h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Fachbereich
Elektrotechnik und Telekommunikation
Schöfferstraße 3
64295 Darmstadt

Diplomarbeit

Entwicklung einer Klassenbibliothek zur Erstellung generischer Sequenzen im Rahmen des *CS* Frameworks

Betreuer: Dr. Holger Brand

Referent Prof. Dr. Antje Wirth

Koreferent: Prof. Dr. Ulrich Schultheiß

Abgabetermin: 31.August 2006

Erstellt von: Maximilian Kugler
Raiffeisenstraße 5b
64319 Pfungstadt
Matrikelnummer 634942
E-Mail: MaximilianKugler@gmx.de

h_da FB E/Telekommunikation

DIPLOMARBEIT

Thema: **Entwicklung einer Klassenbibliothek zur Erstellung generischer Sequenzen im Rahmen des CS Frameworks**

Bearbeiter: **Maximilian Kugler**

Referent: **Prof. Dr. Antje Wirth**

Korreferent: **Prof. Dr. Ulrich Schultheiß**

Abgabe am: **31.August 2006**

h_da FB E/Telekommunikation

Bearbeiter:

Name: Kugler Vorname: Maximilian

Referent/in: Prof. Dr. Antje Wirth Korreferent: Prof. Dr. Ulrich Schultheiß

ERKLÄRUNG

Ich versichere, dass ich die vorliegende Diplomarbeit selbständig erarbeitet habe und dass dabei keine anderen als die angegebenen Quellen und Hilfsmittel verwendet wurden.

Darmstadt, den 30.August 2006 .....
(Unterschrift)

Erklärung zur Verwendung der Diplomarbeit

Ich bin damit einverstanden, dass meine Diplomarbeit vom* 31.August 2006

- im Fachbereich E/T aufbewahrt und nur an Hochschulmitglieder und Hochschulangehörige ausgeliehen wird;
- vollständig oder nur mit Titel und Name des Autors/der Autorin auf elektronischen Datenträgern im Fachbereich E/T für Hochschulmitglieder und Hochschulangehörige zur Einsicht zur Verfügung steht;
- von Hochschulmitgliedern und Hochschulangehörigen nur für die Lehre und wissenschaftliche/berufliche Zwecke auf elektronischen Datenträgern kopiert werden kann;
- in weiteren Exemplaren bei dem/der Referenten/in auch nach Ablauf der Aufbewahrungsfrist von zwei Jahren verbleibt;
  (Diese Aufbewahrungsfrist beginnt mit Ablauf des Kalenderjahres, in dem mir das Ergebnis der Diplomprüfung mitgeteilt worden ist.)
- Ich bitte, mir die für die oben genannten Zwecke nicht benötigten Exemplare der Arbeit nach Ablauf der Aufbewahrungsfrist zuzusenden. Einen ausreichend frankierten und adressierten Umschlag werde ich rechtzeitig einreichen;
- Ich wurde darüber belehrt, dass die nicht abgeholten/zurückgesandten Exemplare nach Ablauf der Aufbewahrungsfrist dem Hessischen Staatsarchiv angeboten bzw. vernichtet werden;
- Diese Erklärung gilt auch für die mit der Diplomarbeit abgegebenen Fotos oder Dias.

Darmstadt, den 30.August 2006 Unterschrift: .....

* Abgabedatum

h_da FB E/Telekommunikation

BEARBEITER:

Name: Kugler Vorname: Maximilian

Geburtstag: 10.11.1981 Matrikel-Nr.: 634942

Adresse: Raiffeisenstrasse 5b 64319 Pfungstadt

Fachgebiet: Verarbeitung

Referent: Prof. Dr. Antje Wirth

Korreferent: Prof. Dr. Ulrich Schultheiß

Thema: Entwicklung einer Klassenbibliothek zur Erstellung generischer Sequenzen im Rahmen des CS Frameworks

Kurzreferat: (max 10 Zeilen)

Die Diplomarbeit befasst sich mit der Erweiterung des an der GSI entstandenen CS Framework um einen generischen Sequenzer. Im Falle des CS Framework handelt es sich um eine Software, die basierend auf LabVIEW eine objektorientierte, ereignisgesteuerte Entwicklungsumgebung für Kontrollsysteme der Experimente an der GSI und anderen Forschungsinstituten realisiert.
Ziel der Arbeit ist es das bestehenden CS Framework mit einer Klassenbibliothek zu erweitern, die einem Entwickler generische Software-Module zur Automatisierung und Ablaufsteuerung zur Verfügung stellt.
Mögliche Anwendungsgebiete sind die Inbetriebnahme und Konfiguartion der Experimentaufbauten oder die Entwicklung von Diagnose- und Testständen.

Die Arbeit wurde mit/ohne Kooperationspartner (extern) durchgeführt:

Institution: Gesellschaft für Schwerionenforschung

Anschrift: Gesellschaft für Schwerionenforschung mbH
Planckstr. 1
64291 Darmstadt

Die Arbeit ist gesperrt: ☐ ja ☒ nein

Unterschrift des Referenten:

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Zusammenfassung

Die Diplomarbeit befasst sich mit der Erweiterung des *CS* Frameworks um einen generischen Sequenzer. Im Falle des *CS* Frameworks handelt es sich um eine Software, die basierend auf LabVIEW eine objektorientierte, ereignisgesteuerte Entwicklungsumgebung für Kontrollsysteme der Experimente an der GSI und anderen Forschungsinstituten zur Verfügung stellt. Das Framework wurde von der Kontrollsystemgruppe der Abteilung Experiment Elektronik an der GSI zur Schaffung einer einheitlichen Software-Architektur entwickelt, die die Wiederverwendbarkeit der entworfenen Software verbessern soll. Hierzu beinhaltet das *CS* Framework Entwurfsvorlagen für die verschiedenen Aufgaben eines Kontrollsystems. Dazu zählen vor allem die Steuerung externer Geräte sowie das Überwachen und Archivieren gerätespezifischer Parameter. Neben der Ansteuerung der Hardware besteht von Seiten der Experimente oft die Anforderung nach einer Automatisierung bestimmter Abläufe, die der Inbetriebnahme oder Konfiguration des Experimentaufbaus dienen. Dies beinhaltet vor allem die Sequenzierung einzelner Zugriffe auf die Hardware. Zur Beeinflussung der Reihenfolge innerhalb einer Sequenz muss die Verwendung weiterer Kontrollstrukturen wie Wiederholungs- und Verzweigungsanweisungen möglich sein.
Aufgrund der ständigen Änderungen, denen ein Experimentaufbau unterliegt, muss das Erstellen und Editieren der Sequenzen durch den Benutzer zur Laufzeit durchgeführt werden können. In diesem Zusammenhang spielt auch die Kompatibilität zwischen beliebigen Geräte, die bereits im *CS* Framework zur Verfügung stehen oder zukünftig implementiert werden, und dem Sequenzer eine wichtige Rolle.
Ein generischer Sequenzer soll die Basisfunktionalität des *CS* Frameworks erweitern, indem es einem Entwickler Entwurfsmuster an die Hand gibt, auf denen er aufbauend seine Sequenzen definieren und testen kann. Dabei soll der Arbeitsaufwand von der Implementierung im Quellcode auf die Konfiguration bereits vorhandener generischer Software-Module verlagert werden, die dem Benutzer anschließend zur flexiblen Verwendung zur Verfügung gestellt werden können.

Kapitel 1

Einleitung

1.1 Die GSI[1]

Das von Bund und dem Land Hessen getragene Forschungsinstitut betreibt seit 1969 in Darmstadt eine Beschleunigeranlage für Schwerionen, die einerseits der Grundlagenforschung im Bereich der Kern- und Atomphysik dient, andererseits Anwendung auf den Gebieten der Materialforschung und Strahlenmedizin findet. Besondere Aufmerksamkeit erregte die Großforschungseinrichtung mit der Entdeckung von sechs neuen chemischen Elementen[2] und der Entwicklung einer weltweit einzigartigen Tumortherapie, mit deren Hilfe seit 1997 über 300 Patienten[3] erfolgreich behandelt werden konnten.
Fortschritte im Bereich der Wissenschaft können nur erzielt werden, wenn die Experimentiergeräte ebenfalls Gegenstand der Forschungsaktivität sind. Daher spielt sowohl die Weiterentwicklung der Beschleunigeranlage als auch der Experimente selbst eine zentrale Rolle. Hierfür unterhält die GSI den Bereich wissenschaftlich technische Infrastruktur, deren Abteilungen sich mit der Betreuung des Beschleunigers und der Experimente beschäftigen, um optimale Voraussetzungen für die Experimentatoren zu schaffen. Mehr als 1000 Wissenschaftler aus über 30 Ländern reisen pro Jahr an die GSI, um die Beschleunigeranlage für ihre Forschungen zu nutzen.
Diese Arbeit entstand im Rahmen der Kontrollsystemgruppe der Abteilung Experiment Elektronik an der GSI, deren Aufgabe die Beratung und Unterstützung der Experimentatoren bei der Entwicklung experimentspezifischer Kontrollsysteme ist.

---

[1] Gesellschaft für Schwerionenforschung; Mehr Informationen siehe http://www.gsi.de.

[2] Bei den sechs neuen Elementen handelt es sich namentlich um Darmstadtium, Hassium, Meitnerium, Roentgenium, Bohrium, Ununbium mit den Ordnungszahlen 107 bis 112

[3] Stand 2005

1.2 Themenstellung

Die Zielsetzung und die daraus resultierenden technischen Anforderungen der Experimente im Bereich der Kernphysik sind einzigartig und in keinem anderen Forschungs- oder Industriezweig anzutreffen. Daher besteht der Aufbau eines Experiments an einer Forschungseinrichtung wie der GSI in der Regel aus Hardware, die direkt vor Ort speziell für das jeweilige Experiment entwickelt und gefertigt wird. Dabei handelt es sich im Falle der GSI vor allem um verschiedene Ionisationskammern[4], Szintilatoren[5] und Halbleiterdetektoren. Für den Betrieb der Detektoren werden darüber hinaus kommerzielle Geräte, wie z.B. Spannungsversorger, Gasventile und Sensoren benötigt.
Angesichts der Einzigartigkeit eines jeden Experimentaufbaus existieren keine kommerziellen Gesamtlösungen, um derartige Systeme zu kontrollieren. Daher liegt die Aufgabe, Kontrollsysteme zu entwickeln, die die Überwachung und Steuerung eines Experiments übernehmen, ebenfalls in den Händen der Forschungseinrichtungen.
Bis auf sicherheitskritische Aspekte, die den Personenschutz oder den Schutz sensibler bzw. teurer Hardware betreffen, besteht ein Kontrollsystem in der Regel aus Software, die über diverse Schnittstellen mit den verschiedenen Hardwarekomponenten eines Experimentaufbaus kommuniziert, um deren Zusammenspiel zu überwachen und regeln.
Zu diesem Zweck entstanden bereits an mehreren Forschungsinstituten weltweit Software-Systeme, wie zum Beispiel EPICS[6], deren Ziel es ist, eine einheitliche Entwicklungsumgebung für Kontrollsysteme zu schaffen. Diese Systeme werden als Framework[7] bezeichnet, da sie Entwurfsvorlagen beinhalten, mit deren Hilfe die Implementierung eines spezifischen Kontrollsystems erleichtert und die benötigte Entwicklungszeit verkürzt werden soll.
Das *CS* Framework[8] stellt ebenfalls ein solches System da. Basierend auf LabVIEW[9] bietet es eine objektorientierte und ereignisgesteuerte Entwicklungsumgebung für verteilte Kontrollsysteme. Entwickelt wurde bzw. wird das *CS* Framework von Dietrich Beck und Holger Brand im Rahmen der Kontrollsystemgruppe als Quellen offene Software an der GSI. Es ist bereits seit meh-

[4] Strahlungsdetektor, der hauptsächlich zur Messung der Strahlungsenergie von Gamma-Strahlen eingesetzt wird; siehe http://de.wikipedia.org/wiki/Ionisationskammer.
[5] Material, welches beim Durchgang von geladenen Teilchen angeregt wird und die Anregungsenergie in Form von Licht wieder abgibt; siehe http://de.wikipedia.org/wiki/Szintillator.
[6] Experimental Physics and Industrial Control Systems; siehe Glossar Seite 87
[7] wörtliche Übersetzung: Rahmenwerk, Gerüst; siehe Glossar Seite 87
[8] Control System Framework; siehe Kapitel 2
[9] Laboratory Virtual Instrument Engineering Workbench; siehe Glossar Seite 87

reren Jahren bei Experimenten wie PHELIX[10] und SHIPTRAP[11] an der GSI oder ISOLTRAP am CERN[12] im Einsatz.

Die Erweiterung des bestehenden *CS* Frameworks durch eine Klassenbibliothek zur Erstellung generischer Sequenzen ist Inhalt dieser Arbeit. Vor allem die Wiederverwendbarkeit steht dabei im Vordergrund. Ziel ist es, eine möglichst allgemeine Klassenbibliothek zu entwerfen, welche Lösungen für unterschiedliche Anwendungen bietet, selbst aber noch keine spezifische Aufgabe erfüllt. Viel mehr soll sie dem Entwickler Entwurfsvorlagen zur Verfügung stellen, die als Ausgangspunkt für dessen Anwendung dient. Deshalb beschreibt der Begriff Framework die Zielsetzung der Software treffender als der Begriff Klassenbibliothek. Die beiden Begriffe werden im Bezug auf die im Rahmen dieser Arbeit entwickelte Software im folgenden Text synonym verwendet.
Mögliche Anwendungsgebiete sind die Inbetriebnahme und Konfiguration experimentspezifischer Kontrollsysteme oder die Entwicklung von Diagnose- und Teststänten. Eine Gemeinsamkeit dieser Anwendungen ist, dass oftmals erst zur Laufzeit der Anwendung feststeht, welche Aktionen in welcher Reihenfolge erfolgen sollen und welche Geräte diese Aktionen ausführen. Im Falle der Testumgebung kann es vorkommen, dass der Benutzer nach Belieben die Testumgebung verändert, indem er Hardwarekomponenten austauscht, hinzufügt oder neu miteinander kombiniert. Möglicherweise will er auch die Abfolge bestimmter Testmuster neu definieren, oder diese austauschen. Ähnliche Anforderungen existieren auch von Seiten der Kontrollsysteme. Abhängig von der Durchführung eines Experiments soll auch das Kontrollsystem dem Benutzer verschiedene Konfigurationen zur Verfügung stellen. Die Klassenbibliothek muss daher generische Module bereitstellen, die der Entwickler verwenden kann, um eine flexible Konfiguration seiner Anwendungen gewährleisten zu können, ohne jedes Mal den Quellcode ändern zu müssen.

[10] Petawatt High Energy Laser for Heavy Ion Experiment; siehe Glossar Seite 88
[11] Das Experiment an der GSI führt Massemessungen an schweren Ionen durch.
[12] Conseil Européen pour la Recherche Nucléaire; siehe Glossar Seite 86

1.3 Überblick über die Arbeit

Software-Entwicklung durchläuft von der Idee bis zum fertigen Produkt mindestens die folgenden drei Entwicklungsphasen[13]:

- Definitionsphase
- Entwurfsphase
- Implementierungs- bzw. Programmierphase

Die Gliederung dieser Arbeit orientiert sich an dem eben vorgestellten Entwicklungsverlauf:

Kapitel 2: Entwicklungsumgebung - *CS* Framework Eine Übersicht über das *CS* Framework soll erläutern, mit welcher Zielsetzung das Framework entwickelt wurde und welche Anforderungen zum Entwurf maßgeblich beigetragen haben. Insbesondere die Gründe für die Entscheidung, LabVIEW als Basis für das *CS* Framework zu verwenden, werden genannt und erklärt.
Eine anschließende Einführung in das *CS* Framework zeigt, welche Werkzeuge einem Entwickler zur Verfügung gestellt werden und vermittelt das Hintergrundwissen, welches für das Verständnis der folgenden Kapitel benötigt wird. Alle Angaben hinsichtlich des *CS* Frameworks beziehen sich auf dessen Version 2.10.

Kapitel 3: Definition Dieser Abschnitt beschäftigt sich mit der Defintionsphase für die zu entwickelnde Klassenbibliothek. Anhand von Anwendungsfalldiagrammen sollen die Anforderungen beschrieben und analysiert werden.

Kapitel 4: Entwurf Anhand der zuvor aufgeführten Anforderungen, beschreibt dieses Kapitel den Entwurf und die Implementierung der Klassenbibliothek. Eine Beschreibung aller enthaltenen Klassen sowie deren Zuständigkeiten, Funktionalitäten und Beziehungen wird mittels der UML[14]-Notation gegeben.

[13] siehe [Balzert2005] Seite 169
[14] Unified Modelling Language; siehe Glossar Seite 88

Kapitel 5: Anwendung: BELAB-Teststand Eine erste Anwendung findet die Klassenbibliothek im Rahmen eines Teststandes für Elektronikentwicklungen des Beschleuniger-Elektronik-Labors (BELAB) der GSI. Der Entwicklungsprozess wurde in enger Zusammenarbeit mit dem BELAB durchgeführt. Dieses Kapitel beschreibt detailliert deren Anforderungen an das Sequenzer-Framework und den Entwurf der BELAB-Klassen, die die softwareseitige Repräsentation der BELAB-Hardware darstellen. Diesbezüglich wird eine kurze Einführung in das Aufgabengebiet des BELABs gegeben.

Kapitel 6: Schlussbetrachtung Abschließend wird im letzten Kapitel eine Bewertung der erstellten Klassenbibliothek gegeben. Vor- und Nachteile werden gegenübergestellt und diskutiert. Mögliche Änderungen und Erweiterungen am Entwurf und der Implementierung sollen aufgezeigt werden.

1.4 Typographische Konventionen

Im Rahmen dieser Arbeit wurden verschiedene Schriftarten und -formate verwendet, um die Lesbarkeit des Textes zu erhöhen und unterschiedliche Informationen besser auseinander halten zu können. Nachfolgend werden diese erklärt:

- Auf Sätze oder Abschnitte, die sinngemäß aus fremden Texten entnommen wurden, folgt die Angabe der Quelle, welche durch eckige Klammern hervorgehoben ist, in der Fussnote. Im Literaturverzeichnis ab Seite 94 findet man eine detaillierte Beschreibung aller verwendeten Quellen.
- Begriffe, die einer Erklärung bedürfen, werden in der dazugehörigen Fußnote entweder erläutert oder es wird auf den Glossar ab Seite 86 verwiesen.
- Begriffe und Namen, die im Zusammenhang mit LabVIEW, dem *CS* Framework oder der Sequenzer-Klassenbibliothek eine spezielle Bedeutung haben, vor allem Methoden-, Objekt- und Klassennamen, werden innerhalb des Textes in `Schreibschrift` dargestellt.

Da das *CS* Framework objektorientierte Ansätze beinhaltet, werden Definition und Entwurf der Klassenbibliothek mit Hilfe der UML dargestellt. Tabelle 1.1 erläutert die im Rahmen dieser Arbeit verwendeten UML-Symbole und deren spezifische Bedeutung im Zusammenhang mit dem *CS* Framework .

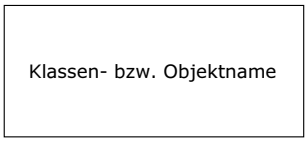
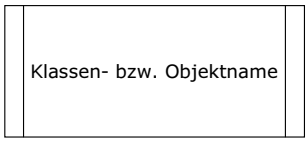
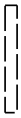
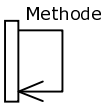
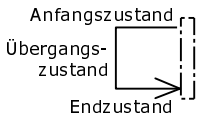
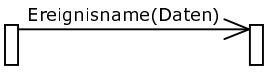
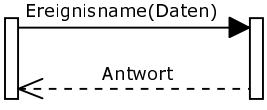
Symbol	Bedeutung
	Passive Klassen besitzen keinen `thread`, der während der Existenz eines Objektes ausgeführt wird. Sie verfügen lediglich über Attribute und stellen Methoden zur Verfügung.
	Aktive Klassen verfügen mindestens über einen `thread`, der während der Existenz eines Objektes ausgeführt wird. Alle Klassen, die von einer aktiven Klasse erben sind automatisch aktiv. Aktive Klassen werden nur also solche ausdrücklich gekennzeichnet, falls in dem entsprechenden Diagramm auch passive Klassen enthalten sind. Die meistens *CS*-Klassen sind aktiv, da sie von einer der drei aktiven *CS*-Basisklassen erben.
	Stellt den `thread` zur Ereignisbehandlung eines Objektes dar.
	Repräsentiert einen `thread`, der von einer beliebigen Instanz einer Unterklasse der Klasse `CSDuT` zur Ausführung elementarer Sequenzen gestartet wird.
	Stellt den `thread` der Zustandsmaschine eines Objektes dar.
	Aufruf einer Methode innerhalb des Ereignis-`threads` eines Objektes. Der Text beinhaltet entweder den Namen oder eine Beschreibung der Methode.
	Zustandsänderungen werden, wie das Symbol zeigt, mit einem Anfangs-, Übergangs- und Endzustand dargestellt.
	Ereignis vom Typ `simple`.
	Ereignis vom Typ `synchronous` und die dazugehörige Antwort.

Tabelle 1.1 UML-Symbole und ihre Bedeutung für das *CS* Framework

Kapitel 2

Entwicklungsumgebung *CS* Framework

2.1 Überblick

2.1.1 Idee

Die Motivation für ein universelles Werkzeug zur Entwicklung von experimentspezifischen Kontrollsystemen entsteht aufgrund der geringen Wiederverwendbarkeit bereits entwickelter Software. Obwohl die Anforderungen, die die Experimente im Allgemeinen an ein Kontrollsystem stellen, viele Gemeinsamkeiten aufweisen, unterscheiden sich die Umsetzungen der Experimente im Einzelnen stark voneinander. Dies führt dazu, dass bei dem Entwurf und der Realisierung eines neuen Systems immer komplett von vorne begonnen werden muss, ohne auf bereits getätigte Entwicklungen zurückgreifen zu können und erhöht damit den benötigten Arbeitsaufwand.[1]
Oftmals kommen ähnliche oder sogar gleiche Geräte bei den Experimenten zum Einsatz. Die Wiederverwendbarkeit der Software zur Gerätesteuerung, beschränkt sich jedoch meist auf die Treiber. Werden jedoch Regelungsmechanismen benötigt, die die Kommunikation von Geräten untereinander voraussetzten, müssen diese komplett neu implementiert werden. Abhilfe kann an dieser Stelle nur durch eine modulare Programmierung, die einheitliche Schnittstellen bereitstellt, geschaffen werden. Dies ermöglicht nach Bedarf des Experiments einzelne Software-Module beliebig miteinander kombinieren zu können, unabhängig von der Art des Gerätes, dessen Schnittstellen oder Treibersoftware.
Auf diesem Grundgedanken aufbauend, besteht das Konzept des *CS* Frameworks in der Kombination von generischen Software-Modulen, die vom Frame-

[1] siehe [Beck2003a]

work zur Verfügung gestellt werden, mit experimentspezifischen Erweiterungen.[2] Abbildungen 2.1 und 2.2 verdeutlichen diese Idee nochmals.

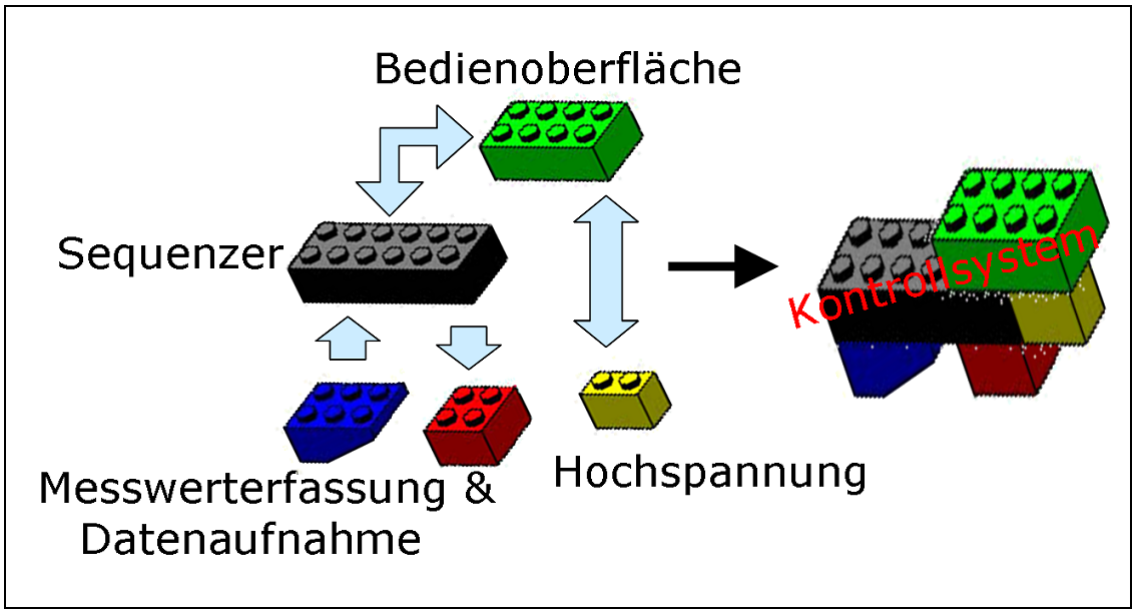


Abbildung 2.1 *CS*-Legobaukastensystem [CSWiki2]

Die Wartung, Dokumentation und Weiterentwicklung des Frameworks ist Aufgabe der Kontrollsystemgruppe, die die Experimente bei der Planung und Implementierung ihrer Kontrollsysteme beratend unterstützt. Die Anpassung des Frameworks an ein Kontrollsystem übernehmen die Experimentatoren selbst. Hierfür stellt das Framework bereits Entwurfsmuster zur Lösung bestimmter Aufgaben, wie beispielsweise dem Implementieren gerätespezifischer Software oder grafischer Bedienoberflächen, bereit.
Diese Zusammenarbeit zwischen Anwendungsentwicklern und Kontrollsystemgruppe gewährleistet, dass Erweiterungen des Frameworks sich an den Anforderungen der Experimente orientieren, ohne deren Wiederverwendbarkeit einzuschränken. Darüber hinaus können Entwicklungen der Experimente direkt in das Framework integriert werden, wie zum Beispiel gerätespezifische Software, die bei anderen Experimenten wieder verwendet werden kann.[3]
Zusammenfassend kann man sagen: Das Ziel des *CS* Frameworks ist die Verbesserung der Wiederverwendbarkeit, Wartbarkeit und Dokumentation von Software durch eine einheitliche Software-Architektur und die damit verbundene Ersparnis von Arbeitsaufwand.

[2] siehe [Beck2003b]
[3] siehe [Beck2005]

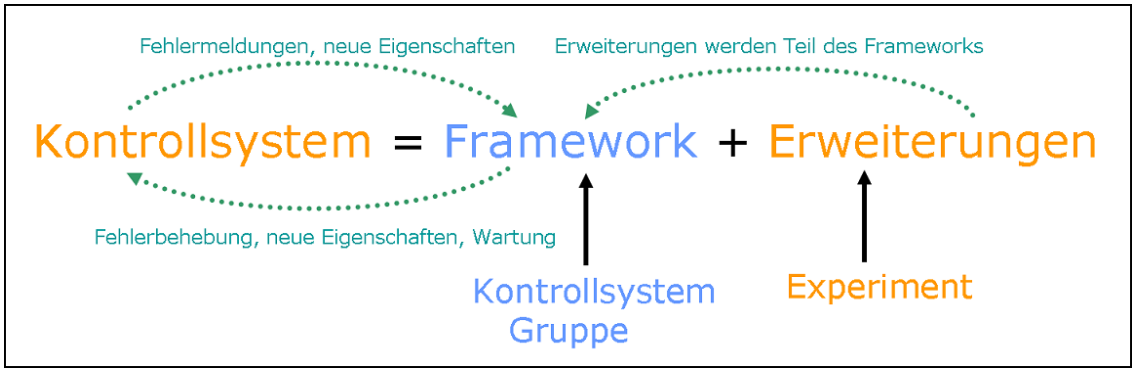


Abbildung 2.2 Idee des *CS* Framework [CSWiki2]

2.1.2 Anforderungen

Im Vergleich zu einer industriellen Produktionsanlage, die eher als statisch zu bezeichnen ist, befindet sich ein Experimentaufbau im ständigen Wandel.[4] Die verwendeten Geräte und deren Konfiguration ändern sich je nach Ziel des Experiments. Oftmals kommt eine Vielzahl an unterschiedlichen Gerätetypen zum Einsatz, die über verschiedene Schnittstellen und Protokolle angesprochen werden. Auch die Anzahl eines Gerätetyps kann mit der Ausbaustufe des Experiments variieren. Zunächst wird getestet, ob ein bestimmtes Gerät den Anforderungen des Experiments gerecht wird, erst anschließend wird die eigentlich benötigte Stückzahl angeschafft. Oder das Experiment expandiert und damit auch die Anzahl an Geräten. Daher muss die zur Hardware passende Software möglichst frei skalierbar sein, ohne hierfür jedes Mal Änderungen am Quellcode durchführen zu müssen.
Ähnlich dynamisch wie der Experimentaufbau verhält es sich auch mit der Kontinuität der Entwickler, bei denen es sich meist um Studenten oder Doktoranden handelt, die dem Experiment maximal für einen Zeitraum von einigen Jahren zur Verfügung stehen. Daher muss der Umgang mit dem Framework bzw. mit dessen Entwicklungsumgebung leicht erlernbar sein, damit der Aufwand für die Einarbeitungszeit in das Framework möglichst gering bleibt. In diesem Zusammenhang ist es von besonderer Wichtigkeit, dass eine zentrale Gruppe das Wissen und die Dokumentation der Software-Entwicklungen verwaltet, da ansonsten mit den studentischen Entwicklern auch das Wissen über die von ihnen entwickelte Software verschwindet.[5]
Eine weitere allgemeine Anforderung ist die Archivierung von Mess- und Konfigurationsdaten, damit der Experimentator nach Durchführung des Experiments zu Analysezwecken auf diese zurückgreifen kann. Ebenfalls ist die

[4] siehe [Beck2003a]
[5] siehe [Beck2003b]

Überwachung von Sensordaten mittels Alarmfunktionalitäten ein elementarer Bestandteil eines Kontrollsystems. Diese so genannten SCADA[6]-Funktionalitäten müssen daher im Kern des Frameworks durch entsprechende Module bereitgestellt werden.
Aus Gründen der Sicherheit ist eine Fernsteuerung der Hardware in der Regel von Nöten. Während der Durchführung eines Experiments wird aufgrund der auftretenden Strahlung der Zugang zum Experimentaufbau gesperrt. Dies hat zur Folge, dass die Bedienung des Kontrollsystems und der direkte Zugriff auf die Hardware, örtlich voneinander getrennt werden müssen. Daher ist die Vernetzung der Rechner, die diese Aufgaben übernehmen, unerlässlich.[7] Zudem erhöht man hiermit die Skalierbarkeit, da man die Module einer Anwendung frei auf verschiedene Rechner verteilen kann. Somit wird die Leistungsfähigkeit eines einzelnen Rechners nicht zum limitierenden Faktor für die Skalierbarkeit eines Kontrollsystems. In diesem Zusammenhang spielt die Plattformunabhängigkeit der Entwicklungsumgebung eine große Rolle, da auf den Rechnern der Experimente oftmals verschiedene Betriebssysteme, hauptsächlich Windows und Linux, zum Einsatz kommen.

2.1.3 Lösung

Das *CS* Framework basiert bis zur aktuellen Version 2.10 vollständig auf LabVIEW 7.1, da dieses eine Vielzahl der oben beschriebenen Anforderungen bereits erfüllt. LabVIEW dient hauptsächlich zur Programmierung von Anwendungen im Bereich der Mess- und Automatisierungstechnik und stellt daher standardmäßig eine Vielzahl von Treiberpaketen zur Ansteuerung diverser Hardware-Schnittstellen wie z.B. Profibus[8], GPIB[9] und RS232[10] zur Verfügung. Ebenso bietet National Instruments das DSC[11] Modul an, welches den Umfang von LabVIEW um SCADA-Funktionalitäten erweitert und den Zugriff auf OPC[12]-Serverapplikationen ermöglicht.[13]
Ein weiterer Vorteil von LabVIEW ist dessen grafische Programmierung, die einen intuitiven Umgang mit der Programmiersprache zulässt. Verglichen mit einer textuellen Programmiersprache, fällt der Einstieg in LabVIEW sehr ein-

[6] Supervisory Control and Data Aquisition; siehe Glossar Seite 88
[7] [Beck2003a]
[8] Process Field Bus ist ein Standard für die Feldbus-Kommunikation in der Automatisierungstechnik; siehe http://de.wikipedia.org/wiki/Profibus
[9] General Purpose Interface Bus; siehe http://de.wikipedia.org/wiki/IEC-625-Bus
[10] Standard für eine serielle Schnittstelle; siehe http://de.wikipedia.org/wiki/RS232
[11] Datalogging and Supervisory Control
[12] Openness, Productivity, Collaboration (vormals für: OLE for Process Control); siehe Glossar Seite 88
[13] siehe [Beck2003a]

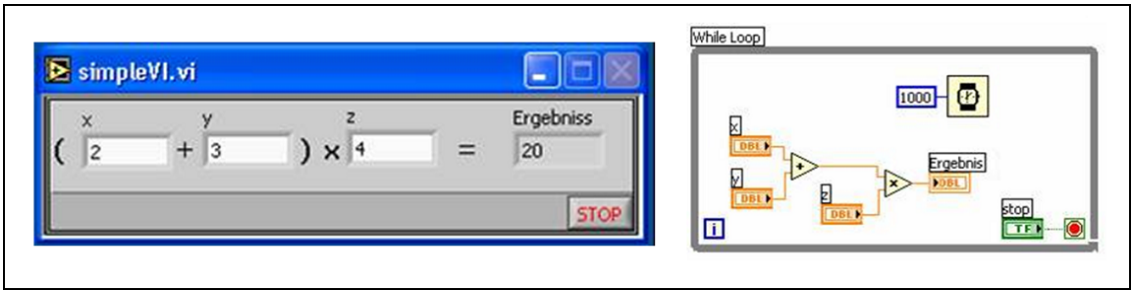


Abbildung 2.3 Front Panel und Block Diagramm in LabVIEW

fach, da der Programmierer sich nicht mit der Einhaltung von befehlsbedingtem Syntax auseinandersetzten muss. Die Implementierung des Programmablaufs erfolgt in LabVIEW durch Erstellen eines Blockdiagramms, in das der Programmierer Funktionen in Form von Blöcken, die in LabVIEW als `VIs`[14] bezeichnet werden, setzt und diese miteinander verbindet. Zu jedem Blockdiagramm existiert ein Front Panel, welches Elemente zur Dialoggestaltung bereitstellt. Die Implementierung von grafischen Bedienoberflächen wird dem Entwickler in LabVIEW daher extrem erleichtert.
Zudem unterstützt LabVIEW Multithreading[15] und verfügt über Mechanismen zur Ereignissteuerung und Synchronisierung, wie Warteschlangen und Semaphoren[16]. In Kombination mit den TCP/IP-Funktionen von LabVIEW wurde damit ein Kommunikationsmechanismus innerhalb des *CS* Framework geschaffen, der es ermöglicht, dass *CS*-Objekte rechnerübergreifend miteinander kommunizieren können. Darüber hinaus wird die CPU-Last aufgrund der Ereignissteuerung solange auf ein Minimum reduziert, wie keine Ereignisse im System auftreten.
Im Hinblick auf Skalierbarkeit und Modularität wurde ein objektorientierter Ansatz für das *CS* Framework gewählt. Da LabVIEW selbst keine Objektorientierung beinhaltet, verwendete man anfangs das Zusatzpaket ObjectVIEW[17] der Firma Vogel Automatisierungstechik, welches objektorientierte Programmierung in LabVIEW integriert. Aus Gründen der geringen Leistungsfähigkeit dieses Paketes wurde ein eigener objektorientierter Ansatz für das *CS* Framework entworfen und mit LabVIEW implementiert.[18]
Zum besseren Verständnis der Inhalte dieser Arbeit folgt ab Seite 13 eine Einführung in das *CS* Framework. Die Objektorientierung und der Ereignismechanismus stehen im Vordergrund der Betrachtung, da daraus maßgebliche Rahmenbedingungen für den Entwurf der Klassenbibliothek resultieren.

[14] Virtual Instrument; siehe Glossar Seite 90
[15] siehe Glossar Seite 88
[16] siehe Glossar Seite 89
[17] Mehr Informationen siehe http://www.vat.de/Download/Objectview.
[18] siehe [Beck2003a]

2.1.4 Status

In der aktuellen Version 2.10 beinhalten die Basisklassen des *CS* Framework die Semaphor geschützte Verwaltung der Objektattribute und die Bereitstellung des rechnerübergreifenden Ereignismechanismus. Außerdem bieten sie Entwurfsvorlagen für grafische Oberflächen und Zustandsmaschinen. Darüber hinaus existieren Klassen, die den Zugriff zum DSC Modul von LabVIEW ermöglichen, und damit SCADA-Funktionalität bereitstellen.
Der Schwerpunkt der letzten Entwicklungen bestand in der Erweiterung des Frameworks um gerätespezifische Klassen, die die jeweiligen Hardwaretreiber kapseln und eine einheitliche Schnittstelle für verschiedenste Gerätetypen innerhalb des *CS* Framework ermöglichen. In diesem Zusammenhang entstanden bis heute an die etwa 60 Geräteklassen für verschiedenste Gerätetypen von unterschiedlichen Herstellern. Entwickelt wurde das *CS* Framework unter Windows. Allerdings konnte es erfolgreich auf Linux portiert werden.[19]
Die Kontrollsysteme, die auf der Grundlage des *CS* Framework bisher entstanden, beschränken sich nicht auf die GSI, sondern verteilen sich bereits auf mehrere Forschungsinstitute. Im Einzelnen handelt es sich dabei um folgende Experimente, deren Kontrollsysteme mit dem *CS* Framework entwickelt wurden bzw. sich momentan im Aufbau befinden (Stand 2005):

GSI (Deutschland)	**CERN** (Schweiz)	**MSU**[20] (USA)
PHELIX SHIPTRAP Motion CaveA[24] RISING[25] FOPI[26]	REXTRAP[21] ISOLTRAP[23]	LEBIT[22]

Tabelle 2.1 Verbreitung des *CS* Frameworks

[20] Michigan State University; Mehr Informationen http://www.msu.edu/.
[21] Mehr Informationen http://rextrap.web.cern.ch/rextrap/.
[22] Steht für Low Energy Beam and Ion Trap. Mehr Informationen http://www.nscl.msu.edu/tech/devices/lebit/.
[23] Mehr Informationen http://rextrap.web.cern.ch/rextrap/.
[24] Schrittmotorsteuerung für fahrbare Blenden und Leuchtschirme. Mehr Information siehe [Beck2005].
[25] Mehr Informationen http://www-aix.gsi.de/~wolle/EB_at_GSI/main.html.
[26] Steht für 4 Pi und spielt auf die Fähigkeit des gleichnamigen Detektors an, den kompletten Raumwinkel erfassen zu können. Mehr Informationen http://www.gsi.de/forschung/kp/kp1/experimente/fopi/index.html/.

[19] siehe [Beck2003b]

2.2 Objektorientierung

Da LabVIEW keine Objektorientierung unterstützt, wurde eine objektorientierte Entwicklungsumgebung für das *CS* Framework mit den Funktionen, die LabVIEW zur Verfügung stellt, realisiert. Die folgenden Abschnitte erläutern wie die grundlegenden Prinzipien der Objektorientierung wie Instanzierung, Attributverwaltung und Vererbung im Rahmen des *CS* Frameworks umgesetzt wurden. Dabei ist allerdings zu beachten, dass es sich um keine Objektorientierung im Sinne einer Programmiersprache wie Java oder C++ handelt. Viel mehr spricht man im Falle des *CS* Frameworks von einem objektorientierten Ansatz. Dies äußert sich beispielsweise in dem komplexen Vererbungsprozess, dessen manuelle Ausführung einerseits einige Zeit in Anspruch nimmt, andererseits ein entsprechendes Hintergrundwissen über das *CS* Framework voraussetzt. Darüber hinaus gibt es viele objektorientierte Prinzipien wie z.B. die Sichtbarkeit von Methoden und Attributen, abstrakte Klasse oder das Singleton[27]-Muster, die im *CS* Framework nur per Konvention existieren. Das heißt, es gibt z.B. keinen Mechanismus, der es verbietet, Sichtbarkeitsgrenzen zu überschreiten oder Klassenmethoden bzw. -attribute missbräuchlich zu verwenden. Der LabVIEW-Compiler weiß von der Objektorientierung des *CS* Framework nichts und kontrolliert daher nur die Einhaltung des LabVIEW Syntax. An dieser Stelle ist die Disziplin des Entwicklers gefordert, sich beim Implementieren an die *CS*-Konventionen zu halten. Hierfür bietet das *CS* Framework zahlreiche Entwurfsvorlagen, anhand derer ein Entwickler seine Klassen aufbauen kann.

Bei Instanzen einer *CS*-Klasse handelt es sich im Allgemeinen nicht um passive Datencontainer, die Methoden zur Datenmanipulation bereitstellen, sondern viel mehr um aktive Prozesse, die miteinander interagieren und unabhängig von Benutzerinteraktionen, aktiv werden können. Hierzu zählt beispielsweise das Versenden von Ereignissen zwischen Objekten oder die Transition zwischen zwei Zuständen als Reaktion auf äußere Veränderungen. Aufgrund der Kombination von objektorientiertem Design, ereignisgesteuerter Kommunikation und nebenläufiger Verarbeitung kann man im Falle des *CS* Frameworks von einem Agentensystem reden. Aktive *CS*-Objekte sind Software Agenten[28], deren generisches Verhalten in Form von Basisklassen vom Framework bereitgestellt wird.

[27] Die Singleton-Eigenschaft einer Klasse sorgt dafür, dass nur ein einziges Objekt dieser Klasse erzeugt werden kann; siehe [Balzert2005] Seite 391

[28] siehe Glossar Seite 89

2.2.1 Klassen und Instanzierung

Eine *CS*-Klasse besteht, wie es in der Objektorientierung üblich ist, aus Attributen, Konstruktor, Destruktor und weiteren Methoden, die auf die Attribute operieren. Zusätzlich besitzt jede aktive *CS*-Basisklasse ein `VI Template`[29]. Dieses enthält den aktiven Code einer Klasse.
Bei der Instanzierung eines Objektes wird der Konstruktor der entsprechenden Klasse aufgerufen, welcher zur Laufzeit eine Instanz des `VI Templates` in den Speicher lädt und eine dazugehörige, eindeutige Referenz erzeugt wird.[30] Hierfür verwendet das *CS* Framework die `VI server`[31]-Funktionen von LabVIEW. Abbildung 2.4 und 2.5 zeigen den dynamischen Aufruf von Methoden und `VI Templates`. Die Referenz auf die Instanz des `VI Templates`

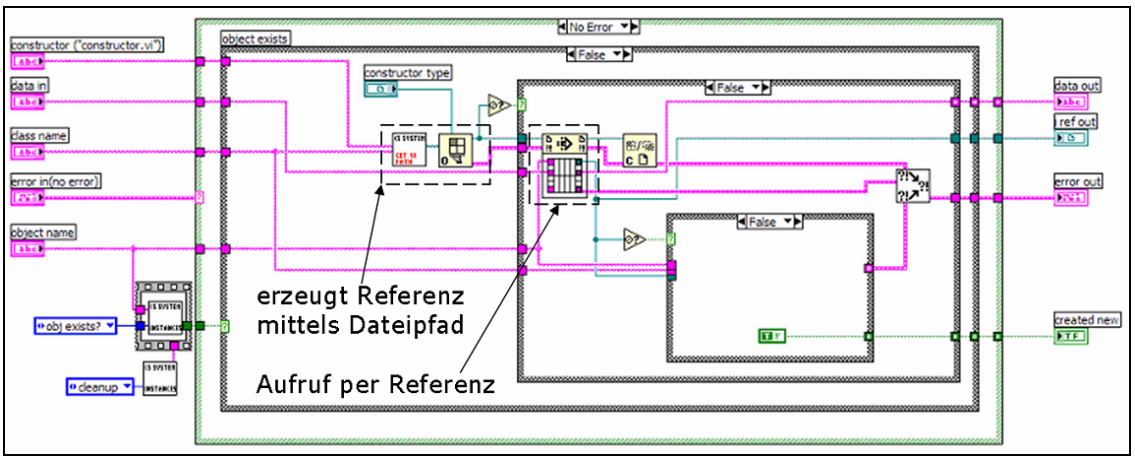


Abbildung 2.4 `CSSystem._new.vi` dient dem dynamischen Aufruf des Konstruktors einer Klasse

speichert die Klasse in ihren Attributen, um diese im Destruktor schließen zu können und den allokierten Platz im Speicher wieder frei zu geben. Unter einer Objektreferenz versteht man im *CS* Framework die Referenz auf die Instanz der Klasse `CSObj`. Sie dient als Oberklasse aller *CS*-Klassen.
Ein `VI Template` einer Klasse besteht aus einem oder mehreren `threads`[32], die die klassenspezifischen Aufgaben ausführen. Eine genaue Beschreibung der `VI Templates` der *CS*-Basisklassen gibt 2.3.2.
Für alle Methoden einer Klasse, die in Form von `VIs` abgespeichert werden, existiert die Namenskonvention *`CLASSNAME`*`.METHODNAME.vi`. Zusammen mit der Verzeichnisstruktur der *CS*-Klassenbibliothek verwaltet das Framework die Dateipfade der `VIs`, die benötigt werden, um zur Laufzeit entsprechende

[29] Virtual Instrument Template; siehe Glossar auf Seite 90
[30] siehe [Brand] Seite 75
[31] siehe Glossar auf Seite 90
[32] siehe Glossar auf Seite 89

Methoden einer Klasse, wie z.B. Konstruktor oder Destruktor aufzurufen.[33]

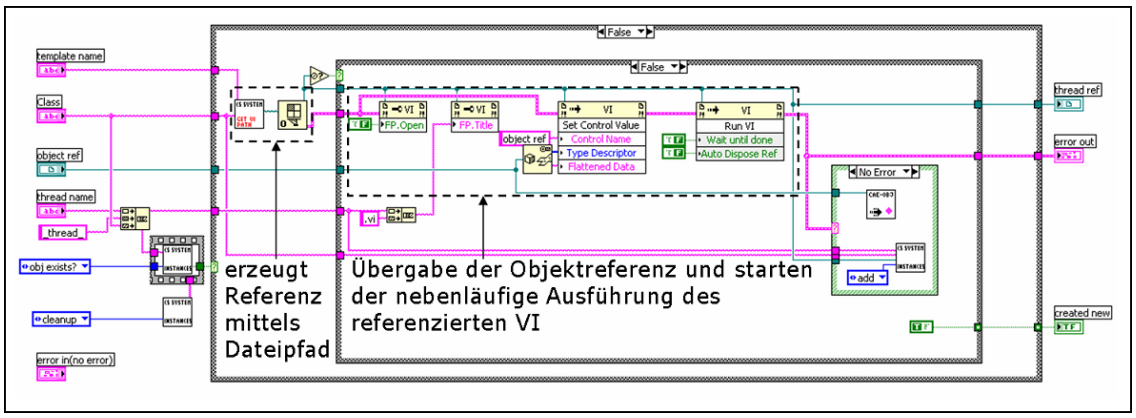


Abbildung 2.5 `CSSystem._startThread.vi` startet eine Instanz eines Klassen-`Templates`

Das *CS* Framework bietet dem Entwickler für den Umgang mit Klassen einen Browser, welcher das Editieren der Methoden ermöglicht und weitere klassenspezifische Informationen wie Vererbungsbeziehungen und Dokumentation darstellt.
Daneben existiert eine Applikationen, die die Vererbung von Klassen vereinfacht, da das manuelle Vererben im *CS* Framework einen relativ großen Zeitaufwand darstellt, der durch diese Anwendung automatisiert und beschleunigt wird.
Für Debugging- und Testzwecke existiert eine Anwendungen, die grundlegende Klassenfunktionalitäten wie den Zugriff auf die Attribute oder die Instanzierung einzelner `Templates` überprüft.

[33] siehe [Wiki1]

2.2.2 Objekt- und Attributverwaltung

Die Objektreferenzen sowie Referenzen auf `Template`-Instanzen werden innerhalb eines *CS*-Systems zusammen mit dem dazugehörigen Objekt- und Klassenamen sowie den Namen der Oberklassen in einem nicht initialisierten Shiftregister gespeichert (siehe Abbildung 2.6). Der Registerinhalt wird aktualisiert sobald eine neue Instanz erzeugt oder eine bereits vorhandene zerstört wird.

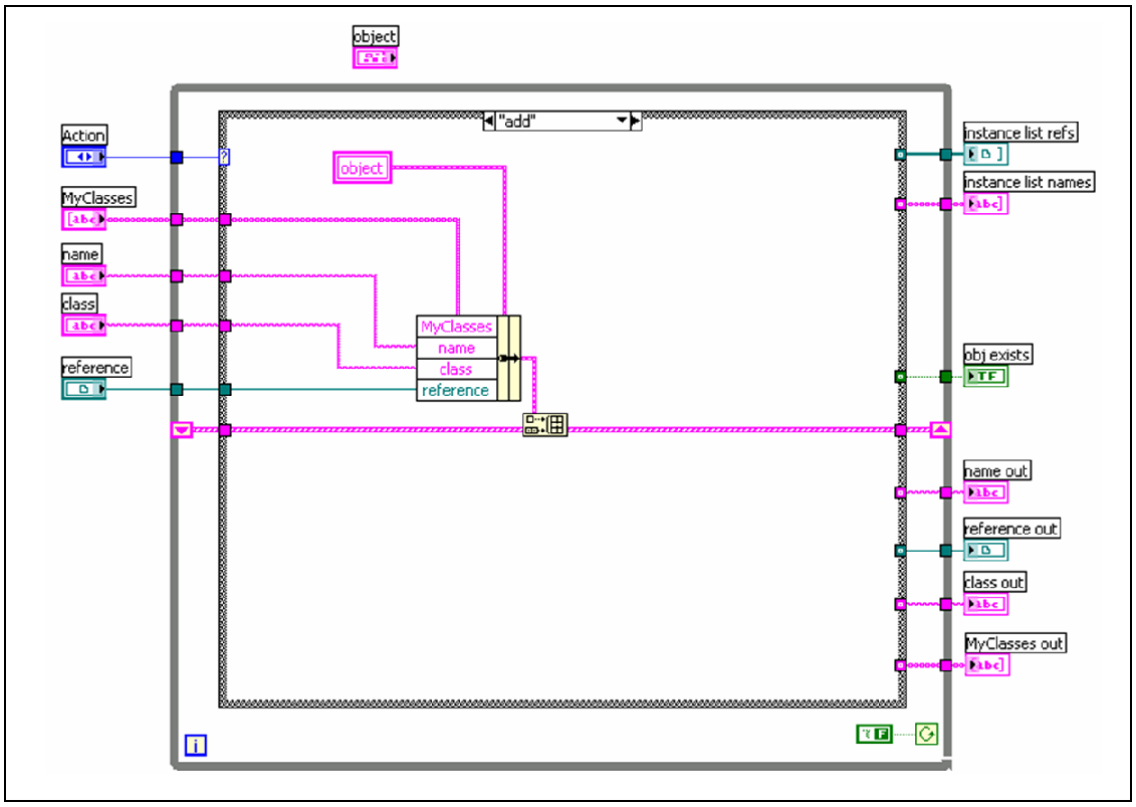


Abbildung 2.6 `CSSystemLib.instances.vi` verwaltet die Objekt- und `thread`-Referenzen innerhalb eines lokalen *CS*-Systems

Derselbe Mechanismus wird zur Attributverwaltung verwendet. Zu jeder Klasse existiert ein `VI` (siehe Abbildung 2.7), welches ebenfalls in einem Shiftregister, die Referenzen und die Attribute aller Instanzen einer Klasse verwaltet. Im Gegensatz zu dem vorher verwendeten ObjektVIEW, welches die Attribute in den Anzeigeelementen des Front Panels speichert, ist der Zugriff auf ein Shiftregister um etwa den Faktor 100 schneller, da ein Thread-Wechsel zwischen dem Blockdiagramm und dem Front Panel vermieden wird.[34]

[34] siehe [Brand] Seite 75

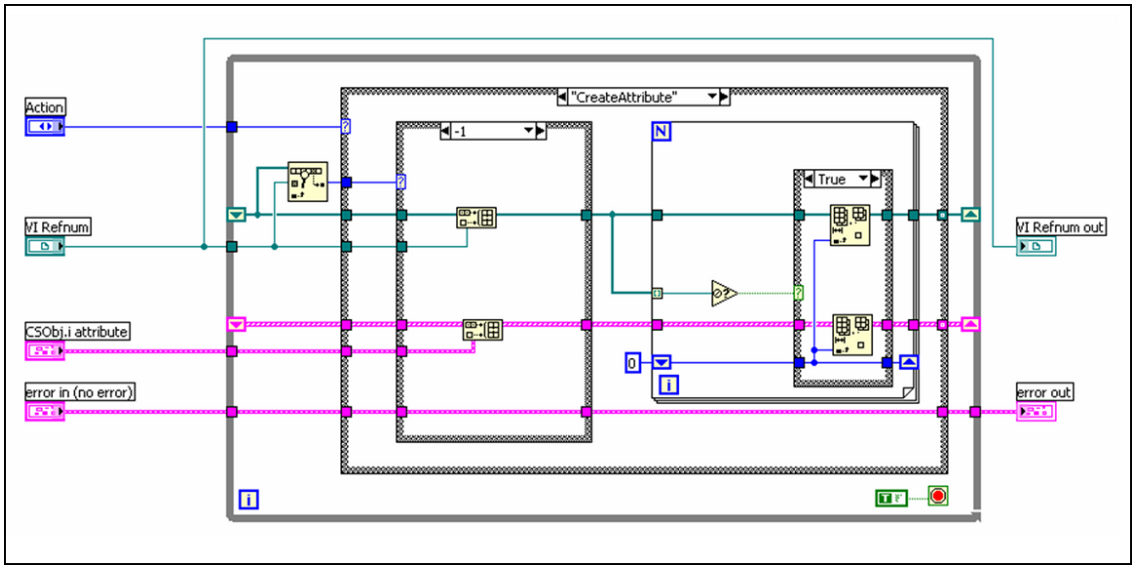


Abbildung 2.7 VI zur Verwaltung der Attribute aller Instanzen einer Klasse

Die grundlegenden Eigenschaften eines *CS*-Objektes sind in der Basisklasse `CSObj` zusammengefasst. Sie beinhaltet die Zugriffsmethoden auf die Attribute eines Objektes. Da im *CS* Framework ein Objekt aus mehreren `threads` bestehen kann, die asynchron nebeneinander laufen, benötigt man eine Möglichkeit, den parallelen Zugriff auf die Attribute eines Objekts zu verhindern, um so deren Konsistenz sicherzustellen. Ein solcher Ausschlussmechanismus ist mit Hilfe von Semaphoren, die von LabVIEW zur Verfügung gestellt werden, realisiert, und kann bei jedem Attributzugriff optional verwendet werden.

2.2.3 Vererbung

Innerhalb des *CS* Frameworks spiegeln sich Vererbungsbeziehungen in der Schachtelung von Konstruktoren wieder. Im Falle der einfachen Vererbung wird der Konstruktor der Oberklasse im Konstruktor der Unterklasse aufgerufen. Mehrfachvererbung ist ebenfalls möglich, indem mehrere Konstruktoraufrufe innerhalb eines Konstruktors stattfinden. Kommt es dadurch zum mehrfachen Aufruf des Konstruktors einer Oberklasse, wird diese immer nur einmal vererbt.
Zu beachten ist in diesem Zusammenhang auch, dass ein Konstruktoraufruf außerhalb eines Konstruktors eine neue Instanz der Klasse erzeugt und nicht der Vererbung dient. Soll jedoch innerhalb eines Konstruktors eine neue Instanz erzeugt werden, geschieht dies mittels des `new`-Operators (siehe Abbildung 2.4).[35] Auf diese Weise können assoziative Klassenbeziehungen wie Kompositionen oder Aggregationen implementiert werden.

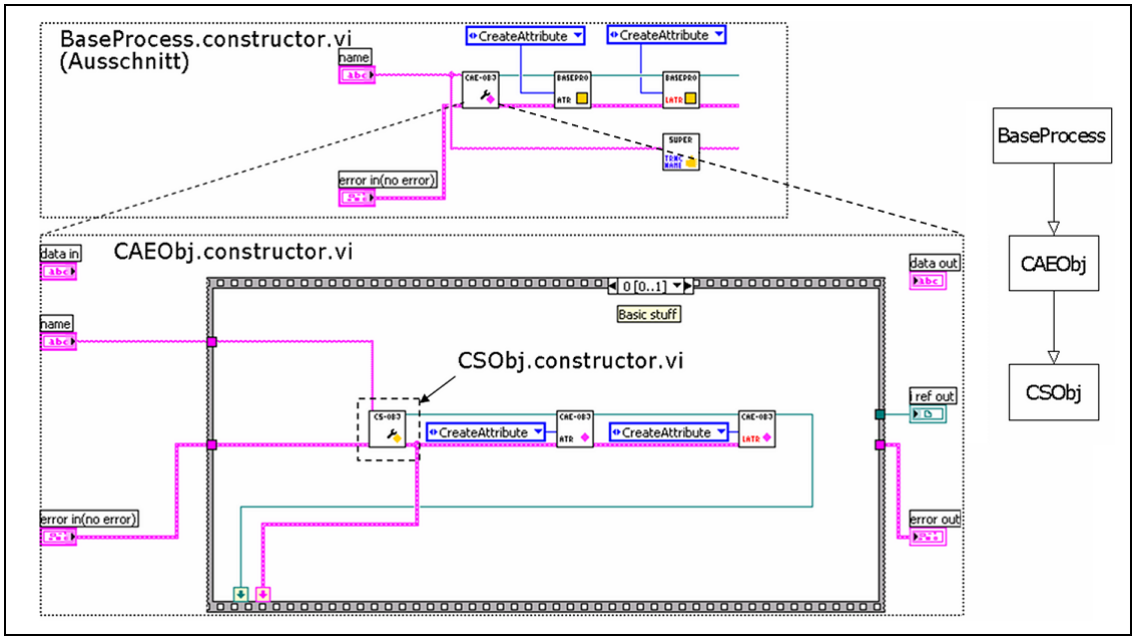


Abbildung 2.8 Vererbung durch geschachtelte Konstruktoren

[35] siehe [Brand] Seite 76

2.3 Klassenbibliothek

2.3.1 Drei-Schichten-Architektur

Um die Modularität und Flexibilität des Frameworks zu gewährleisten, wird eine Drei-Schichten-Architektur verwendet, wie sie häufig in der Informatik anzutreffen ist.[36] Durch einheitliche Schnittstellen und Kapselung zusammengehöriger Aufgaben in Klassen, können Komponenten einer Schicht ohne großen Aufwand ausgetauscht werden, ohne den kompletten Quellcode einer Anwendung überarbeiten zu müssen. Dies setzt allerdings eine strikte Kategorisierung der *CS*-Klassen in Geräteschicht, Anwendungsschicht und Operationsschicht voraus.
Gemäß dem Schichtenmodell, dessen Grundgedanke häufig Ausgangspunkt für den Entwurf von modularen Informationssystemen darstellt, benutzen weiter oben liegende Schichten die Funktionalitäten der unteren Schichten. Das wohl bekannteste Beispiel hierfür ist das OSI-Modell[37]. Die Schichten des *CS* Frameworks und deren Aufgaben werden im Folgenden näher erläutert:

Geräteschicht Die unterste der drei Schichten stellt die Schnittstelle zur Hardware her. Dazu kapseln Geräteklassen die Treibersoftware, um eine einheitliche Schnittstelle für den Zugriff auf Geräte innerhalb des *CS* Framework zu ermöglichen. Die Instanz einer Geräteklasse repräsentiert in der Regel eine Hardwarekomponente. Zur Laufzeit kann man durch Instanzierung weiterer Objekte ein Kontrollsystem dynamisch an die tatsächliche Anzahl an Geräten und Gerätetypen anpassen.

Anwendungsschicht Klassen der Anwendungsschicht verwenden Instanzen der Geräteschicht, um die Gerätefunktionalitäten in einen für den Anwender sinnvollen Zusammenhang zu bringen. Typische Aufgaben der Anwendungsschicht sind die Bereitstellung von SCADA-Funktionalität, Sequenzern oder eine Benutzerverwaltung.

Operationsschicht Auf dieser Ebene wird typischerweise die Schnittstelle zum Benutzer in Form von grafischen Bedienoberflächen hergestellt. Durch die Aufteilung der Klassen in Schichten können Bedienoberflächen auf eine spezifische Anwendung angepasst werden ohne die Prozesse im Hintergrund verändern zu müssen.

---

[36] siehe [Balzert2005] Seite 448

[37] Open Systems Interconnection; Schichtenmodell für die Kommunikation offener, Informationsverarbeitender Systeme. siehe http://de.wikipedia.org/wiki/OSI-Schichtenmodell

2.3.2 Basisklassen

Das *CS* Framework stellt fünf abstrakte[38] Basisklassen bereit. Mittels Vererbung kann der Entwickler darauf aufbauend seine eigenen Klassen entwerfen. Einen wichtigen Mechanismus stellen in diesem Zusammenhang virtuelle Methoden[39] der Basisklassen dar, deren Schnittstelle bereits in den Basisklasse definiert ist. Die Methode selbst wird jedoch erst in der Unterklasse implementiert:

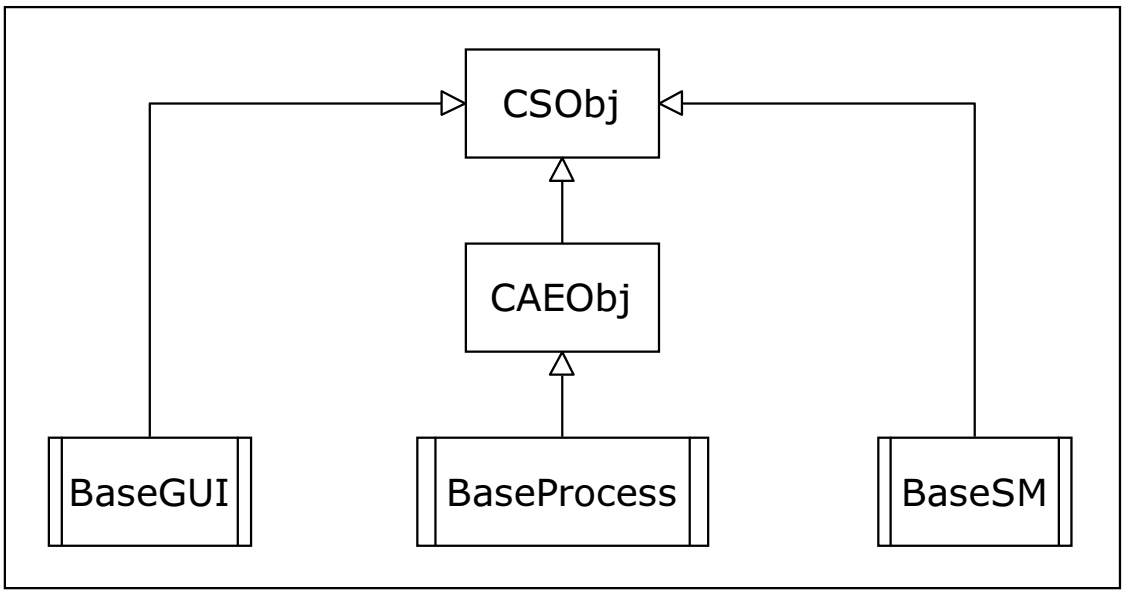


Abbildung 2.9 Basisklassen des *CS* Frameworks

CSObj Die Basisklasse aller *CS*-Klassen erzeugt in ihrem Konstruktor eine Referenz auf die Instanz ihres `VI Templates`. Diese wird an die Konstruktoren der abgeleiteten Klassen zurückgegeben und dient als Objektreferenz, die eindeutig das Objekt identifiziert. Mit ihrer Hilfe können sowohl Informationen wie Instanz-, Klassenname und die Namen der Oberklassen abgefragt, als auch auf die Objektattribute zugegriffen werden.
Zusätzlich verwaltet die Klasse einen Semaphor, der den wechselseitigen Zugriff auf gemeinsam genutzte Ressourcen durch verschiedene `threads` eines Objektes verhindert. Dieser Mechanismus wird vor allem für den konsistenten Schreib- und Lesezugriff auf die Objektattribute verwendet.
Es handelt sich nicht um eine aktive Klasse, da das dazugehörige `VI Template` nur der Erzeugung der Objektreferenz dient, selbst jedoch keine `threads` oder sonstigen Code beinhaltet.

[38] Abstrakte Klassen dienen im Gegensatz zu konkreten Klassen nur der Vererbung. Es können keine Objekte von abstrakten Klassen instanziert werden; siehe [Balzert2005] Seite 305
[39] siehe Glossar auf Seite 90

CAEObj Die Klasse stellt die Methoden für die Handhabung des Ereignismechanismus bereits. Dazu gehört das Erzeugen, Versenden und Empfangen von Ereignissen. Die Klasse besitzt selbst keinen `thread`, der eintreffende Ereignisse verarbeitet, sondern bietet nur die Methoden, die eine solche Klasse benötigt.
Da in LabVIEW die Ereignismethoden keinen einheitlichen Wartemechanismus bereitstellen, unterstützt die Klasse `CAEObj` nur den Austausch von gepufferten Ereignisse über Warteschlangen[40]. Sollen ungepufferte Ereignisse[41] versandt werden, erzeugt die entsprechende `CAEObj`-Instanz ein Objekt der Klasse `NotifierObj`, welches die Behandlung von ungepufferten Nachrichten in einem eigenen Thread durchführt. Dies bietet den Vorteil, dass die Behandlung verschiedener Ereignistypen sich nicht gegenseitig blockieren, sondern nebenläufig ausgeführt werden. Die Kapselung in unterschiedliche Klassen ermöglicht die Erweiterung um weitere Ereignistypen ohne Veränderung bestehender Klassen.

BaseGUI Diese Klasse dient als Vorlage für die Implementierung von Benutzeroberflächen. Neben einem `thread` für das GUI[42], beinhaltet die Klasse `BaseGUI` Methoden, um auf Parameter des GUIs programmatisch zugreifen zu können. Dazu zählen beispielsweise, die Größe und Position des Front Panels und deren Bedienelemente oder dessen Sichtbarkeit.

BaseProcess Das `Template` des `BaseProcess` besteht aus zwei `threads`: die Ereignisschleife und die periodische Schleife, welche beide über eine Watchdog[43]-Funktionalität verfügen.[44] Die periodische Schleife dient der Durchführung von zyklischen Aktionen, was z.B. im Falle mancher Geräte benötigt wird, um diese periodisch auszulesen oder abzufragen.
Die wichtigere Funktionalität besteht in der Ereignis-Schleife, die die Kommunikation zwischen Objekten ermöglicht. Als Unterklasse der Klasse `CAEObj` verfügt die Klasse `BaseProcess` über Methoden zur Ereignissteuerung. Jede Instanz einer Unterklasse der Klasse `BaseProcess`, auch `Prozess`[45] genannt, besitzt mit der Ereignis-Schleife einen `thread`, der auf eintreffende Ereignisse wartet. Sobald ein Ereignis sich in der Warteschlange befindet, wird die Ereignis-Schleife aktiv und ruft über ihre virtuelle Methode die entsprechenden Methode der Unterklasse auf. Zu diesem Zweck besitzt jede

[40] In LabVIEW mit Queue bezeichnet.
[41] In LabVIEW mit Notifier bezeichnet.
[42] Graphical User Interface; deutsche Übersetzung: grafische Benutzeroberfläche
[43] siehe Glossar Seite 90
[44] siehe [Beck2003b]
[45] siehe Glossar auf Seite 88

Klasse, die vom `BaseProcess` vererbt hat, eine Methode namens *CLASSNAME*`.ProcCases.vi`, in die der Entwickler seine Ereignisbehandlungsmethoden implementiert. Ebenso müssen in einer weiteren Methode namens *CLASSNAME*`.ProcEvents.vi` die Namen der Ereignisse, die ein Objekt der Klasse empfangen kann, definiert werden. Dasselbe Prinzip wird auch bei der periodischen Schleife verwendet. Die Schnittstelle ist in der Basisklasse `BaseProcess` implementiert, die aufgerufene Methode in der Unterklasse. Die entsprechende Methode für die periodische Schleife heisst *CLASSNAME*`.ProcPeriodic.vi`.

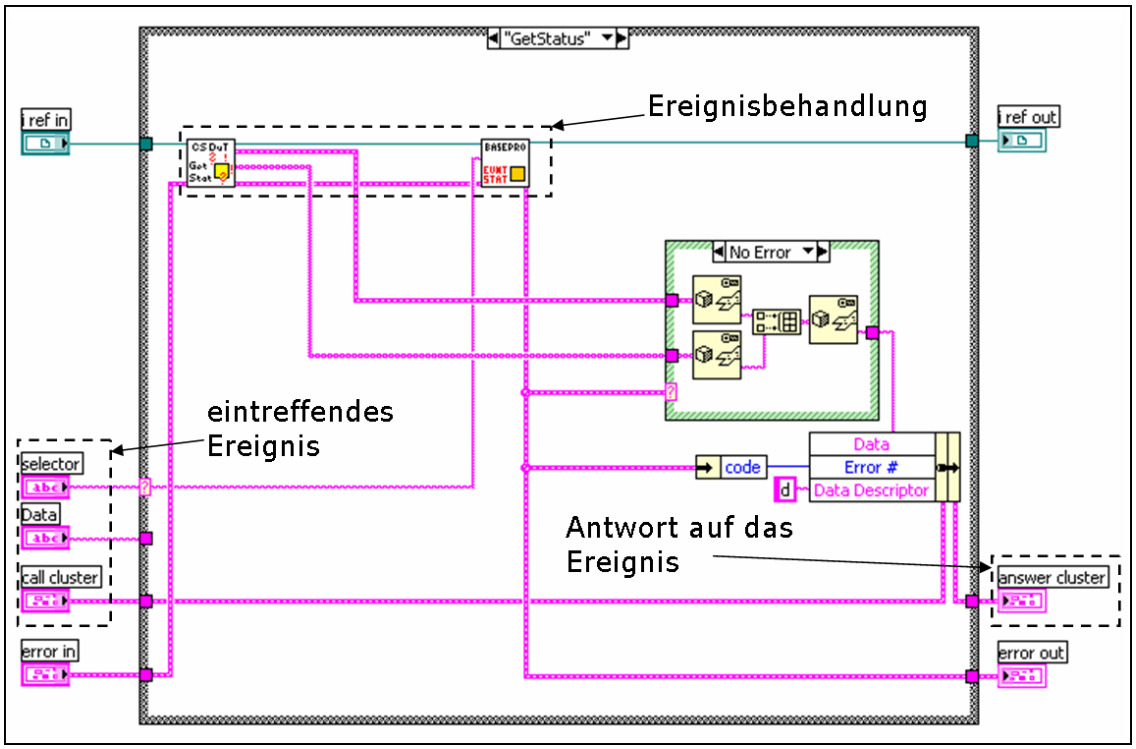


Abbildung 2.10 Beispiel eines `ProcCases.vi` für die Ereignisbehandlungsmethode eines `Prozesses`

Eine weitere Fähigkeit des `BaseProcess` besteht in dem Auslesen einer Datenbank, die Datensätze für `Prozesse`, enthält.[46] Da die Klasse `BaseProcess` meist als Oberklasse für Geräteklassen verwendet wird, enthält die Datenbank in diesen Fällen meist Daten zur Konfiguration der Hardware, wie zum Beispiel Busadressen.

[46] siehe [Brand]

BaseSM Die Klasse `BaseSM` startet in ihrem `Template` einen `thread`, der eine Zustandsmaschine realisiert. Die Zustände und deren Transitionen werden jedoch in der Methode *`CLASSNAME`*`.ProcState.vi` der Unterklasse vom Entwickler implementiert, die vom `thread` der Basisklasse aufgerufen wird. Die Methoden zur Initialisierung und Zustandsänderung werden durch die Basisklasse gekapselt. Die Klasse `BaseSM` stellt zusätzlich Methoden für den Entwickler zur Verfügung, um Zustandwechsel anzustoßen, oder den aktuellen Zustand abzufragen. Durch Kombination mit anderen Basisklassen können Transitionen z.B. als Reaktion auf ein Ereignis oder eine Benutzerinteraktion ausgelöst werden.

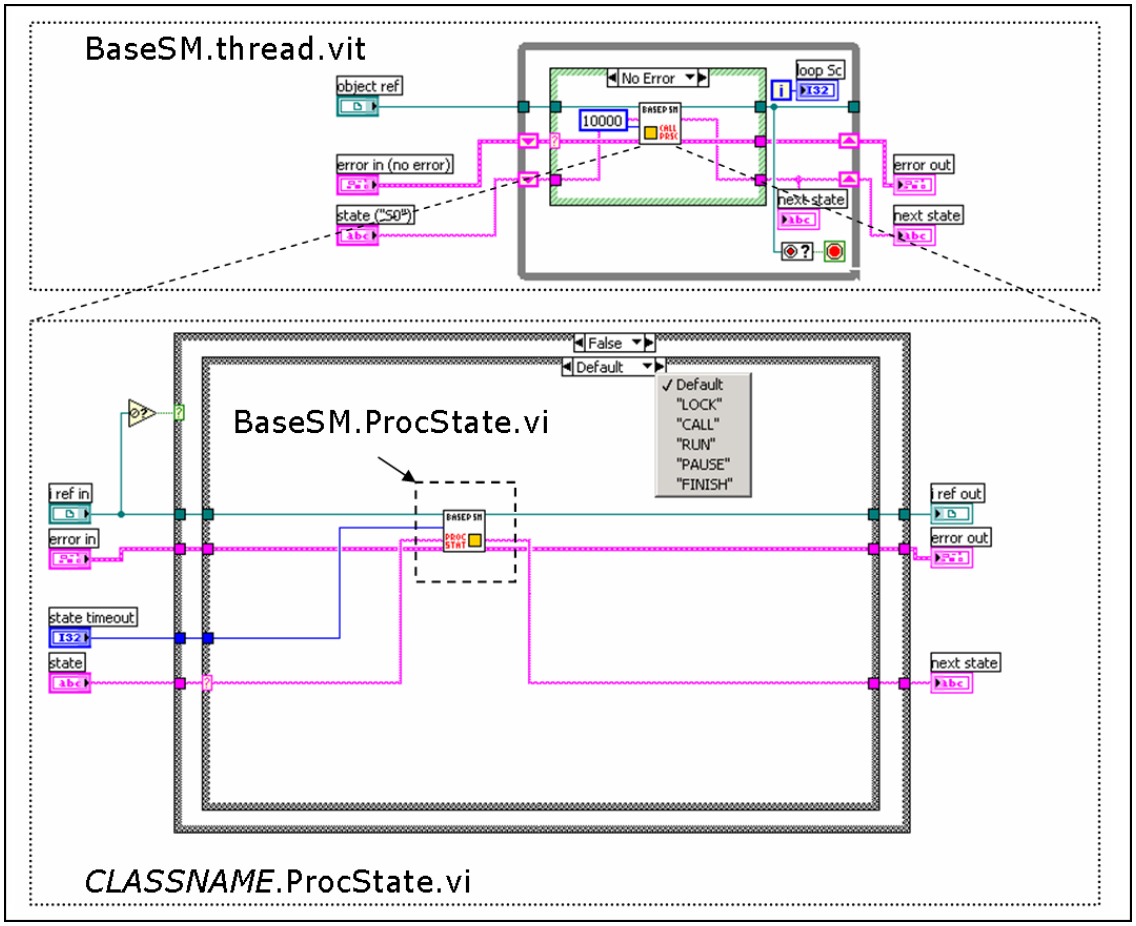


Abbildung 2.11 virtueller Methodenaufruf am Beispiel der Klasse BaseSM

2.3.3 System-Klassen

Neben den Basisklassen, die dem Entwickler als Grundlage für eigene Klassen dienen sollen, beinhaltet das *CS* Framework noch weitere Klassen, die für den Betrieb des Frameworks wichtig sind. Wie das dazugehörige Klassendiagramm (siehe Abbildung 2.12) zeigt, sind diese Klassen alle durch Vererbung aus den drei Basisklassen entstanden.

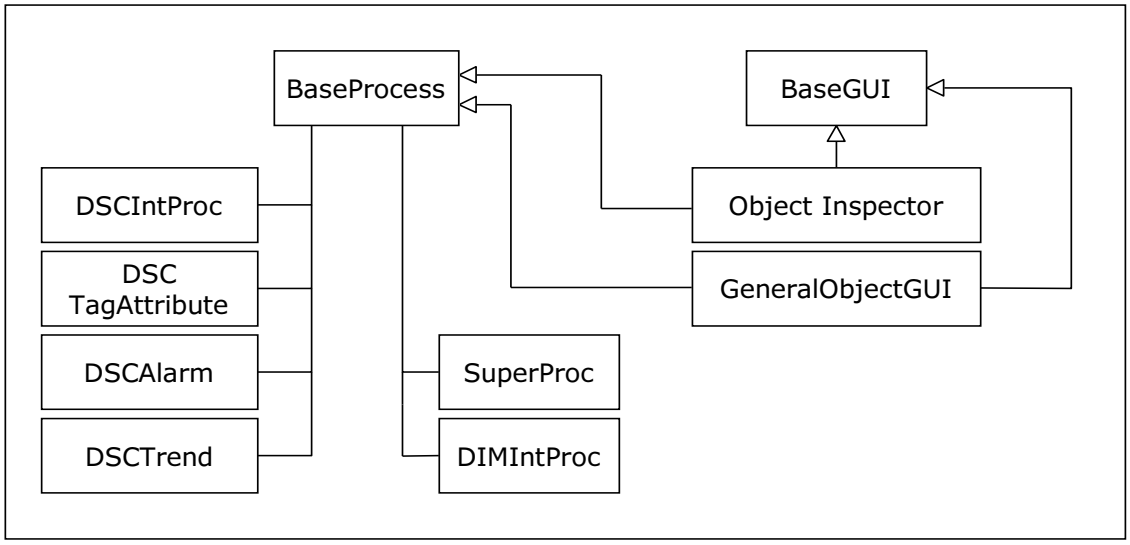


Abbildung 2.12 System-Klassen des *CS* Framework

SuperProc Eine Instanz namens `Super_`*RECHNERNAME* der Klasse `SuperProc` wird beim Start eines *CS*-Systems instanziert. Sie dient als Container-Objekt für alle *CS*-Objekte und ist per *CS*-Konvention ein Singleton-Objekt.[47] Ihre Aufgabe besteht im Erzeugen, Zerstören und Reservieren von Prozessen. Reservierte Prozesse können, solange noch bestehende Reservierungen existieren, nicht vom `Super` zerstört werden.

ObjectInspector Die Instanz dieser Klasse dient dem Entwickler als Werkzeug zum Überwachen und Debuggen. Sie zeigt alle *CS*-Instanzen und deren Status an. Je nach Klasse des Objektes können Informationen über die Ereigniswarteschlange, die Zustandsmaschine, die Lebenszeit der Instanz, usw. angezeigt werden. Zusätzlich können die aktuelle Ereignisrate des *CS*-Systems sowie aufgetretene Ereignisse mit zusätzlichen Informationen wie Sender, Empfänger und gesendete Daten abgefragt werden. Ebenfalls ermöglicht sie dieselben Informationen von anderen *CS*-Systemen über das Netzwerk abzufragen und anzuzeigen.

[47] siehe [Brand] Seite 89

GeneralObjectGUI Ähnlich wie der `Object Inspector` stellt diese Klasse dem Entwickler eine GUI zur Verfügung, mit deren Hilfe Objekte instanziert und zerstört werden können. Darüber hinaus können Ereignisse an Objekt versandt werden, um während der Entwicklungsphase deren Verhalten zu testen oder im Betrieb Einstellungen vorzunehmen. All diese Funktionalitäten sind optional auch automatisierbar, sodass beim Start oder auf Wunsch des Entwicklers eine Liste von Objekten instanziert wird und diesen bestimmte Ereignisse zugeschickt werden, um ein System in einen Anfangszustand zu versetzten, oder verschiedene Einstellungen zu laden.

SCADA-Klassen Die SCADA Funktionalitäten werden durch Interface Klassen in das *CS* Framework integriert. Auf diese Weise können verschiedene SCADA Module innerhalb des Frameworks benutzt oder gegeneinander ausgetauscht werden. Für den Zugriff auf das LabVIEW DSC Modul existieren im *CS* Framework bereits folgende Klassen:

`DSCIntProc`: Bietet Schreib- und Lesezugriff auf Prozessvariablen (`tags`) des DSC Moduls. Darüber hinaus verfügt die Klasse über eine Überwachungsfunktionalität: Jeder *CS*-Prozesse kann sich für `tags` bei der `DSCIntProc`-Instanz registrieren und wird über alle Werteänderungen der `tags` durch die `DSCIntProc`-Instanz per Ereignis benachrichtigt.

`DSCTagAttr`: Ermöglicht Eigenschaften von `tags`, wie z.B. deren Alarmeinstellungen oder Wertebereiche, zur Laufzeit zu manipulieren.

`DSCAlarm`: Bietet Zugriff auf die Echtzeitdatenbank des DSC Moduls, um Alarmzustände von `tags` abzufragen.

`DSCTrend`: Bietet Zugriff auf die Echtzeitdatenbank des DSC Moduls, um Werteverläufe von `tags` abzufragen.

2.3.4 Netzwerk-Klassen

Die Kommunikation im Netzwerk, basierend auf TCP/IP, realisieren im *CS* Framework folgende drei Klassen:

- `QueueListener`
- `QueueClient`
- `QueueServer`

Sollen Objekte über Rechnergrenzen miteinander kommunizieren, muss zunächst eine Verbindung zwischen den Rechnern aufgebaut werden. Dafür wird als erstes ein Objekt der Klasse `QueueClient` instanziert und die Verbindungsanfrage an das Listener-Objekt des Zielrechners übertragen. Diese startet auf ihrem System ein Objekt der Klasse `QueueServer`, welches die Verbindung mit dem dazugehörigen Client-Objekt aufrechterhält. Um den Zustand der Verbindung zu überwachen, werden Ping-Ereignisse ausgetauscht, um eine automatische Wiederaufnahme einer Verbindung nach einer Unterbrechung zu ermöglichen. Diese Verbindung ist unidirektional und muss daher in der entgegengesetzten Richtung ebenfalls aufgebaut werden, um eine wechselseitige Kommunikation zu etablieren.[48]

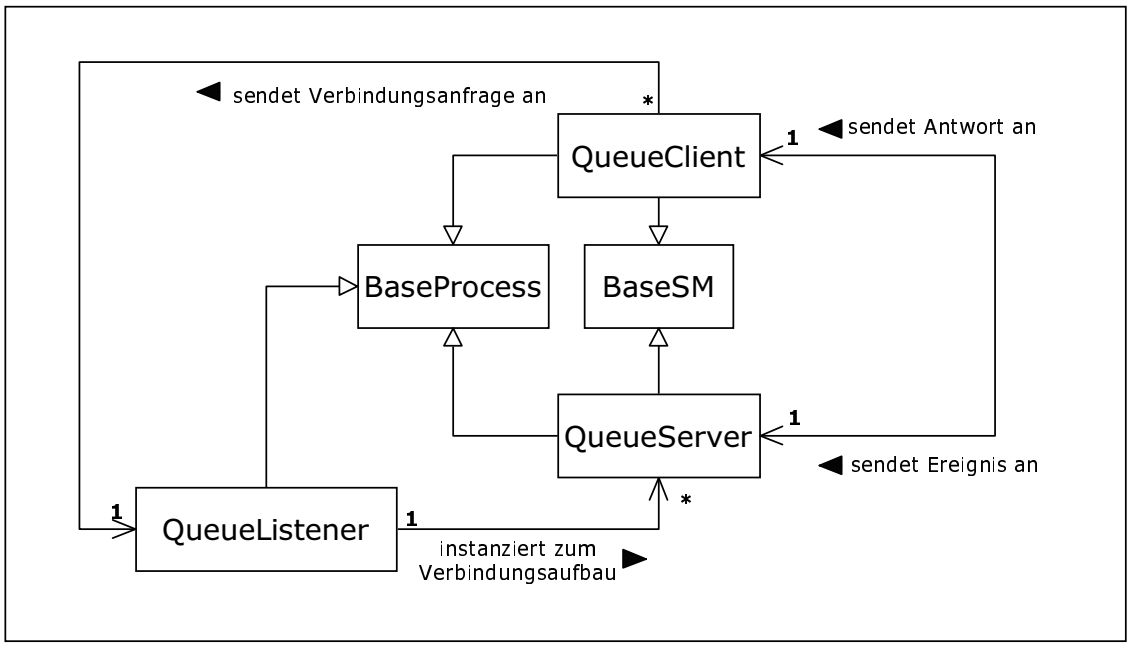


Abbildung 2.13 Netzwerk-Klassen des *CS* Framework

[48] siehe [Brand] Seite 108ff

2.4 Ereignismechanismus

In der Regel werden Methoden einer Klasse im *CS* Framework nicht direkt aufgerufen, sondern per Ereignis ausgelöst.[49] Hierzu verwendet das *CS* Framework intern die Ereignismethoden von LabVIEW.
Der Austausch von Ereignissen kann gepuffert oder ungepuffert geschehen. Im Fall der gepufferten Nachricht wird das Ereignis in einer Warteschlange zwischengespeichert werden, bis der `thread` eines `Prozesses` bereit ist sie zu empfangen. Ungepufferte Ereignisse überschreiben bereits vorhandene Ereignisse, falls der Empfänger nicht in der Lage ist bei deren Eintreffen, sie zu verarbeiten.
Der Ereignismechanismus wird dem Entwickler durch die Klasse `CAEObj` zur Verfügung gestellt. Deren Unterklasse `BaseProcess` verwendet dessen Methoden in einem `thread`, der auf eintreffende Ereignisse reagiert. Dem Entwickler wird auf diese Weise die Möglichkeit geboten, entweder die Vorlagen des `BaseProcess` zu nutzen, oder direkt von der Klasse `CAEObj` zu erben, um seinen eigenen Ereignis-`thread` zu implementieren.

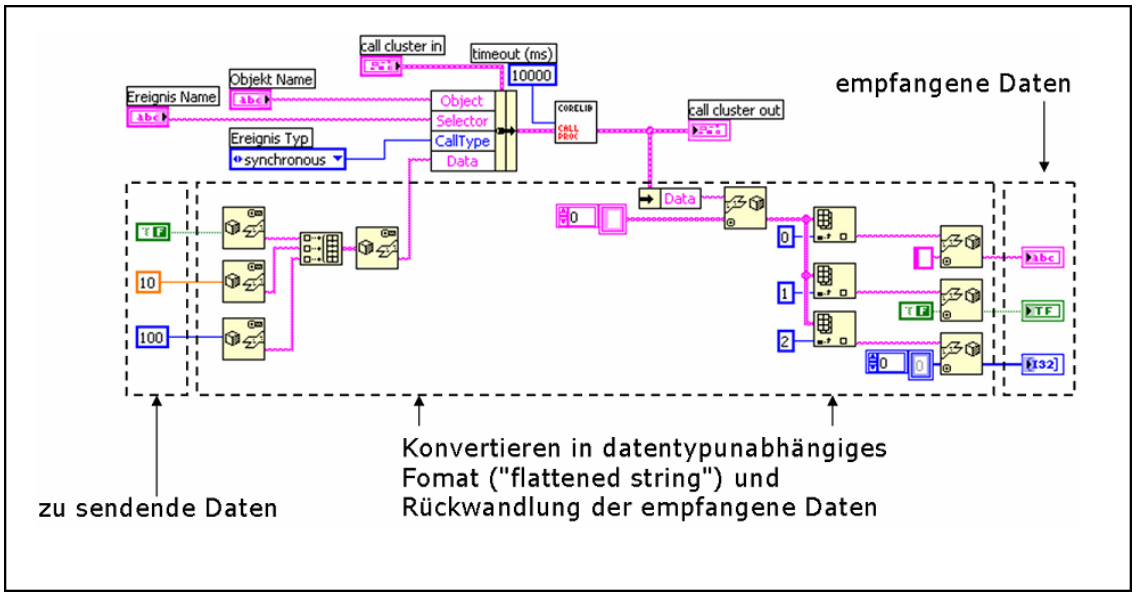


Abbildung 2.14 Beispiel für das Senden eines Ereignisses im *CS* Framework

[49] siehe [Beck2005]

Wie Abbildung 2.14 zeigt müssen lediglich Objektname und Ereignisname zum Versenden eines Ereignisses angegeben werden. Ansonsten ist die Verarbeitung von Ereignissen vollständig durch die Basisklassen `CAEObj` und `BaseProc` gekapselt. Jedes Objekt kann Ereignisse mit Hilfe der Methode `call process` versenden. Allerdings sind nur `Prozesse` in der Lage Ereignisse zu empfangen, solange man nicht einen eigenen Ereignis-`thread` implementiert.
Eine einheitliche Schnittstelle für den Ereignismechanismus ermöglicht, dass Objekte beliebig miteinander kommunizieren können. Hierfür existiert in *CS* Framework das `call cluster`. Um eine beliebige Anzahl unterschiedlicher Datentypen an ein Ereignis anhängen zu können, muss eine Konversion in eine einheitliche Darstellung der Daten durchgeführt werden. Dafür werden alle Datentypen in einen `flattened string` gewandelt und in einem Array zusammengefasst. Dieses wird anschließend ebenfalls in einen `flattened string` konvertiert und in das `call cluster` eingefügt. Auf der Empfängerseite müssen diese Daten ausgepackt und in ihren ursprünglichen Datentyp zurückgewandelt werden. Die Antwort auf ein Ereignis geschieht ebenfalls mit Hilfe des `call cluster`. Daher müssen Daten, die dem Sender mitgeteilt werden sollen, nach demselben Prinzip zunächst vom Empfänger konvertiert und vom Sender wieder zurückgewandelt werden. Abbildung 2.14 zeigt den Aufruf der Methode `call process` und die Konversion der Daten beim Versenden des Ereignisses und Empfangen der Antwort.

2.4.1 Ereignistypen

Nach Empfang eines Ereignisses und der Ausführung der damit verknüpften Methoden, wird eine Antwort versandt. Sender und Empfänger eines Ereignisses können im *CS* Framework identisch sein. Das heißt ein Objekt kann an seinen Ereignis-`thread` Ereignisse versenden.
Wie sich der Empfänger nach Versenden der Nachricht verhält und an wen die Antwort adressiert wird, entscheidet der Typ des Ereignisses. Das *CS* Framework stellt folgende vier Ereignistypen zur Verfügung:

`simple:` Der Sender verschickt ein Ereignis an den Empfänger und erwartet keine Antwort. Der Empfänger versendet ebenfalls keine Antwort.

`synchronous:` Der Sender verschickt ein Ereignis und wartet die Antwort ab. Optional kann ein Zeitintervall angegeben werden, nachdem der Sender das Warten abbricht, falls die Antwort noch nicht eingetroffen ist (timeout case).

`asynchronous:` Der Sender verschickt ein Ereignis und wartet nicht. Die Antwort wird an ein drittes Objekt weitergeleitet, welches vom

Sender zuvor definiert werden muss.

`notifier`: Im Gegensatz zu den anderen drei Typen, handelt es sich hierbei um eine ungepufferte Nachricht. Falls das adressierte Objekt ein Ereignis nicht empfangen kann, überschreiben eingehende Ereignisse das jeweils wartende Ereignis. Es wird also immer das zuletzt eintreffende Ereignis empfangen, alle vorherigen Ereignisse werden dabei überschrieben. Ansonsten verhält sich dieser Typ wie `simple`.

2.4.2 Kommunikation im Netzwerk

Ereignisse können lokal oder im Netzwerk versandt werden. Der Verbindungsaufbau sowie die Kommunikation im Netzwerk mit Hilfe der TCP/IP-Funktionen von LabVIEW sind durch die Netzwerk-Klassen vom Entwickler gekapselt. Zur Adressierung eines Objektes im Netzwerk muss er lediglich folgende Namenskonvention für das Zielobjekt befolgen: `ObjektName@RechnerName`.
Zusätzlich ist noch zu beachten, dass ein Ereignis, welches über eine noch nicht bestehende Verbindung übertragen werden soll, zunächst die Instanzierung eines Objektes der Klasse `QueueClient` auf dem sendeseitigem Rechner anstößt, die den Verbindungsaufbau übernimmt. Wird der Client vor dem Ereignis nicht erzeugt, sorgt das erste zu übertragende Ereignis für dessen Instanzierung und den damit verbundenen Verbindungsaufbau. Das Ereignis selbst wird nicht an den Zielrechner versandt und geht verloren.

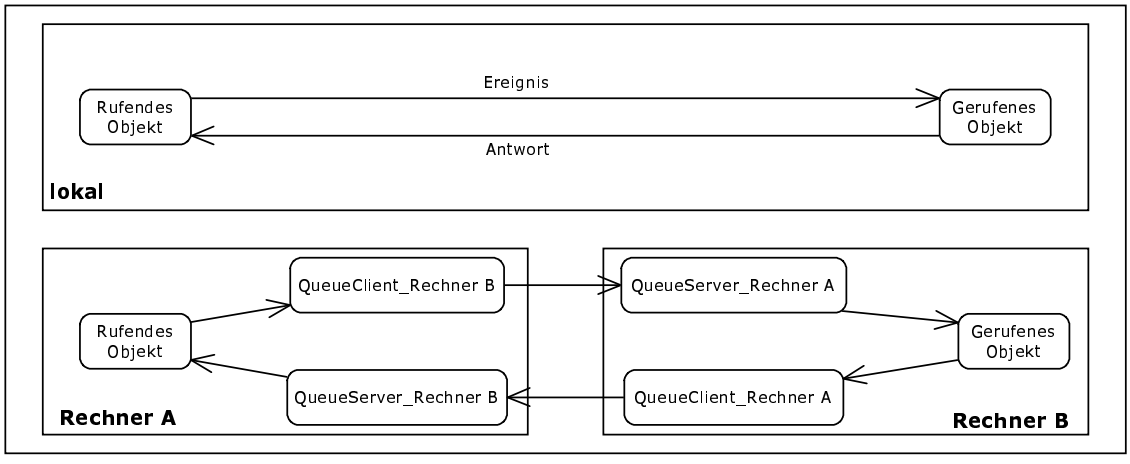


Abbildung 2.15 Lokale (oben) und Rechnerübergreifende (unten) Ereignisse

2.5 Leistungsfähigkeit

Eines der Ziele bei der Entwicklung des *CS* Frameworks war zu überprüfen, wann die Grenze der Skalierbarkeit für ein auf LabVIEW basierendes System erreicht ist. Da es sich bei Kontrollsystemen im Allgemeinen um verteilte Systeme handelt, zieht man als Maß hierfür die maximale Anzahl an Rechnern und die Anzahl an aktiven Objekten pro Rechner heran. Das größte Kontrollsystem, das mit dem *CS* Framework bisher entwickelt wurde, ist beim Experiment PHELIX im Einsatz. Es umfasst an die 20 Rechner, auf denen jeweils zwischen 100 und 200 Objekte laufen.[50]

Der begrenzende Faktor für die Objektanzahl ist der Arbeitsspeicher. Der Grund dafür liegt in der Speicherverwaltung von LabVIEW. Da Objekte einer Klasse denselben Quellcode verwenden, jedoch asynchron darauf zugreifen, würden sich die Objekte gegenseitig blockieren, solange nur eine Instanz Zugriff auf den Code erhalten kann. Daher verwendet man für die `VIs`, welche den entsprechenden mehrfach benötigten Quellcode beinhaltet, die `reentrant`[51]-Option von LabVIEW. Das bedeutet, dass im Speicher für jeden Aufruf eines `VI´s`, eine Kopie im Speicher abgelegt wird. Daher wächst die Anzahl an benötigten Kopien mit der Anzahl an Objekten, die dasselbe `VI` verwenden.[52]

Ein weiteres Leistungskriterium im Zusammenhang mit der Objektanzahl ist die Dauer für die Erzeugung von Objekten. Im Falle von Geräteinstanzen, kommt es häufig vor das mehrere hundert Objekte instanziert werden sollen. Der Vorgang des Instanzierens eines Objekts dauert jedoch einige Sekunden, was dazu führt, dass das Starten eines Kontrollsystemrechners mehrere Minuten dauern kann. Darüber hinaus verhält sich die benötigte Zeit nicht linear zur Anzahl an zu instanzierenden Objekten, sondern wächst schneller als die Objektzahl. Verantwortlich für dieses Verhalten sind die Mechanismen von LabVIEW, die zur Instanzierung verwendet wurden.[53]

Unabhängig von der Skalierbarkeit ist die maximale Ereignisrate bzw. die Dauer eines Ereignisses ein weiteres Maß für die Leistungsfähigkeit des Frameworks. Da hauptsächlich Ereignisse verwendet werden, um Objekte miteinander interagieren zu lassen, ist die Geschwindigkeit der Kommunikation der begrenzende Faktor für die Reaktionszeit eines Kontrollsystems. Die Dauer eines `synchronen` Ereignisses zwischen zwei Objekten auf einem Rechner liegt unter einer Millisekunde. Wird ein Ereignis über das Netzwerk übertragen, verschlechtert sich dieses Ergebnis auf etwa 15 Millisekunden.[54] Ursächlich

[50] siehe [Beck2005]
[51] deutsche Übersetzung: ablaufinvariant
[52] siehe [Beck2003b]
[53] siehe [Beck2003b]
[54] siehe [Beck2005]

für diese Ergebnisse sind im Falle des lokalen Ereignisses die von LabVIEW bereitgestellten Ereignismechanismen, sowie dessen TCP/IP-Funktionen für die Kommunikation im Netzwerk.
Die begrenzenden Faktoren für die Skalierbarkeit und Reaktionszeit werden maßgeblich durch die Leistungsfähigkeit von LabVIEW bestimmt. Aus diesem Grund wird der Ereignismechanismus in der nächsten Version des *CS* Frameworks, die im Laufe dieses Jahres fertig gestellt werden soll, auf DIM[55] basieren und erweitert damit die von LabVIEW verwendeten Ereignismechanismen. Damit soll zum einen der Umgang mit Ereignissen für den Entwickler vereinfacht, zum anderen eine verbesserte Ereignisrate erzielt werden.

[55] Distributed Information Management; siehe Glossar Seite 86

Kapitel 3

Definition

3.1 Motivation

Die Kommunikation innerhalb des *CS* Frameworks geschieht, wie bereits in 2.4 erläutert, über Ereignisse, die im Falle von Geräteklassen die Schnittstelle zu einzelnen Funktionen der Hardware herstellen. Um beispielsweise einen Spannungsgenerator auf einen bestimmten Ausgangspegel hochzufahren, schickt man dem Objekt, das die softwareseitige Repräsentation des Gerätes darstellt, ein Ereignis, welches innerhalb der Klasse die Treibersoftware veranlasst, das entsprechende Kommando an das Gerät zu übermitteln.
Bei der Verwendung von Hardwarekomponenten im Rahmen eines Experiments müssen häufig viele solcher hardwarespezifischen Aktionen ausgeführt werden, um den Experimentaufbau in einen bestimmten Zustand zu versetzten. Oft möchte man auch Aktionen zusammenfassen und später durch ein einzelnes Ereignis deren Abarbeitung anstoßen. Zum Beispiel könnte der Experimentator durch einen einzelnen Knopfdruck die Konfiguration des Experiments ändern, ohne selbst über das nötige Hintergrundwissen bezüglich der Eigenschaften und Einstellungen aller beteiligten Geräte zu verfügen. Die einzelnen Konfigurationen und Aktionen der Hardware, die automatisiert werden sollen, müssen im Vorfeld von einem Hardware-Spezialisten zu Sequenzen zusammengefasst werden, die dem Anwender später als Auswahl untrennbarer Aktionen zur Verfügung stehen. Abbildung 3.1 stellt den eben beschriebenen Anwendungsfall nochmals da.
Neben der sequentiellen Ereignisverarbeitung existiert im Falle von Regelungs-, Test- und Überwachungsvorgängen die Anforderung nach weiteren Kontrollstrukturen wie Wiederholungen und Verzweigung. Hierzu sind Analyse- bzw. Vergleichsmethoden nötig, die auf eine Sequenz folgen, um anschließende Bedingungsabfragen zu ermöglichen.

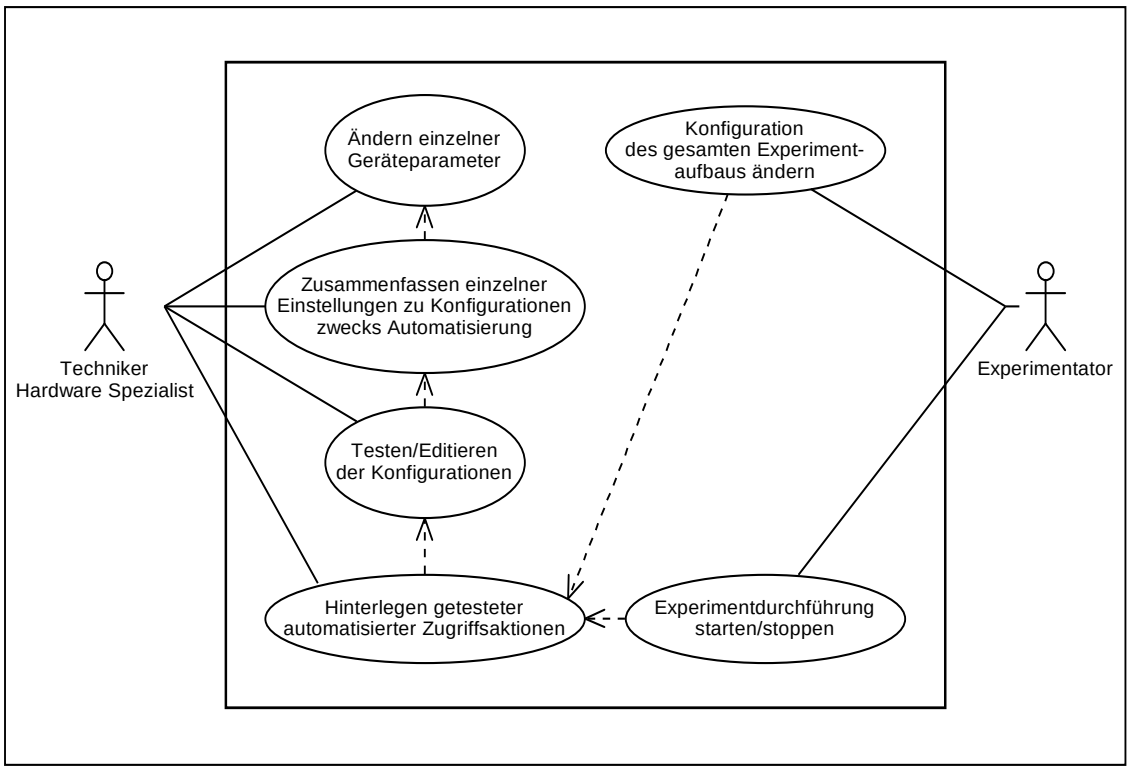


Abbildung 3.1 Aufgaben und Rollen bei der Entwicklung einer Ablaufsteuerung

Bisherige Lösungen, die die eben genannten Aufgaben erledigen, wurden in der Vergangenheit spezifisch für das jeweilige Experiment implementiert. Eine allgemeine Lösung steht bisher nur mit eingeschränkter Funktionalität in Form des `UserGOG`[1] zur Verfügung. Mit dessen Hilfe besteht bereits die Möglichkeit eine Gruppe von Objekten zu instanzieren und anschließend eine beliebige Anzahl von Ereignissen auszulösen. Allerdings fehlen Kontrollstrukturen, wie Entscheidungen oder Wiederholungen, die die Ausführungsreihenfolge der Aktionen beeinflussen. Darüber hinaus ist die Funktionalität in den Code der Klasse `GeneralObjectGUI` integriert und steht nicht in Form einer einzelnen Klasse zur Verfügung. Es ist daher unmöglich durch Ereignisse oder Vererbung Zugriff auf die entsprechenden Methoden zu erlangen. Ebenso ist die Abarbeitung der Instanzierung sowie der Ereignisse statisch im Code festgelegt und verläuft in jedem Fall streng sequentiell. Eine parallele bzw. nebenläufige[2] Verarbeitung von Teilsequenzen, die keine Abhängigkeiten untereinander besitzen, ist nicht möglich.

[1] Name einer Instanz der Klasse `GeneralObjectGUI`; siehe 2.3.3
[2] siehe Glossar Seite 89

3.2 Primäre Anforderungen

Anhand der beschriebenen Anwendungsfälle und des momentanen Entwicklungsstandes des *CS* Frameworks, lassen sich folgende primäre Anforderungen an eine allgemeine Sequenzer-Klassenbibliothek formulieren:

- Einheitliche Schnittstelle zum Sequenzer ermöglicht beliebige *CS*-Klassen mit einem Sequenzer zu verwenden.
- Zusammenfassen von Ereignissen beliebiger `Prozesse` zu Sequenzen, deren Abarbeitung durch eine einzelnes Ereignis angestoßen werden kann.
- Beliebige Verknüpfung von Sequenzen mit Hilfe von Kontrollstrukturen, wie Verzweigungen und Wiederholungen.
- Konfigurierbarkeit der Verarbeitungslogik von Sequenzen.
- Modularität des Sequenzer-Frameworks ermöglicht die Anpassung an spezifische Anwendungen
- Parallelisierung bzw. Nebenläufigkeit von Sequenzen muss möglich sein.
- Reservierungsmechanismus der die wechselseitige Interaktion nebenläufiger Sequenzen verhindert und damit den Zugriff auf die verwendeten `Prozesse` regelt.

3.3 Anforderungsanalyse

Ziel der Entwicklung ist eine modulare Software, welche sich nahtlos in das bereits bestehende *CS* Framework integrieren lässt. Aus Sicht des Frameworks handelt es sich bei Aufgaben wie Ablaufsteuerung bzw. Sequenzierung von Aktionen um Zuständigkeiten der Anwendungsschicht. Sie verwendet die darunterliegende Geräteschicht, um deren Funktionalität in einen für den Anwender sinnvollen Zusammenhang zu bringen[3]. Aus diesem Grund ist es notwendig, entsprechende Schnittstellen vorzusehen, die die Kompatibilität von beliebigen Geräteklassen mit dem Sequenzer ermöglichen. Abbildung 3.2 fasst diesen Grundgedanken zusammen.

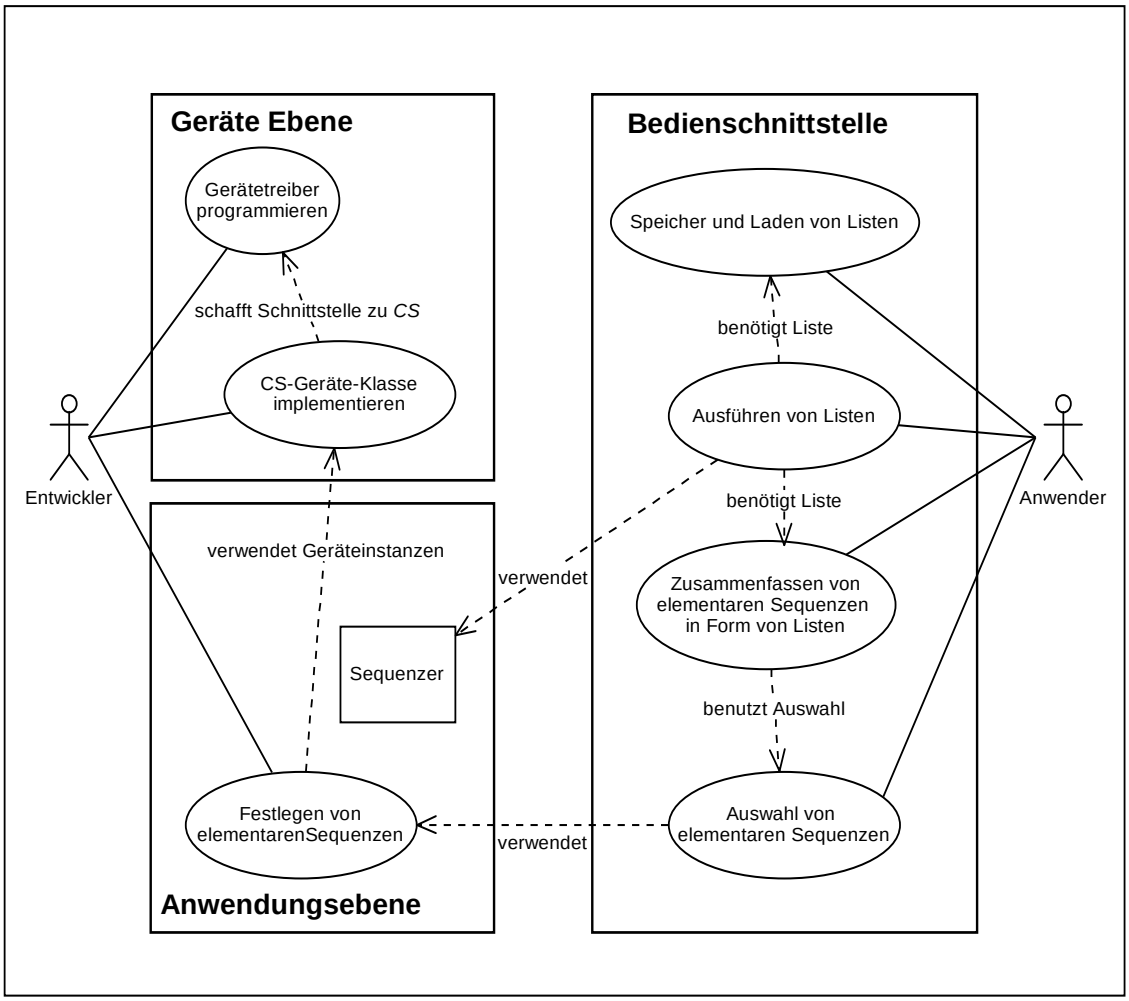


Abbildung 3.2 Integration eines generischen Sequenzers in das *CS* Framework

[3] siehe 2.3.1

3.3.1 Elementare Sequenzen

Die Schnittstelle zwischen dem Sequenzer und den bestehenden *CS*-Klassen ist eine elementare Eigenschaft, die in Form einer Basisklasse vorliegen muss. Eine mögliche Lösung ist die Schnittstelle in eine bereits bestehende Basisklasse wie z.B. dem `BaseProcess` zu implementieren. Dies hätte den Vorteil, dass jede Geräteklasse durch Vererbung Teil eines Sequenzers sein kann. Der Entwickler kann dann parallel zum Implementieren der Gerätefunktionen die Definition elementarer Sequenzen vornehmen. Nachteil ist dass eine elementare Sequenz möglicherweise sich nicht auf ein Gerät beschränkt und daher keine gerätespezifische Eigenschaft darstellt. Darüber hinaus stellt die Änderung einer Basisklasse, die bereits als Oberklasse für etliche Klassen dient, einen Eingriff in das bestehende Framework da und kann zur Instabilität des Frameworks führen.
Eine alternative Lösung besteht in einer neuen Basisklasse, die als abstrakte Klasse nur der Vererbung dient. Sie beinhaltet die Abarbeitung der Sequenzen und definiert hierzu, ähnlich wie andere *CS*-Basisklassen, virtuelle Methoden, die durch den Entwickler in den abgeleiteten Klassen überschrieben werden können. Diese müssen folgende Aufgaben abdecken:

- Definition der Ereignisse und deren Reihenfolge, die eine elementare Sequenz beinhaltet.
- Deklaration der Daten, die im Rahmen der Sequenz ausgetauscht werden sollen.
- Analyse-Methode, die das Ergebnis einer elementaren Sequenz zurückgibt.
- Nebenläufige elementare Sequenzen ermöglichen.
- Reservierung der `Prozesse`, die in nebenläufigen Sequenzen angesprochen werden.

3.3.2 Verknüpfung elementarer Sequenzen

Nachdem der Entwickler seine elementaren Sequenzen implementiert hat, muss die Möglichkeit bestehen diese miteinander zu verknüpfen. Wie aus 3.2 hervorgeht wird diese Aufgabe durch den Anwender übernommen. Das bedeutet die Kombination von Sequenzen muss durch Konfiguration möglich sein. Der Anwender soll nicht gezwungen werden neuen Quellcode schreiben zu müssen. Viel mehr soll er die Möglichkeit haben eine Auswahl an Sequenzen zu treffen und diese mit Kontrollstrukturen zu versehen, um auf die Ausführungsreihenfolge Einfluss nehmen zu können. Eine Gruppierung mehrerer elementarer Sequenzen, die durch Kontrollstrukturen verknüpft sind, wird im Folgenden mit dem Begriff der Liste bezeichnet. Listen sollen mittels einer Bedienoberfläche vom Anwender definiert und ausgeführt werden können. Folgende Aufgaben müssen dafür von der Software erledigt werden:

Verwaltung und Speicherung von Listen, kann durch eine Klasse realisiert werden, die als reiner Datenconainer fungiert. Ein Listenelement beinhaltet die Information über die Sequenz und deren Ausführung. Die Elemente werden in einer doppelt verketteten Liste verwaltet. Dies ermöglicht Verzweigungen innerhalb der Liste. Hierfür benötigt jedes Listenelement noch zusätzliche Steuerinformationen zur Navigation zwischen den Listenelementen.

Interpretation und Ausführung der Listen übernimmt der eigentliche Sequenzer. Er benötigt Zugriff auf eine Liste, um deren Elemente auszulesen. Mit Hilfe der Steuerinformationen eines Listenelementes wird das jeweils nächste Element bestimmt, abhängig von dem Ergebnis des aktuellen Listenelements.
Um die Ausführung einer Liste für den Entwickler zu visualisieren, muss die Möglichkeit bestehen, dem Sequenzer mitteilen zu können, ob die Ausführung ohne Unterbrechung von Anfang bis Ende der Liste oder Elementweise erfolgen soll. D.h. die Abarbeitung einer Liste muss zum einen von Benutzerinteraktionen beeinflussbar sein, so dass ein Pausieren oder Abbrechen durch den Benutzer möglich ist. Zum anderen ist eine Elementweise Verarbeitung einer Liste notwendig, die dem Entwickler zur Fehlersuche dient.

Archivierung der Daten, die während der Ausführung einer Liste anfallen. Das bedeutet, dass nach der Abarbeitung einer Liste die Ergebnisdaten der einzelnen elementaren Sequenzen im Zusammenhang mit den Listenin-

formationen in eine Datei geschrieben werden. Diese kann zu einem späteren Zeitpunkt wieder geladen werden, und dient der Protokollierung.

Schachtelung von Listen kann durch einen hierarchischen Aufbau von Sequenzern erreicht werden. Das bedeutet, ein Sequenzer besitzt die Möglichkeit mittels der Listenanweisungen eine anderen Sequenzer zu starten, der seinerseits wieder eine Liste ausführt, in der wiederum Anweisung zum Start weiterer Sequenzer enthalten sein können.

Editieren von Listen wird durch eine grafische Benutzeroberfläche ermöglicht, die sowohl den Zugriff auf die Liste als auch den Sequenzer benötigt. Dadurch kann der Anwender zunächst seine Liste anlegen und deren Elemente konfigurieren. Anschließend aktiviert er die Ausführung der Liste durch den Sequenzer, um seine Liste zu debuggen.
Das GUI dient zunächst als allgemeines Werkzeug für den Entwickler für die Erstellung von Listen. Es muss, gemäß des Schichtenmodells[4] des Frameworks, getrennt von Sequenzer und Liste implementiert werden, damit es jederzeit durch eine neue Oberfläche ersetzt werden kann, die an die Anforderungen einer spezifischen Anwendung angepasst ist. Sowohl der Sequenzer als auch die Liste können unverändert bleiben, nur die Schnittstelle zum Benutzer wird ausgetauscht.

[4] siehe 2.3.1

3.3.3 Nebenläufigkeit

Sequenzen Damit die Abarbeitung der Sequenzen per Ereignis angestoßen werden kann, muss die neue Basisklasse von `BaseProcess` oder `CAEObj` erben, um Ereignisse empfangen zu können. In diesem Zusammenhang ist auch die Frage von nebenläufigen Sequenzen von Bedeutung. Sobald ein Objekt mehrere Ereignisse zur Sequenzbearbeitung empfängt, kann es diese nur sequentiell verarbeiten, solange die Verarbeitung innerhalb des Ereignis-`threads` durchgeführt wird. Das bedeutet die Sequenzen blockieren sich gegenseitg. Daher muss die Verarbeitung in separate `threads` verlagert werden, die von dem Ereignis-`thread` aus gestartet werden.
Durch die Nebenläufigkeit entsteht die Möglichkeit, dass sich Sequenzen, die dieselben `Prozesse` verwenden, gegenseitig stören. Der `Prozess` führt lediglich die Ereignisse aus, die er empfängt, unterscheidet dabei allerdings nicht zu welcher Sequenz das jeweilige Ereignis gehört. Der Zugriff auf die Ereignisse eines `Prozesses` muss daher durch einen Reservierungsmechanismus, der von den aufrufenden Objekten verwendet wird, geregelt werden.

Listen Sobald die Nebenläufigkeit von Sequenzen durch eine Basisklasse möglich ist, können mehrere Listen nebenläufig ausgeführt werden ohne sich gegenseitig zu blockieren. Der Reservierungsmechanismus, der von der Basisklasse verwendet wird, regelt in diesem Fall den Zugriff auf gemeinsam genutzte `Prozesse`.
Um dem Anwender eine Möglichkeit zu geben, Einfluss auf die Reservierungsvorgänge zu nehmen, kann optional eine Reservierung der Objekte, die der Sequenzausführung dienen, angestrebt werden. Da sowohl der Zugriff auf die `Prozesse` als auch das Anstoßen einer Sequenz per Ereignis erfolgt, kann in beide Fällen derselbe Reservierungsmechanismus verwendet werden.

3.3.4 Reservierungsmechanismus

Der Begriff der Reservierung bezieht sich im Zusammenhang mit dem Sequenzer - Framework auf den Ereignis-Mechanismus eines `Prozesses`. Diese Funktionalität ist in den *CS*-Basisklassen bisher noch nicht vorhanden, soll aber in der nächsten Version des *CS* Frameworks als Eigenschaft der Basisklasse `CAEObj` enthalten sein. Um trotzdem die Reservierung von Ereignissen eines `Prozesses` zu ermöglichen, muss dies in einer Form implementiert werden, die es zulässt, zu einem späteren Zeitpunkt ersetzt werden zu können, ohne allzu großen Eingriff in den Quellcode anderer Sequenzerklassen vornehmen zu müssen. Die einfachste Möglichkeit dies zu gewährleisten, besteht in der Kapselung durch eine eigene Klassen, deren Instanzen stellvertretend für das Zielobjekt die Ereignisse empfangen und diese weiterleiten, falls der Sender über die Reservierung verfügt.
Das stellvertretende Objekt muss hierfür folgende Anforderungen erfüllen:

- Ereignisse des Zielobjekts müssen ebenfalls dem stellvertretenden Objekt bekannt sein, um sie empfangen zu können.
- Weiterleiten der empfangene Ereignisse an das Zielobjekt.
- Die Reservierung muss über einen eindeutigen Schlüssel erfolgen.
- Speicherung des Reservierungszustandes.
- Verarbeitung von Reservierungsanfragen in Abhängigkeit des aktuellen Reservierungszustands.
- Speicherung und Benachrichtigung anfragender Objekte bei Änderung des Reservierungszustandes.

Kapitel 4

Entwurf

4.1 Übersicht über die Klassenbibliothek

4.1.1 Begriffsdefinitionen

Der folgende Abschnitt enthält Begriffe, die im Zusammenhang mit der Sequenzer-Klassenbibliothek spezielle Bedeutung haben. Aus diesem Grund wird an dieser Stelle eine Definition der entsprechenden Bezeichnungen gegeben, um Unklarheiten zu vermeiden. Begriffe im Text, die sich auf die folgenden Definitionen beziehen, sind in `Schreibschrift` dargestellt.

Begriff	**Definition**
Testklasse	Beliebige Unterklasse der Klasse `CSDuT`. Ein `Testobjekt` ist eine Instanz einer beliebigen Testklasse.
elementare Sequenz	Abfolge von Ereignissen, deren Abarbeitung innerhalb eines `threads` der Klasse `CSDuT` durchgeführt wird.
Liste	Instanz der Klasse `CSList`, die in einer mehrfach verketteten Liste eine beliebige Anzahl an Listenelementen verwaltet, an welche Daten angehängt werden können. Bezieht sich dieser Begriff nicht auf die vorangehende Definition, ist anstatt der Listeninstanz das Listenattribut, welches die doppelt verkettete Liste speichert, gemeint.
Sequenzer	Instanz der Klasse `CSSeqExec`, die die Abarbeitung elementarer Sequenzen anstößt.
Reservierung	Mechanismus, der den exklusiven Zugriff auf einen `Prozess` per Ereignisse ermöglicht. Die `Reservierung` eines `Prozess` übernimmt ein Objekt der Klasse `CSProxy`.
Stellvertreter	Instanz der Klasse `CSProxy`.

Tabelle 4.1 Begriffe des Sequenzer-Frameworks und deren Definition

4.1.2 Klassenbeschreibung

Im Folgenden wird zunächst ein Überblick über die Klassen, die das Sequenzer-Framework bereitstellt, gegeben. Im Anschluss werden die einzelnen Klassen im Detail vorgestellt.
Abbildung 4.1 zeigt die Vererbungsbeziehungen zwischen den Klassen des Sequenzer-Framework und den *CS*-Basisklassen. Ein detailliertes Klassendiagramm, das Attribute und Methoden der Sequenzer-Klassen beinhaltet, befindet sich in Anhang B auf Seite 92.

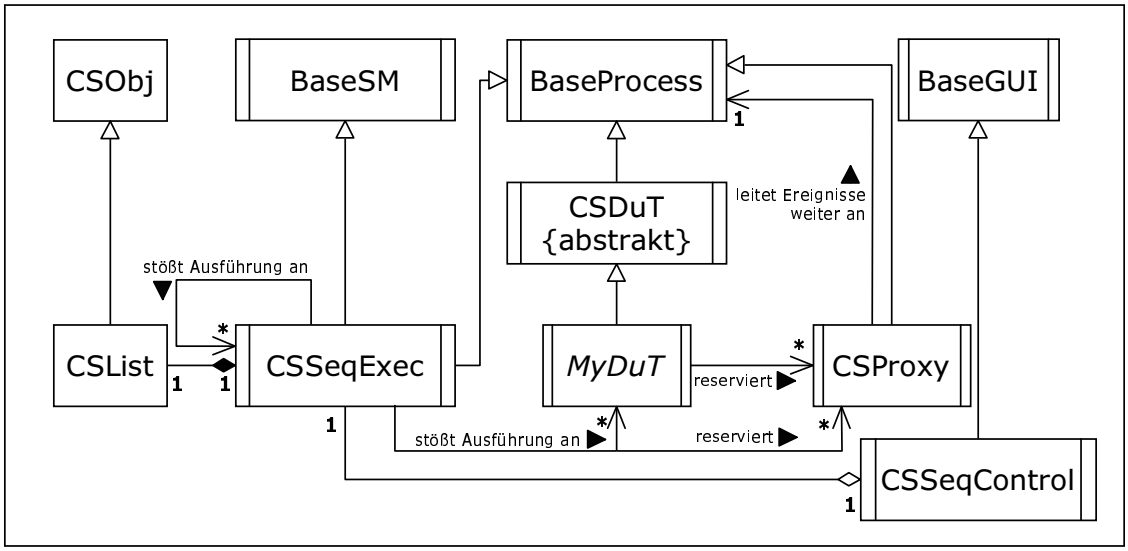


Abbildung 4.1 Vererbungsbeziehungen zwischen der Sequenzer-Klassenbibliothek und dem *CS* Framework

CSDuT dient als Basisklasse des Sequenzer-Framework und stellt die Schnittstelle zwischen einem `Sequenzer` und einer beliebigen Anzahl an `Prozessen` her. Die Verarbeitung `elementarer Sequenzen` sowie die `Reservierung` der genutzten `Prozesse` ist durch die Basisklasse gekapselt. Durch Vererbung von der Klasse `CSDuT` können konkrete Testklassen implementiert werden, in denen `elementaren Sequenzen` definiert werden. Die Abarbeitung einer solchen `elementaren Sequenz` kann dann von einem beliebigen Objekt aus per Ereignis angestoßen werden.
Der Klassenname DuT[1] entstand in der Definitionsphase, die in Zusammenarbeit mit dem BELAB[2] durchgeführt wurde. Deren Anwendung des Sequenzer-Framework besteht in einem Teststand[3] für Elektronik-Komponenten. Im

[1] Device under Test
[2] Beschleuniger-Elektronik-Labor an der GSI; Mehr Informationen zum BELAB findet man unter [BELAB1].
[3] siehe Kapitel 5

Rahmen eines Teststandes bezeichnet der Ausdruck DuT den Prüfling. In der Software beinhaltet die gleichnamige Klasse und deren Unterklassen die Ereignisabfolgen und deren Ausführung, die verschiedene Testreihen für einen Prüflingstyp realisieren.

CSProxy stellt den Reservierungsmechanismus für die ereignisgesteuerte Kommunikation bereit. Eine Instanz dieser Klasse fungiert als `Stellvertreter` eines `Prozesses`.
Ein `Stellvertreter` leitet Ereignisse, die er empfängt an den `Prozess` weiter, für welchen er stellvertretende Funktion einnimmt, falls der Sender sich im Vorfeld dessen `Reservierung` gesichert hat oder keine `Reservierung` vorliegt.

CSList ist eine passive Klasse, deren Instanzen doppelt verkettete Listen[4] verwalten, die im Kontext des Sequenzer-Frameworks Ausführungs- und Steueranweisungen für den `Sequenzer` enthalten.

CSSeqExec stellt die eigentliche Sequenzerklasse dar. Durch Vererbung von der Klasse `BaseSM` verfügt sie über eine Zustandsmaschine, die die Listenverarbeitung übernimmt. Zusätzlich handelt es sich bei einem `Sequenzer` um einen `Prozesse`, der über eine Ereignisbehandlungsmethode verfügt. Daher kann per Ereignis Einfluß auf die Listenverarbeitung genommen werden. Jeder `Sequenzer` erzeugt eine `Liste`. Somit können über den `Sequenzer` per Ereignis auch Listenoperationen aufgerufen werden.

CSSeqControl implementiert ein GUI zum Erstellen und Testen von Listen. Neben Bedienelementen, die das Editieren und Verwalten von Listen ermöglichen, erlaubt die Oberfläche die manuelle Steuerung eines `Sequenzers`, wie z.B. dem Einzelschritt-Modus für die Listenverarbeitung.

[4] siehe Glossar auf Seite 87

4.1.3 Anwendungsbeispiel

Um deutlich zu machen in welcher Form die Klassen bzw. deren Instanzen miteinander interagieren, zeigt Abbildung 4.2 das Zusammenspiel der Klasseninstanzen in den einzelnen Schichten des *CS* Frameworks für eine einfache Beispielanwendung.

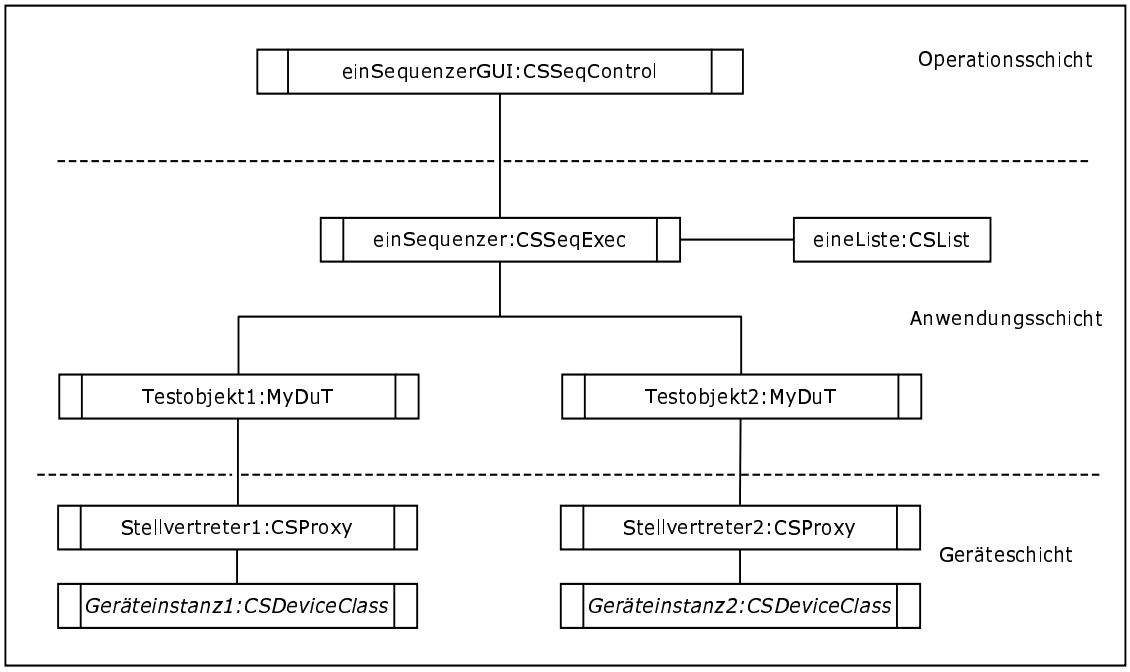


Abbildung 4.2 Objektbeziehungen in einer einfachen Beispielanwendung

Der `Sequenzer` als zentrale Instanz der Anwendung hat Zugriff auf die `Liste`, die die Ausführungs- und Steueranweisungen verwaltet. Sie liefert die Informationen in welcher Reihenfolge `elementare Sequenzen` von den `Testobjekten` ausgeführt werden sollen.
Nachdem der `Sequenzer` ein Listenelement ausgelesen hat, versendet er ein entsprechendes Ereignis an ein `Testobjekt`. Dieses reserviert die `Prozesse`, die in der auszuführenden Sequenzen verwendet werden, um im Anschluss die entsprechenden Ereignisse an die `Prozesse` zu versenden. Danach werden die `Reservierungen` aufgehoben, das Ergebnis der Sequenz mit den im `Testobjekt` implementierten Analyse-Methoden generiert und an den aufrufenden `Sequenzer` zurückgesendet. Anhand des Ergebnisses und der Steuerinformation des Listenelements bestimmt er das nächste Element, holt dies aus der Liste und stößt die Ausführung der darin enthaltenen `elementaren Sequenz` an.

4.2 Die Reservierungsklasse - CSProxy

4.2.1 Stellvertreterfunktion

Wie bereits in den Anforderungen[5] beschrieben wird für das Sequenzer-Framework ein eigener Reservierungsmechanismus durch die Klasse `CSProxy` implementiert, die später leicht aus dem Framework entfernt werden kann. Ersetzt werden soll die Klasse mit der nächsten Version des *CS* Frameworks, in der die *CS*-Basisklasse `CAEObj` die Reservierung der Ereignisbehandlung eines `Prozesses` ermöglichen wird.
`Stellvertreter` empfangen die Ereignisse ihres `Prozesses` und prüfen ob sie diese weiterreichen dürfen. Hierzu benötigt ein `Stellvertreter` zunächst die Namen aller Ereignisse seine `Prozesses`, damit er in der Lage ist dessen Ereignisse empfangen zu können. Abbildung 4.3 beschreibt den Austausch von Ereignissen zwischen `Stellvertreter` und dem dazugehörigen `Prozess` beginnend mit der Instanzierung des `Stellvertreters`.

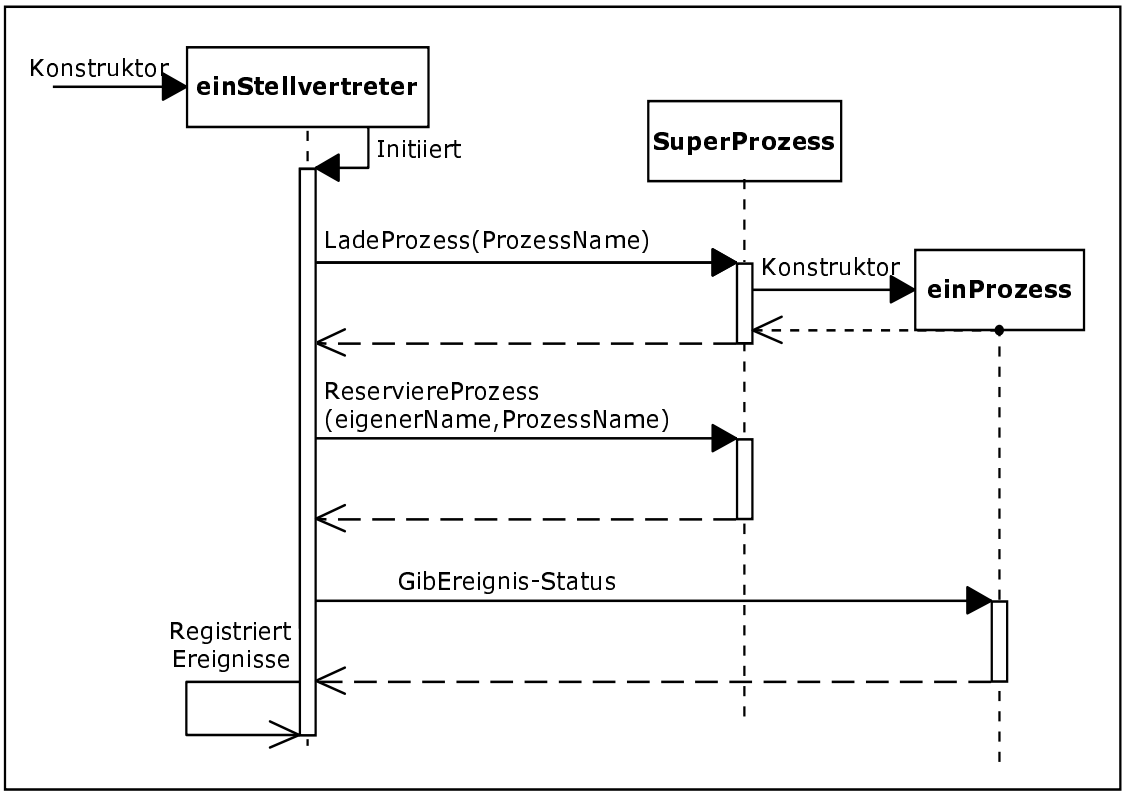


Abbildung 4.3 Initialisierungsvorgang eines Stellvertreters

[5] siehe Kapitel 3.3.4

4.2.2 Reservierungsmechanismus

Die `Reservierung` erfolgt per Ereignis. Der Sender verschickt eine Reservierungsanfrage an den entsprechenden `Stellvertreter`. Diese beinhaltet einen eindeutigen Schlüssel, der von `Stellvertreter` und Sender gespeichert wird. Will der Sender Ereignisse an den reservierten `Prozess` verschicken, adressiert er zunächst den `Stellvertreter` und hängt den Schlüssel an die Daten an. Der `Stellvertreter` vergleicht seinen Schlüssel mit dem Gesendeten. Bei Übereinstimmung wird das Ereignis an den `Prozess` weitergeleitet und die dazugehörige Antwort an den Sender zurückgegeben. Sind die Schlüssel nicht identisch bzw. es wurde keine Schlüssel mitgeschickt, teilt der `Stellvertreter` dem Sender mit, dass bereits eine `Reservierung` vorliegt und das Ereignis nicht zum `Prozess` durchgereicht werden konnte. Liegt keine `Reservierung` vor, werden alle Ereignisse ohne Prüfung an den `Prozess` weitergeleitet.

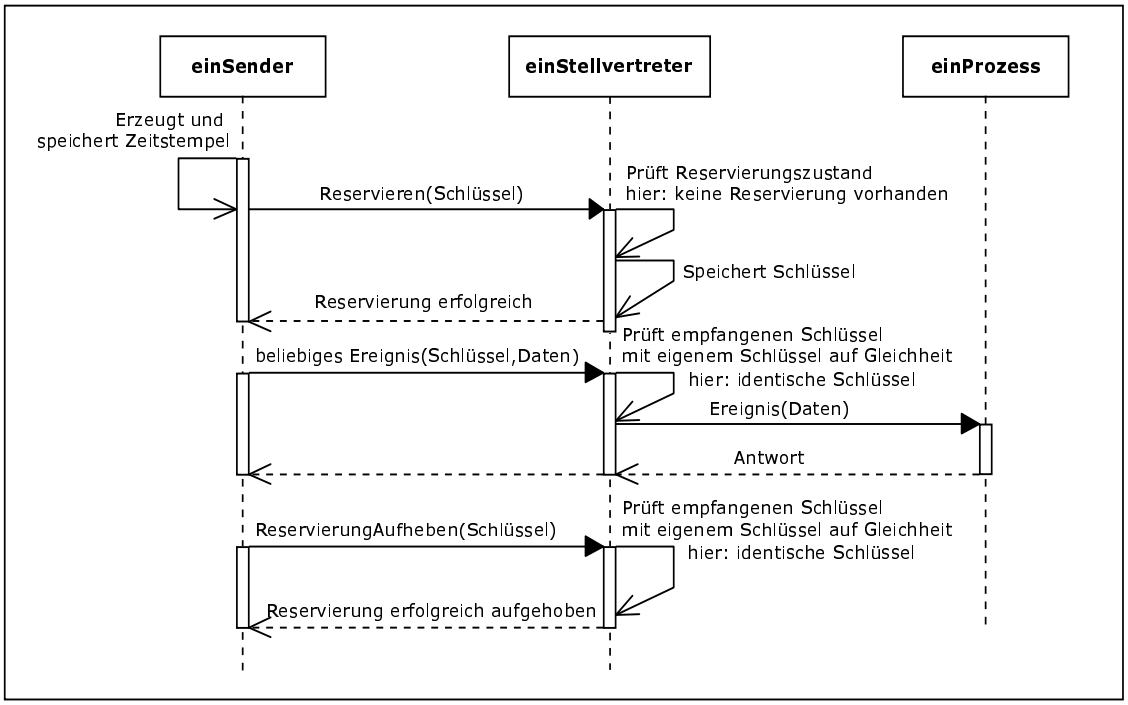


Abbildung 4.4 erfolgreiche Reservierung

Der Schlüssel, der eine `Reservierung` identifiziert, muss eindeutig sein. Aus diesem Grund besteht ein Schlüssel aus dem Namen des Senders und einem Zeitstempel, der vom Sender generiert wird. Diese beiden Daten werden vom `Stellvertreter` auf Gleichheit geprüft und entscheiden, ob Ereignisse des Senders an den reservierten `Prozess` weitergeleitet werden.
Entsteht die Situation, dass trotz bereits vorhandener `Reservierung` ein Objekt entweder eine Reservierungsanfrage startet oder ohne `Reservierung` ver-

sucht Ereignisse über den Stellvertreter an einen Prozess zu versenden, wird in beiden Fällen dem sendenden Objekt mitgeteilt, dass eine Reservierung bzw. die Durchstellung des Ereignisses aufgrund einer bestehenden Reservierung nicht möglich ist.

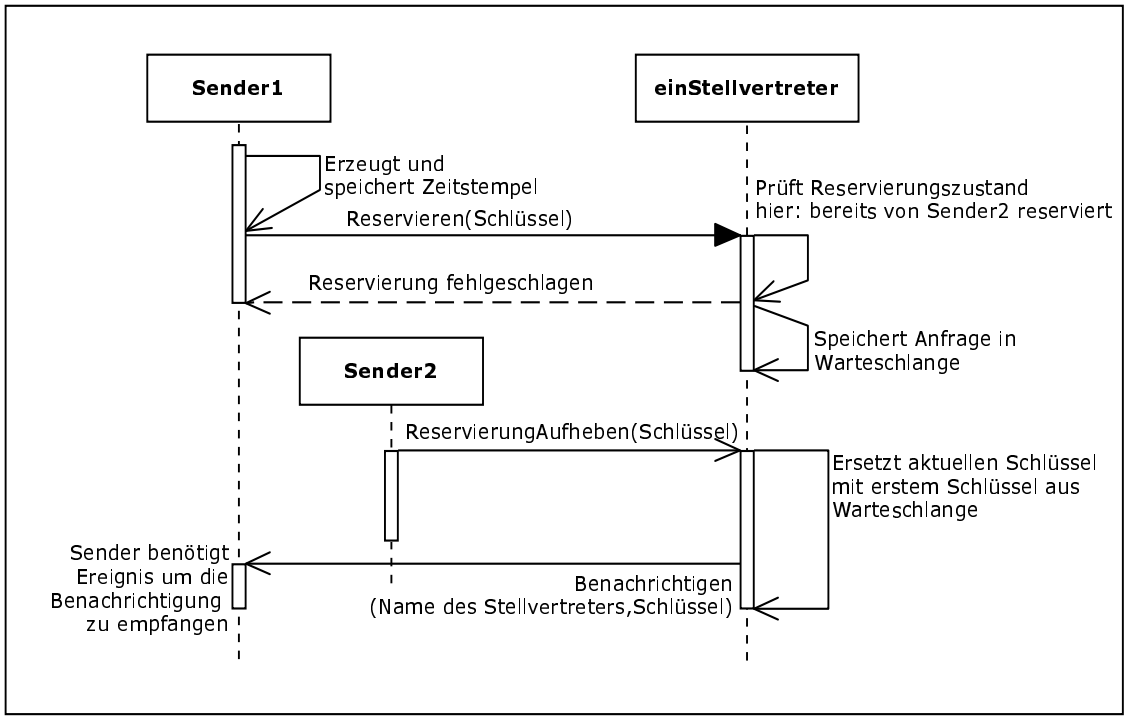


Abbildung 4.5 Benachrichtigung bei erfolgloser Reservierungsanfrage

Sobald Objekte asynchron einen Stellvertreter beanspruchen, erhält immer nur ein Objekt die gewünschte Reservierung. Für die restlichen Objekte besteht entweder die Möglichkeit entsprechend der erfolglosen Anfrage ihre Verarbeitung fortzusetzen, oder in regelmäßigen Zeitintervallen die Anfrage zu wiederholen, bis sich der Reservierungszustand ändert und der Stellvertreter die Reservierung an das nächste Objekt vergeben kann.
Als Alternative zu dem eben beschriebenen Polling[6] kann der Stellvertreter aktiv werden, indem er das wartende Objekte benachrichtigt, sobald sich dessen Reservierungszustand ändert. Durch das Polling würde nur unnötig oft abgefragt, ob sich der Zustand geändert hat und in Abhängigkeit der Periodizität verschlechtert sich die Reaktionszeit auf eine Statusänderung im ungünstigsten Fall um eine Abfrageperiode. Darüber hinaus ist die Kommunikation im *CS* Framework ereignisgesteuert, und ermöglicht damit die Nachteile, die durch Polling entstehen, zu negieren.

[6] In der Informatik bezeichnet der Begriff die automatische, periodische Statusabfrage von Hard- oder Software; siehe http://de.wikipedia.org/wiki/Polling_%28Informatik%29

Daher speichert sich ein Stellvertreter trotz bestehender Reservierung alle Reservierungsanfragen und getrennt davon die reservierungslosen Zugriffsversuche. Wird dann die Reservierung aufgehoben, rückt die nächste Reservierungsanfrage nach und das dazugehörige Objekt wird per Ereignis benachrichtigt (siehe Abbildung 4.5). Sind keine Reservierungen mehr vorhanden werden alle Objekte benachrichtigt, die seit der letzten Reservierung versucht haben, Ereignisse ohne Reservierung an den Stellvertreter zu versenden (siehe Abbildung 4.6).

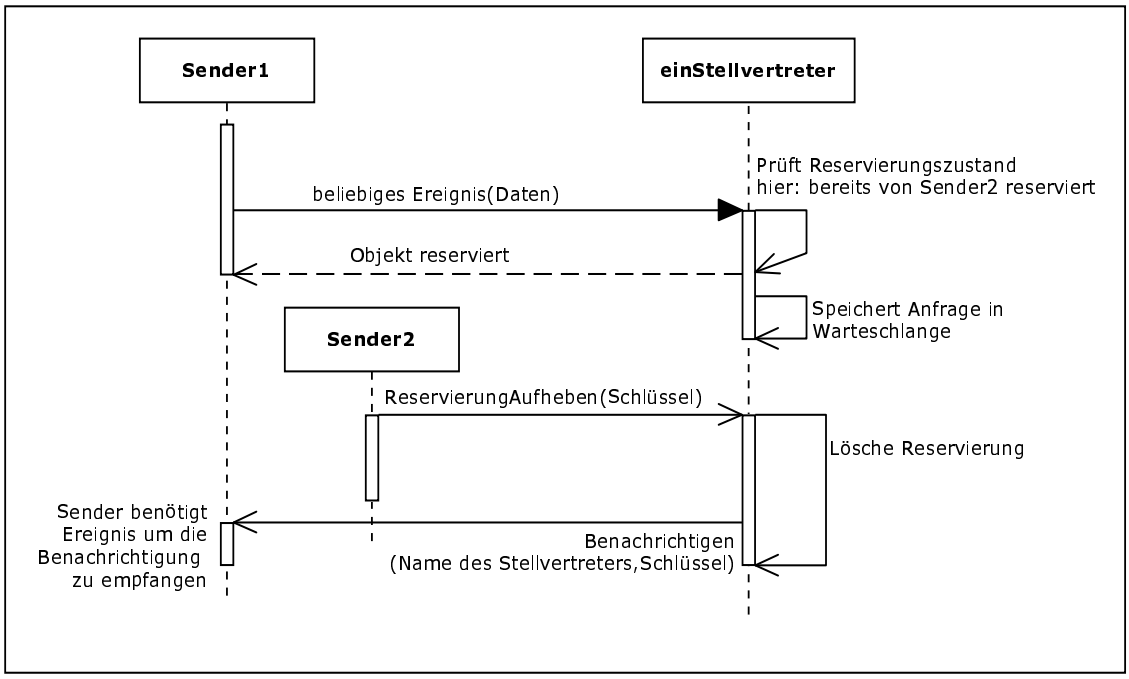


Abbildung 4.6 Benachrichtigung bei reservierungslosem Ereignis

Für die Benachrichtigung durch den Stellvertreter benötigt ein Objekt allerdings die Fähigkeit, Ereignisse zu empfangen und es muss die entsprechenden Ereignisse des Stellvertreters kennen, um sie empfangen können.
Die Aufhebung einer Reservierung ist in allen Fällen durch das reservierende Objekt unerlässlich, ansonsten bleibt der Stellvertreter für die Dauer seiner Existenz für alle Zugriffe gesperrt.
Im Rahmen des Sequenzer-Frameworks verläuft die Reservierung unsichtbar für den Entwickler. Dieser braucht sich keine Gedanken über die Generierung und Verwaltung der Schlüssel oder dem Versenden der Reservierungsanfragen zu machen. Er entscheidet lediglich, ob die Reservierung stattfinden soll oder nicht, indem er entweder den Stellvertreter adressiert oder direkt den Prozess.

4.3 Die Basisklasse - CSDuT

4.3.1 Verwendung durch den Entwickler

Als abstrakte Klasse, dient die neue Basisklasse, lediglich als Oberklasse, die dem Entwickler durch Vererbung generische Basisfunktionen zur Verfügung stellt. Dabei handelt es sich im Falle der Klasse `CSDuT` um die nebenläufige Abarbeitung `elementarer Sequenzen` und die Verwaltung der benötigten `Reservierungen`. Die Klasse definiert aus Sicht des Entwicklers vier neue virtuelle Methoden, die er in jeder seiner abgeleiteten Klassen überschreiben muss.
Um dem Entwickler diese Arbeitsschritte zu erleichtern, werden im Zuge der Vererbung die `VIs`, die die virtuellen Methoden überladen, bereits in das neue Klassenverzeichnis kopiert. Direkt nach der Vererbung beinhalten diese `VIs` lediglich funktionslose Vorlagen, die der Entwickler noch mit Code füllen muss, der die generischen Verarbeitungsvorschriften spezifiziert. Bei den vier virtuellen Methoden handelt es sich um folgende `VIs`:

`ProcSequence:` Dieses `VI` dient der Definition der `elementaren Sequenzen`, die ein `Testobjekt` dem `Sequenzer` zur Verfügung stellt. Dies beinhaltet die Ereignisnamen und deren Reihenfolge.
Neben dem Namen eines Ereignisses benötigt man, wie in 2.4 beschrieben, zum Versenden eines Ereignisse noch den Namen des Zielobjekts. Würde man diese im Rahmen einer `elementaren Sequenz` fest in den Quellcode schreiben, könnte man zur Laufzeit die entsprechende Sequenz nur auf bestimmte Instanzen anwenden, obwohl die Ereignisse von allen Instanzen einer Klasse verstanden werden.
Aus diesem Grunde gibt der Entwickler anstatt dem Objektnamen einen Platzhalter, der für alle Instanzen einer Klasse steht. Erst im Rahmen einer Liste kann der Anwender definieren welche konkreten Instanzen die Platzhalter ersetzen soll.

`ProcStimulate:` Im Rahmen der Ausführung einer `elementaren Sequenz` muss es möglich sein, Daten an die Ereignisse anhängen zu können. Diese Daten können erst während der Ausführung der Sequenz anfallen, aus einer Datei ausgelesen werden oder durch die Objektattribute bereitgestellt werden. Möglicherweise muss ein Ereignis innerhalb einer oder mehrerer Sequenzen mit unterschiedlichen Daten versandt werden. Diese Aufgaben erledigt der Entwickler durch die Methode `ProcStimulate.vi`.

ProcRespond: Beinhaltet die Antwort auf ein Ereignis Daten, die für die weitere Ausführung der elementaren Sequenz bzw. für die Ergebnisfindung relevant sind, kann der Entwickler in der Methode ProcRespond.vi bestimmen wie die empfangenen Daten weiterverarbeitet werden.

ProcCompare: Nach der Durchführung einer elementaren Sequenz wird ein Ergebnis an die aufrufende Instanz zurückgegeben. Die Funktionen, die der Ergebnisfindung dienen und die Daten auf die sie operieren, werden in der Methode ProcCompare.vi vom Entwickler implementiert.

4.3.2 Klasseninterne Verarbeitung

Die Abarbeitung einer elementaren Sequenz wird per Ereignis ausgelöst. Im Ereignis-thread des adressierten Testobjekts wird zunächst überprüft, ob die angeforderte Sequenz dem Testobjekt bekannt ist und alle Ersetzungen der Platzhalter in der Sequenz, die vom aufrufenden Objekt mitgegeben wurden, vorhanden sind. Anschließend werden die nötigen Reservierungsanforderungen an die Stellvertreter der Prozesse geschickt und deren Antworten in den Attributen abgelegt.
Die sequentielle Ausführung der Ereignisse einer elementaren Sequenz wird nicht in dem Ereignis-thread des Testobjekts durchgeführt. Stattdessen wird für jede elementare Sequenz ein neuer thread gestartet, der das Versenden der Ereignisse an die Prozesse übernimmt. Dadurch kann ein Testobjekt mehrere elementare Sequenzen nebenläufig abarbeiten[7]. Dies wird benötigt, sobald zwei nebenläufig arbeitende Sequenzer in ihren Listen auf das gleiche Testobjekt zugreifen. Wenn zusätzlich dieselben Prozesse in den Sequenzen verwendet werden, regelt der Reservierungsmechanismus, dass die Zugriffe auf die Prozesse sich nicht überschneiden.
Für das Starten eines threads besitzt die Klasse CSDuT ein VIT[8]. Wird ein Testobjekt per Ereignis aufgefordert eine elementare Sequenz auszuführen, erzeugt das Objekt eine Instanz des VIT´s (siehe Abbildung 4.7). Über die Attribute werden die nötigen Daten an den thread übergegeben. Konnten im Ereignis-thread nicht alle Prozesse reserviert werden, wartet der thread für ein definiertes Zeitinterval auf die Benachrichtigung durch die Stellvertreter auf die fehlenden Reservierungen. Hierfür besitzt jeder thread den Zugriff auf einen LabVIEW-Notifier. Falls alle fehlenden Reservierungen im Ereignis-thread eingetroffen sind, aktiviert dieser den wartenden thread, um

[7] siehe 3.3.3
[8] Virtual Instrument Template; siehe Glossar Seite 90

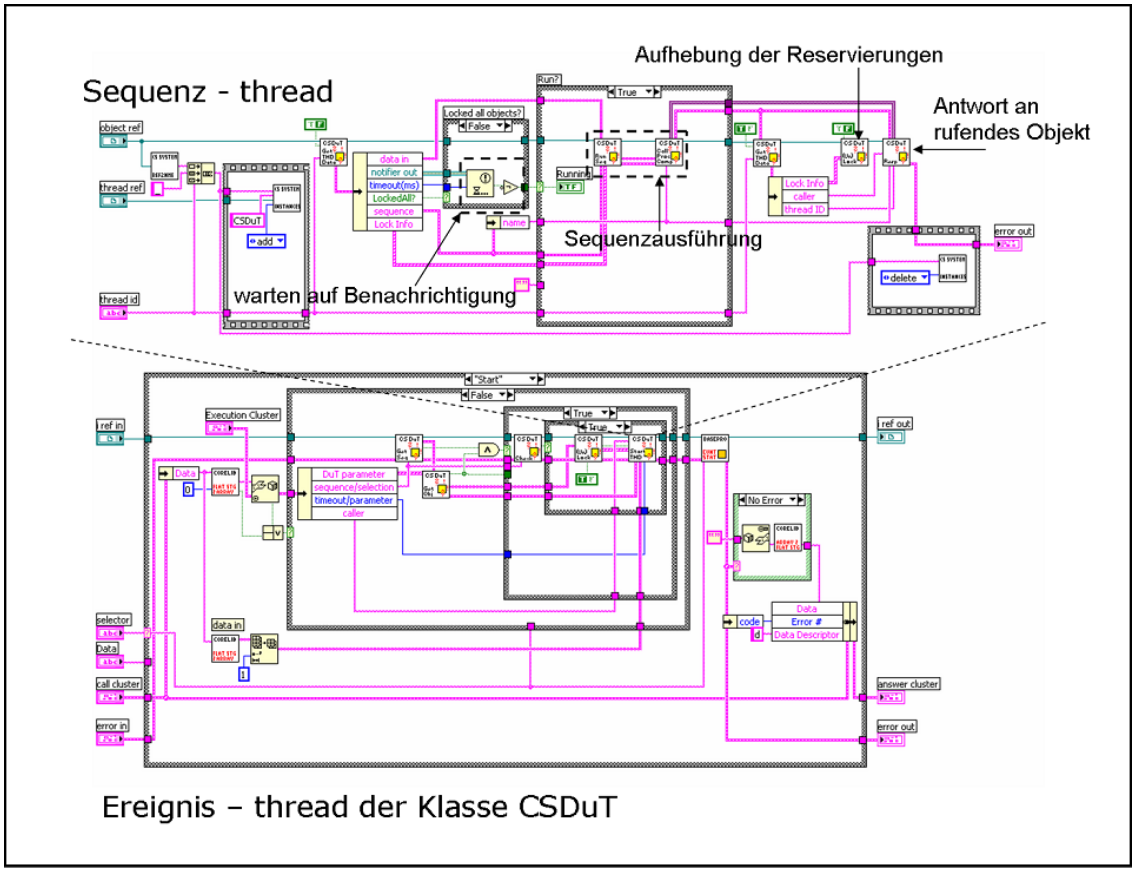


Abbildung 4.7 Starten einer Instanz des VIT´s der Klasse CSDuT

seine elementare Sequenz auszuführen. Wartet der thread für das definierte Zeitinterval erfolglos auf die Benachrichtigung, bricht er das Warten ab und gibt eine Fehlermeldung an das aufrufende Objekt zurück.
Konnten alle Reservierungen getätigt werden, führt der thread seine elementare Sequenz unter Verwendungen der Methoden ProcStimulate.vi und ProcRespond.vi aus. Im Anschluss wird die Methode ProcCompare.vi aufgerufen und in Abhängigkeit der Testklasse und des Namens der Sequenz ein Ergebnis bestimmt, welches an das aufrufende Objekt geschickt wird.
In jedem Fall muss ein thread bevor er beendet ist, alle Reservierungen, die mit ihm verbunden sind aufheben, damit die entsprechenden Stellvertreter von anderen Testobjekten verwendet werden können.

4.3.3 Reservierungsszenarien

Durch die nebenläufige Verarbeitung elementarer Sequenzen entstehen verschiedene Szenarien, auf die der Reservierungsmechanismus reagieren muss, ohne dass sich Sequenzen gegenseitig blockieren oder es zu wechselseitigen Zugriffen auf die gleichen Prozesse kommt.
Abbildung 4.8 zeigt den Fall, dass zwei threads von einem Testobjekt gestartet werden, die unterschiedliche Prozesse reservieren. In dieser Situation ist ein Reservierung nicht notwendig, solange keine weitere Instanz auf dieselben Prozesse zugreift.

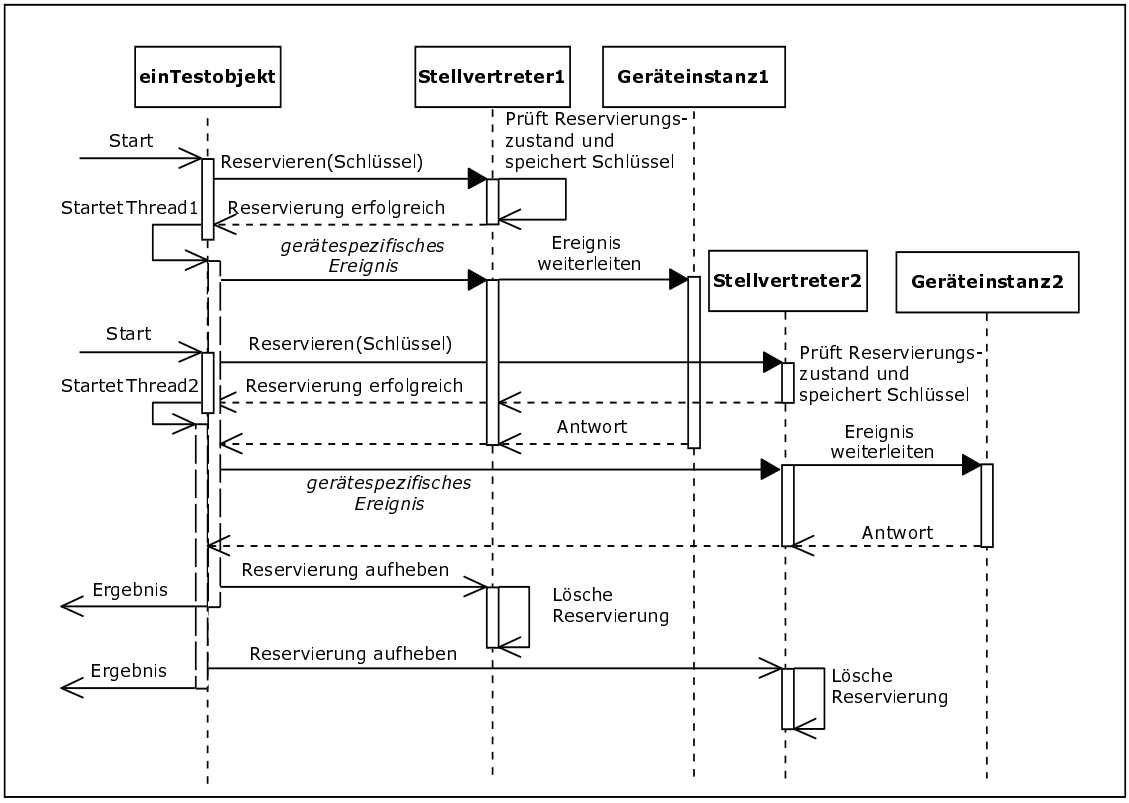


Abbildung 4.8 Nebenläufige Verarbeitung zweier elementarer Sequenzen ohne gemeinsam genutzte Prozesse

Überschneiden sich die Mengen der Prozesse mehrerer nebenläufiger Sequenzen, ist deren Reservierung notwendig. Immer nur ein thread erhält die Reservierung für alle benötigten Prozesse. Sobald dessen Verarbeitung beendet ist und er seine Reservierungen aufgehoben hat, benachrichtigen die Stellvertreter die wartenden threads.
Das obere Diagramm in Abbildung 4.9 zeigt den Ablauf bei erfolgreicher Benachrichtigung durch den Stellvertreter. Nachdem der erste thread seine Reservierungen für die Prozesse aufgehoben hat, sendet der Stellvertreter

ein Ereignis an das Testobjekt. Dieses merkt sich alle von ihm gestarteten threads und kennt den Reservierungsstatus der einzelnen Prozesse. Bei jeder Benachrichtigung wird geprüft welcher thread nun alle benötigten Reservierungen erhalten hat. Dieser wird mittels seines Notifier aktiviert und beginnt die Abarbeitung der Sequenz.
Damit durch das Warten mehrerer threads auf die Reservierung nicht ein Deadlock[9] entsteht, existiert die Möglichkeit eine Zeitinterval zu definieren nach dessen Ablauf der thread das Warten abbricht und seine Reservierungen aufhebt, um Deadlocks zu vermeiden und anderen threads den Zugriff auf die entsprechenden Prozesse zu ermöglichen. Diese Situation beschreibt das untere Diagramm in Abbildung 4.9.

[9] deutsche Übersetzung: Tödliche Verklemmung; Bezeichnung für den Zustand, wenn mindestens zwei Prozesse auf die Zuteilung einer Ressource, die bereits von dem jeweils anderen Prozess reserviert wurden, warten.

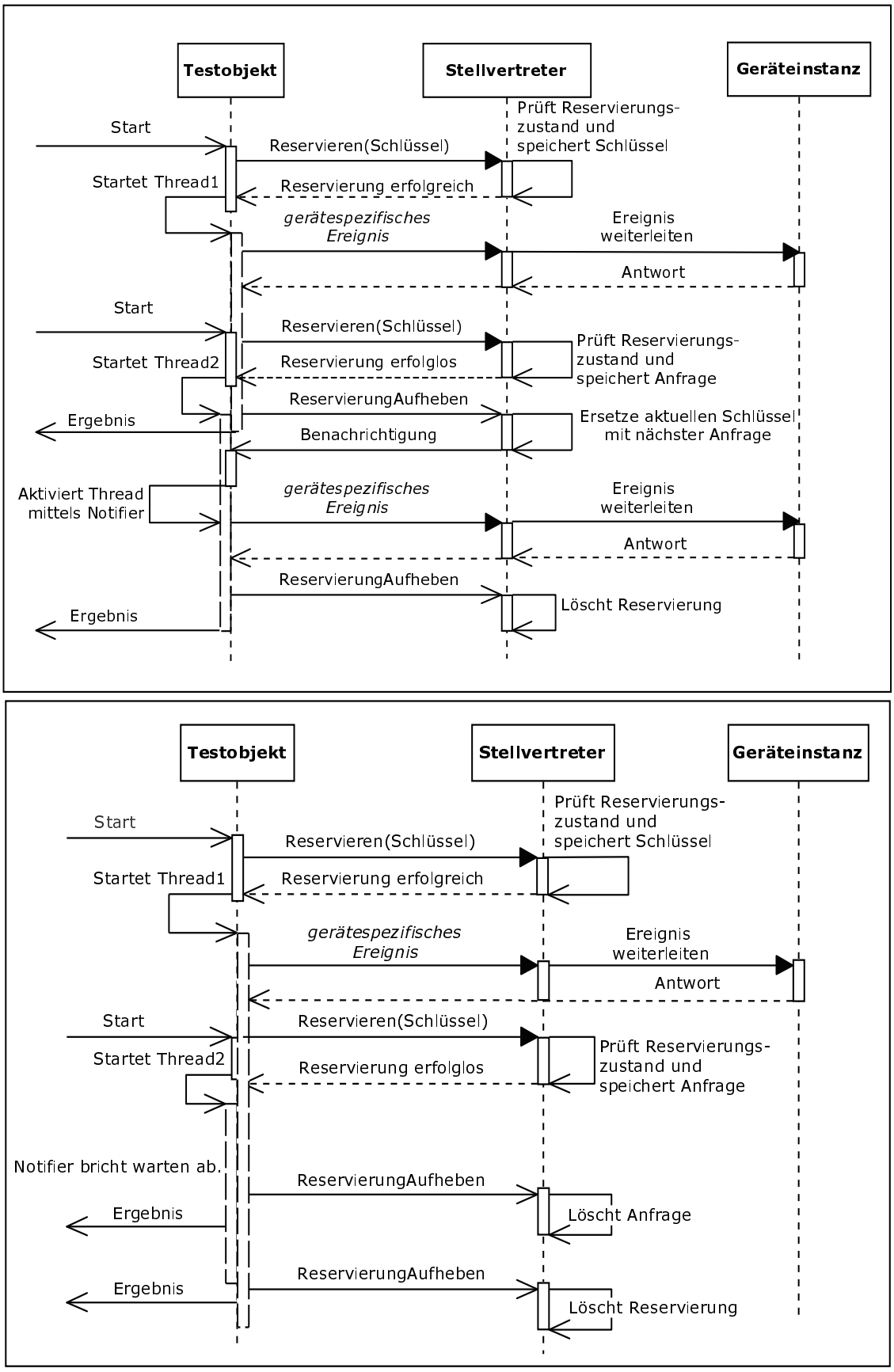


Abbildung 4.9 Nebenläufige Verarbeitung zweier elementarer Sequenzen mit gemeinsam genutzten Prozessen
oben: Die Reservierung wird rechtzeitig aufgehoben.
unten: Der zweite thread bricht das Warten auf die Reservierung ab.

4.4 Der listengesteuerte Sequenzer

4.4.1 Die Listenklasse - CSList

Wie in Abbildung 4.1 dargestellt erbt die Klasse `CSList` nur von der *CS*-Basisklasse `CSObj` und beinhaltet daher keinen aktiven Code. Es handelt sich um eine passive Datenklasse, die eine doppelt verkettete Liste verwaltet. Ein Element besitzt bis zu vier Verweise auf andere Listenelemente und verfügt über ein Zeichenkettenfeld zur Speicherung von Daten. Ähnlich der Datenkonversion, die beim Versenden von *CS*-Ereignissen verwendet wird, können mit Hilfe der `flatten to string`-Funktion von LabVIEW eine variable Anzahl verschiedener Datentypen an ein beliebiges Listenelement angehängt werden.

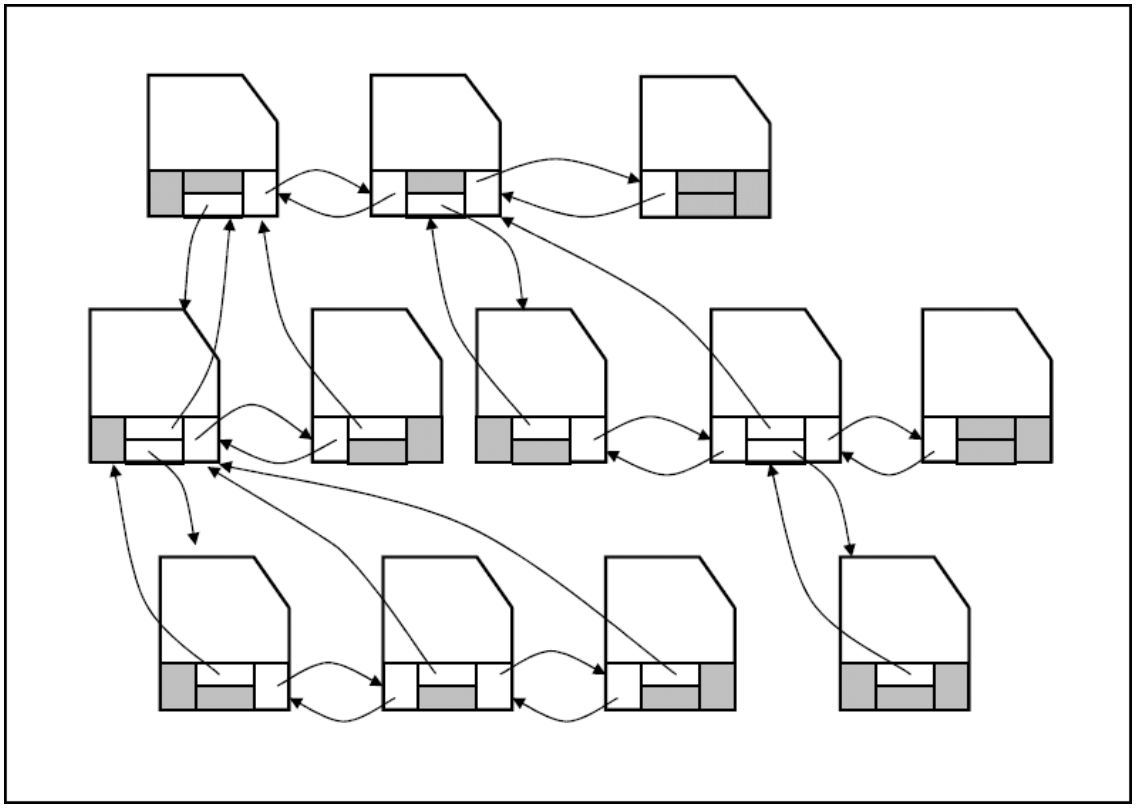


Abbildung 4.10 Beispiel einer Listenstruktur

Mit den Verweisen der Listenelemente lässt sich eine hierarchische Listenstruktur realisieren. Dabei sind Elemente, die sich in einer Hierarchie-Ebene befinden immer doppelt miteinander verknüpft. Das bedeutet, ein Element verweist jeweils auf das Folgende, welches wiederum einen Verweis auf das Vorangehende besitzt. Jedes Listenelement kann zusätzlich auf eine eigene Unterebene verweisen. Alle Elemente der Unterebene verweisen ihrerseits auf

das übergeordnete Element. Dadurch sind beliebige Verzweigungen innerhalb einer Liste möglich. Abbildung 4.10 zeigt ein Beispiel für die Struktur einer Liste.
Eine Liste verwaltet ihre Elemente in einem Feld und verfügt über ein Attribut, das ähnlich eines Zeigers durch Indizierung des Feldes auf das aktuelle Listenelement verweist. Um die Position des Listenzeigers zu manipulieren, stellt die Klasse Methoden bereit, die anhand der Verweise eines Listenelements den Index des neuen Elements bestimmen und das Zeigerattribut aktualisieren. Weitere Listenoperationen ermöglichen das Lesen und Schreiben von Listeninhalten aus bzw. in Dateien zur persistenten Speicherung.
Im Kontext des Sequenzer-Frameworks übernimmt eine Liste die Speicherung zusammenhängender elementarer Sequenzen und deren Steuerungsinformationen, die dem Sequenzer als Ausführungsanweisungen dienen (siehe Tabelle 4.2).

4.4.2 Die Sequenzerklasse - CSSeqExec

Ein Sequenzer besteht aus einer Liste und einer Zustandsmaschine. Die Liste liefert die Ausführungsanweisungen, die von der Zustandsmaschine verarbeitet werden.

Die Beziehung zur Listenklasse wurde als Komposition[10] implementiert (siehe Abbildung 4.1). Mit der Instanzierung eines Sequenzers wird automatisch eine Liste erzeugt[11], deren Objektreferenz vom Sequenzer verwaltet wird. Der Aufruf von Listenoperationen durch den Sequenzer erfolgt daher nicht per Ereignis, sondern direkt über die Listenreferenz. Verglichen mit dem ereignisgesteuerten Aufruf ist diese Programmiertechnik performanter, bietet allerdings nicht die Möglichkeit zur Laufzeit Einfluss auf die Reihenfolge der aufgerufenen Methoden auszuüben, da der Aufruf statisch im Quellcode implementiert ist.
Sequenzer und Liste bilden durch ihrer Klassenbeziehung für andere Objekte eine Einheit. Sollen Sequenzer- oder Listenmethoden per Ereignis aufgerufen werden, muss in beiden Fällen der Sequenzer adressiert werden, dessen Ereignisbehandlung den Zugriff auf Methoden beider Klassen beinhaltet. Somit ist sichergestellt, dass die Manipulation von Listenattributen durch dritte Objekte nie ohne vorausgehende Prüfung durch den dazugehörigen Sequenzer

[10] In der Objektorientierung bezeichnet die Komposition eine spezielle Form der Assoziation. Es handelt sich um eine starke Aggregation, in der Objekte der aggregierten Klasse nicht ohne das erzeugende Objekt existieren können.
[11] siehe 2.2.3

stattfindet. Zum Beispiel verhindert der Sequenzer in seiner Ereignisbehandlungsmethode, dass während der Ausführung einer Liste außer dem Sequenzer selbst kein anderes Objekt auf die Listenelemente zugreifen kann.

Die Verarbeitung einer Liste erfolgt durch die Zustandsmaschine des Sequenzers (siehe Abbildung 4.11). Per Ereignis erhält der Sequenzer den Befehl die Bearbeitung seiner Liste zu starten.

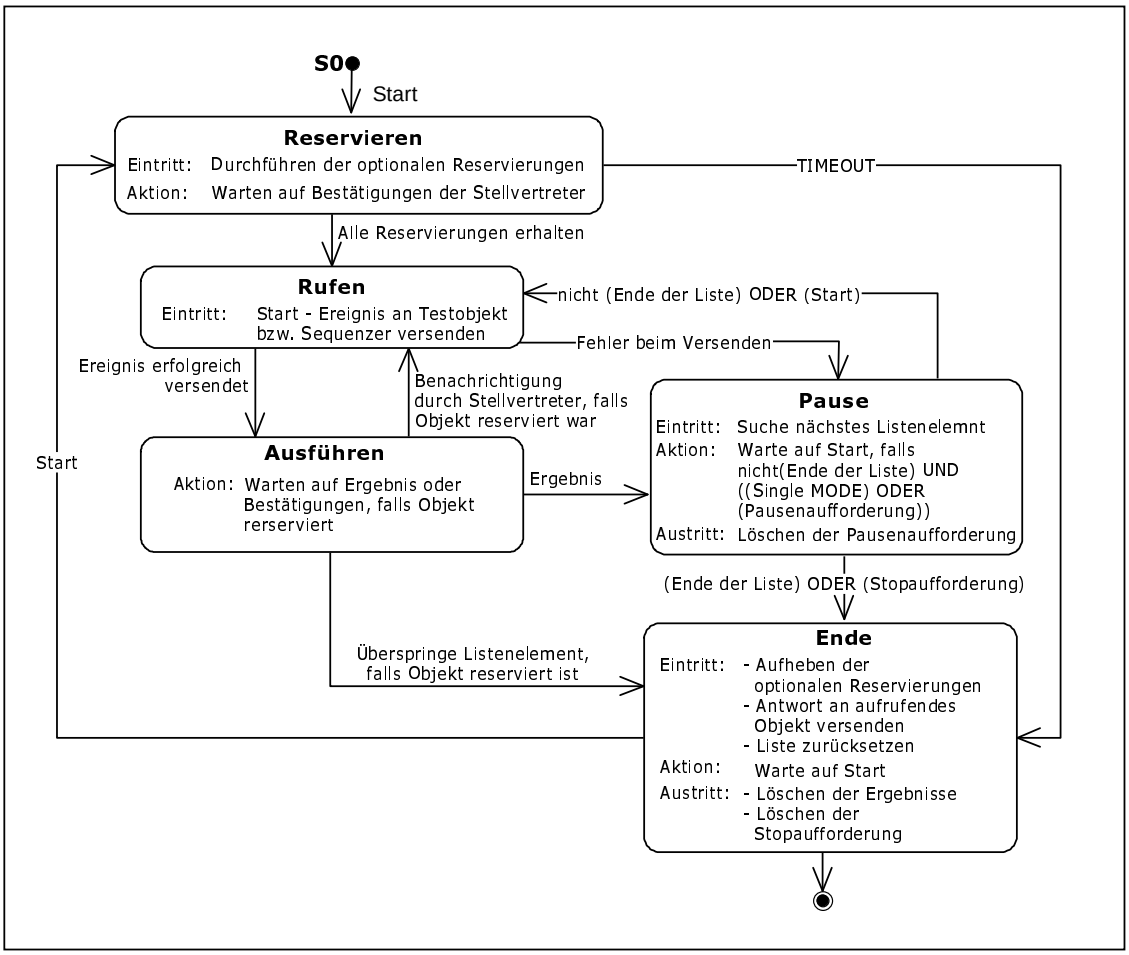


Abbildung 4.11 Zustandsmaschine der Klasse CSSeqExec

Dazu wechselt er zunächst in den Zustand **RESERVIEREN**. Ähnlich der Reservierung der Prozesse durch die Testobjekte können zu Begin der Listenausführung, die angesprochenen Testobjekte durch Stellvertreter reserviert werden (siehe Abbildung 4.12). Falls das angesprochene Objekt kein Stellvertreter ist wird die Reservierung für den folgenden Ablauf ignoriert. Dies gilt sowohl für die Reservierung durch einen Sequenzer als auch für die Testobjekte.
Der Zeiger der Liste verweist bei Ausführungsbeginn auf das erste Listenelement, das die Namen der Stellvertreter und das Zeitinterval beinhaltet,

welches der `Sequenzer` abwartet, um alle notwendigen `Reservierungen` von Seiten der Stellvertreter zugesichert zu bekommen. Als Eintrittsaktion des Zustands **RESERVIEREN** werden die Daten des ersten Listenelements ausgelesen, die Reservierungsanfragen versandt und der Listenzeiger auf das nächste Element gerückt. Konnten alle `Reservierungen` innerhalb des Zeitintervals dem `Sequenzer` zugeteilt werden, beginnt dieser mit der eigentlichen Listenabarbeitung, indem er in den Zustand **RUFEN** wechselt. Anderenfalls wechselt er direkt in den Zustand **ENDE**.

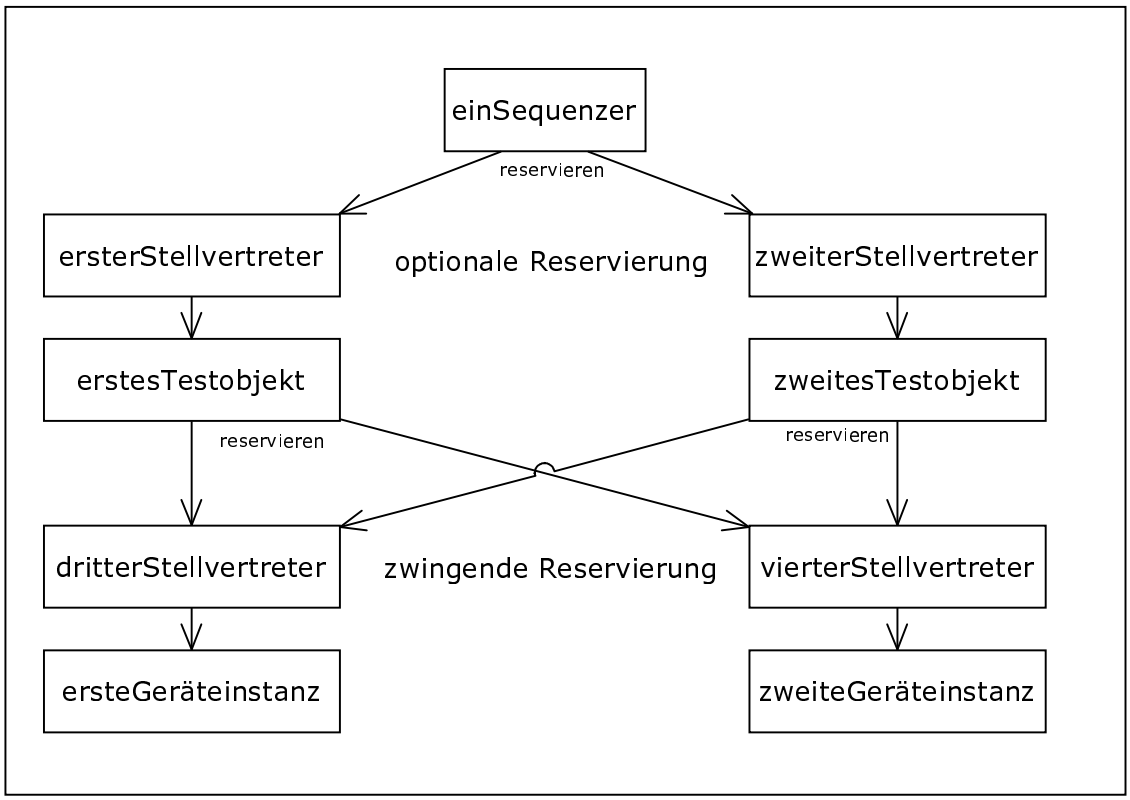


Abbildung 4.12 optionale und zwingende Reservierung

Im Zustand **RUFEN** werden die Ausführungsanweisungen aus dem Datenfeld des aktuellen Listenelements ausgelesen und ein entsprechendes Ereignis versandt. Zur Ausführung eines Listenelements beinhalten dessen Datenfeld folgende Informationen:

gerufenes Objekt	Name des `Testobjekt` bzw. dessen Stellvertreters, falls optional eine `Reservierung` zu Begin der Listenausführung durchgeführt wurde.
Ausführungsparameter	
rufendes Objekt	Name des aufrufenden `Sequenzers`.
Sequenz	Bezeichnung der `elementaren Sequenz`, deren Ausführung vom `Sequenzer` aus angestoßen wird.
Timeout	Zeitinterval in Millisekunden, welches der `thread` eines `Testobjekts` auf die `Reservierung` der `Prozesse` warten soll. timeout=-1 -> `Testobjekt-thread` wartet bis er die Benachrichtigung erhält.
Platzhalter	Ersetzungen für die Platzhalter einer `elementaren Sequenz`.
Steuerparameter	
Ignoriere Ergebnis	Falls aktiviert, wird unabhängig von dem Ergebnis einer `elementaren Sequenz` das nächste Listenelement bestimmt.
Aktion bei erfolgreicher Sequenz	Entscheidet über welchen Verweis das nächste Listenelement bestimmt wird, falls die Sequenz erfolgreich war oder `Ignoriere Ergebnis` aktiviert ist. Mögliche Aktionen sind: *Next*: Der Listenzeiger rückt auf das nächste Element der gleichen Listenebene. *Down*: Der Listenzeiger rückt auf das erste Element der darunter gelegenen Listenebene. *Up&Next*: Der Listenzeiger rückt auf das nächste Element der darüber gelegenen Listenebene.
Aktion bei fehlgeschlagener Sequenz	Entscheidet über welchen Verweis das nächste Listenelement bestimmt wird, falls die Sequenz fehlgeschlagen ist. Dieser Parameter wird ignoriert, falls `Ignoriere Ergebnis` aktiviert ist. Mögliche Aktionen siehe oben.

Tabelle 4.2 Ausführungs- und Steuerparameter eines Listenelements

Der `Sequenzer` adressiert das zu rufende Objekt und hängt die entsprechenden Ausführungsparameter an das Ereignis an. Falls das Ereignis fehlerfrei an das Zielobjekt übertragen werden konnte, wechselt der `Sequenzer` in den Zustand `RUN` und wartet auf die Antwort des gerufenen Objekts. Falls das `Testobjekt` bereits durch einen anderen `Sequenzer` reserviert wurde, verharrt

der Sequenzer solange in dem Zustand **AUSFÜHREN**, bis er die Benachrichtigung durch den Stellvertreter empfängt, dass sich der Reservierungszustand des Stellvertreters geändert hat. Dann ändert er seinen Zustand zu **RUFEN** und versendet erneut die Ausführungsanfrage an das zu rufende Objekt.
Nach Empfang der Antwort des Testobjekts nimmt der Sequenzer den Zustand **PAUSE** ein. Die Antwort beinhaltet das Ergebnis der elementaren Sequenz und die dazugehörige Daten. Zusammen mit den Steuerparametern des Listenelements bestimmt der Sequenzer das nächste Listenelement. Existiert an der verwiesenen Stelle der Liste kein Element, wechselt der Sequenzer in Abhängigkeit des Betriebsmodi direkt in den Zustand **ENDE** oder beginnt erneut mit der Listenausführung.
Der Sequenzer verfügt über verschiedene Betriebsmodi, um die Ausführung der Liste zu beeinflussen:

Attribut	**Datentyp**	**Erklärung**
Schleifen-Modus	enumeration (FOR/WHILE)	*iterativ*(=FOR): wiederholt Liste n-mal n<=0 bei Ausführungsbeginn bedeutet endlose Ausführung der Liste *konditional*(=WHILE):n>=0 -> Abbruch, falls Listenergebnis erfolglos ist n<0 -> Abbruch, falls Listenergebnis erfolgreich ist
Schleifen-Parameter (n)	i32	siehe Schleifen-Modus
Ausführungsmodus	Boolean	*auto*(=TRUE): Listenelemente werden ohne Pause abgearbeitet. *single*(=FALSE): Schrittweise Bearbeitung der Listenelemente. Sequenzer verharrt im Zustand **PAUSE** nach jedem Listenelement.
Stop	Boolean	TRUE -> Sequenzer wechselt nach dem aktuellen Listenelement in den Zustand **ENDE**. FALSE -> keine Auswirkung.
Pause	Boolean	TRUE -> Sequenzer wartet nach dem aktuellen Listenelement im Zustand **PAUSE**. FALSE -> keine Auswirkung.

Tabelle 4.3 Attribute der Klasse CSSeqExec zur Listenverarbeitung

Sobald der Sequenzer die Ausführung der Liste beendet hat, wechselt er in den Zustand **ENDE**. Dort werden alle Reservierungen aufgehoben und anschließend das Ergebnis des zu letzt ausgeführten Listenelements als Listenergebnis an das aufrufende Objekt zurückgegeben. Als Letztes wird die aktuelle Listenposition wieder auf den Index 0 zurückgesetzt.

4.5 Anwendungsszenarien

4.5.1 Geschachtelte Sequenzer

Um eine beliebige Schachtelung von Wiederholungs- und Verzweigungsstrukturen zu ermöglichen, muss eine Liste die Ausführung anderer Listen beinhalten, die ihrerseits wiederum auf weitere Listen verweisen können. Daher kann ein Sequenzer nicht nur `Testobjekte` per Ereignis anstoßen kann, sondern ist ebenfalls in der Lage andere `Sequenzer` zu steuern.

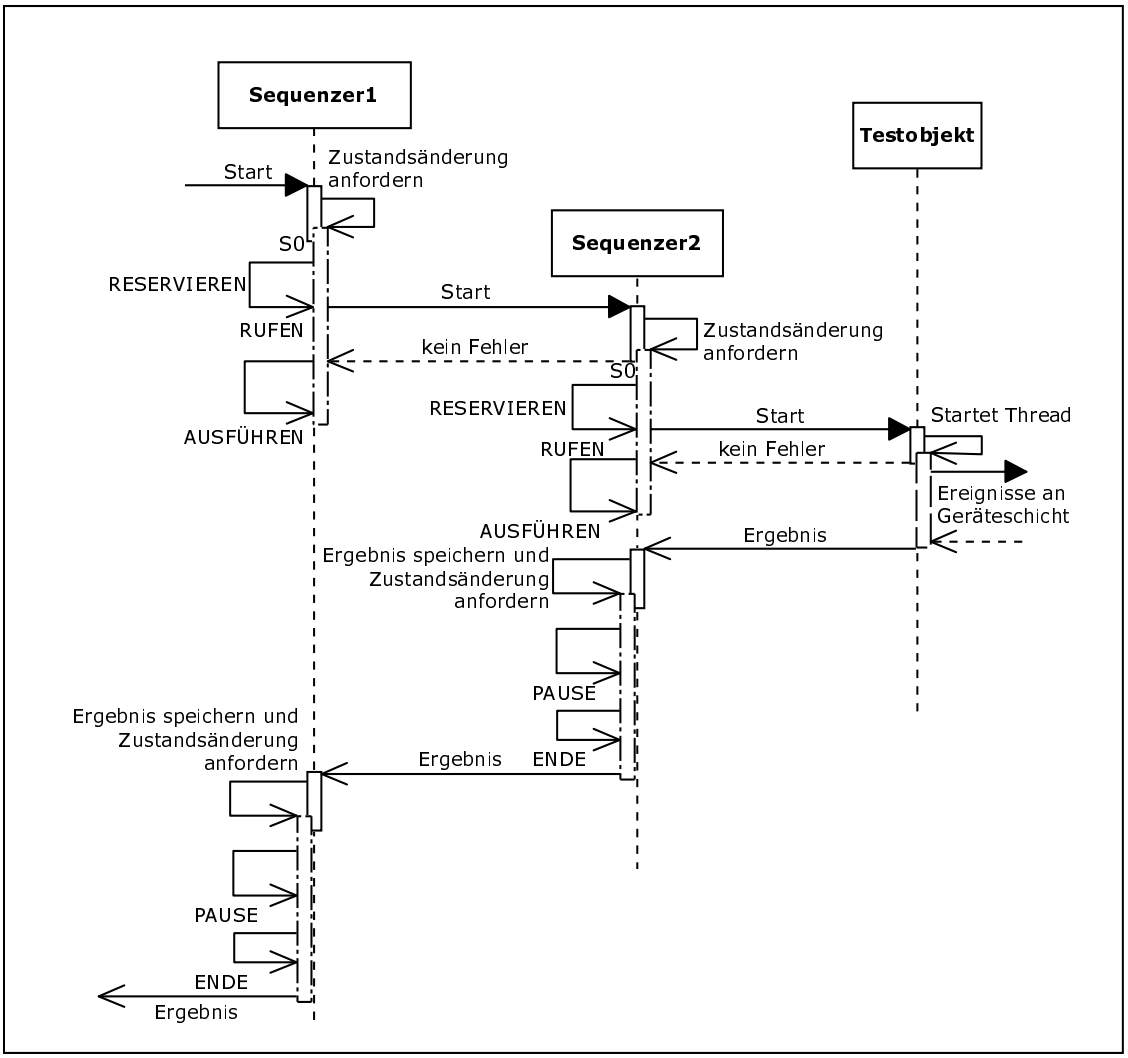


Abbildung 4.13 geschachtelte Sequenzer: Sequenzer1 ruft Sequenzer2 auf. Optionale Reservierungsvorgänge sind aus Gründen der Übersichtlichkeit nicht dargestellt.

Die Möglichkeit, dass Sequenzer andere Sequenzer starten können, erlaubt die beliebige Schachtelung von Kontrollstrukturen innerhalb einer komplexen Abfolge elementarer Sequenzen. Listen stellen die Möglichkeit von Verzweigungen zur Verfügung, die durch einen Sequenzer von einer Wiederholungsanweisung umschlossen werden. Verglichen mit einer Programmiersprache repräsentiert der Sequenzer eine Schleifenstruktur, deren Anweisungen in Form von Listenelementen durch bedingte Sprunganweisungen miteinander verknüpft werden können.
Abbildung 4.13 zeigt im Detail welche Ereignisse zwischen geschachtelten Sequenzern ausgetauscht werden und beschreibt die Transitionen, die dabei in den Zustandsmaschinen der Sequenzer stattfinden.

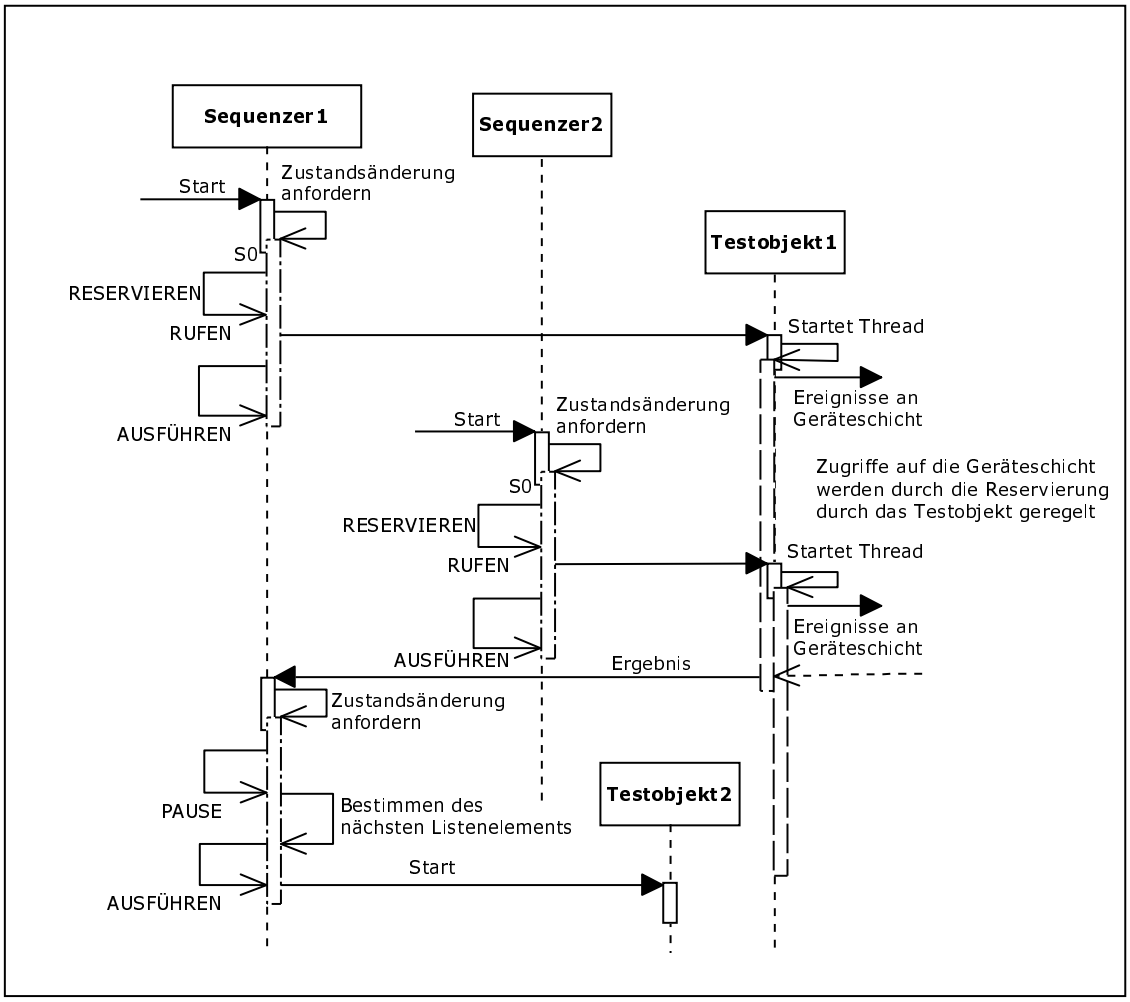


Abbildung 4.14 Nebenläufige Sequenzer ohne optionale Reservierung

4.5.2 Nebenläufige Sequenzer

Im Gegensatz zur bereits erwähnten Schachtelung von `Sequenzern`, besteht auch die Möglichkeit, mehrere `Sequenzer` unabhängig voneinander zu starten, um sie nebenläufig auszuführen. Für diese Art der Anwendung existiert der Reservierungsmechanismus durch die Stellvertreter, da die asynchron arbeitenden `Sequenzer` dieselben `Testobjekte` und damit dieselben `Pozesse` zur gleichen Zeit verwenden können. Dadurch entstehen unterschiedliche Szenarien abhängig davon, ob beteiligte `Sequenzer` von der Möglichkeit der optionalen `Reservierung` Gebrauch machen.

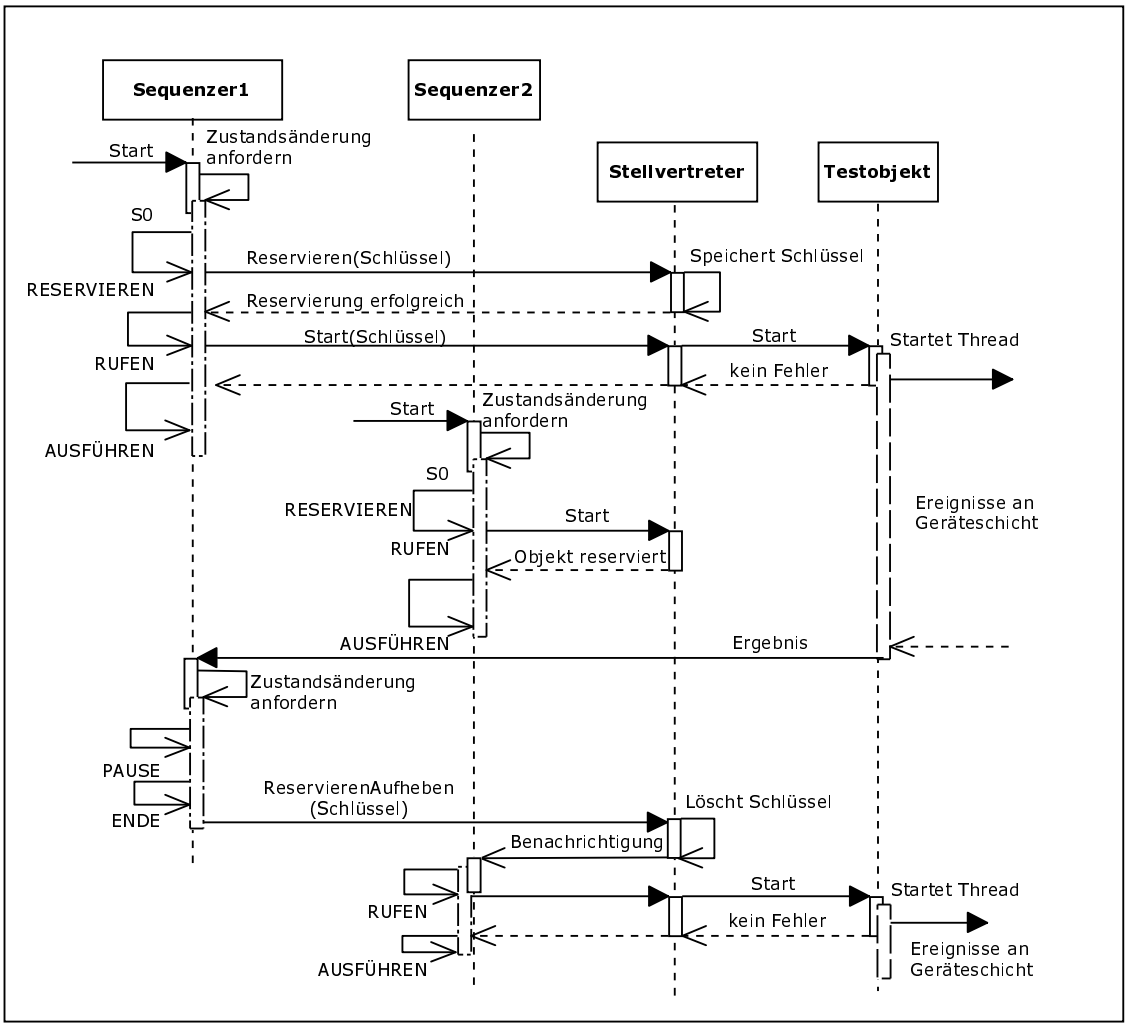


Abbildung 4.15 Sequenzer1 verwendet optionale Reservierung, Sequenzer 2 nicht.

Solange keine optionale Reservierung verwendet wird, verhindern die Stellvertreter der Prozesse, dass es zu parallelen Zugriffen auf einen Prozess kommt. Werden im Zuge der nebenläufigen Verarbeitung Testobjekte angesprochen, die dieselben Prozesse verwenden, regelt die Reservierung die Zugriffsreihenfolge (siehe Abbildung 4.14).
Für den Fall, dass nicht alle beteiligten Sequenzer eine optionale Reservierung vornehmen, müssen Diejenigen warten, die keine Reservierung vorgenommen haben, bis sie die Benachrichtigung durch die Stellvertreter erhalten. Solange die Reservierungen nicht aufgehoben werden, warten die blockierten Sequenzer im Zustand **AUSFÜHREN**. Abbildung 4.15 verdeutlicht die Abläufe für dieses Szenario.

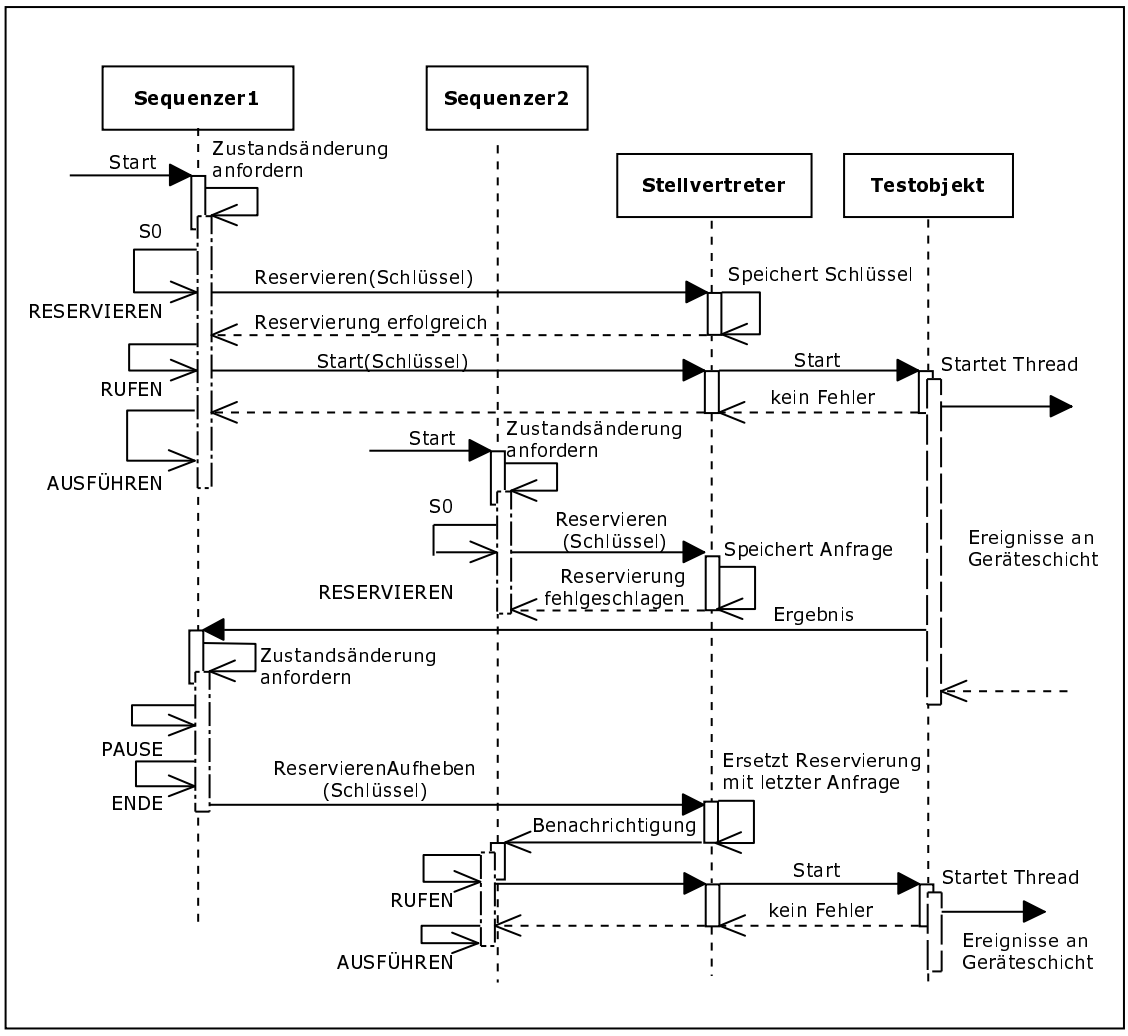


Abbildung 4.16 Beide Sequenzer reservieren das Testobjekt.

Erfolgt eine Reservierung gemeinsam genutzter Testobjekte oder Sequenzer durch mehrere Sequenzer, erhält entweder einer oder keiner alle Reservierungen. In diesem Zusammenhang können Deadlocks entstehen, falls Sequenzer ewig auf ihre Reservierungen warten sollen. Falls einer der Sequenzer über alle benötigten Reservierungen verfügt, startet dieser die Listenverarbeitung. Die Übrigen warten im Zustand **RESERVIEREN** bis die Reservierungen aufgehoben wurden, oder brechen das Warten nach Ablauf eines vorher definierten Zeitintervals ab. Für den Fall, das keiner alle Reservierungen erhält, da sich die Reservierungsanfragen zeitlich überschnitten haben, ist es vom Timeout-Parameter der einzelnen Sequenzer abhängig, ob alle das Warten abbrechen oder einzelne noch rechtzeitig benachrichtigt werden können (siehe Abbildung 4.16).

Alternative Reservierungskonzepte Benötigte eine Instanz mehrere Reservierungen, führt diese die Reservierungsanfragen sequentiell durch. Fehlgeschlagene Reservierungen führen dazu, dass die betreffende Instanz auf die Benachrichtigungen der Stellvertreter wartet. Falls das Warteinterval mehrerer Instanzen nicht begrenzt ist, kann dies zu Deadlocks führen.
Eine Alternativlösung besteht darin, zunächst zu prüfen, ob alle benötigten Reservierungen möglich sind und falls diese Bedingung zutrifft, die Ausführung der Reservierung zu starten. Dadurch entfällt die Möglichkeit, dass sich Reservierungen zeitlich überschneiden und negiert das Auftreten von Deadlocks. Anstatt auf die einzelnen Reservierungen zu warten, wird abgewartet bis die Möglichkeit besteht alle Reservierungen zu erhalten und erst dann zu reservieren. Dieses Konzept kann durch die Modellierung des Reservierungsmechanismus mittels Petri-Netzen[12] realisiert werden.

[12] Mathematisches Modell von nebenläufigen Systemen; siehe http://wikipedia.org/wiki/Petrinetz

4.6 Die GUI Klasse - CSSeqControl

Um den Programmieraufwand für den Entwickler auf die Implementierung der elementaren Sequenzen zu beschränken, ist mit der Klasse CSSeqControl eine GUI implementiert, die mit einen beliebigen Sequenzer kommunizieren kann. Als Schnittstelle zwischen Entwickler und Sequenzer-Framework, stellt sie den Zustand eines Sequenzers und der dazugehörigen Liste da. Darüber hinaus dient die GUI dem Anlegen, Editieren und Ausführen von Listen. Abbildung 4.17 zeigt die Oberfläche des threads der Klasse CSSeqControl. Tabelle 4.4 und 4.5 beinhalten eine Beschreibung der auf der GUI befindlichen Anzeige- und Bedienelemente und ihrer Menüleisten-Funktionen.
Das GUI versendet periodisch Ereignisse an ihren Sequenzer, um dessen Eigenschaften und Zustand abzufragen und die empfangenen Daten für den Benutzer anzuzeigen. Zwischen diesen zyklischen Ausleseaktionen, reagiert das GUI auf die Eingaben des Benutzers, der einerseits die Listeninhalte editieren und andererseits den Sequenzer steuern will. Um die Konsistenz des Listeninhalts während der Verarbeitung durch den Sequenzer zu gewährleisten, kann das GUI zwischen zwei Modi umgeschaltet werden. Dadurch wird verhindern, dass der Benutzer während der Ausführung der Liste, deren Elemente verändert.
Darüber hinaus prüft der Sequenzer anhand seines Zustandes, ob eintreffende Ereignisse, die Listenmanipulierende Methoden auslösen, ausgeführt werden dürfen. Dadurch ist sowohl durch die GUI-Klasse als auch durch die Ereignisbehandlung des Sequenzers sichergestellt, dass eine Veränderung der Listeninhalte während deren Ausführung nicht stattfinden kann.

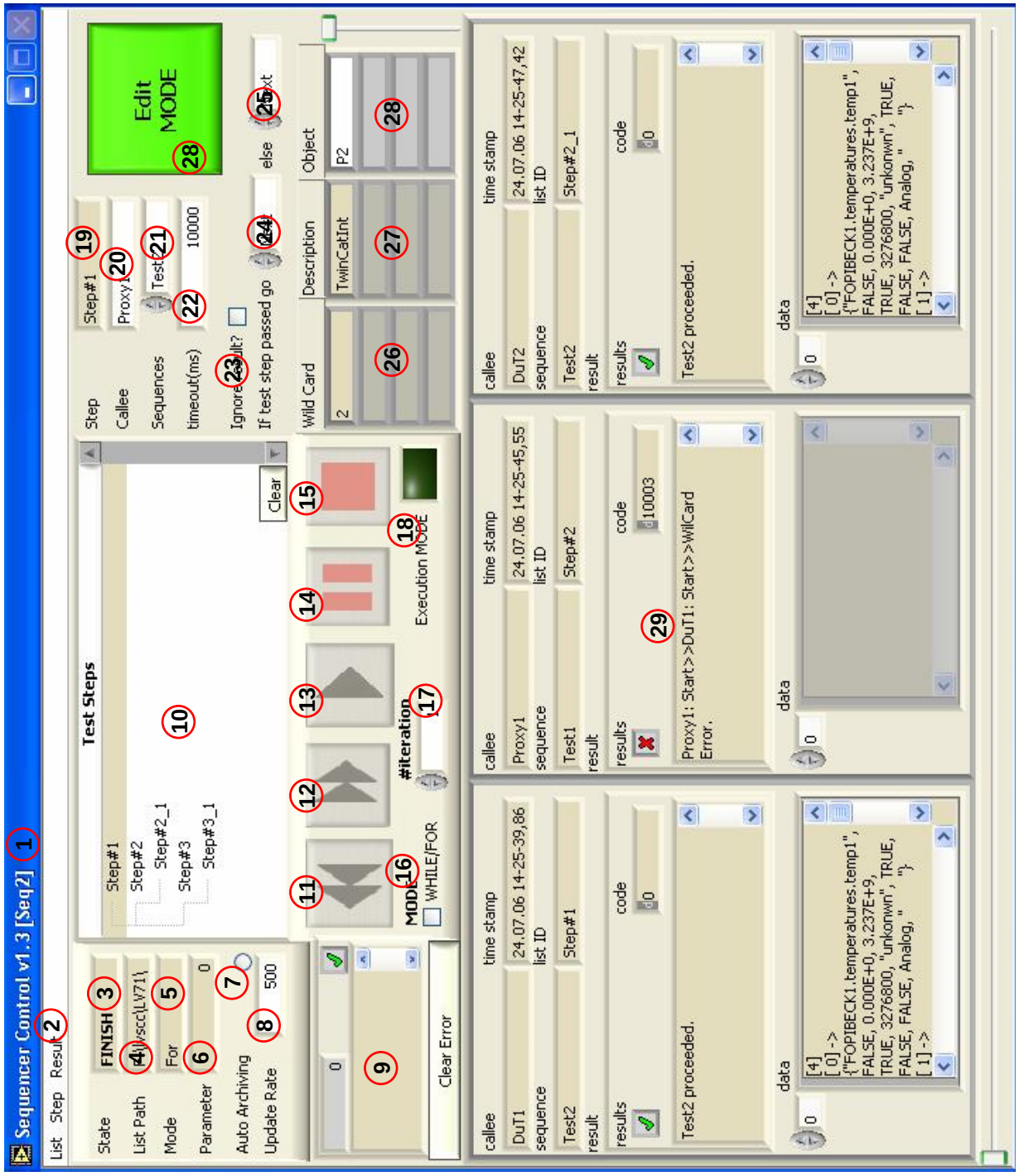


Abbildung 4.17 Bedienoberfläche der Klasse `CSSeqControl`

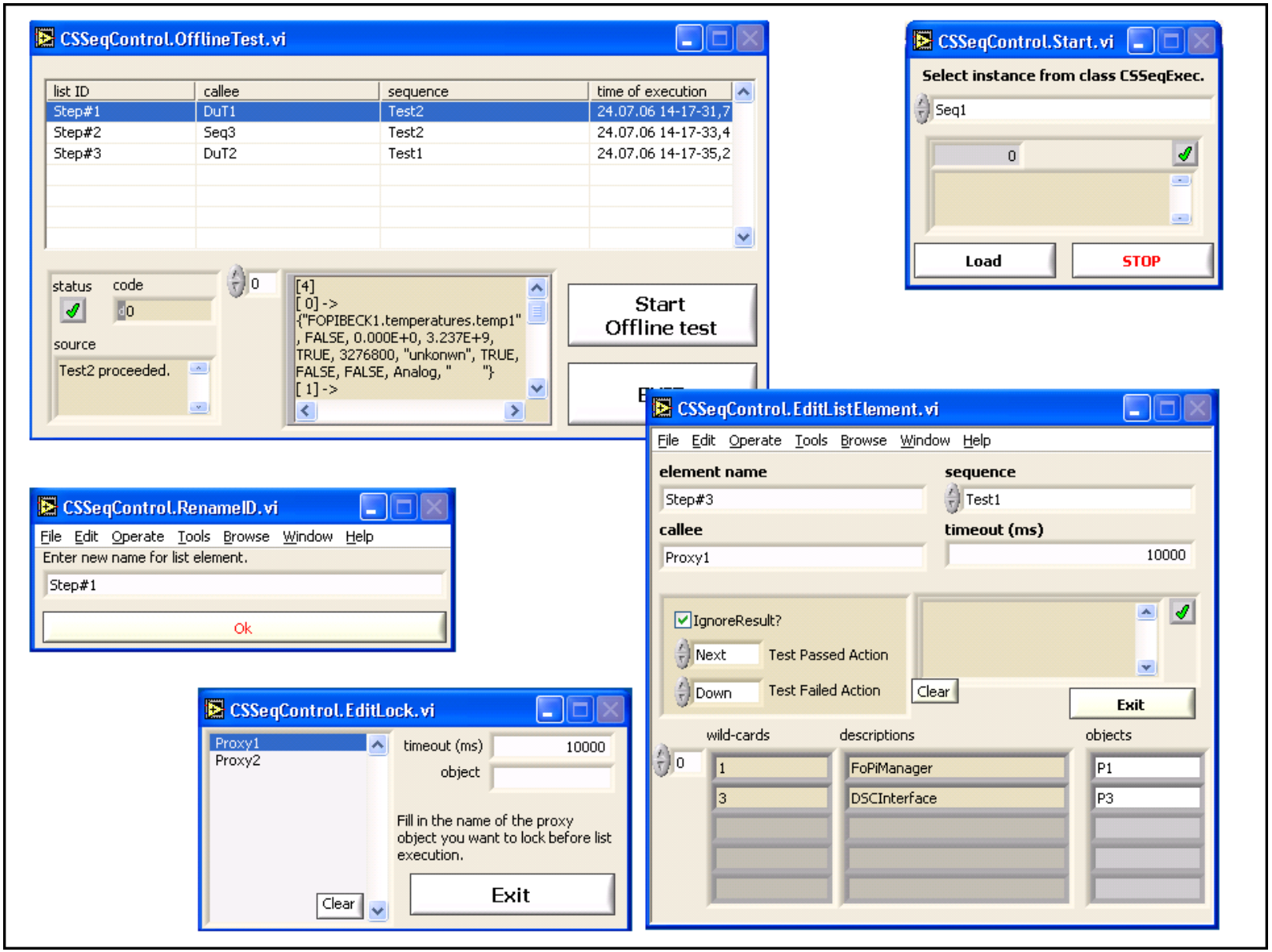


Abbildung 4.18 Fenster von links oben nach rechts unten: a.) Überprüfung bereits vorhandener Ergebnisse; b.) Startoberfläche zum Auswählen des Sequenzers; c.) Umbenennen eines Listenelements ; d.) Editieren der optionalen Reservierungen einer Liste; e.) Anlegen eines neuen Listenelements

Tabelle 4.4 Bedien- und Anzeigeelemente des Sequenzer-GUIs

Nummer	Funktion
1	Die Kopfzeile beinhaltet den Namen des Sequenzers.
2	Menüleiste siehe Tabelle 4.5
3	Aktueller Zustand des Sequenzers
4	Pfad der momentan geladenen Listendatei
5	Aktueller Schleifen-Modus des Sequenzers (siehe Tabelle 4.3)
6	Aktueller Schleifen-Parameter des Sequenzers (siehe Tabelle 4.3)
7	Automatische Archivierung der Testergebnisse (de-)aktivieren. Falls die Option aktiviert ist, legt der Sequenzer nach jeder Listenausführung ein XML[13]-Datei an, die die Ergebnisse der Listenelemente beinhaltet. Das Verzeichnis kann über den Datenbankeintrag des Sequenzers festgelegt werden.
8	Aktualisierungsrate, definiert das Zeitinterval in Millisekunden, nachdem die GUI ihre Bedienelemente mit den Attributwerten des Sequenzers aktualisiert.
9	Fehleranzeige, die sowohl Fehler durch den Benutzer als auch Fehler, die in Hintergrundprozessen, wie z.B. dem Sequenzer entstehen, darstellt. Es wird immer der zu letzt aufgetretene Fehler dargestellt. Die Fehleranzeige kann per Knopfdruck zurückgesetzt werden.
10	Darstellung der Listenelemente und deren Verweise mittels einer Baumstruktur. Die Listenebenen werden ähnlich einer Verzeichnisstruktur, wie man sie von der Datei- und Ordnerverwaltung in Windows kennt, dargestellt. Listenelemente können ähnlich wie Dateien im Windows Explorer durch einfaches Verschieben neu angeordnet werden. Das unterlegte Listenelement zeigt die aktuelle Position des Listenzeigers an.
11	Falls der Sequenzer sich im Zustand **PAUSE** befindet, ermöglicht das Bedienelement dem Benutzer die Position des Listenzeigers Elementweise entgegen der Ausführungsrichtung zu verändern ohne dabei die Ausführung der Listenelemente auszulösen.
12	Setzt den Ausführungsmodus des Sequenzers auf single und startet anschließend die Listenverarbeitung (siehe Tabelle 4.3).

[13] Extended Markup Language; siehe http://de.wikipedia.org/wiki/XML

Tabelle 4.4 Bedien- und Anzeigeelemente des Sequenzer-GUIs

Nummer	Funktion
13	Setzt den Ausführungsmodus des Sequenzers auf auto und startet anschließend die Listenverarbeitung (siehe Tabelle 4.3).
14	Fordert den Sequenzer auf nach dem aktuell zu bearbeitenden Listenelement im Zustand **PAUSE** zu verharren.
15	Fordert den Sequenzer auf nach dem aktuell zu bearbeitenden Listenelement in den Zustand **FINISH** zu wechseln.
16	Setzt den neuen Schleifen-Modus des Sequenzers. Die Änderungen werden erst nach Betätigung von Bedienelement 11 oder 12 wirksam.
17	Setzt den neuen Schleifen-Parameter des Sequenzers. Die Änderungen werden erst nach Betätigung von Bedienelement 11 oder 12 wirksam.
18	Zeigt den aktuellen Ausführungsmodus des Sequenzers an.
19	Anzeige für den Namen des aktuellen Listenelements.
20	Hier wird der Name des Testobjekts bzw. des Stellvertreters eingegeben (siehe Tabelle 4.2). Existiert das Objekt nicht wird es automatisch erzeugt.
21	Hier kann der Name der elementaren Sequenz ausgewählt werden. Das Bedienelement stellt bereits die Namen aller implementierten Sequenzen des angewählten Testobjektes zur Auswahl.
22	Eingabe für das Zeitinterval, dass der thread eines Testobjekts maximal auf die Reservierung der Prozesse wartet (siehe Tabelle 4.2).
23	(De-)Aktiviert den Steuerparameter Ignoriere Ergebnis (siehe Tabelle 4.2).
24	Auswahl für die Listenoperation bei erfolgreichem Sequenzergebnis (siehe Tabelle 4.2).
25	Auswahl für die Listenoperation bei fehlerhaftem Sequenzergebnis (siehe Tabelle 4.2).
26	Anzeige der Platzhalter, die in einer Sequenz vorkommen.
27	Anzeige der Beschreibung für die einzelnen Platzhalter.
28	Dient der Deklaration der Ersetzungen für die Platzhalter des angewählten Listenelements (siehe Tabelle 4.2).

Tabelle 4.4 Bedien- und Anzeigeelemente des Sequenzer-GUIs

Nummer	Funktion
29	Schaltet den Modus der GUI zwischen Konfigurations- und Steuerungsmodus um. Im Steuerungsmodus können nur die Bedienelemente betätigt werden, die zur Ausführung der Liste dienen. Der Konfigurationsmodus deaktiviert die zur Listenausführung benötigten Bedienelemente und aktiviert Jene die zur Bearbeitung der Liste verwendet werden. Wechselt der Benutzer vom Konfigurations- in den Steuermodus wird die Liste mit den Modifikationen durch die GUI aktualisiert.
30	Die optionale Visualisierung der Sequenzergebnisse beinhaltet zu jedem ausgeführten Listenelemente das dazugehörige Ergebnis bestehend aus: Name des `Testobjekts` und der Sequenz, Zeitstempel der Ausführung, Name des Listenelements sowie Ergebnis mit Kommentar und Daten.

Tabelle 4.5 Funktionen der Menüleiste des Sequenzer-GUIs

Menüpunkt	Unterpunkt	Funktion
Liste	Schreiben	Legt die aktuell geladene Liste in einer Datei ab.
Liste	Lesen	Liest eine Liste aus einer Datei, lädt diese in die `Liste` und aktualisiert die entsprechenden Bedien- und Anzeigeelemente der GUI.
Liste	Wiederherstellen	Liest die aktuell geladenen Liste aus der `Liste` und aktualisiert die entsprechenden Bedien- und Anzeigeelemente der GUI.
Liste	Einstellung der optionalen `Reservierung`	Öffnet ein weiteres Fenster, in welchem der Benutzer die zu reservierenden `Stellvertreter` und das zu wartende Zeitinterval definieren kann (siehe Abbildung 4.18d).
Liste	Beenden	Schliesst das Fenster, zerstört die GUI-Instanz und den dazugehörigen `Sequenzer`.

Tabelle 4.5 Funktionen der Menüleiste des Sequenzer-GUIs

Menüpunkt	**Unterpunkt**	**Funktion**
Listenelement	Hinzufügen	Öffnet einen weiteres Fenster, in dem einen neues Listenelement konfiguriert werden kann, um es anschließend in die Liste mit aufzunehmen. Das Element kann entweder ans Ende der Liste gesetzt oder als letztes Element in die nächst untere Ebene des aktuell angewählten Listenelements eingefügt werden (siehe Abbildung 4.18e).
Listenelement	Umbenennen	Öffnet einen weiteren Dialog, der es ermöglicht, das momentan angewählte Listenelement umzubenennen (siehe Abbildung 4.18c).
Listenelement	Überspringen	Überspringt das aktuelle Listenelement, falls gerufenes Objekt nicht antwortet, da es möglicherweise bereits reserviert ist.
Ergebnisse	Sichern	Öffnet einen Dateidialog, um die aktuellen Ergebnisse der Liste in einer XML Datei abzulegen.
Ergebnisse	Laden	Öffnet einen Dateidialog, um Ergebnisse einer Liste aus einer XML Datei auszulesen.
Ergebnisse	Offline Test	Öffnet ein neues Fenster, welches die aktuellen Ergebnisse darstellt und eine Überprüfung der Ergebnisse anhand der Vergleichsmethoden des dazugehörigen `Testobjekts` ermöglicht. Hierzu muss der Entwickler allerdings die Daten, die für die Bestimmung des Ergebnisses relevant sind an den `Sequenzer` zurückgegeben haben (siehe Abbildung 4.18a).
Ergebnisse	Ergebnisse betrachten	Erweitert das Fenster des GUI, damit im unteren Bereich die Ergebnisse der einzelnen Listenelemente sichtbar werden.
Ergebnisse	Auto Archivierung	(De-)Aktiviert das automatische Abspeichern der Ergebnisse nach einer Listendurchführung.

Kapitel 5

Anwendung: BELAB[1] - Teststand

5.1 Einführung

Der Anstoß für die Erweiterung des *CS* Frameworks um eine Sequenzer-Klassenbibliothek wurde durch die Anfrage des BELAB gegeben, eine Test-Software für deren Elektronik zu entwerfen.
Das BELAB befasst sich hauptsächlich mit dem Anschluss von Geräten der Beschleunigeranlage an das rechnergesteuerte Kontrollsystem der GSI.[2] Seit 1972 werden im BELAB Bussysteme und die dazugehörige Anpasselektronik entwickelt, um die Vielfalt an externen Geräten, durch ein einheitliches Medium miteinander zu verbinden und eine Schnittstelle zum Kontrollsystem des Beschleunigers herzustellen. Um den Anforderungen durch den komplexen und ständig wachsenden Aufbau der Beschleunigeranlage gerecht zu werden, entstanden in der langjährigen Entwicklungsarbeit des BELAB eine Vielzahl unterschiedlicher Einzelkomponenten und Bussysteme. Dabei handelt es sich vor Allem um proprietäre[3] Entwicklungen der GSI, die weltweit an keiner anderen Stelle zur Anwendung kommen.

[1] Kurzform für Beschleuniger-Elektronik-Labor; Teil der Gruppe Beschleuniger-Elektronik an der GSI
[2] siehe [BELAB1]
[3] Im Falle von Hardware bezeichnet der Begriff Entwicklungen, die nicht den allgemein anerkannten Standards entsprechen.

5.2 Die Test-Software

5.2.1 Anwendungsfälle

Mit der Fertigung der im BELAB entwickelten Hardware werden zum größten Teil externe Firmen beauftragt. Nach Bestückung der Platinen muss eine Qualitätskontrolle durch die Herstellerfirmen erfolgen. Eine Testsoftware, die das BELAB den Firmen zur Verfügung stellt, reduziert den dafür benötigten Zeit- und Kostenaufwand und definiert einheitliche Qualitätsmaßstäbe für die Hersteller.
Neben der Fertigungskontrolle existieren weitere Anwendungsfälle für die Software. Sowohl die Inbetriebnahme und Steuerung der Hardware im Betrieb als auch die Unterstützung der Entwickler und Techniker bei der Fehler-Diagnose sind Aufgaben, die durch ein entsprechendes Software-System erleichtert werden.
Ähnlich variable wie die Anwendungsgebiete sind die damit verbundenen Benutzer, die über unterschiedliches Wissen verfügen und verschiedene Anforderungen an die Software stellen. Zum Beispiel will ein Techniker, der eine neue Komponente in Betrieb nehmen will, lediglich testen, ob die Hardware fehlerfrei arbeitet. Hierzu benötigt er eine Auswahl an umfangreichen Testprozeduren, hinter denen sich im Einzelnen mehrere elementare Hardwarezugriffe in einer bestimmten Reihenfolge verbergen. Der Hardware-Entwickler dagegen möchte einzelne elementare Zugriffe in beliebiger Reihenfolge auf die Hardware ausüben, um Ursache und Ort eines Fehlers im Detail bestimmen zu können.

5.2.2 Anforderungen

Die Definitionsphase des Sequenzer-Frameworks wurde maßgeblich durch die Anforderungen des BELAB geprägt. Im Gegensatz zu Experimenten, welche Sequenzer hauptsächlich für die Automatisierung definierter Abläufe wie z.B. der Inbetriebnahme eines Detektors verwenden, sind die Anforderungen des BELAB-Teststandes an einen generischen Sequenzer weitaus umfangreicher.
Testdaten und Vergleichswerte, wie z.B. Bitmuster oder Rauschanregungssignale, müssen zur Laufzeit verändert werden können. Neben der Sequenzierung einzelner elementarer Aktionen muss im Rahmen eines Teststandes die Möglichkeit bestehen, nach der Ausführung einer Sequenz eine abschließende Analyse der angefallenen Testdaten durchführen zu können. Dabei kann es sich um einfache Vergleichsfunktionen bis hin zu komplexeren mathemati-

schen Algorithmen handeln. Das Ergebnis und die dazugehörigen Daten müssen zur Dokumenation des Tests abgespeichert werden.
Im Rahmen eines Teststandes existiert zudem die Anforderung nach der Identifizierung der Testumgebung. Dazu gehören neben dem Prüfling alle Geräte, die am Test beteiligt sind. Zum Beispiel soll eine Digitale-I/O-Karte getestet werden. Hierfür wird ein Testbitmuster benötigt, das als Anregungssignal dient. Die einwandfreie Funktion des Gerätes, das das benötigte Bitmuster generiert, ist Vorraussetzung für den erfolgreichen Ablauf des Tests. Daher muss die Überprüfung des Bitmuster-Generators Bestandteil des Tests sein.
Um so komplexer und umfangreicher die Testumgebung wird, desto mehr Abfragen müssen im Vorfeld eines Test durchgeführt werden, um sicher zu stellen, dass die Testumgebung bereit ist und während eines Tests auftretende Fehler nur auf den Prüfling zurückzuführen sind und nicht auf Komponenten der Testumgebung.
Im Falle des BELAB kann eine Testumgebung aus vielen Komponenten bestehen. Um dem Anwender das Testen einzelner Komponenten zu erleichtern, muss die Test-Software sicherstellen, dass alle Bedingungen für einen Test erfüllt sind und falls dies nicht der Fall ist, den Anwender darüber in Kenntnis setzen, welche Fehler in der Testumgebung herrschen. Hierfür werden Testprozeduren benötigt, die einerseits eine ständige Kontrolle der Testumgebung durchführen, andererseits keine Interaktion mit den eigentlichen Tests auslösen. Sollen diese Test-Routinen nicht durch den Benutzer ausgelöst, sondern von der Software angestoßen werden, müssen diese nebenläufig zu den vom Benutzer angestoßenen Tests ablaufen. Daraus resultiert die Notwendigkeit von Reservierungsmechanismen, die die nebenläufigen Prozesse synchronisert, da diese auf dieselben Hardware-Ressourcen zugreifen.
Angesichts der Vielzahl der Hardware-Komponenten des BELAB muss die Test-Software modular aufgebaut werden, um die Eigenschaften der Hardware entsprechend abbilden und mit der Entwicklung im Bereich der Hardware mithalten zu können.

5.3 Beschreibung der BELAB-Hardware

Der Anschluss eines beliebigen Gerätes bzw. einer Anlage an das Kontrollsystem der GSI geschieht über den so genannten Device-Bus. Dabei handelt es sich um eine proprietäre Entwicklung der GSI. Das Übertragungsprotokoll basiert auf dem MIL-1553B Protokoll [4].
Die Aufgabe des Device-Bus ist es, über zugehörige Anpass-Elektroniken wie Interface- und I/O-Karten, Geräte wie Netzteile für Magnete oder Hochfrequenz-Sender mit den Kontrollrechnern (VME-Rahmen) zu verbinden (siehe Abbildung 5.1). Der Bus arbeitet nach dem Master-Slave-Prinzip. Der Master (SE = Steuereinheit) befindet sich im Kontrollrechner und kommuniziert mit den Slaves in Form der Interface-Karten (IFK).[5]

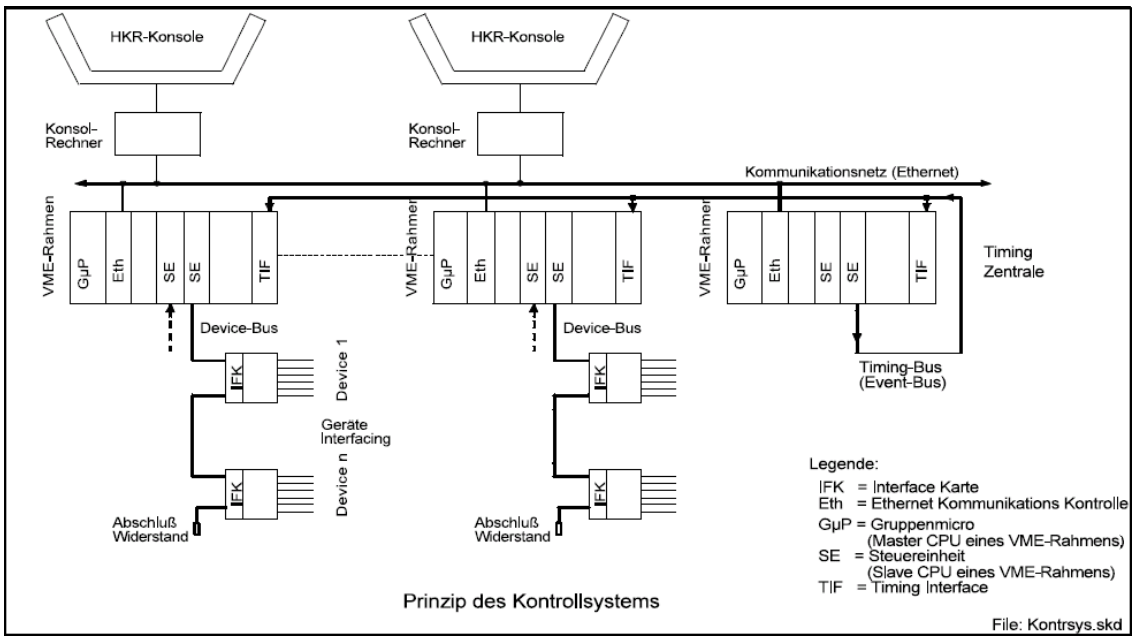


Abbildung 5.1 Aufbau des GSI-Kontrollsystems [BELAB2]

Da auf jeder Interface-Karte der komplette Anschluss an den Device-Bus untergebracht ist, steht wenig Platz für weitere Anwendungen zur Verfügung. Daher etablierten sich das so genannte Backplane[6]-Modul, das Platz für eine Interface-Karte, bis zu vier ADC/DAC[7]-Karte und eine Schaltkarte bietet.
Im nächsten Schritt entstand unterhalb des Device-Busses ein weiteres Bussystem: der Modul-Bus. Er vereinfacht den Anschluss von Anwender-Elektronik

[4] Spezifikation MIL-STD 1553 vom 30.04.1975; siehe http://www-wnt.gsi.de/belab/Betriebsmittel/Allgemein/Devicebus/mil_1553b.pdf
[5] siehe [BELAB1]
[6] Rückwandverdrahtung
[7] Analog-Digital- und Digital-Analog-Converter

über Anpasskarten an den Device-Bus und ermöglicht darüber hinaus die Abfrage eines eindeutigen Identifikationskodes, um Fehlbestückungen zu vermeiden.[8]

Für Labortests und Anlagen-Service vor Ort existieren Einsteckkarten für ISA- und PCI-Schnittstellen, die als Device-Bus-Master die Kommunikation mit den IFKs über einen herkömmlichen PC ermöglichen (siehe Abbildung 5.2). Dadurch können Geräte unabhängig von der SE sehr komfortable getestet werden, bevor sie an das Kontrollsystem angeschlossen werden.[9]

In der Software stellt eine DLL[10] die notwendigen Funktionen bereit, um über den Device-Bus auf die externe Hardware zuzugreifen. Diese kann ebenfalls von LabVIEW aus aufgerufen werden, um auf die BELAB-Hardware zuzugreifen.

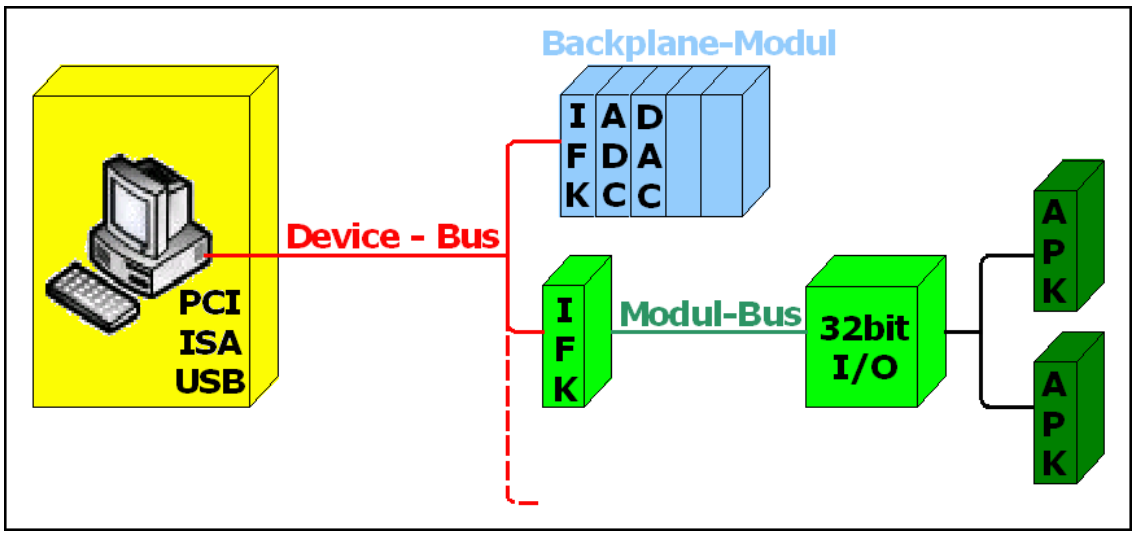


Abbildung 5.2 BELAB-Bussysteme und Hardware

[8] siehe [BELAB3]

[9] siehe [BELAB2]

[10] Dynamic Link Library; Eine unter Microsoft Windows verwendete Programmbibliothek, die von mehreren Anwendungen gleichzeitig verwendet werden kann. Der Vorteil einer DLL besteht in der Modularisierung häufig verwendeten Quellcode. Durch den Austausch einer DLL können Erweiterungen einfach für viele Anwendungen wirksam werden. Einzige Bedingung ist, dass die Schnittstelle der DLL sich nicht verändert; siehe http://de.wikipedia.org/wiki/Bibliothek_(Programmierung)

5.4 Entwurf der BELAB-Geräteschicht

Der Einsatz des Sequenzer-Framework im Rahmen eines BELAB-Teststandes benötigt im ersten Schritt den Zugriff auf die BELAB-Hardware innerhalb des *CS* Frameworks. Aufgrund der Vielfalt unterschiedlicher Karten- und Bustypen sowie ständiger Neuentwicklungen muss eine entsprechende Klassenhierarchie für die BELAB-Hardware implementiert werden, die spätere Erweiterungen zulässt, ohne die bereits vorhandene Software grundsätzlichen Änderungen unterziehen zu müssen. Darüber hinaus ist das Ziel ein Software-System zu entwerfen, das die Beziehungen der Hardware bestmöglich abbildet, um die Modularität der BELAB-Hardware in der Software ebenfalls vorzufinden.
Abbildung 5.3 gibt einen vereinfachten Überblick über den Entwurf der BELAB-Geräteklassen. Eine detaillierte Variante, die alle konkreten Geräteklassen beinhaltet, befindet sich im Anhang B auf Seite 93. Die Farbcodierung der Klassen entspricht Abbildung 5.2.
Für jeden Geräte- bzw. Kartentyp existiert eine Klasse, deren Instanzen die Repräsentation der realen Komponenten übernehmen. Die Kommunikation zwischen den Geräteinstanzen der Software verläuft ebenso mehrstufig, wie es in der Hardware aufgrund der verschiedenen Bussysteme der Fall ist. Dabei existiert immer eine Kombination aus einem Busmaster-Objekt und den dazugehörigen Slave-Objekten. Alle Slaves kennen ihren Master. So kennt beispielsweise ein Modulbus-Slave seinen Modulbus-Master, der seinerseits wiederum in seiner Rolle als MIL-Slave seinen MIL-Master kennt. Sollen Zugriffe auf eine Hardwarekomponente ausgeführt werden, wird die entsprechende Software-Instanz per Ereignis adressiert. Diese löst eine Kette weiterer Ereignisse aus. Jedes Objekt in einer solchen Kette hängt seine Daten wie z.B. Bus-Adresse, Funktionscodes, etc. an und reicht das Ereignis an ihren jeweiligen Master weiter, bis das Ereignis bei der MIL-Master-Instanz angelangt ist, die den eigentlich Aufruf der DLL und damit den Zugriff auf die Hardware durchführt. Die Antwort wird auf demselben Weg wieder zurück an das aufrufende Objekt versandt.
Das Durchreichen von Ereignissen bildet die Bus-Kommunikation in der Hardware ab und ermöglicht dieselbe Modularisierung, wie sie die Hardware bietet. Hierbei ist zu beachten, dass der Weg über die einzelnen Instanzen, den eine Aktion in der Software beschreibt, entgegengesetzt zur Bearbeitung in der Hardware verläuft. In der Software löst das Slave-Objekt eine Aktion aus. Die Hardware arbeitet allerdings nach dem Master-Slave-Prinzip. Aktionen werden in der Hardware folglich immer vom MIL-Master initiiert. Daher muss immer verdeutlicht werden, ob man sich auf Abläufe in Hard- oder Software

bezieht.
Da in der Hardware immer mehrere Kartentypen existieren, die als Master bzw. Slave fungieren werden die Bus-spezifischen Eigenschaften in abstrakte Klassen zusammengefasst. Konkrete Geräteklassen, die einem bestimmten Gerätetyp repräsentieren, können dann durch Vererbung von den abstrakten Klassen implementiert werden. Auf diese Weise stehen Eigenschaften und Operationen, die die Vernetzung und die Kommunikation über die verschiedenen Bussysteme in der Software abbilden, getrennt von gerätespezifischen Eigenschaften und Funktionen zur Verfügung. Mögliche Erweiterungen in Form von neuen Buskomponenten können zu einem späteren Zeitpunkt durch Vererbung wieder auf die bereits implementierten Eigenschaften und Schnittstellen zurückgreifen ohne diese neu implementieren zu müssen.
Ein Beispiel für eine kommende Neuerung ist die Erweiterung der MIL-Master-Karten um eine USB-Variante, die sich bereits in der Entwicklung befindet. Bisher ist es nur möglich mit dem PCI-MIL-Interface einzelne Schreib- und Lesezugriffe durchzuführen. In der Zukunft soll die USB-Variante eine Blockweise Verarbeitung ermöglichen. Dies muss in den Entwurf der MIL-Master-Klassen einbezogen werden. Die MIL-Master-Klasse zeichnet sich daher durch verschiedene Betriebsmodi aus. Je nach Schnittstelle muss zwischen Einzel- und Blockverarbeitung umgeschaltet werden können. Im Blockmodus sammelt ein MIL-Master-Objekt Zugriffsanfragen von verschiedenen Objekten, bis es aufgefordert wird, die gesammelten Befehle auszuführen. Hierbei ist vor Allem zu beachten, dass Befehlsanfragen von unterschiedlichen Sendern stammen und sich aufgrund der ereignisgesteuerten Kommunikation zeitlich überschneiden können. Daher muss das MIL-Master-Objekt die Reihenfolge der Befehle anhand der zugehörigen Senderinstanzen bestimmen und die Antworten ebenfalls in korrekter Reihenfolge an die Sender zurückverschicken.

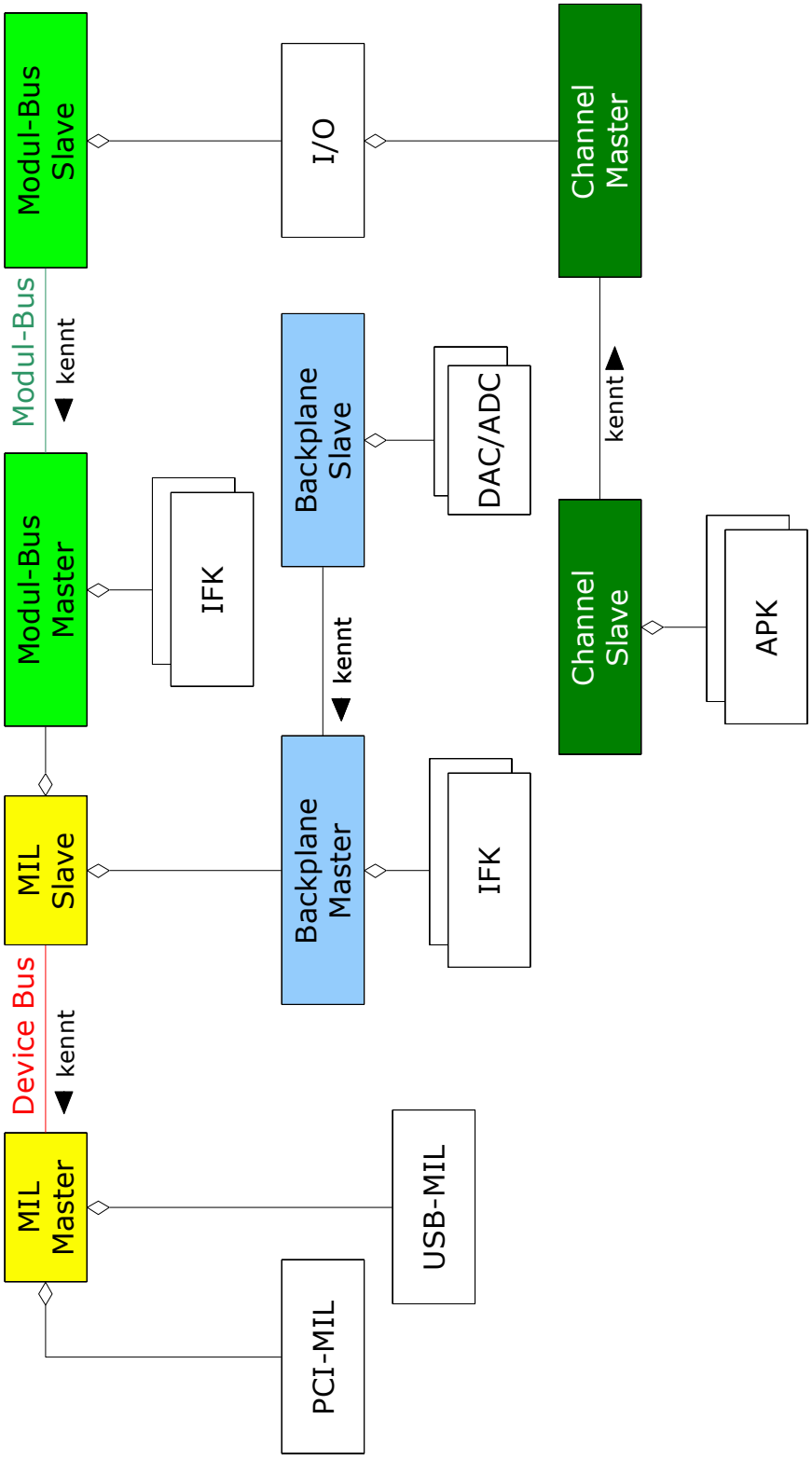


Abbildung 5.3 vereinfachtes Klassendiagramm der BELAB-Geräteschicht

5.5 Erste Anwendungsergebnisse

Der Entwurf der BELAB-Geräteschicht verlief parallel zur Entwicklung des Sequenzer-Framework. Die Implementierung der BELAB-Geräteklassen konnte im Verlauf dieser Arbeit nicht komplett fertig gestellt werden. Aus diesem Grund kam es bisher nur zu einem ersten Anwendungsversuch, mit dem Ziel elementare Hardwarezugriffe von einem Sequenzer aus anzustoßen, um die Schnittstelle zwischen den BELAB-Klassen und dem Sequenzer-Framework zu testen. Zu diesem Zweck wurde ein Testobjekt implementiert, welches grundlegende Funktionen einer Interface-Karte dem Sequenzer zur Verfügung stellt. Abbildung 5.4 zeigt die Ereigniskette, die in der Software angestoßen wird, um von einem Sequenzer den Zugriff auf die Interface-Karte durchzuführen. Die Test-Anwendung zeigt, dass der Zugriff auf die BELAB-Hardware ausgehend von einem Sequenzer möglich ist und einzelne Funktionen der BELAB-Geräteklassen mit dem Sequenzer-Framework zu Tests zusammengefasst werden können.
Ein kritischer Faktor hinsichtlich der Leistungsfähigkeit besteht jedoch in der zahlreichen Verwendung von Ereignissen. Einerseits benötigt die BELAB-Geräteschicht ein Ereignis pro Aktion zwischen den Master- und Slave-Objekten. Andererseits verwendet das Sequenzer-Framework Ereignisse sowohl für den Zugriff auf die BELAB-Geräteinstanzen als auch für die internen Reservierungen der Stellvertreterinstanzen und das Starten elementarer Sequenzen der Testobjekte. Angesichts einer Übertragungszeit eines synchronen, lokalen Ereignisses im Bereich von einer Millisekunde[11], entsteht bei mehrfachem Durchreichen eines Ereignisses ein erhöhter zeitlicher Aufwand, solange jede einzelne Aktion auf die Hardware, den gesamten Weg durch die Sequenzer- und Geräteschicht nehmen muss.
Im Falle von Testreihen, die sich über einen langen Zeitraum erstrecken oder innerhalb einer vorgegebenen Zeit abgeschlossen sein müssen, ist es notwendig die vielen Hardwarezugriffe, aus denen ein Test besteht, durch eine Methode der MIL-Masterklasse zusammenzufassen. Die Ereigniskette, die den Test anstößt muss in diesem Fall nur einmal durchlaufen werden. Das Testergebnis steht im Sequenzer dann erst nach Beendigung des Tests zur Verfügung.
Die Erweiterung der Geräteklassen um spezifische Test-Methoden kann sukzessiv vorgenommen werden. Zunächst verfügen die Geräteklassen über Methoden, die die elementaren Funktionen der Hardware zur Verfügung stellen. Sollen spezifische Tests implementiert werden, kann deren Ablauf zuvor vom

[11] siehe 2.5

Entwickler mit den elementaren Funktionen entworfen und getestet werden und anschließend durch entsprechende Methoden fest in den Quellcode der Geräteklassen eingefügt werden. Per Ereignis stehen diese Methoden dann dem Sequenzer-Framework zur Verfügung. Somit können auch umfangreiche Tests mit der maximalen Geschwindigkeit, die die Hardware zulässt, ausgeführt werden.

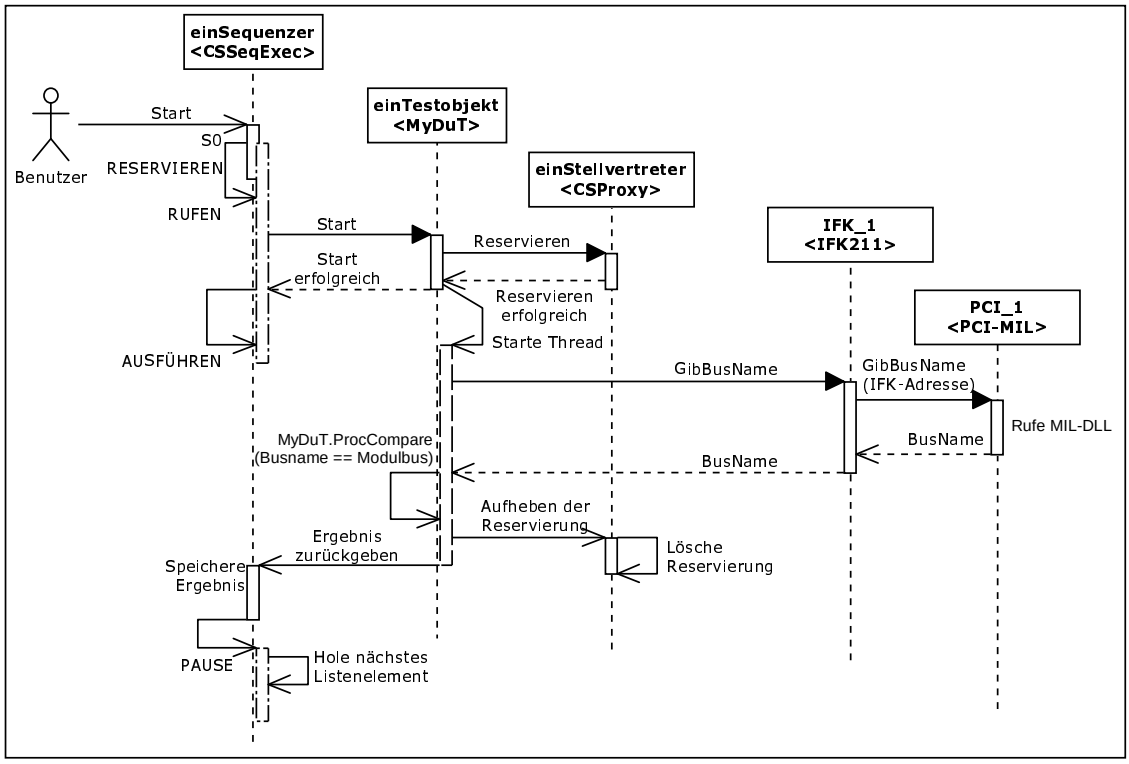


Abbildung 5.4 Interaktion zwischen Sequenzer-Framework und BELAB-Klassen im Rahmen einer ersten Test-Anwendung

Kapitel 6

Schlussbetrachtung

6.1 Ergebnis

Im Rahmen dieser Arbeit wurde die Entwicklung einer Klassenbibliothek mit dem *CS* Framework vorgestellt. Diese erweitert die Anwendungsschicht des bestehenden Frameworks um einen generischen Sequenzer.
Als Schnittstelle zu bereits bestehenden *CS*-Klassen dient die neue Basisklasse `CSDuT`, die es dem Entwickler erlaubt, einzelne Ereignisse zu sequenzieren und die daraus entstehenden Sequenzen nebenläufig abzuarbeiten. Ihr übergeordnet ermöglicht die Klasse `CSSeqExec`, die Sequenzen mit Kontrollstrukturen wie Wiederholungen und Verzweigungen zu versehen und in Form von Listen zusammenzufassen und abzulegen. Eine Liste kann weiterer Listen beinhalten und ermöglicht damit die Schachtelung von Kontrollstrukturen und erhöht die Wiederverwendbarkeit bereits vorhandener Listen. Für das Erstellen, Editieren und Testen einer Liste steht dem Entwickler eine grafische Oberfläche zur Verfügung.
Der durch die nebenläufige Verarbeitung entstehende gleichzeitige Zugriff auf einen `Prozess` durch mehrere Instanzen der Sequenzer-Klassen wird durch den Reservierungsmechanismus der Klasse `CSProxy` synchronisiert. Verklemmungen, die aufgrund konkurrierender Reservierungen zum Stillstand der Anwendung führen, werden durch Wartezeiten vermieden. Die Reservierung ist durch die Sequenzer-Klassen gekapselt und kann optional vom Entwickler verwendet werden.
Eine erste Beispielanwendung im Rahmen des BELAB-Teststandes hat die Funktionalität der Sequenzer-Klassenbibliothek unter Beweis gestellt und erlaubt weitere Entwicklungen, die auf den Sequenzer-Klassen aufbauen.

6.2 Ausblick

6.2.1 Modularität und Leistungsfähigkeit

Einzelne Aktionen einer Sequenz lösen abhängig von der Schachtelungstiefe der Liste, von der aus sie angestoßen werden, und der Anzahl an benötigten Reservierungen eine Kette an Ereignissen zwischen Objekten der Sequenzer-Klassen aus, bevor sie ausgeführt werden. Die vergleichsweise langsame Verarbeitung von Ereignissen im Bereich von Millisekunden erhöht daher je nach Länge der Ereigniskette die Verarbeitungszeit einer Sequenz.
Dies äußert sich vor Allem im Falle der Reservierung. Durch die stellvertretende Funktion der Klasse `CSProxy` verdoppelt sich die Anzahl an Ereignissen, die benötigt werden, um mit einem reservierten Objekte zu kommunizieren. Abhilfe bringt an dieser Stelle die nächsten Version des *CS* Frameworks, die die Reservierung eines `Prozesses` durch die *CS*-Basisklasse `CAEObj` bereitstellt und damit das Weiterreichen der Ereignisse durch stellvertretende Instanzen erspart. Darüber hinaus verbessert der neue Ereignismechanismus basierend auf DIM die Ereignisrate des *CS* Frameworks.
Besteht die Notwendigkeit die Verarbeitungszeit zeitkritischer Sequenzen zu beschleunigen muss die Anzahl an Ereignissen, die zu deren Ausführung nötig sind, reduziert werden. Hierzu können zwei Maßnahmen ergriffen werden. Erstens besteht die Möglichkeit, Aktionen innerhalb einer Liste in eine Sequenz, die durch die Klasse `CSDuT` definiert wird, zu verlagern. Dadurch reduziert man die Anzahl an Ereignisse, die innerhalb der Sequenzer-Schicht ausgetauscht werden. Im zweiten Schritt können innerhalb der `Prozesse`, die von der Sequenzer-Schicht verwendet werden, Aktionen zusammengefasst werden, die später durch ein einzelnes Ereignis ausgelöst werden. Bei der Entscheidung, an welcher Stelle Aktionen zusammengefasst werden, muss der Entwickler sich zwischen Modularität und Leistungsfähigkeit entscheiden. Umso schneller eine Sequenz verarbeitet werden soll, desto statischer muss sie in der Software hinterlegt werden, um die zusätzlichen Kommunikationswege, die aufgrund der Modularität der Software bestehen, nicht für jede Einzelaktion durchlaufen zu müssen. Wird sowohl Flexibilität und Geschwindigkeit gefordert, müssen für beide Kriterien unterschiedliche Varianten vorgesehen werden, die an die jeweiligen Anforderungen angepasst sind.

6.2.2 Schnittstelle zum *CS* Framework

In der aktuellen Version der Sequenzer-Klassenbibliothek müssen zunächst Sequenzen mit Hilfe der Basisklasse `CSDuT` definiert werden, bevor diese im Rahmen von Listen verwendet werden können. Das bedeutet, der Entwickler muss anfangs Klassen implementieren, die die Schnittstelle zwischen einer bestehenden *CS*-Klasse und dem Sequenzer-Framework herstellt.
Ein Alternative, die ohne das Erstellen neuer Klassen auskommt, besteht in der Erweiterung der Klasse `CSSeqExec`, deren Instanzen dann direkt Ereignisse an beliebige `Prozesse` versenden anstatt über Schnittstellen-Klassen zu kommunizieren. In diesem Fall enthält ein Listenelement die Informationen über ein Ereignis eines beliebigen `Prozess`.
Dadurch verliert man allerdings die Möglichkeit der nebenläufigen Verarbeitung, da die Ereignisbehandlung eines `Prozess` sequentiell arbeitet. Darüber hinaus entfällt die Implementierung spezifischer Analyse-Methoden am Ende einer Sequenz, die nur durch die Klasse `CSDuT` bereitgestellt werden und über die ein beliebiger `Prozess` nicht verfügt. Der Vorteil besteht lediglich darin, dass keine neuen Klassen implementiert werden müssen und der Ereignisaustausch zwischen Sequenzer und Schnittstellen-Objekt entfällt, was die Verarbeitungszeit wiederum beschleunigt. Abhängig von der Komplexität der Sequenzen und der damit verbundenen Anwendung entscheidet der Entwickler, welche der beiden Möglichkeiten er für die Lösung seiner Aufgabe verwendet.

Anhang A

Glossar

CERN(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.
[siehe http://public.web.cern.ch/Public/Welcome.html]

Datenfluss

Kausale Abhängigkeiten zwischen Funktionen, die entscheiden, ob serielle oder parallele bzw. nebenläufige Verarbeitung der vorhandenen Funktionen möglich ist.
[siehe http://de.wikipedia.org/wiki/Datenfluss]

DIM (Distributed Information Management)

Ereignisgesteuerte Kommunikationsschnittstelle für verteilte Systeme, welches 1990 am CERN entwickelt wurde. Es basiert auf der Client-Server Architektur, d.h. ein Server publiziert Dienste, die eine beliebige Anzahl an Clients in Anspruch nehmen können.
[siehe http://dim.web.cern.ch/dim/]

EPICS (Experimental Physics and Industrial Control Systems)
Sammlung von Quellen offenen Software-Werkzeugen und Bibliotheken, entstanden aus einer gemeinschaftlichen Entwicklung, die weltweit für den Entwurf von verteilten Kontrollsystemen für wissenschaftliche Experimentaufbauten, wie Teilchendetektoren und Teleskope genutzt wird. Aber auch kommerzielle Systeme wie z.B. Ölförderungsanlagen benutzen EPICS. Obwohl die Entwicklung von EPICS an mehreren Orten von verschiedenen Gruppen vorangetrieben wird, wurde der Grundstein für das extrem flexible System am National Laboratory in Los Alamos gelegt.
[siehe http://www.aps.anl.gov/epics/]

Framework
Begriff aus der Softwaretechnik, der insbesondere im Rahmen der Objektorientierten Softwareentwicklung sowie bei Komponentenbasierten Entwicklungsansätzen verwendet wird. Das Framework definiert die Schnittstellen und den Kontrollfluss der Anwendung. Ziel eines Frameworks ist die Wiederverwendung von Modulen, die durch das Framework definiert sind und dem Anwender als Vorlage dienen sollen.
[siehe http://de.wikipedia.org/wiki/Framework]

LabVIEW (Laboratory Virtual Instrument Engineering Workbench)
Grafische Programmiersprache von National Instruments, die häufig in der Mess- und Automatisierungstechnik zur Datenaufnahme und -verarbeitung verwendet wird. Die grafische Programmierung erfolgt mit der Programmiersprache G, die auf dem Modell des Datenflusses basiert und die Programmierung nebenläufiger Verarbeitung mit geringem Aufwand ermöglicht.
[siehe http://www.ni.com/labview/whatis/]

mehrfach verkettete Liste
Verkettete Listen gehören zu den dynamischen Datenstrukturen, die eine Speicherung von einer im Vorfeld nicht bestimmten Anzahl von miteinander in Beziehung stehenden Werten einfacher oder zusammengesetzter Datentypen erlauben. Im Falle einer einfach verketteten Liste verweist ein Element immer auf das Nachfolgende. Mehrfachverkettung entsteht, wenn ein Listenelement mehrere Verweise auf andere Listenelemente verwaltet, zum Beispiel auf das vorausgehende und das nachfolgende Element.
[http://de.wikipedia.org/wiki/Liste_(Datenstruktur)]

Multithreading
Fähigkeit einer Hard- oder Software Aktivitäten, auch threads genannt, parallel zueinander ausführen zu können. Im Falle der Software spricht man von quasi paralleler Ausführung oder Nebenläufigkeit, da solange nur ein Prozessor beteiligt ist, nur eine zeitliche Aufteilung der Prozessornutzung durch die einzelnen Aktivitäten stattfindet Die hierbei entstehenden Thread-Wechsel führen zu einer Reduzierung der Systemleistung, solange keine Hardware beteiligt ist, die das Multithreading unterstützt.
[siehe http://de.wikipedia.org/wiki/Multithreading]

OOP (Objektorientierte Programmierung)
Anwendung der Konzepte Klasse, Objekt, Attribut, Methode, Vererbung, Polymorphismus und dynamisches Binden bei der Programmierung [Balzert2005].

OPC (Openness Productivity Collaboration)
Standardisierte Software-Schnittstelle, die es Anwendungen unterschiedlichster Hersteller ermöglicht, Daten auszutauschen.

PHELIX (Peta Watt High Enery Laser for Heavy Ion Experiments)
Die im Aufbau befindliche Konstruktion einer Höchstleistungs-Laseranlage ist ein gemeinsames Projekt der Atomphysik und der Plasmaphysik an der Gesellschaft für Schwerionenforschung. Der Aufbau des Lasersystems erlaubt erstmals die Untersuchung extremer Materiezustände und der dichten Plasmen. Die Kombination aus Hochleistungslaser und Ionenstrahlen soll die Tür zu völlig neuen Forschungsmöglichkeiten an der GSI öffnen.
[siehe http://www.gsi.de/forschung/phelix]

Prozess
Ein Prozess bezeichnet den Ablauf eines Computerprogramms auf einem Prozessor und kann mehrere threads des Betriebssystems belegen, falls dies Multithreading unterstützt.
[siehe http://de.wikipedia.org/wiki/Prozess_%28Informatik%29]
Im Rahmes des *CS* Framework ist ein `Prozess` eine Instanz einer beliebigen Unterklasse der Klasse `BaseProc`. Die grundlegende Eigenschaft eines `Prozesses` ist die Fähigkeit der Ereignisbehandlung. D.h. ein `Prozess` ist in der Lage, Ereignisse zu empfangen und auf diese zu reagieren.

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.

Semaphore

In der Informatik wird mit diesem Begriff eine Datenstruktur zur Prozesssynchronisation bezeichnet (Wikipedia).

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.

[siehe http://de.wikipedia.org/wiki/Semaphor(Informatik)]

Software Agent

Softwareeinheit, die eigenständig Entscheidung anhand definierter Regeln treffen kann, um z.B. auf äußere Veränderungen zu reagieren. Agenten können miteinander kommunizieren und im Falle intelligenter Agenten, verfügen diese sogar über die Fähigkeit aus Erfahrungen zu lernen und ihr Verhalten dementsprechend zu verändern. Mobile Agenten können ihren Ausführungsort z.B. über eine Netzwerkverbindung wechseln.

[siehe http://de.wikipedia.org/wiki/Software-Agent]

Transition

Bezeichnet einen Zustandsübergang eines Automaten.

thread

Programmaktivitäten, die nebenläufig, d.h. quasi-parallel, ausgeführt werden [Balzert2005] Seite 823.

In der Informatik ein Ausführungsstrang beziehungsweise eine Ausführungsreihenfolge der Abarbeitung der Software.

[siehe http://de.wikipedia.org/wiki/Thread_%28Informatik%29]

Im Kontext von LV bezeichnet der Begriff Teile des Quellcode, die über keine Datenabhängigkeit mit anderen Codefragmenten verbunden sind. Platziert man einfach zwei Schleifen im Blockdiagramm und verbindet diese nicht miteinander werden beide Schleifen von LabVIEW asynchron ausgeführt.

UML (Unified Modelling Language)

Notation zur grafischen Darstellung Objektorientierter Konzepte [Balzert2005][Seite145]. Die weitgehend programmiersprachenunabhängige Symbolsprache umfasst 13 verschiedene Diagrammtypen, mit denen verschiedene Aspekte eines Softwaresystems beschrieben werden können.

[siehe http://de.wikipedia.org/wiki/UnifiedModelingLanguage]

VI (Virtual Instrument)
Bezeichnung für (Unter-)Programm im Rahmen der grafischen Programmiersprache LabVIEW . Ein VI besteht aus einem Front Panel zur Erstellung von Bedienoberflächen und einem Blockdiagramm, das der Implementierung des Quellcodes dient.

virtuelle Methode
Sollen Methoden der Oberklasse in der Unterklasse modifiziert oder komplett neu definiert werden, müssen diese als virtuelle Methoden deklariert werden, damit sie von der Kindklasse überladen werden können.
Im *CS* Framework wird dieses Prinzip im Falle der Basisklassen verwendet, um Mechanismen der Oberklasse mit Code der Unterklasse auszuführen.

VI server
LabVIEW-Funktionsbibliothek, die eine Programmierschnittstelle für die Steuerung von LabVIEW und dessen VIs zur Verfügung stellt. Sie ermöglicht das Abfragen und Setzten der Eigenschaften von LabVIEW-Applikationen sowie das programmatische Aufrufen von VIs lokal oder über das Netzwerk.

VIT (Virtual Instrument 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.

Watchdog
Technischer Begriff für eine Komponente eines Systems, die Überwachungsaufgaben übernimmt. Im Fehlerfall wird diese aktiv, um wieder andere Prozesse auszulösen, die den Fehler beheben können.
[siehe http://de.wikipedia.org/wiki/Watchdog]

Anhang B

UML-Diagramme

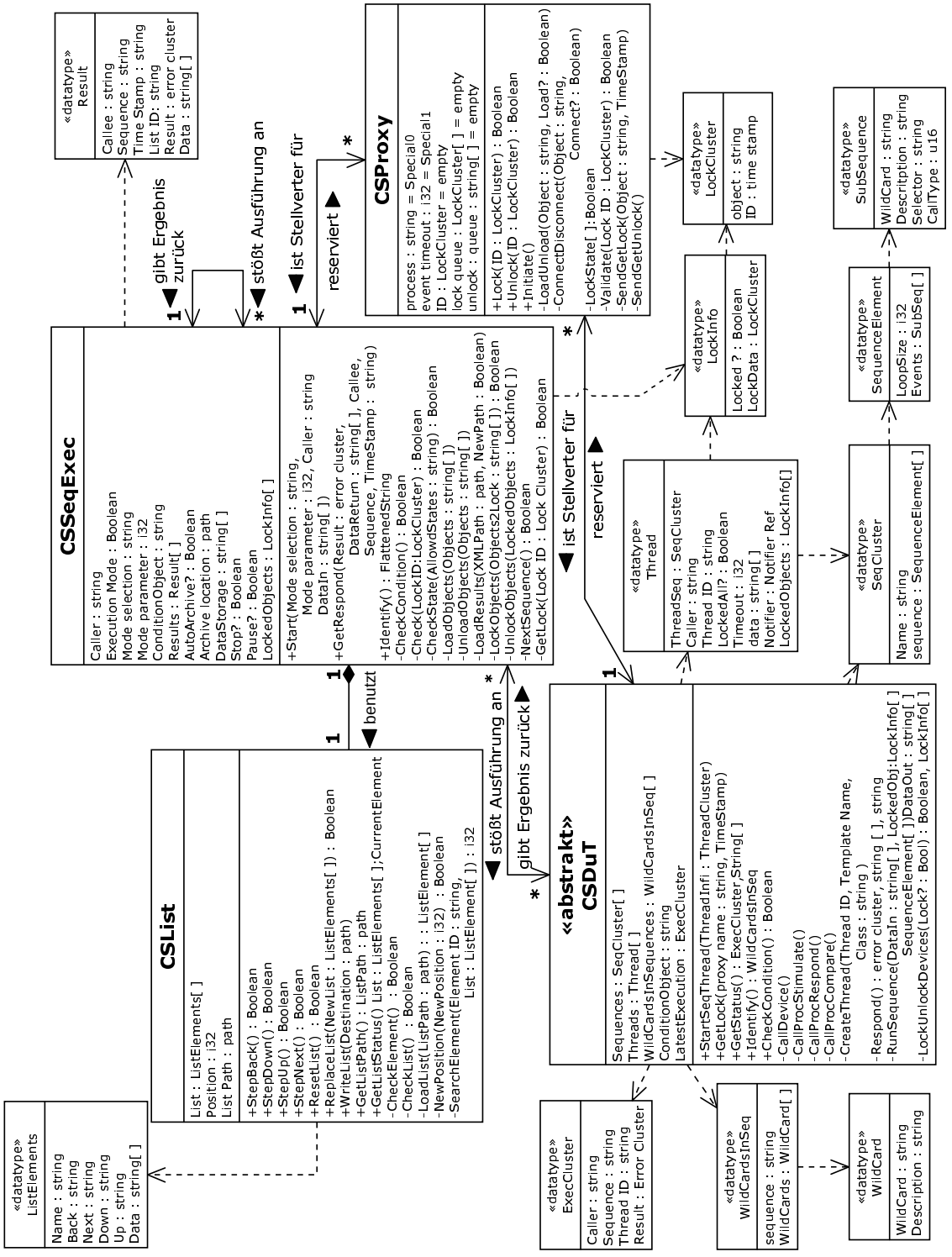


Abbildung B.1 detailliertes Klassendiagramm der Sequenzer-Klassenbibliothek

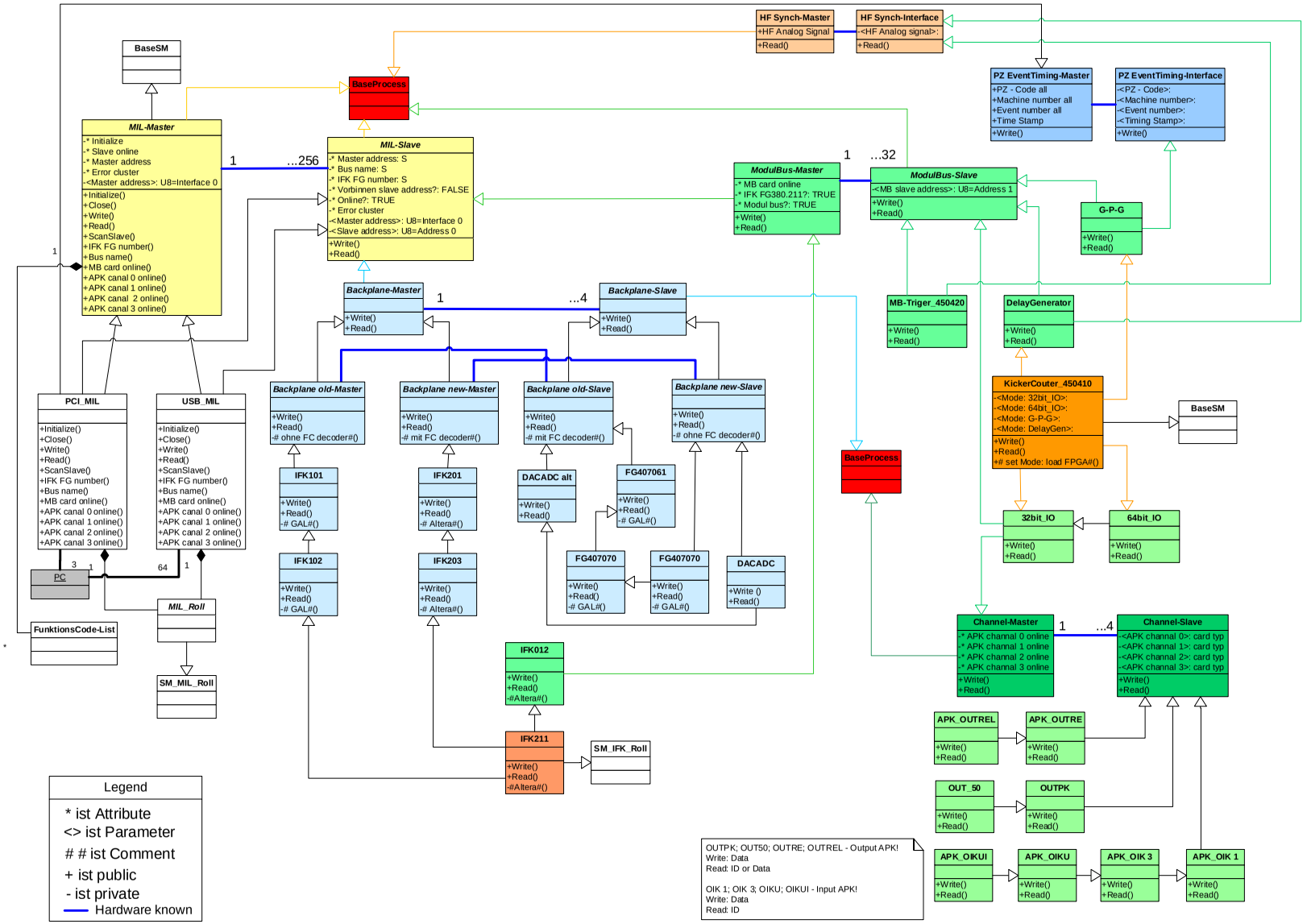


Abbildung B.2 detailliertes Klassendiagramm der BELAB-Geräteschicht [http://wiki.gsi.de/cgi-bin/view/CSframework/CSSeqPrimaryRequirements]

Literaturverzeichnis

[Balzert2005] H. Balzert
Lehrbuch Grundlagen der Informatik
2.Auflage
Spektrum Akademischer Verlag, ISBN 3-8274-1410-5

[Beck2003a] D.Beck, H.Brand
Ein allgemeines Kontrollsystem für Experimenteinrichtungen an der GSI
„Virtuelle Instrumente in der Praxis 2003"
München, Germany, Herausgeber R. Jamal and H. Jaschinski, ISBN 3-7785-2908-0 (2003) 30-34

[Beck2003b] D.Beck, H.Brand, F.Herfurth
CS - A Control System Framework for Experiments (not only) at GSI
„The IX International Conference on Accelerator and Large Experimental Physics Control Systems"
ICALEPCS 2003, Gyeongju, Korea

[Beck2005] D.Beck, H.Brand
Status and Developement of the CS Control System Framework
„SEI Frühjahrstagung 2005"
Darmstadt, Germany, Herausgeber F. Wulf

[Beck2005] D. Beck
Anwendung des allgemeinen Kontrollsystems *CS* Framework auf eine flexible Schrittmotorsteuerung an der GSI
„Virtuelle Instrumente in der Praxis 2004"
Fürstenfeldbruck, Germany, Herausgeber R. Jamal and H. Jaschinski, ISBN 3-7785-2932-3, 250-254

[Brand] H.Brand
CS-Kurshandbuch
Version 1.2
Ausgabe November 2005

[Wiki1] http://wiki.gsi.de/cgi-bin/view/CSframework/CSObjectOrientation
(Stand:27.06.2006)

[Wiki2] http://wiki.gsi.de/cgi-bin/view/CSframework/CSDocuments
(Stand:29.07.2006)

[BELAB1] http://www-wnt.gsi.de/belab/
(Stand:21.07.2006)

[BELAB2] Device-Bus und Geräte-Interfacing
G. Englert/T. Györffy/R.Werkmann
Beschleuniger Elektronik Labor
24.09.97

[BELAB3] Modulbus Manual FG 450.xxx
G.Englert/T.Györffy/R.Werkmann
Beschleuniger Elektronik Labor
18.06.98