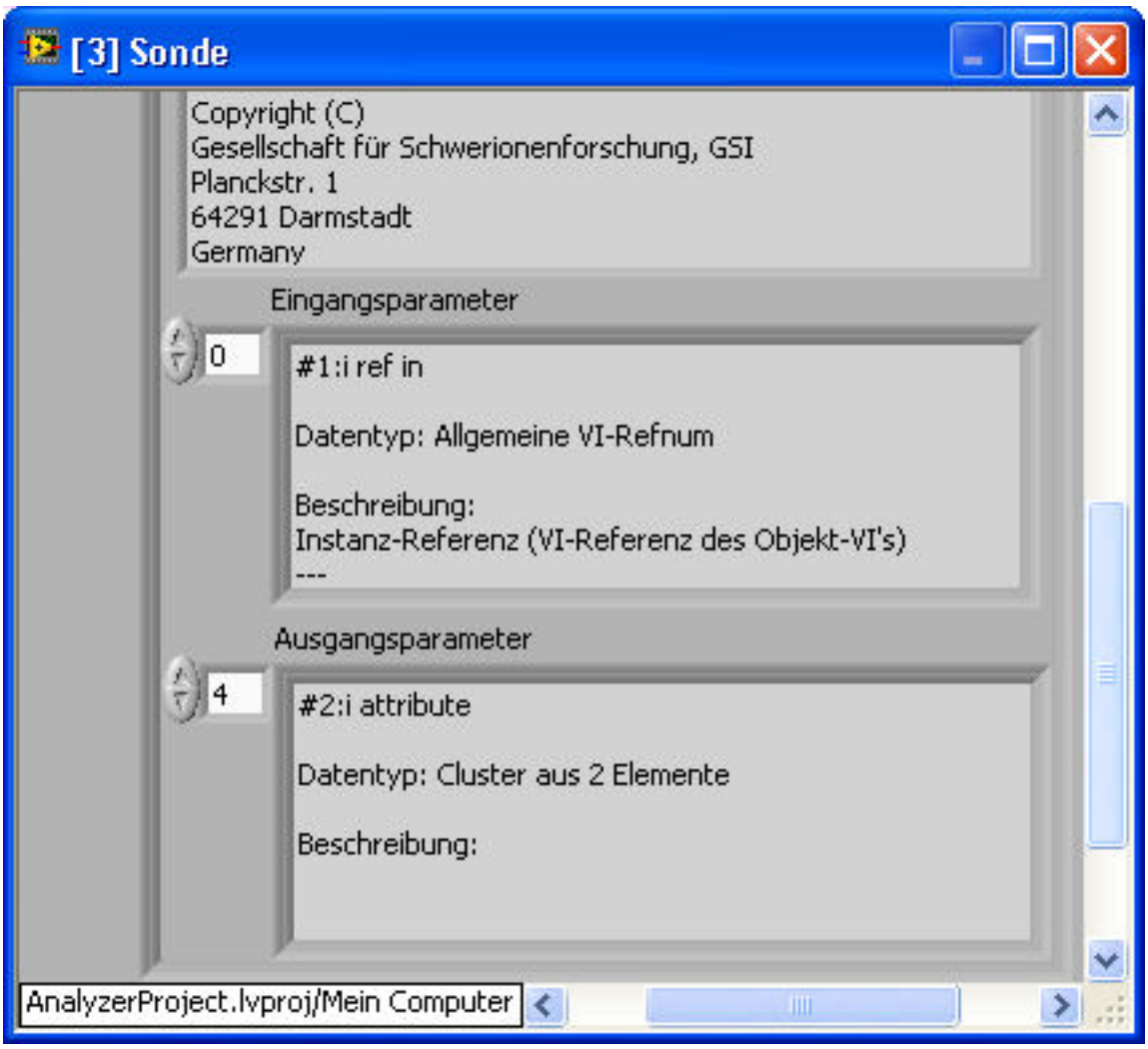


Abbildung 38: Arrays der Ein- und Ausgangsparameter der Methoden der Klasse 4Wins-Server

Unterschiede zwischen Entwurf und Codierung: Zwar sind *getVisibilityOfMethods* und *getDescription*, *getIOParameters* und *getInheritance* im Entwurf parallel modelliert (siehe Abb. 22), werden hier aber seriell codiert, da es einfacher ist, auf bereits bekannte Informationen aus ExtrahierteMetadaten zuzugreifen, als die selbe Information in jeder parallelisierten Methode wieder neu herauszufinden. Dazu müssen die Methoden allerdings aufeinander warten, bis die jeweils neusten Informationen aus der Control Extrahierte-Metadaten zur Verfügung stehen. Weiterhin ist es im Zuge der Modularisierung einfacher und sinnvoller die aktuell herausgefundene Information abzuspeichern, und dann an das nächste Modul weiterzugeben, anstatt alle Informationen erst separat zu suchen und am Ende zusammenzubauen.

Parallel zur Datenakquisition der Methoden wird *getInheritance* ausgeführt. Da es für jede *CS*-Klasse nur eine Vererbungshierarchie gibt, die sich nicht für jede Methode ändert,

macht es keinen Sinn die Vererbungshierarchie zusammen mit den Informationen über Methoden in der Control abzuspeichern.

6.2.3 Beschreibung des Parsers

Im folgenden Abschnitt wird der Walkthrough fortgesetzt, um die Funktionsweise des Parsers zu erläutern.
Der Parser schreibt die in der Control ExtrahierteMetadaten gespeicherten Daten jetzt in eine Textdatei, die später als Java Quellcode verwendet wird.
Wie im Abschnitt Diskussion und Lösungsansatz erläutert, werden hier zur Repräsentation der Mehrfachvererbung von der Benutzerschnittstelle zwei Optionen Angeboten. Alle Klassen können mit normalen Vererbungsbeziehungen durch das Schlüsselwort "extends" bei einfacher Vererbung und zusätzlich willkürlicher Verwendung des Schlüsselworts "implements" erzeugt werden. Die zweite Option ist, alle Klassen als Interfaces mit mehrfacher Verwendung des Schlüsselworts "extends" zu erzeugen. In diesem Walkthrough wurde die normale Klassenerzeugung ausgewählt.
Weiterhin werden hier zum Erzeugen der Namen von Klassen, Methoden, Attributen und Ein- und Ausgangsparameter die VI's *deleteLeadingNumber* und *eraseUnparseableSymbols* verwendet. Da es in LabVIEW keine Namenskonvention gibt, werden hier häufig Symbole verwendet, die in Java nicht zugelassen sind. Um möglichst syntaktisch korrekten Quellcode zu erzeugen werden diese daher geändert. Für Klassennamen, die mit einer Zahl beginnen, wird daher ein Dialog implementiert, der den Benutzer nach einem neuen Namen für die modellierte Klasse fragt, da hier nicht automatisch entschieden werden kann, wie der Name geändert werden soll. Die restlichen Namen werden allerdings aufgrund der geringeren Priorität ohne Dialog einfach geändert. Dabei werden einige Zeichen durch Unterstriche ersetzt, andere ganz entfernt.
Die Codierung enthält außerdem noch statisch angelegte Klassen, die in einem festen Verzeichnis gespeichert sind, und vom Parser in das vom Benutzer angegebene Zielverzeichnis kopiert werden. Diese Klassen dienen zur Unterstützung der LabVIEW Datentypen, die kein primitives Äquivalent in Java besitzen. Sie sind lediglich definiert und enthalten keinen Code. Dadurch werden mögliche Fehler reduziert, die beim Import eines Dateisystems durch ein CASE-Tool auftreten können. Außerdem ist es so einfacher möglich, die entsprechenden komplexen Datentypen bei der Verwendung des CASE-Tools in einem erzeugten Diagramm zu modellieren. Die statisch hinzugefügten Klassen gehören einem eigenen Paket, "csSupport.javaDatatypeSupport", an.
Ein weiteres Problem, das während der Codierung auftritt, ist die Handhabung von Clustern als Ein- und Ausgangsparameter von Methoden. In LabVIEW existieren aufgrund

der Datenflusssteuerung typischerweise sehr viele Ein- und Ausgangsparameter bei Methoden. Oft sind diese zu Clustern zusammengefasst, um eine gewisse Semantik zu erhalten, so z.B. bei Error Clustern (siehe auch Abb. 46). Die Zuordnung zu einer semantischen Einheit ist aber durch keine Konvention gestützt, und kann willkürlich erfolgen. Deshalb ist es nicht möglich, einen solchen Cluster als komplexen Datentyp durch eine eigene Klasse automatisiert zu modellieren, wie etwa beim Paket "javaDatatypeSupport". Wird der Cluster selbst als eigener, nicht weiter spezifizierter Datentyp modelliert, also als Klasse Cluster, dann gehen die Informationen über die tatsächlichen Parameter verloren. Alternativ ist es möglich, alle Attribute des Clusters als Ein- und Ausgangsparameter der Methode selbst zu modellieren. Dabei geht dann allerdings die durch die Zugehörigkeit zum Cluster beschriebene Semantik verloren.
Für Eingangsparameter wird vorläufig die Variante implementiert, bei der alle Attribute der Cluster direkt als Eingangsparameter der Methode modelliert werden. Um die syntaktische Korrektheit des erzeugten Codes zu gewährleisten, werden die durch die Auflösung eines Clusters doppelt auftretenden Namen von Attributen mit einer laufenden Nummer versehen (siehe auch Abb. 42). Die Cluster werden dabei allerdings nicht vollständig aufgelöst, sondern werden als Parameter mit dem Datentyp Cluster vor ihren beinhalteten Parametern zusätzlich aufgeführt. Alternativ dazu ist für eine spätere Version des Dokumentationswerkzeuges vorgesehen, die im Cluster beinhalteten Werte durch einen nicht näher spezifizierten Datentyp "Clusterelement" zu klassifizieren.
Für Ausgangsparameter wird die Variante der Klassendarstellung implementiert. Es wird also bei mehr als einem Ausgangsparameter ein nicht näher spezifizierter Datentyp mit dem Namen "Cluster" zurückgegeben.
Als Alternative dazu ist noch eine an C angelehnte Methodik zu nennen, bei der eine Übergabe der Parameter per Referenz stattfindet. Dabei wird der Rückgabetyp bei mehr als einem Ausgangsparameter als "void" modelliert. Gleichzeitig werden die tatsächlichen Ausgangsparameter in der Liste der Eingangsparameter aufgeführt, wobei diese ein Präoder Suffix erhalten, so dass sie als Ausgangsparameter erkennbar sind. Bei dieser Variante treten allerdings dieselben Probleme auf, wie für Eingangsparameter beschrieben.

In diesem Abschnitt können keine Screenshots gemacht werden, da die Sondendarstellung des Debuggers von LabVIEW nur einen sehr kleinen einsehbaren Ausschnitt mit Scrollbar erlaubt. Daher wird der erzeugte Quellcode in Typewriter-Schriftart dargestellt.

Fortsetzung des Walkthroughs: Als erstes wird der Code gezeigt, der von der Methode *parseTemplate* erzeugt wird (Abb. 39). Sie erstellt das Gerüst einer Java Klasse, bestehend aus der Definition des Klassennamens inklusive Rumpf, einem leeren Konstruk-

tor, sowie Kommentaren zur Orientierung für den Parser und den späteren menschlichen Leser. Weiterhin werden eine Paketdeklaration und zwei Importbefehle geschrieben.
Am Ende wird die erzeugte Datei von der jeweils aktuellen Methode geschlossen, zur nächsten Methode weitergegeben und dort wieder geöffnet.

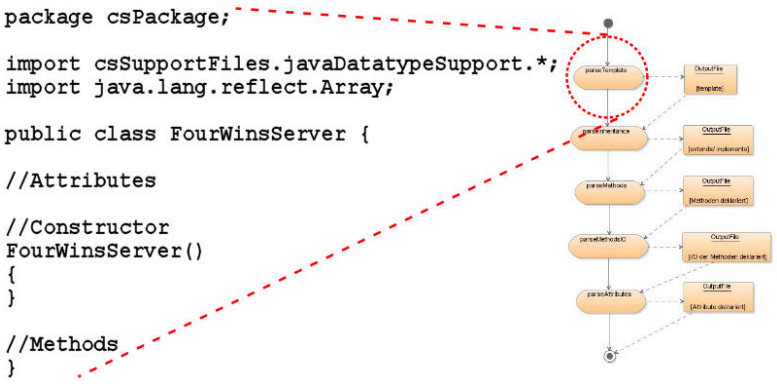


Abbildung 39: *parseTemplate*

Die erzeugte Textdatei wird von der nächsten Methode, *parseInheritance*, wieder geöffnet. Zusätzlich wird die temporäre Datei, in der die Vererbungshierarchie der aktuell bearbeiteten Klasse gespeichert ist, geöffnet.
Der Parser sucht dann nach dem Klassennamen und fügt hinter diesen die Vererbung an (Abb. 40). Dabei wird für die Option "make all classes" die erste Klasse, die der Parser in der temporären Datei findet, als Vererbungsbeziehung mit "extends" geschrieben, und die folgenden als "implements". Dabei kann die Reihenfolge der Klassen, die mit "extends" oder "implements" geschrieben werden, nicht bestimmt werden (siehe Kapitel 5.4, Diskussion Lösungsansatz). Die Klasse 4WinsServer erbt allerdings nur von der Klasse BaseProcess.

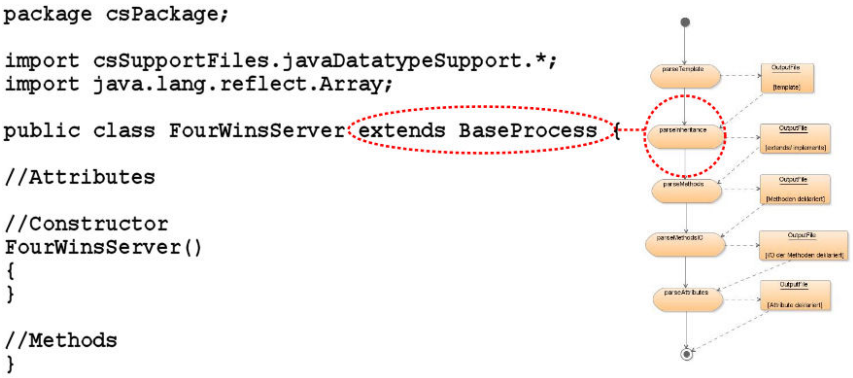


Abbildung 40: *parseInheritance*

Die Methode “parseMethods” sucht nach der Kommentarzeile “//Methods”. Angefangen in der nächsten Zeile werden die Methodenname inklusive Sichtbarkeit und leeren Methodenrumpf in jeweils eine eigene Zeile geschrieben (Abb. 41). In der Prototypversion werden zur Vollständigkeit der *CS*-Klasse noch alle gefundenen VI als Methode geführt, so zum Beispiel auch der Konstruktor. Für spätere Versionen ist allerdings vorgesehen, unnötige oder redundante Informationen auszuschließen. Zusätzlich werden zum leeren Standardkonstruktor weitere Konstruktoren, je nach *CS*-Klasse, mit entsprechenden Eingangsparametern erzeugt. Destruktoren existieren zwar in Java nicht, allerdings ist es aufgrund der vollständigen Abbildung der *CS*-Klassen auf Java-Quellcode sinnvoll den *CS*-Destruktor auf eine Methode abzubilden. Weiterhin können auch in Java Speicherverwaltungstätigkeiten manuell vorgenommen werden.

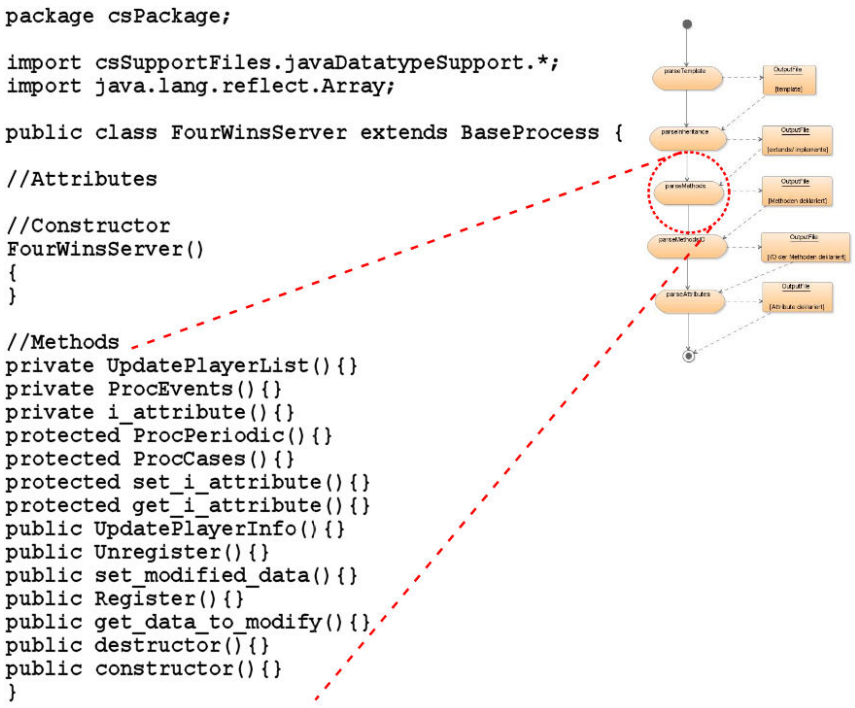


Abbildung 41: *parseMethods*

Die Methode *parseMethodsIO* sucht den Namen der aktuell bearbeiteten Methode, und schreibt die Ein- und Ausgangsparameter (Abb. 42).
Dabei werden die Felder der Ein- und Ausgangsparameter der Control “ExtrahierteMetadaten” durchsucht, um Namen und Datentyp zu finden. Den LabVIEW Datentypen werden dabei primitive Java Datentypen und normalisierte komplexe Datentypen in Form der im Paket javaDatatypeSupport enthaltenen Klassen zugeordnet.
Weiterhin ist hier die Lösung der beschriebenen Clusterdarstellungsproblematik zu sehen. Die Rückgabetypen der Methoden sind vom Typ “Cluster”, ohne weitere Spezifikation der

eigentlichen Inhalte. Des Weiteren ist die Nummerierung von Parametern zu sehen, die als Lösung zu doppelt auftretenden Namen durch Auflösung eines Clusters implementiert wurde.

```
package csPackage;

import csSupportFiles.javaDatatypeSupport.*;
import java.lang.reflect.Array;

public class FourWinsServer extends BaseProcess {

//Attributes

//Constructor
FourWinsServer()
{
}

//Methods
private Cluster UpdatePlayerList( VIRefnum i_ref_in,
    Array Player_Array, Cluster Player1, String Name, boolean busy,
    String Object, Cluster error_in_, boolean status, int code, String source ){}
private Cluster  ProcEvents( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source ){}
private Cluster i_attribute( Cluster CLASS_i_attribute,
    Array Player_Array, Cluster Player1, String Name, boolean busy,
    String Object, Cluster DIM_Service_IDs, int PlayerList, int PlayerListSize,
    int Action, VIRefnum VI_Refnum, Cluster error_in_, boolean status,
    int code, String source ){}
protected Cluster ProcPeriodic( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source, int ms_timeout_ ){}
protected Cluster ProcCases( Cluster call_cluster, String Object,
    String Selector, int CallType, String Async_Object, String Async_Selector,
    String Data_Descriptor, Array Data1, int Numeric, int Error_, int Expire,
    String selector, VIRefnum i_ref_in, Cluster error_in_, boolean status,
    int code, String source, Array Data2, int Numeric3 ){}
protected Cluster set_i_attribute( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source, Cluster i_attribute,
    Array Player_Array, Cluster Player1, String Name, boolean busy,
    String Object, Cluster DIM_Service_IDs, int PlayerList,
    int PlayerListSize ){}
protected Cluster get_i_attribute( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source ){}
public Cluster UpdatePlayerInfo( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source, Cluster Player, String Name,
    boolean busy, String Object, int ms_timeout_ ){}
public Cluster Unregister( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source, int ms_timeout_, String Object ){}
public Cluster set_modified_data( VIRefnum i_ref_in,
    Cluster error_in_, boolean status, int code, String source,
    Cluster i_attribute, Array Player_Array, Cluster Player1, String Name,
    boolean busy, String Object, Cluster DIM_Service_IDs, int PlayerList,
    int PlayerListSize ){}
public Cluster Register( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source, Cluster Player, String Name,
    boolean busy, String Object, int ms_timeout_ ){}
public Cluster get_data_to_modify( VIRefnum i_ref_in,
    Cluster error_in_, boolean status, int code, String source,
    int ms_timeout_ ){}
public Cluster destructor( Cluster error_in, boolean status, int code,
    String source, VIRefnum i_ref_in ){}
public Cluster  constructor( Cluster error_in, boolean status, int code,
    String source, String data_in, String name ){}
}
```

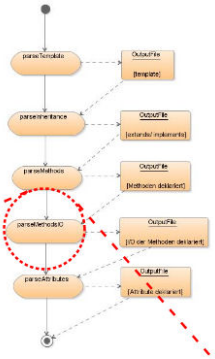


Abbildung 42: *parseMethodsIO*

Am Ende werden die Attribute durch die Methode *parseAttributes* geschrieben (Abb. 43). Dabei sucht die Methode nach der Kommentarzeile "//Attributes" und schreibt die Attribute, die in der Form der Eingangsparameter der als VI gespeicherten Control "i

attribute" vorliegen jeweils in eine eigene Zeile. Da auch hier die genannte Clusterdarstellungsproblematik auftritt, wird dieselbe Lösung verwendet wie beim Erzeugen der Eingangsparameter der Methoden.

```
package csPackage;

import csSupportFiles.javaDatatypeSupport.*;
import java.lang.reflect.Array;

public class FourWinsServer extends BaseProcess {

//Attributes
int PlayerListSize;
int PlayerList;
Cluster DIM_Service_IDs;
String Object;
boolean busy;
String Name;
Cluster Player1;
Array Player_Array;
Cluster CLASS_i_attribute;

//Constructor
FourWinsServer()
{
}

//Methods
private Cluster UpdatePlayerList( VIRefnum i_ref_in,
    Array Player_Array, Cluster Player1, String Name, boolean busy,
    String Object, Cluster error_in_, boolean status, int code, String source ){}
private Cluster  ProcEvents( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source ){}
private Cluster i_attribute( Cluster CLASS_i_attribute,
    Array Player_Array, Cluster Player1, String Name, boolean busy,
    String Object, Cluster DIM_Service_IDs, int PlayerList, int PlayerListSize,
    int Action, VIRefnum VI_Refnum, Cluster error_in_, boolean status,
    int code, String source ){}
protected Cluster ProcPeriodic( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source, int ms_timeout_ ){}
protected Cluster ProcCases( Cluster call_cluster, String Object,
    String Selector, int CallType, String Async_Object, String Async_Selector,
    String Data_Descriptor, Array Data1, int Numeric, int Error_, int Expire,
    String selector, VIRefnum i_ref_in, Cluster error_in_, boolean status,
    int code, String source, Array Data2, int Numeric3 ){}
protected Cluster set_i_attribute( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source, Cluster i_attribute,
    Array Player_Array, Cluster Player1, String Name, boolean busy,
    String Object, Cluster DIM_Service_IDs, int PlayerList,
    int PlayerListSize ){}
protected Cluster get_i_attribute( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source ){}
public Cluster UpdatePlayerInfo( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source, Cluster Player, String Name,
    boolean busy, String Object, int ms_timeout_ ){}
public Cluster Unregister( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source, int ms_timeout_, String Object ){}
public Cluster set_modified_data( VIRefnum i_ref_in,
    Cluster error_in_, boolean status, int code, String source,
    Cluster i_attribute, Array Player_Array, Cluster Player1, String Name,
    boolean busy, String Object, Cluster DIM_Service_IDs, int PlayerList,
    int PlayerListSize ){}
public Cluster Register( VIRefnum i_ref_in, Cluster error_in_,
    boolean status, int code, String source, Cluster Player, String Name,
    boolean busy, String Object, int ms_timeout_ ){}
public Cluster get_data_to_modify( VIRefnum i_ref_in,
    Cluster error_in_, boolean status, int code, String source,
    int ms_timeout_ ){}
public Cluster destructor( Cluster error_in, boolean status, int code,
    String source, VIRefnum i_ref_in ){}
public Cluster  constructor( Cluster error_in, boolean status, int code,
    String source, String data_in, String name ){}
}
```

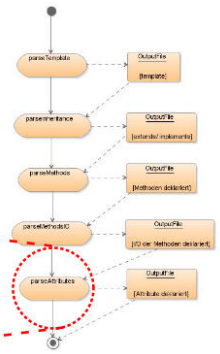


Abbildung 43: *parseAttributes*

6.2.4 Import und Darstellung im CASE-Tool

Die fertig erstellte Java-Datei wird jetzt vom CASE-Tool importiert und dargestellt.
Der erste Schritt ist der Import der Quelldateien in den Workspace des Programms. Dadurch entsteht eine Kopie der importierten Datei im Workspace, und der in der Datei enthaltene Quellcode kann durch einen Editor bearbeitet und begutachtet werden.
Im nächsten Schritt wird ein neues Klassendiagramm erzeugt oder ein bestehendes geöffnet, und die Java Datei kann per Drag 'n' Drop ins Diagramm gezogen werden. Dadurch wird automatisch eine Klassendarstellung der Datei im Diagramm erzeugt. Das Diagramm kann jetzt verändert und gespeichert werden.

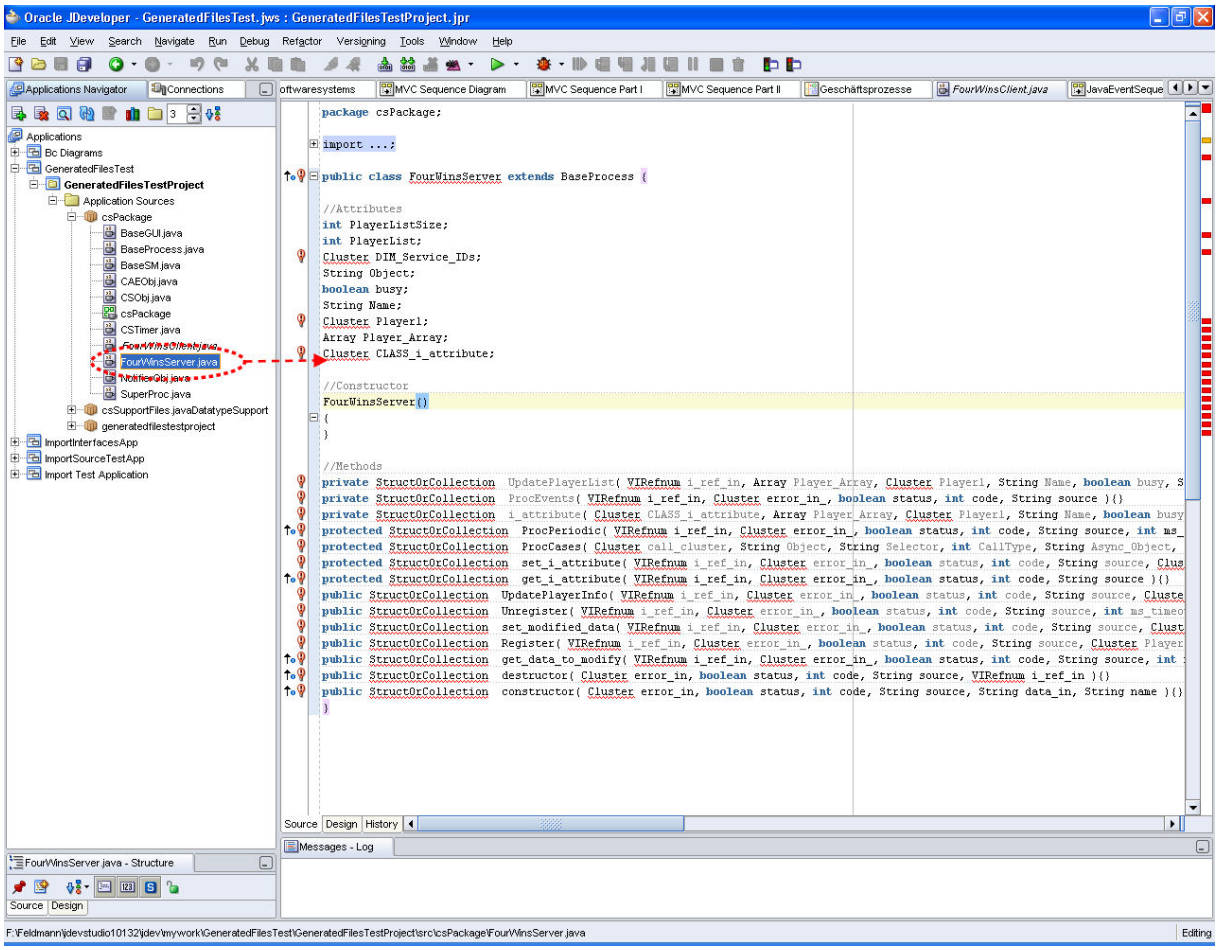


Abbildung 44: Ansicht des importierten Quellcodes im Editor

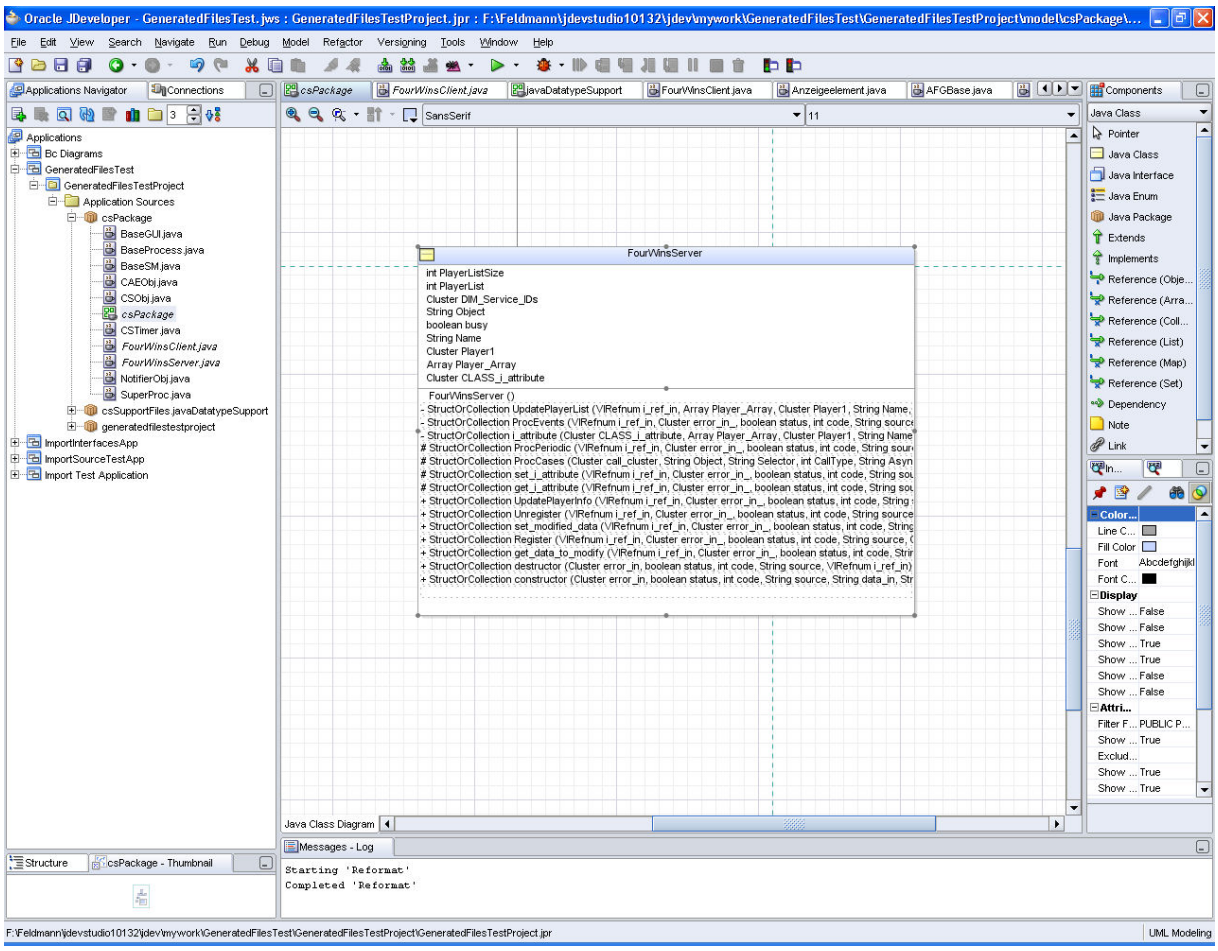


Abbildung 45: Ansicht der importierten Klasse im Diagramm

6.3 Tests

Das Dokumentationswerkzeug wird im Rahmen einer Implementierung als Prototyp nur sehr wenigen Tests unterzogen. Automatisierte Tests mit vielen Eingabewerten werden nicht durchgeführt, da die normale LabVIEW-Entwicklungsumgebung keine automatisierten werkzeuggestützten Testsysteme unterstützt. Zwar würden sich Datenflusstests aufgrund der Struktur der LabVIEW-Programme eignen, sind allerdings im Rahmen dieser Arbeit zu aufwendig. Auch werden keine Performancetests gemacht, da die Performance bei der Benutzung der entwickelten Software innerhalb akzeptabler Grenzen bleibt und auch nicht systemkritisch ist. Getestet wird lediglich im Rahmen der normalen Anwendung des Werkzeugs durch Funktionalitätstests. Das bedeutet, die Komponenten werden mit normalen *CS*-Klassen und deren VI getestet, die nicht speziell für einen bestimmten Test konstruiert werden. Weiterhin werden gängige Programmierrichtlinien zur Fehlervermeidung eingehalten.

Dabei wird auf eine möglichst vollständige Verwendung von Error Clustern geachtet. Ein Error Cluster ist ein standardisierter Datentyp von LabVIEW zur Fehlerbehandlung. Er

besteht aus einem Boolean, “Status”, einem Integer “Code” und einem String “Source” (siehe Abb. 46). Status zeigt an, ob bei der Ausführung ein Fehler aufgetreten ist. Code zeigt den Error Code, über den anhand einer Tabelle weitere Informationen zum aufgetretenen Fehler gefunden werden können. Source beschreibt die Quelle des aufgetretenen Fehlers. Error Cluster sind für die meisten komplexen VI von LabVIEW schon implementiert. Error Cluster werden durch VI durchgereicht. Für VI, die Error Cluster unterstützten, existiert daher oft ein Eingang, mindestens aber ein Ausgang für einen Parameter des Datentyps Error Cluster. Dabei wird der Ausgang eines VI mit dem Eingang des nächsten VI verbunden, wobei der Fehler so durch das ganze Programm fließt und am Ende angezeigt wird. So können die Felder “Source” und “Code” der Standard-Error Cluster von LabVIEW auch erweitert werden, um eigene Fehlerbeschreibungen hinzuzufügen, wie etwa einen eigenen Fehlercode mit näherer Beschreibung des Fehlers. Weiterhin ist es auf diese Weise möglich, am Anfang einer Routine den Error Cluster am Eingang auf Fehler zu prüfen, und eine geeignete Fehlerbehandlung einzuleiten. Dabei wird ein VI, das mit dem Error Status “true” aufgerufen wird, per Konvention nicht ausgeführt. Stattdessen wird nur der Fehler durchgereicht, um ähnlich des “try{} catch(Exception e){}” Konzepts von Java am Ende einer Sequenz eine zentrale Fehlerbehandlung durchzuführen, anstatt für jedes VI extra eine eigene Fehlerbehandlung schreiben zu müssen.

Abbildung 46: Error Cluster

Ein Gesamtsystemtest wird im Rahmen der Dokumentation des gesamten *CS*-Systems durchgeführt. Entscheidend für die Funktionsfähigkeit des Prototyps ist dabei, Quellen zu generieren, die sich ohne Fehler vom CASE-Tool importieren lassen. Eine möglichst vollständige und fehlerfreie Generierung aller Einzelkomponenten jeder *CS*-Klasse ist wünschenswert, aber im Rahmen der Implementierung des Prototyps nicht notwendig, da es sich dabei in erster Linie um eine Studie handelt.

6.4 Bedienung des Dokumentationswerkzeuges

In diesem Abschnitt wird die Handhabung des Dokumentationswerkzeuges beschrieben. Da das Dokumentationswerkzeug ein Prototyp ist, wird über diesen Abschnitt hinaus kein Benutzerhandbuch geschrieben.
Es folgt ein Abschnitt zur Installation des Programms. Danach wird die Benutzerschnittstelle des LabVIEW Teils beschrieben. Zur Handhabung des CASE-Tools wird auf Kapitel 6.2.4 verwiesen, da die Verwendung des JDeveloper nur eine Empfehlung an den Benutzer ist, und der Ablauf des Imports für CASE-Tools immer ähnlich der Beschreibung im Walkthrough abläuft.

6.4.1 Installation

Um das Dokumentationswerkzeug verwenden zu können, ist eine vorinstallierte Version 8.2 von LabVIEW auf dem zur Benutzung vorgesehenen PC erforderlich. Weiterhin ist ein bereits installiertes CASE-Tool notwendig (siehe auch Tabellen 1 und 2).
Sind diese Voraussetzungen erfüllt, wird das Dokumentationswerkzeug an einen beliebigen Ort auf dem PC entpackt, und das VI “documentateCS.vi” durch einen Doppelklick gestartet.

6.4.2 Benutzerschnittstelle

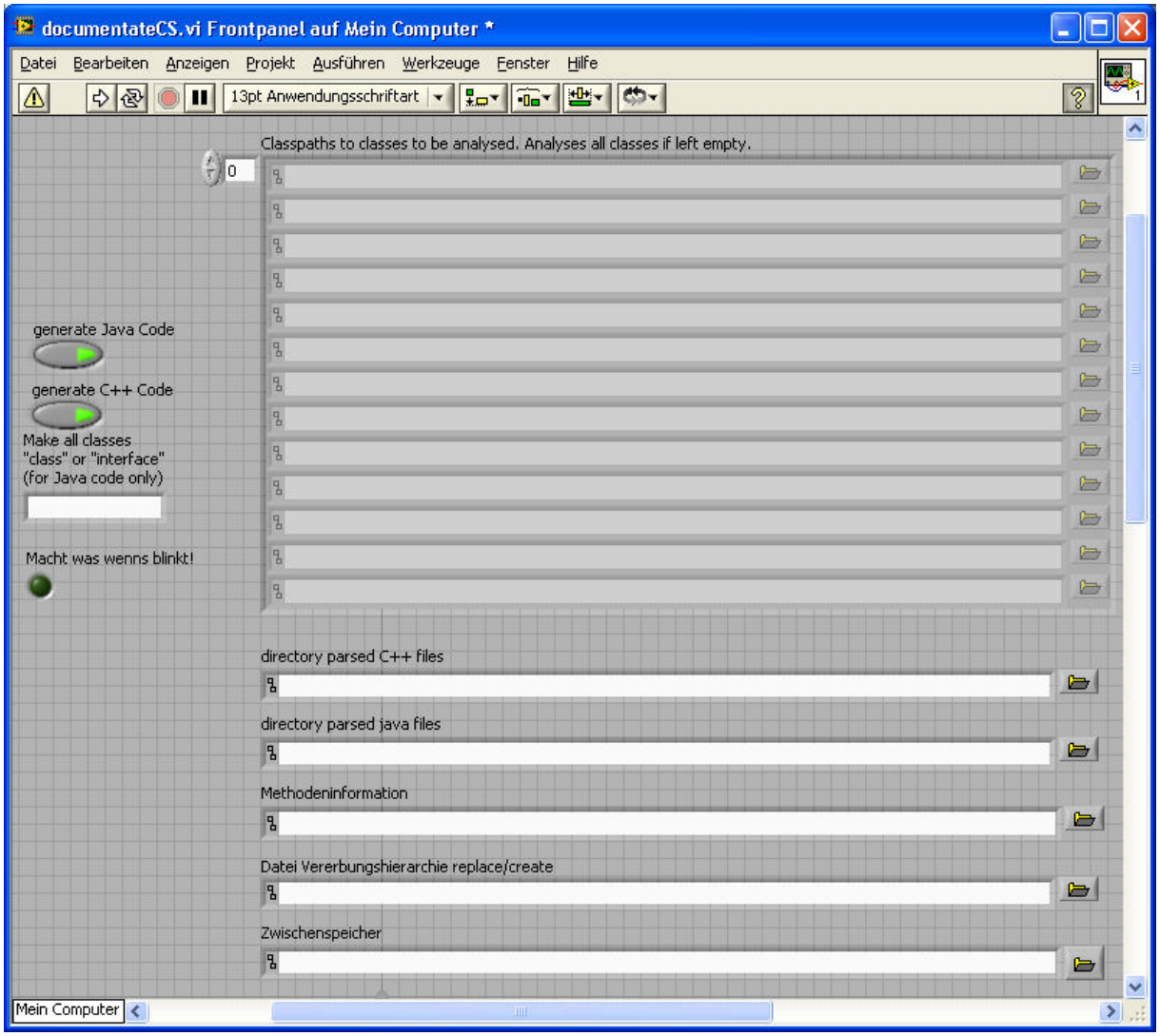


Abbildung 47: Benutzerschnittstelle

Nachdem die Datei “documentateCS” gestartet wurde, öffnet sich die Benutzerschnittstelle.
Sie besteht aus verschiedenen Eingabefeldern:

- Das große Array im unteren Bereich dient zur Eingabe der zu dokumentierenden *CS*-Klassen. Das Array ist unbeschränkt und es ist möglich, über die sichtbare Größe hinaus noch mehr Klassen anzugeben. Dabei müssen in den einzelnen Feldern des Arrays die Verzeichnisse zu den Klassen eingetragen werden. Wird das Programm ohne Eintrag in diesem Array gestartet, werden automatisch alle Klassen des momentan auf dem PC installierten *CS*-Systems bearbeitet.

- Auf der linken Seite sind zwei Kippschalter zu sehen. Sie stellen Booleans dar, mit denen angegeben werden kann, welches Format vom Dokumentationswerkzeug

erzeugt werden soll. Zur Auswahl stehen Java und C++ Code. Im Rahmen der iterativen Codierung, bei der wechselseitig verschiedene CASE-Tools getestet wurden, entstand auch eine Prototyp zur Generierung von C++ Code. Daher wird diese Option offen gehalten. Sie ist aber nur zum Testen gedacht, erzeugt keinen fehlerfreien Code, und ist deshalb nicht mit allen CASE-Tools kompatibel.

- Darunter befindet sich ein Textfeld zur Eingabe ob normaler Java Code erzeugt werden soll, oder ob alle Klassen als Interfaces modelliert werden sollen.

- Es existiert eine Status-Anzeige, um verfolgen zu können, ob das Programm noch am Arbeiten ist. Nach Bearbeitung einer Klasse wechselt die Anzeige von "An" auf "Aus" und wieder zurück. Dies dient zur Orientierung bei der Beurteilung, ob das Programm abgestürzt ist und eventuell manuell beendet werden sollte.

  In zukünftigen Versionen des Dokumentationswerkzeuges kann diese Statusanzeige allerdings durch einen Statusbalken ersetzt werden, der die gerade in Bearbeitung befindliche Komponente anzeigt.

- Weiterhin sind mehrere einzelne Felder zur Angabe von Verzeichnissen zu sehen. Die obersten zwei dienen zur Angabe der Zielverzeichnisse des erzeugten Java und C++ Code. Die unteren drei geben temporäre Dateien an, die das Programm zur Laufzeit benötigt. Existieren diese noch nicht, werden sie vom Programm neu angelegt. Sie sind nur zu Debugging-Zwecken nötig, müssen aber in der momentanen Version angegeben werden.

6.5 Integration und Gesamtsystemtest

In diesem Abschnitt werden die Integration der einzelnen Module sowie der anschließende Gesamtsystemtest durch die Dokumentation des gesamten *CS*-Systems beschrieben.

6.5.1 Systemintegration

Die Systemintegration beschreibt, wie die einzelnen Komponenten zusammengebaut werden.

Datenakquisition: Die Datenakquisition besteht aus den Komponenten Vererbung, Sichtbarkeit, Beschreibung und Ein- und Ausgabeparametern. Die Komponenten werden einzeln entwickelt und getestet. Iterativ werden hier die Ein- und Ausgangsparameter der entwickelten VI aufeinander abgestimmt. Wichtig ist dabei vor allem die Klasse ExtrahierteMetadaten, die während dieses Prozess' konkrete Gestalt annimmt. Da das endgültige

Format nicht schon von vornherein klar ist, wird dieser Datentyp auch später während der Integration der Datenakquisition und des Parsers noch weiter angepasst werden. Mit Format ist die Art der Speicherung der Einzelinformationen gemeint, aus denen diese Klasse zusammengesetzt ist.

Parser: Die Integration des Parsers verläuft linear durch die Teile Vorlage, Vererbung, Methoden, Methodenparameter und Attribute. Hier werden als Eingangsparameter der entwickelten VI im Wesentlichen die erzeugte Datei und der Datentyp ExtrahierteMetadaten benötigt. Dabei bauen alle Teile auf der durch *parseTemplate* erzeugten Vorlage auf. Die Vererbung, Methoden und Attribute können nach Codierung und Einzeltest einfach zusammengebaut werden, da hier keine weitere Abstimmung von Parametern erforderlich ist. Das VI *parseMethodsIO* wird dabei mit *parseMethods* integriert, wobei diese Integration durch das Fehlen von Abhängigkeiten parallel zu den restlichen Tätigkeiten stattfinden kann.

Am Ende werden Parser und Datenakquisition zusammengebaut. Dabei entsteht das abschließende Format für den Datentyp ExtrahierteMetadaten. Da ExtrahierteMetadaten die wesentliche Verbindung der Teile Parser und Datenakquisition darstellt, sind hier noch Abstimmungen vorzunehmen.

LabVIEW Programm und CASE-Tool: Die Integration des Parsers und des CASE-Tools bezieht sich vor allem auf die Korrektheit des erzeugten Codes, und dem damit verbundenen möglichst fehlerfreien Import in das CASE-Tool. Hier wird die Codeerzeugung iterativ soweit verbessert bis ein korrekter Import durch das vorgeschlagene CASE-Tool möglich ist.

6.5.2 Systemgesamttest

Nach der Systemintegration kann das gesamte System durch Anwendung auf das komplette *CS*-System getestet werden. Hierbei wird für jede Klasse des *CS*-Systems eine Javadatei erzeugt und anschließend vom CASE-Tool importiert und dargestellt. Der Systemgesamttest wird jeweils für die Optionen “Interfaceerzeugung” und “Klassenerzeugung” durchgeführt.
Die Ergebnisse des Systemgesamttests bestätigen dabei die in den Anforderungen spezifizierten Ziele. Der Systemgesamttest dauert für Klassen und Interfaces jeweils etwa 15 Minuten. Dabei benötigt das LabVIEW Programm ca. 10 Minuten um alle Klassen zu analysieren und zu erzeugen. Weitere 5 Minuten werden für die Darstellung im CASE-Tool benötigt. Dabei entstehen die Diagramme 48, 49, 50 und 51, die die Anforderungen

bestätigen.

7 Schlussbetrachtung

Im letzten Kapitel wird eine Schlussbetrachtung durchgeführt, in der die Ergebnisse der Arbeit noch einmal kurz zusammengefasst werden. Zusätzlich wird ein Ausblick auf zukünftige Entwicklungen und Erweiterungen zum Dokumentationswerkzeug gegeben.

7.1 Ergebnisse

Die Bachelor-Thesis zeigt die Machbarkeit der Erzeugung von UML-Klassendiagrammen für das objektorientierte Rahmenwerk CS der graphischen Programmiersprache LabVIEW mit Hilfe eines CASE-Tools durch die Verwendung von Austauschformaten.

Durch die Implementierung eines in LabVIEW geschriebenen Prototyps ist es möglich, die zur Erzeugung eines Klassendiagramms nötigen Informationen aus den VI's der *CS*-Klassen zu extrahieren und daraus Textdateien zu erzeugen, die als Java Quellcode verwendet werden. Diese können dann von einem CASE-Tool importiert und als Klassendiagramm dargestellt werden.
Die Bachelor-Thesis wird dabei ordnungsgemäß durch Erreichen der Anforderungen an die Arbeit und mit Unterstützung der Betreuer an der GSI und der TU-Darmstadt beendet.
Kritische Punkte sind die Verwendung einer anderen Programmiersprache als Austauschformat, sowie die Unterstützung von LabVIEW für die dafür notwendigen Tätigkeiten.
Java eignet sich als Austauschformat nur bedingt. Da es keine Mehrfachvererbung unterstützt, kommt es zwangsläufig zu verfälschten Darstellungen. Diese liegen zwar im Rahmen der Anforderungen an die Arbeit, stellen jedoch ein generelles Problem dar. C++ als Austauschformat lässt zwar Mehrfachvererbung zu, ist jedoch weniger Kompatibel bei der Verwendung eines CASE-Tools zur Darstellung des Klassendiagramms. Geeignet erscheint auch das Austauschformat XMI, da es für den Austausch von Modellinformationen gedacht ist. Dieser gute Ansatz scheitert jedoch an der Kompatibilität des Formats zu CASE-Tools und sogar zu sich selbst, was sich in der Unverträglichkeit verschiedener XMI-Versionen niederschlägt. Vollständige Lösungen für diese Probleme existieren im Moment nicht, weshalb die Bachelor-Thesis sowohl die Machbarkeit als auch die Probleme damit aufzeigt.
Kritisch ist auch die Unterstützung von LabVIEW für die dafür notwendigen Tätigkeiten zu sehen. Das in LabVIEW geschriebene Dokumentationswerkzeug operiert sehr oft auf Strings, da dies oft die einzige Möglichkeit ist, Informationen von LabVIEW zu erhalten. Kommt eine neue Version von LabVIEW auf den Markt, kann es durch eine Veränderung der Formate dieser Strings zu Inkompatibilitäten und daraus entstehenden Wartungs-

arbeiten am Werkzeug kommen. Solche strukturellen Informationen zu Elementen der Programmiersprache sollten generisch durch eine einheitliche Schnittstelle abrufbar sein. Weiterhin wird die Roundtrip-Engineering Fähigkeit von LabVIEW selbst eingeschränkt, da es keine Möglichkeit gibt, Elemente der Programmiersprache durch eine entsprechende Schnittstelle zu erzeugen. Da der Quellcode nur in Binärform und nicht textbasiert vorliegt, kann auch hier kein Quellcode durch einen Parser erzeugt werden. Es ist lediglich möglich, bereits bestehende Elemente in ein neues LabVIEW VI programmatisch zu kopieren. Darauf baut der Roundtrip-Mechanismus des Endevo UML Modellers auf. Diese Möglichkeit konnte allerdings aufgrund des damit einhergehenden enormen Arbeitsaufwandes in der Bachelor-Thesis nicht umgesetzt werden. Wünschenswert wäre auch hier eine Schnittstelle zur generischen Erzeugung von LabVIEW-Sprachelementen.

7.2 Ausblick

In diesem Abschnitt wird ein Ausblick auf die zukünftige Anwendung des Dokumentationswerkzeuges gegeben. Außerdem werden mögliche zukünftige Entwicklungen erläutert, die das Dokumentationswerkzeug zur Produktreife führen können.

7.2.1 Anwendung

Das Dokumentationswerkzeug wird in Zukunft vor allem von cs -Entwicklern der Abteilung Experiment-Elektronik für Kontrollsysteme benutzt. Im Zuge der objektorientierten Modellierung von Systemen kommt es außerdem auch als Hilfe beim Design neuer Systeme zum Einsatz. Das bedeutet der *CS*-Entwickler kann sich mit Hilfe des Werkzeuges einen Überblick über bestehende Systeme verschaffen und diese sofort auf Entwurfsebene im selben CASE-Tool erweitern oder neue entwerfen. Nach der Planung im Entwurf wird dieser dann in LabVIEW implementiert. Mit steigender Verbreitung des *CS*-Frameworks wird das Dokumentationswerkzeug auch an anderen Forschungseinrichtungen weltweit zum Einsatz kommen.

7.2.2 Zukünftige Entwicklungen

Um das Werkzeug zur Produktreife zu führen stehen noch einige Entwicklungen aus:

- Error Cluster wird als eigener Datentyp erkannt und behandelt. Dabei werden die Eigenschaften des Error Clusters nicht mehr als Parameter der Methoden aufgeführt in denen Error Cluster als Ein- oder Ausgangsparameter vorkommen, da der Error Cluster ein Standarddatentyp von LabVIEW ist, dessen Inhalt und Funktion weitge-

hend bekannt sind. Angelehnt an die Fehlerbehandlung in Java ist der Error Cluster als Unterklasse der Klasse Exception geplant.

- Optionen zur Auflösung von Clustern als Ein- und Ausgabeparameter werden realisiert
- verbesserte Selektion und Modellierung spezieller Methoden, z.B. Konstruktor, Destruktor, “i attribut.vi” etc.
- generell verbesserte Fehlerbehandlung

Weitere zukünftige Entwicklungen können die vollständige Implementierung eines Parsers für C++ oder XMI sein. Der Entwicklung eines Parsers zur Erzeugung von funktionsfähigem XMI Code werden allerdings weitere Standardisierungsbestrebungen bezüglich des Austauschs von Modell- und Diagramminformationen dieses Formats vorausgehen müssen.
Als Wünschenswerte zukünftige Erweiterung ist noch die Dokumentation des Ereignisinterfaces zu nennen. Dabei muss ein Konzept zur Modellierung erstellt werden. Angelehnt an den Eventmechanismus von Java können hier Listenerklassen und Eventklassen erzeugt werden. Jede *CS*-Klasse, die Unterklasse von BaseProcess ist, kann dann ihre eigene Listenerklasse implementieren. Auf der anderen Seite gibt es passend dazu eine Oberklasse für Events die in BaseProcess definiert sind. Die Ereignissklasse der konkreten *CS*-Klasse erbt dann von dieser und definiert ihre eigenen Ereignisse, die von der jeweiligen Listenerklasse abgefangen werden. So ist eine Abbildung der *CS*-Ereignisse auf die Methoden ihrer Ereignisklasse und der Descriptoren der Ereignisse auf die Parameter der Methode möglich.

A Glossar

CASE-Tool

Ein CASE-Tool (Computer Aided Software Engineering) ist ein Werkzeug zur Unterstützung des Softwareentwicklungsprozess. Es zeichnet sich durch automatisierte Unterstützung in den Bereichen des Entwurfs und der Programmierung von Software aus.

CERN

Europäische Organisation für Kernforschung, Conseil Européen pour la Recherche Nucléaire

Bei der Europäischen Organisation für Kernforschung handelt es sich um die weltweit grösste Grossforschungseinrichtung auf dem Gebiet der Teilchenphysik. Die in der Schweiz gelegene Anlage besteht aus 2 Speicherringen und mehreren Teilchenbeschleunigern, mit deren Hilfe Elementarteilchen wie Elektronen und Protonen beschleunigt und zur Kollision gebracht werden, um die Zusammensetzung der Materie zu erforschen. Gegründet wurde das Institut am 29. September 1954 in Meyrin in der Nähe von Genf, indem elf europäische Länder ein entsprechendes Übereinkommen unterzeichneten.

Command Pattern

In der objektorientierten Programmierung ist das Command Pattern ein Designattern, bei dem Objekte benutzt werden, um Aktionen zu repräsentieren. Ein Command-Objekt kapselt eine Aktion und deren Parameter.

Compilersprache

Programmiersprache, deren Programme vor der Ausführung vollständig in Maschinencode übersetzt werden. Im Gegensatz dazu stehen Interpretersprachen, deren Programme Befehl für Befehl übersetzt und gleich ausgeführt werden (siehe http://de.encarta.msn.com/encyclopedia_761571301/Compilersprache.html).

DTD

Eine Dokumenttypdefinition ist ein Satz an Regeln, der benutzt wird, um Dokumente eines bestimmten Typs zu repräsentieren. Ein Dokumenttyp ist dabei eine Klasse ähnlicher Dokumente, wie beispielsweise Telefonbücher oder Inventurdatensätze. Die Dokumenttypdefinition besteht dabei aus Elementtypen, Attributen von Elementen, Entitäten und Notationen. Konkret heißt das, dass in einer DTD die Reihenfolge, die Verschachtelung der Elemente und die Art des Inhalts von Attributen festgelegt wird - kurz gesagt: die Struktur des Dokuments.

G

G ist die LabVIEW eigene Programmiersprache. LabVIEW ist die Entwicklungsumgebung, während G der unter LabVIEW erzeugte Code ist. Die graphische Programmiersprache G kann nicht zu einer textbasierten Sprache re-interpretiert werden. Es ist nicht möglich G aufzudecken und den darunter liegenden Textcode zu sehen, da keiner existiert. G bleibt also der Quellcode mit dem Entwickler arbeiten.

GOOP

Graphical Object Oriented Programming

Das Graphical Object-Oriented Programming Framework wird von Endevo und National Instruments entwickelt. Es handelt sich dabei um eine Umgebung zur Objektorientierten Programmierung in LabVIEW.

Interpreter

In der Informatik versteht man unter einem Interpreter ein Computerprogramm, das Befehle ausführt, die in einer Programmiersprache geschrieben sind. Die Terminologie Interpreter trifft dabei sowohl auf Programme zu, die Quellcode ausführen, der schon in ein Zwischenprodukt übersetzt wurde, als auch auf Programme, die Quellcode übersetzen und ausführen (siehe http://en.wikipedia.org/wiki/Interpreter_%28computing%29).

MATLAB

MATLAB ist eine kommerzielle, plattformunabhängige Software des Unternehmens The MathWorks. zur Lösung mathematischer Probleme und zur grafischen Darstellung der Ergebnisse (siehe http://de.wikipedia.org/wiki/MATLAB).

monolithisch

Allgemein bezeichnet man Objekte, die aus einem Stück bestehen, als monolithisch.

Notification

Die Melder-Funktionen werden benötigt, wenn die Ausführung eines Blockdiagramms so lange ausgesetzt werden soll, bis Daten aus einem anderen Teil des Blockdiagramms oder einem anderen VI in derselben Applikationsinstanz zur Verfügung stehen. Für die Kommunikation mit VIs auf anderen Computern sind Melder nicht geeignet. So kann zum Beispiel damit nicht über ein Netzwerk oder einen VI-Server kommuniziert werden. Anders als bei den Queue-Funktionen werden die gesendeten Meldungen bei Melder-Funktionen nicht gepuffert. Wenn beim Absenden einer Meldung keine Knoten für deren Empfang bereitstehen, geht sie beim Senden der nächsten Meldung verloren. Melder sind mit Queues vergleichbar, die in der Größe begrenzt sind und deren Inhalt einzeln abgearbeitet wird.

Observer Pattern

Das Observer Pattern ist ein in der Programmierung verwendetes Designpattern um den Zustand von Objekten in einem Programm zu überwachen.

Okkurenz

Mit den Occurrence-Funktionen werden separate synchrone Aktivitäten gesteuert.

Diese Funktionen werden insbesondere dann verwendet, wenn die Ausführung eines VIs oder Blockdiagrammteils zurückgestellt werden soll, bis ein anderes VI oder ein Teil eines Blockdiagramms seine Aufgabe beendet hat. Normalerweise ist dazu eine laufende Statusabfrage erforderlich.

Queue

Mit den Queue-Funktionen lässt sich eine Queue für den Datenaustausch zwischen verschiedenen Stellen im Blockdiagramm oder zwischen den Blockdiagrammen zweier VIs erstellen. Anders als die Melder-Funktionen werden die Daten bei Queue-Funktionen gepuffert.

Rapid Prototyping

Das Prototyping ist eine Methode der Softwareentwicklung, die schnell zu ersten Ergebnissen führt und frühzeitiges Feedback bezüglich der Eignung eines Lösungsansatzes ermöglicht.

Remote Procedure Calls

Mit Hilfe von RPC können über ein Netzwerk Funktionsaufrufe auf entfernten Rechnern durchgeführt werden (siehe http://de.wikipedia.org/wiki/Remote_Procedure_Call).

Rendezvous

Rendezvous dienen zur Synchronisierung von getrennten, parallel ablaufenden Tasks an bestimmten Stellen des Datenflusses. Jeder am Rendezvous teilnehmende Task wartet, bis alle angegebenen Tasks bereitstehen; erst dann wird deren Ausführung fortgesetzt.

SCADA

Supervisory Control and Data Acquisition

"Konzept zur Überwachung und Steuerung technischer Prozesse. Dies beinhaltet vor allem das Darstellen von Messwerten und Betriebszuständen für den Anwender, automatische Regelungsmechanismen und die Aufnahme und Archivierung von Messdaten." [Kugl06]

Semaphore

"In der Informatik wird mit diesem Begriff eine Datenstruktur zur Prozesssynchronisation bezeichnet. Greifen nebenläufige Programmteile auf gemeinsame Daten zu, wird ein Ausschlussmechanismus benötigt, der den Zugriff regelt. Solange Daten von einem Thread beschrieben werden, muss der Zugriff für alle anderen Threads gesperrt sein, ansonsten kommt es zu asynchronen Zugriff, die zu einer Inkonsistenz der Daten führen." [Kugl06]

Simulink

Simulink ist eine Software des Herstellers The MathWorks zur Modellierung von mathematischen Systemen. Simulink ist ein Zusatzprodukt zu MATLAB und benötigt dieses zum Ausführen (siehe http://de.wikipedia.org/wiki/Simulink).

Thread

In der Informatik steht "Thread" als Kurzform für "Thread of Execution", also für den "Faden der Ausführung". Durch Threads wird einem Programm die Möglichkeit gegeben, sich selbst in mehrere gleichzeitig laufende Aufgaben aufzuteilen (siehe http://en.wikipedia.org/wiki/Thread_(computer_science)).

Threadsicherheit

Threadsicherheit ist ein Programmierkonzept, das im Kontext von Multi-Threaded-Programmen Anwendung findet. Ein Stück Code ist threadsicher, wenn es korrekt während der gleichzeitigen Ausführung mehrerer Threads funktioniert. Es muss speziell der Anforderung genügen, mehreren Threads den Zugriff auf dieselben gemeinsam genutzten Daten zu gewährleisten, sowie der Anforderung, gemeinsam genutzte Daten zu jeder Zeit nur von einem einzigen Thread aus Zugriff zu gewähren (siehe http://en.wikipedia.org/wiki/Thread_safety).

UML

UML (Unified Modeling Language), eine durch die Object Management Group standardisierte, universelle grafische Sprache zur Beschreibung aller Arten objektorientierter Softwaresysteme. Sie zielt auf eine Vereinigung der bekanntesten Beschreibungsnotationen und ist schwerpunktmäßig auf die Produkte der Softwaretechnik gerichtet (siehe http://lexikon.meyers.de/meyers/UML).

VI Template

"Vorlage oder Muster für ein VI. Zur Laufzeit können von einer Vorlage beliebig viele Instanzen in den Speicher geladen werden, auf die mittels eindeutiger Referenzen zugegriffen werden kann." [Kugl06]

Watchdog

"Technischer Begriff für die Komponente eines Systems, die Überwachungsaufgaben übernimmt. Im Fehlerfall wird diese aktiv, um wieder andere Prozesse auszulösen, die den Fehler beheben können." [Kugl06]

B Diagramme

Abbildung 48: Paketdiagramm der *CS*-Klassen

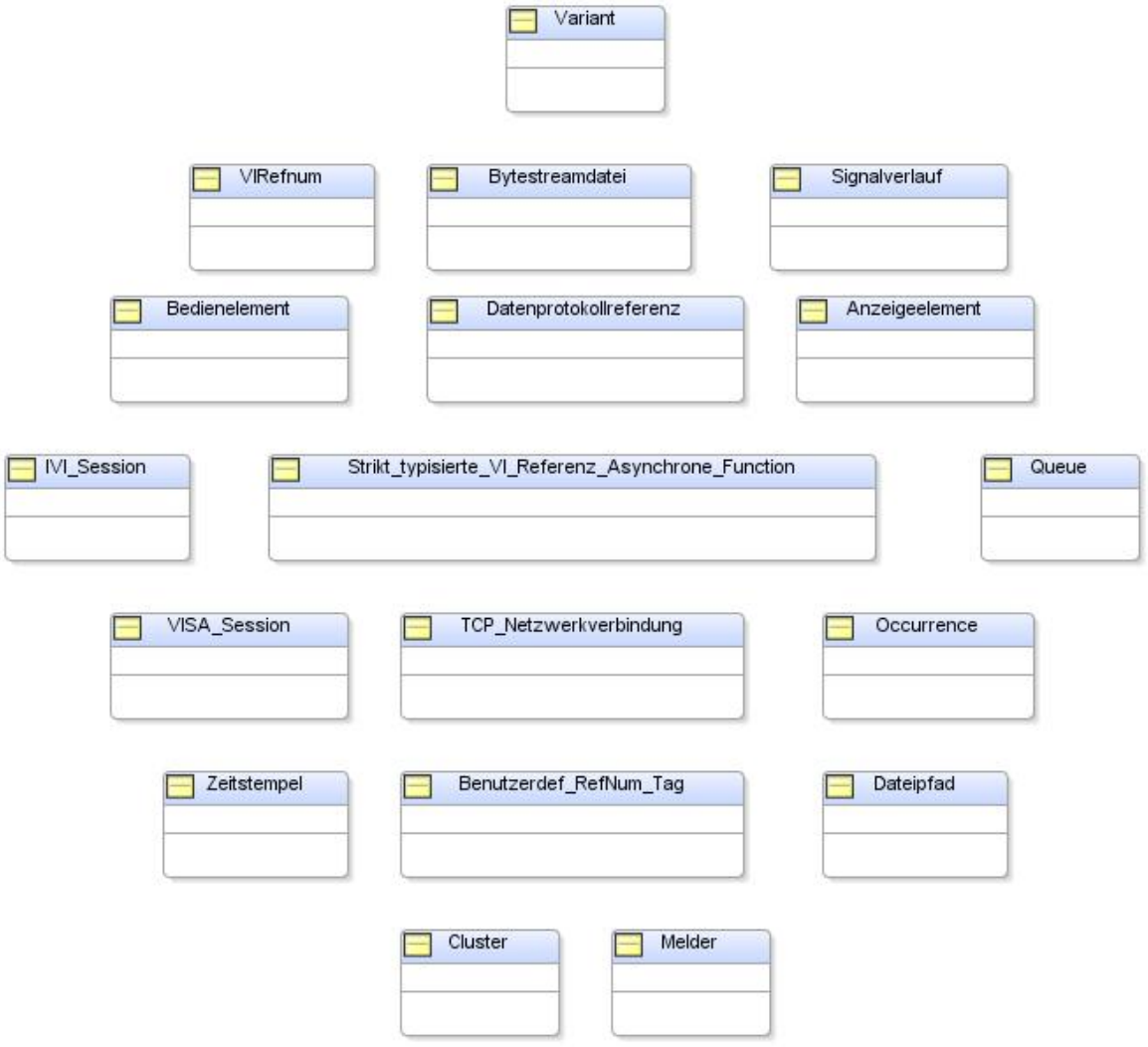


Abbildung 49: Klassendiagramm des javaDatatypeSupport-Pakets

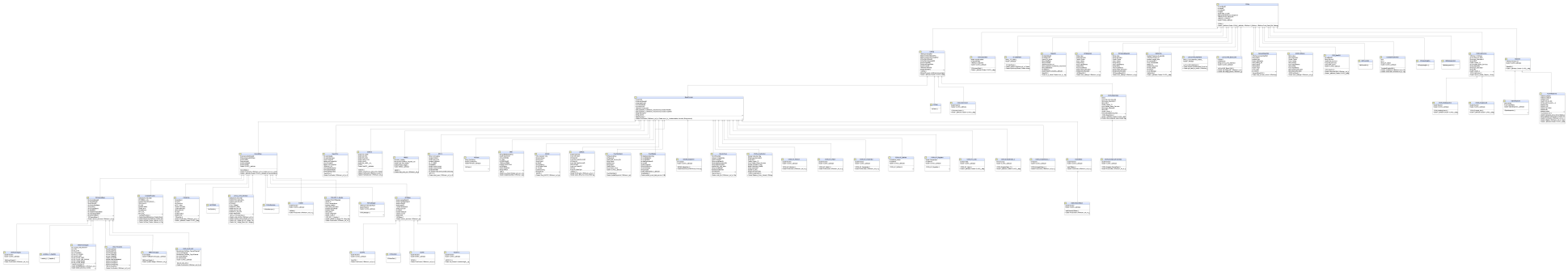


Abbildung 50: Klassendiagramm des *CS*-Systems

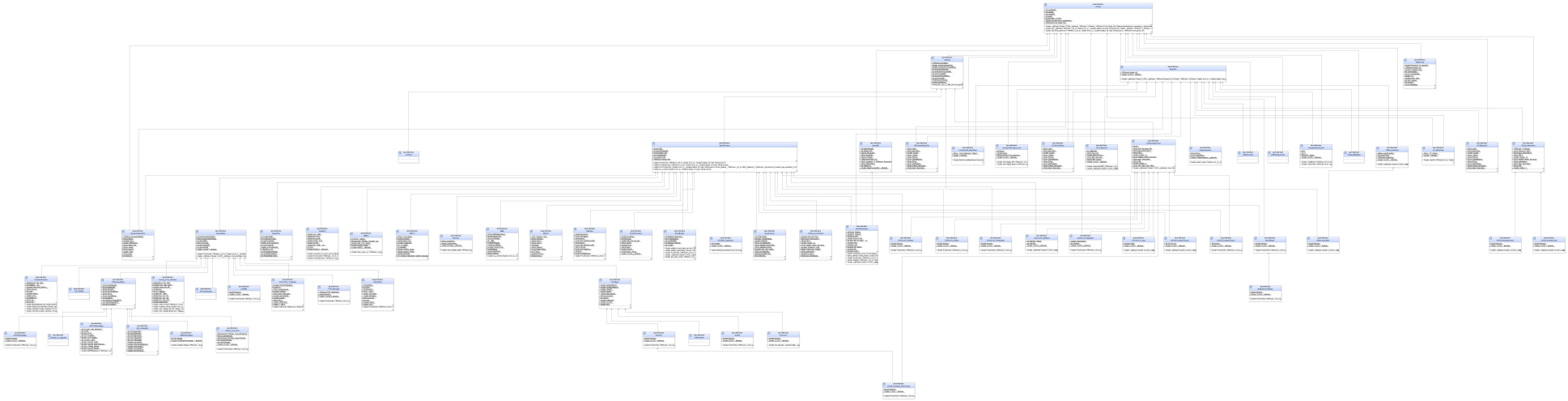


Abbildung 51: “Interfacediagramm” des *CS*-Systems

C Literaturverzeichnis

Literatur

[BBGH05] D. Beck, H. Brand, S. Götte, F. Herfurth, C. Rauth, R. Savreux, S. Schwarz, C. Yazidjian
THE CS FRAMEWORK - A LABVIEW BASED APPROACH TO SCADA SYSTEMS
Geneva, 10 - 14 Oct 2005, PO1.051-6 (2005)
10th ICALEPCS Int. Conf. on Accelerator & Large Expt. Physics Control Systems

[BeBH03] D. Beck, H. Brand, F. Herfurth
CS - A CONTROL SYSTEM FRAMEWORK FOR EXPERIMENTS (NOT ONLY) AT GSI
2003
Proceedings of ICALEPCS2003, Gyeongju, Korea

[Beck05] Dr. Dietrich Beck
Stand und Entwicklung des *CS*-Kontrollsystem- Frameworks
SEI Frühjahrstagung, April 2005
Herausgeber F. Wulf

[Bran02] Dr. Holger Brand
A Framework for Experiment Control Systems
2002, CBM Collaboration-Meeting

[Bran05] Dr. rer. nat. Holger Brand
Einführung in das *CS*-Framework
Version 1.2, November 2005
Brand New Technologies

[Bran07] Dr. Holger Brand
Kontrollsystemaktivitäten an der GSI
SEI Frühjahrstagung März 2007

[DHTT00] Christian Heide Damm, Klaus Marius Hansen, Michael Thomsen, Michael Tyrsted
Tool Integration: Experiences and Issues in Using XMI and Component Technology

2000
Proceedings of TOOLS-Europe2000, Mont-Saint-Michel & Saint-Malo

[Gasp06] Distributed Information Management System, C. Gaspar. 2006, http://dim.web.cern.ch/dim/, Zugriff am 09.2007

[GMPR06] K.-D. Groß, C. Meier, I. Peter, J. Reiß
Ionenstrahlen im Kampf gegen Krebs, Tumortherapie an der GSI
März 2006

[GrLP06] K.-D. Groß, J. Leroudier, I. Peter
Die GSI, Ein Überblick
August 2006

[Harm98] Development of LabVIEW Master Page for Experiment 877: The Effect of Magnetic Field on the Speed of Light by Akilah Jaha Harmon. 1998, http://sist.fnal.gov/archive/1998-topics/Harmon/Html/Paper.html, Zugriff 08.2007

[Holl05] Introduction to Modern Data Acquisition with LabVIEW and MATLAB by Matt Hollingsworth. 2005, http://www.phys.utk.edu/labs/modphys/, Zugriff 09.2007

[Kugl06] Maximilian Kugler
Entwicklung einer Klassenbibliothek zur Erstellung generischer Sequenzen im Rahmen des *CS*-Frameworks.
2006, Gesellschaft für Schwerionenforschung mbH

[LVUG05] LabVIEW User Group Central Europe e.V. (LVUG). 2005, http://www-w2k.gsi.de/lvug/techniken.htm, Zugriff 08.2007

[LVWI07] LabVIEW Wiki. 2007, http://wiki.lavag.org, Zugriff 09.2007

[NIES06] National Instruments
LabVIEW - Erste Schritte mit LabVIEW
2006, National Instruments

[NIGr05] National Instruments
LabVIEW - Grundlagen
2005, National Instruments

[NIHi06] National Instruments
LabVIEW - Hilfe
2006, National Instruments

[Pich06] Herbert Pichlik
G++ - Neue Wege in der Informationstechnologie
2006, LabVIEW User Group Central Europe e.V

[Schw07] Alexander Schwinn
Entwicklung von Objekt- und Petri-Netzen im Rahmen eines Kontrollsystems
2007, Gesellschaft für Schwerionenforschung mbH