IAS-Forschungsberichte

Forschungsbericht
Institut für Automatisierungs- und Softwaretechnik

Hrsg.: Prof. Dr.-Ing. Dr. h. c. P. Göhner

Stjepan Dujmović

Anwendungsentwicklung mit Komponenten-Frameworks in der Automatisierungstechnik

Band 1/2002

Universität Stuttgart

Anwendungsentwicklung mit Komponenten-Frameworks in der Automatisierungstechnik

Von der Fakultät Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde eines
Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung

Vorgelegt von
Stjepan Dujmović
aus Zagreb (Kroatien)

Hauptberichter:	Prof. Dr.-Ing. Dr. h. c. Peter Göhner
Mitberichter:	Prof. Dr.-Ing. Dr. h. c. mult. Paul J. Kühn
Tag der Einreichung:	21.11.2001
Tag der mündlichen Prüfung:	18.07.2002

Institut für Automatisierungs- und Softwaretechnik
der Universität Stuttgart

2001

IAS-Forschungsberichte

Band 1/2002

Stjepan Dujmović

Anwendungsentwicklung mit Komponenten-Frameworks in der Automatisierungstechnik

D 93 (Diss. Universität Stuttgart)

Shaker Verlag
Aachen 2002

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Dujmović, Stjepan:
Anwendungsentwicklung mit Komponenten-Frameworks in der Automatisierungstechnik / Stjepan Dujmović.
Aachen : Shaker, 2002
(IAS-Forschungsberichte ; Bd. 2002,1)
Zugl.: Stuttgart, Univ., Diss., 2002
ISBN 3-8322-0672-8

Printed in Germany.

ISBN 3-8322-0672-8

Shaker Verlag GmbH • Postfach 101818 • 52018 Aachen
Telefon: 02407 / 95 96 - 0 • Telefax: 02407 / 95 96 - 9
Internet: www.shaker.de • eMail: info@shaker.de

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Automatisierungs- und Softwaretechnik (IAS) der Universität Stuttgart.

Mein besonderer Dank gilt

Herrn Prof. Dr.-Ing. Dr. h.c. P. Göhner für die Übernahme des Hauptberichts, die Betreuung dieser Arbeit, die zahlreichen Anregungen und Diskussionen, die es ermöglicht haben, dass die Arbeit in der vorliegenden Form abgeschlossen werden konnte,

Herrn Prof. Dr.-Ing. Dr. h.c. mult. P. Kühn für die Übernahme des Mitberichts,

allen Kolleginnen und Kollegen am Institut für die angenehme Arbeitsatmosphäre und die gute Zusammenarbeit,

allen beteiligten Studenten, die durch ihre Studien- und Diplomarbeiten einen wesentlichen Anteil an der Implementierung der vorgestellten Konzepte haben und

meinen Eltern für die langjährige Unterstützung während meiner Ausbildungzeit.

Nicht zuletzt möchte ich meiner Frau Tatjana und meinen beiden Söhnen Bruno und Oliver für die Geduld, die Nachsicht und den Ansporn danken, die sie mir während der Entstehung dieser Dissertation entgegengebracht haben.

Stuttgart, Oktober 2001

Stjepan Dujmović

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ADL	**A**rchitecture **D**escription **L**anguage
API	**A**pplication **P**rogramming **I**nterface
CAN	**C**ontroller **A**rea **N**etwork
COM	**C**omponent **O**bject **M**odel
CORBA	**C**ommon **O**bject **R**equest **B**roker **A**rchitecture
DCOM	**D**istributed **C**omponent **O**bject **M**odel
DOM	**D**ocument **O**bject **M**odel
DTD	**D**ocument **T**ype **D**efinition
FIFO	**F**irst-**I**n-**F**irst-**O**ut
FIT	**F**ramework **I**nstantiation **T**ool
FML	**F**ramework **M**arkup **L**anguage
GUI	**G**raphical **U**ser **I**nterface
HTML	**H**yper**T**ext **M**arkup **L**anguage
IC	**I**ntegrated **C**ircuit
IDL	**I**nterface **D**efinition **L**anguage
MFC	**M**icrosoft **F**oundatation **C**lasses
OLE	**O**bject **L**inking and **E**mbedding
OMG	**O**bject **M**anagement **G**roup
OPC	**O**LE for **P**rocess **C**ontrol
ORB	**O**bject **R**equest **B**roker
PC	**P**ersonal **C**omputer
SLIO	**S**erial **L**ink **I**nput **O**utput
SPS	**S**peicher **P**rogrammierbare **S**teuerung
UML	**U**nified **M**odeling **L**anguage
XML	e**X**tensible **M**arkup **L**anguage

Begriffsverzeichnis

Aktor: Einheit zur Umsetzung von Stellinformation tragenden Signalen geringer Leistung in leistungsbehaftete Signale einer zur Prozessbeeinflussung notwendigen Energieform

Bibliothek: organisierte Sammlung von wiederverwendbaren Software-Einheiten, die voneinander größtenteils unabhängig sind und nach Bedarf in eigenen Programmen verwendet werden können

Document Type Definition: logische Struktur für eine Klasse von XML-Dokumenten, welche die zur Verfügung stehenden XML-Elemente, XML-Attribute und ihre zulässige Schachtelung festlegt

Echtzeit-Software: Software, deren Korrektheit nicht nur von logischen Ergebnissen abhängt, sondern auch von dem Zeitpunkt, zu dem die Ergebnisse berechnet werden

Framework: generische Lösung für einen definierten Anwendungsbereich, welche die Architektur und den Kontrollfluss darauf aufbauender Anwendungen definiert. Durch Frameworks sind viele Entwicklungsentscheidungen zugunsten einer effizienteren Anwendungsentwicklung vorweggenommen

Framework-Instanziierung: Entwicklung individueller Anwendungen auf der Grundlage eines Frameworks

Frozen Spots: invariable Bestandteile eines Frameworks, welche die Gemeinsamkeiten des Anwendungsbereichs repräsentieren

Hot Spots: variable Bestandteile eines Frameworks, welche für die Verwirklichung von Flexibilität verantwortlich sind

Idiom: Low-Level Muster (siehe Muster), das während der Implementierungsphase eingesetzt wird und spezielle Eigenschaften einer Programmierspache ausnützt

Invertierter Kontrollfluss: framework-typische Kontrollfluss-Umkehrung, bei der nicht der anwendungsspezifische Quellcode, sondern der invariable Framework-Code den globalen Kontrollfluss steuert (auch "Hollywood-Prinzip" genannt)

Komponente: vorgefertigte Software-Einheit, die bewußt für die Mehrfachverwendung ausgelegt wurde. Durch Auswahl, Konfiguration und Parametrierung entstehen aus einzelnen Komponenten vollständige Anwendungen.

Komponenten-Framework: Framework, das aus verschiedenen Komponenten und einer Frameworkbeschreibung besteht. Die Komponenten sind für die Implementierung der Funktionalität zuständig, während die Frameworkbeschreibung die Summe aller Anwendungen definiert, die aus den Kompontenen erstellt werden können.

Muster: abstrakte Lösungsvorschrift für ein typisches Problem, welches mehrfach auftritt und mit den Mitteln der Softwaretechnik gelöst werden kann

Sensor: Einheit zur Umsetzung von physikalischen oder chemischen Größen in eine elektronische Größe

Software-Architektur: wesentliches Merkmal einer komponentenbasierten Anwendung, welches durch die verwendeten Komponenten und die Verbindungsstrukturen zwischen den einzelnen Komponenten charakterisiert wird

Tag: definierte Folge von Zeichen, die den Beginn oder das Ende eines logischen Abschnitts oder eines logischen Elements in einem XML-Dokument repräsentieren

Zusammenfassung

Software spielt bereits heute eine wesentliche Rolle bei der Automatisierung von Anlagen und Produkten. Die zunehmende Komplexität der Automatisierungssoftware in Kombination mit steigenden funktionalen und qualitativen Anforderungen machen es notwendig, nach geeigneten Konzepten für die Wiederverwendung von Software in der Automatisierungstechnik zu suchen.

In der vorliegenden Arbeit wird ein Verfahren für die Wiederverwendung von Software in der Automatisierungstechnik vorgestellt, welches sowohl die Wiederverwendung von Implementierungscode als auch von domänenspezifischen Architekturen ermöglicht. Das Verfahren basiert auf der Framework-Technologie, die bezüglich der Benutzungsfreundlichkeit, der Werkzeugunterstützung und der Anwendbarkeit in der industriellen Automatisierungstechnik um wesentliche Aspekte erweitert wird. Für die Spezifikation von invariablen und variablen Framework-Elementen wird eine neue, XML-basierte Beschreibungssprache eingeführt, welche die formalisierte Dokumentation von Framework-Informationen ermöglicht. Für die Unterstützung des Entwicklers bei der frameworkbasierten Anwendungsentwicklung wird eine neues Werkzeug konzipiert und realisiert, welches den Entwickler schrittweise durch den Prozess der Anwendungsentwicklung bis hin zur lauffähigen Anwendung führt. Mit Hilfe der Beschreibungssprache für Frameworks und dem Werkzeug für die frameworkbasierte Anwendungsentwicklung wird ein effizienter Entwicklungsprozess verwirklicht, der die Bedürfnisse des Anwendungsentwicklers in besonderem Maße berücksichtigt.

Das vorgeschlagene Verfahren ermöglicht die Erstellung von individueller Automatisierungssoftware, ohne dazu weitreichende Programmierkenntnisse vorauszusetzen. Dadurch wird die Anwendungsentwicklung auch für Personengruppen zugänglich, die über detaillierte Kenntnisse des Anwendungsbereichs verfügen, ohne gleichzeitig Programmierexperten zu sein. Weiterhin kann das Verfahren zusammen mit bestehenden Komponentenmodellen eingesetzt werden, wodurch die Integration in bestehende Entwicklungsprozesse deutlich vereinfacht wird. Auf diese Weise wird ein effizienter Entwicklungsprozess definiert, der die funktionalen und qualitativen Herausforderungen zukünftiger Automatisierungssoftware bewältigen kann.

Abstract

Today, software is playing an important role for the automation of industrial plants and products. The continuously increasing complexity of automation software in combination with growing functional and qualitative requirements creates the demand for an adequate software reuse concept for industrial automation.

This thesis proposes an efficient reuse technique for industrial automation software, which allows the reuse of implementation code and domain-specific architectures. The technique is based on the framework technology which is essentially extended with respect to usability, tool support and suitability for industrial automation. For specifying constant and variable framework elements a new XML-based description language is introduced, which allows the formalized documentation of framework information. For supporting the creation of framework-based applications a new tool is designed and realized, which leads the developer stepwise through the application development process to an executable application. With the help of the framework description language and the tool for framework-based application development an efficient development process is realized emphasizing the needs of the application developer.

The proposed technique permits the creation of individual automation software without demanding thorough programming knowledge as a prerequisite. This quality makes application development approachable for people who possess extensive domain knowledge without being programming experts. Furthermore, the technique works together with current component models thus clearly simplifying integration with an existing development process. By this means an efficient development process is defined which can cope with functional and qualitative challenges of future automation software.

1 Einleitung und Motivation

> "Wenn du ein Schiff bauen willst, dann trommle nicht die Männer zusammen, um Holz zu beschaffen, Aufgaben zu vergeben und die Arbeit einzuteilen, sondern lehre die Männer die Sehnsucht nach dem weiten, endlosen Meer."
>
> (Antoine de Saint-Exupéry)

1.1 Software in der Automatisierungstechnik

1.1.1 Bedeutung der Software

In der Automatisierungstechnik nimmt Software einen immer größeren Stellenwert ein [Doug99b]. Aufgaben, die in der Vergangenheit durch spezielle Hardware gelöst wurden, werden zunehmend durch entsprechende Softwarelösungen ersetzt. Software ersetzt aber nicht nur entsprechende Hardware-Lösungen, sondern ermöglicht gleichzeitig die Realisierung von zusätzlicher Funktionalität, die mit Hardware nur sehr schwer oder kostenintensiv verwirklicht werden könnte. Bei immer mehr Produkten ist deshalb die durch Software realisierte Funktionalität das differenzierende Wettbewerbsmerkmal, während die Hardware eine eher sekundäre Rolle spielt.

Parallel zur Verlagerung von Aufgaben von der Hardware zur Software ist zusätzlich ein Trend zur Verwendung von Standard-Hardware zu beobachten [PTH97]. Gründe für diesen Trend sind vor allem in den hohen Entwicklungskosten für Spezial-Hardware zu suchen, die vor dem Hintergrund von leistungsfähiger und kostengünstiger Hardware "von der Stange" nicht mehr gerechtfertigt sind. Als Beispiel sei in der Steuerungstechnik der Trend zur Soft-SPS genannt, die konventionelle PC-Hardware verwendet und nicht auf spezialisierte Automatisierungs-rechner angewiesen ist.

Sowohl die Verlagerung der Funktionalität von der Hardware zur Software, als auch die steigende Verwendung von Standard-Hardware weisen dem Software-Entwicklungsprozess eine Schlüsselposition zu, die zunehmend dominanter wird. Bereits heute ist ein flexibler und effizienter Software-Entwicklungsprozess ein wesentlicher Wettbewerbsfaktor in der Automatisierungstechnik, der zukünftig sogar noch an Bedeutung gewinnen wird. Trotz dieser steigenden Bedeutung für die Automatisierungstechnik wird die Rolle der Software in vielen Bereichen unterschätzt. Das vollständige Potenzial flexibler Software-Lösungen ist bei weitem noch nicht ausgeschöpft. Dieses Potenzial kann mit einem flexiblen und effizienten Software-

Entwicklungsprozess mobilisiert werden, der die Wiederverwendung von Entwurfs- und Implementierungsinformationen ermöglicht.

1.1.2 Problemstellung bei der Entwicklung von Automatisierungssoftware

In den letzten Jahrzehnten wurden in Form von neuen Vorgehensmodellen, Entwicklungsmethoden und Softwarewerkzeugen wesentliche Anstrengungen unternommen, um die Entwicklung von Softwaresystemen als typische Ingenieursdisziplin zu etablieren [MAMY99]. Die industrielle Praxis hat leider gezeigt, dass die bisher erzielten Erfolge auf dem Gebiet der Softwaretechnik nicht ausreichen, um die Probleme bei der Softwareentwicklung zu bewältigen. Auch heute dominieren in vielen Bereichen der Softwareentwicklung die "künstlerischen" Aspekte, die keinen oder nur einen schwachen Bezug zur strengen Systematik typischer Ingenieursdisziplinen aufweisen.

Als Hauptursache für die Schwierigkeiten bei der Softwareentwicklung für Automatisierungssysteme ist vor allem die zunehmende Komplexität von Systemen zu nennen [Göhn98]. Während die Software in diesem Bereich zu Beginn lediglich für einen beschränkten, gut überschaubaren Funktionsumfang zuständig war, wird mittlerweile immer mehr Funktionalität durch Software realisiert. Zusätzlich ist moderne Software in der Automatisierungstechnik – anders als im Bereich der allgemeinen Datenverarbeitung – nicht nur für die reine Produktfunktionalität verantwortlich, sondern muss auch viele nichtfunktionale Eigenschaften, wie z.B. Echtzeitanforderungen oder knappe Hardware-Ressourcen, berücksichtigen [LaGö98, Thal97]. Abbildung 1.1 skizziert das Zusammenspiel der Automatisierungssoftware mit der Umgebung, die aus verschiedenen Betriebssystemen, Kommunikationsbussen und unterschiedlicher Hardware bestehen kann.

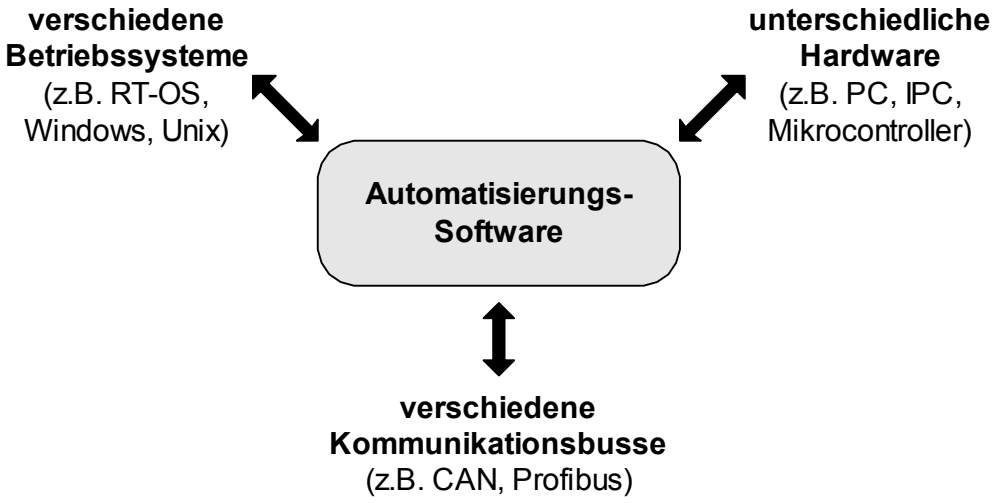

Abbildung 1.1: Automatisierungssoftware im Kontext

Diese Rahmenbedingungen führen dazu, dass die fortwährende Neuentwicklung von Software heute nicht mehr wirtschaftlich vertretbar ist. Die Neuimplementierung einer Steuerungssoftware für jede Produktvariante würde vergleichsweise der Konstruktion eines neuen Fernsehapparats aus einzelnen Kleinstbausteinen (Widerstände, Transistoren) entsprechen, ohne dabei auf vorhandene integrierte Bauelemente (sog. Integrated Circuits, ICs) zurückzugreifen. Der Software-Entwicklungsprozess muss deshalb weg von der klassischen Quellcodeprogrammierung hin zur mehrfachen Verwendung von bereits erstellten Software-Bausteinen geführt werden.

Die Motivation für die Mehrfachverwendung von Software ist dieselbe wie beim Einsatz von Normteilen in der Fertigungsindustrie. Normteile sind billiger, zuverlässiger und in der Regel leichter zu reparieren oder zu ersetzen als individuell gefertigte Bauteile. Software für Automatisierungssysteme wird deshalb in der Zukunft nicht mehr individuell programmiert werden, sondern durch Auswahl, Konfiguration und Parametrierung bereits bestehender Software-Bausteine erstellt werden. Zukünftige Anwendungsprogramme in der Automatisierungstechnik werden deshalb hauptsächlich aus wieder verwendeten Software-Bausteinen bestehen, wobei individuelle Neuimplementierungen möglichst gering gehalten werden. Aus diesem Grund sind Techniken notwendig, welche die Software-Wiederverwendung in der Automatisierungstechnik unterstützen.

1.1.3 Frameworks in der Automatisierungstechnik

Im Bereich der allgemeinen Datenverarbeitung wurden in den letzten Jahren eine Reihe unterschiedlicher Konzepte für die Mehrfachverwendung von Software vorgestellt. Eine besondere Form der komponentenbasierten Softwareentwicklung ist der Einsatz von objektorientierten Frameworks [JoFo88, Roge97a]. Durch Frameworks werden viele Entwurfsentscheidungen bereits vorweggenommen und die Möglichkeit, als Entwickler Einfluss zu nehmen und damit Fehler zu machen, ist reduziert. Im Gegensatz zur allgemeinen komponentenbasierten Entwicklung, bei der unabhängige Komponenten wieder verwendet werden, sind bei Frameworks die einzelnen Komponenten nicht unabhängig voneinander, sondern Bestandteile einer fest vorgegebenen, übergeordneten Architektur, die für diesen speziellen Anwendungsbereich zugeschnitten ist [FaJo99].

Frameworks bieten sich vor allem für die Bereiche der Automatisierungstechnik an, die durch eine große Variantenvielfalt gekennzeichnet sind [DaKn96, GDH00, Moli97]. Statt für jede Variante neue Software zu entwickeln bzw. aus Komponenten neu aufzubauen, ist es auch auf Kosten geringerer Flexibilität günstiger, lediglich die Framework-Bestandteile flexibel zu halten, die zur Verwirklichung von verschiedenen Varianten unbedingt notwendig sind. Alle übrigen Bestandteile, die in allen resultierenden Anwendungen unverändert wiederkehren (z.B. Architekturen, Funktionalität), können als feste Bestandteile des Frameworks definiert werden.

Obwohl Frameworks einen effizienten Software-Entwicklungsprozess und eine deutliche Produktivitätssteigerung ermöglichen, werden diese Vorteile in der Regel durch Schwierigkeiten bei der Anwendungsentwicklung erkauft [BBE95, MBF99]. Dem Benutzer aus der Automatisierungstechnik bleibt diese Technologie normalerweise verschlossen, da sie einen hohen initialen Einarbeitungsaufwand erfordert, ohne ihm sofort den resultierenden Nutzen zu offenbaren.

1.2 Ziel der Unterstützung der frameworkbasierten Anwendungsentwicklung

Aus der Problemstellung der frameworkbasierten Softwareentwicklung in der Automatisierungstechnik leitet sich die Zielsetzung für die vorliegende Arbeit ab: Durch die Bereitstellung geeigneter Konzepte und Verfahren soll die Wiederverwendung in der Automatisierungstechnik unterstützt und die frameworkbasierte Anwendungsentwicklung einfacher und effizienter gestaltet werden. Es sollen Konzepte aufgezeigt werden, welche innerhalb der Grenzen definierter Anwendungsbereiche individuelle Software-Lösungen ermöglichen, die sowohl eine Variantenvielfalt im Detail, als auch übergeordnete Gemeinsamkeiten des dedizierten Anwendungsbereichs umfassen. Bei dieser Zielsetzung haben folgende Aspekte eine besondere Bedeutung.

Anwendbarkeit für den industriellen Anwender

Das in dieser Arbeit vorgestellte Verfahren soll besonders die Bedürfnisse des industriellen Anwendungsentwicklers berücksichtigen, der mit Hilfe eines Frameworks individuelle Anwendungen erstellen möchte. Dieser Anwendungsentwickler wird normalerweise Expertenwissen innerhalb seines Fachgebiets besitzen, ohne gleichzeitig Programmierexperte zu sein. Um diese Anwendergruppe ausreichend zu unterstützen, hat die Einfachheit des Verfahrens hohe Priorität – Die Fragestellung "Was soll entstehen" und nicht das "Wie soll es entstehen" steht im Vordergrund dieser Arbeit. Diese Anforderung ist insofern wichtig, da nur verhältnismäßig wenige Experten im Bereich der Automatisierungstechnik gleichzeitig auch Software-Experten sind [HoRi99].

Verträglichkeit mit bestehenden Technologien

Jedes neuartige Konzept, welches konsequent die Abkehr von bisherigen Techniken und Methoden fordert, wird zwangsläufig mit Akzeptanzschwierigkeiten zu kämpfen haben. Aus diesem Grund sollen in der vorliegenden Arbeit bestehende Technologien weitestgehend berücksichtigt und in das vorgeschlagene Konzept möglichst nahtlos integriert werden.

Unterstützung nichtfunktionaler Anforderungen

In vielen Bereichen der Automatisierungstechnik spielen nichtfunktionale Anforderungen eine wichtige Rolle für die Funktionsfähigkeit des Gesamtsystems. Dazu gehören beispielsweise Echtzeitanforderungen, Speicherplatzeffizienz und besondere Anforderungen an die Sicherheit und Zuverlässigkeit von Automatisierungssystemen. Ein Verfahren für die Automatisierungstechnik muss diese Besonderheiten auf angemessene Weise berücksichtigen.

Zur Verwirklichung dieser Ziele soll die aus der Softwaretechnik bekannte Frameworktechnologie zugrunde gelegt werden, durch neue Konzepte ergänzt und in Richtung Benutzungsfreundlichkeit für den industriellen Anwender erweitert werden. Das Ergebnis dieser Arbeit soll ein effizientes, werkzeuggestütztes Verfahren für die frameworkbasierte Anwendungsentwicklung in der Automatisierungstechnik sein.

1.3 Abgrenzung zu ähnlichen Themen

Die hier vorgestellten Ideen und Konzepte liegen an der Grenze zu weiteren wichtigen Themenbereichen, die jedoch bewusst nicht weiter vertieft werden. Aus diesem Grund sollen diese verwandten Bereiche kurz genannt und auf weiterführende Literatur verwiesen werden.

- *Domain Engineering*: Wichtige Fragen, die sich in der Entwicklungsphase von Frameworks stellen, sind beispielsweise die Abgrenzung des betrachteten Anwendungsbereichs und die Unterteilung des Bereichs in einzelne Komponenten mit einem definierten Funktionsumfang. Diese Fragen werden von verschiedenen Domain Engineering Methoden aufgegriffen, die dafür Antworten präsentieren [CzEi00, Jost00]. Diese Arbeit geht davon aus, dass diese entwicklungsbezogenen Aspekte bereits gelöst sind und die Anwendungsentwicklung mit Frameworks im Vordergrund steht.

- *Komponentenmodelle*: Diese Arbeit setzt auf bestehenden Komponentenmodelle auf, die verbreitet sind und auch in der Automatisierungstechnik eine entsprechende Relevanz besitzen. Es wird nicht versucht, ein neues Komponentenmodell zu entwickeln, welches im Vergleich zu den bestehenden Modellen in engen Bereichen der Automatisierungstechnik bessere Eigenschaften aufweisen können [GuNä99, PSW99, SCSP95]. Obwohl dieses Thema nicht in Widerspruch zu den in dieser Arbeit vorgeschlagenen Konzepten steht, wird dieser Weg nicht weiter verfolgt.

1.4 Gliederung der Arbeit

Im zweiten Kapitel werden Konzepte und Verfahren vorgestellt, die im Bereich der modernen Softwaretechnik zur Unterstützung der Wiederverwendung eingesetzt werden und im Wesentlichen den Stand der Forschung auf diesem Gebiet wiedergeben. Es wird weiterhin gezeigt, dass Frameworks eine besonders effiziente Basistechnologie für die Unterstützung der Wiederverwendung darstellen.

Kapitel 3 erläutert die Probleme und Schwierigkeiten, die aus heutiger Sicht an die Framework-Technologie gekoppelt sind. Ein besonderes Augenmerk wird dabei auf die Bedürfnisse des Anwendungsentwicklers gelegt, der mit dem Framework individuelle Anwendungen erstellen möchte.

Auf der Basis der in Kapitel 3 aufgezeigten Schwierigkeiten wird in Kapitel 4 das Konzept der Komponenten-Frameworks eingeführt und erläutert. Komponenten-Frameworks verbinden die Vorzüge der Framework- und der Komponenten-Technologie und bieten eine gute Ausgangslage für die werkzeuggestützte Anwendungsentwicklung.

Kapitel 5 stellt das Verfahren der werkzeuggestützten Anwendungsentwicklung auf der Basis von Komponenten-Frameworks vor. Dabei wird das vorgeschlagene Verfahren diskutiert und alle Kernelemente des Verfahrens vorgestellt.

Im Kapitel 6 wird detailliert auf die verwendete Spezifikationssprache eingegangen, die zur Beschreibung von Komponenten-Frameworks eingesetzt wird und die als Grundlage für die Werkzeugunterstützung von Frameworks dient. In diesem Kapitel werden die wichtigsten Elemente der Spezifikationssprache eingeführt und diskutiert.

Kapitel 7 behandelt ausführlich das Werkzeug, das den Anwendungsentwickler bei der Arbeit mit Komponenten-Frameworks unterstützt. Es wird sowohl der Entwurf als auch die Realisierung des Werkzeuges behandelt.

Zur Veranschaulichung der vorgeschlagenen Ideen und Konzepte wird in Kapitel 8 das vorgeschlagene Verfahren auf ein konkretes Beispiel angewendet. Als Beispiel wurde die Steuerungssoftware für Aufzugsysteme gewählt. Es wird das realisierte Komponenten-Framework für Aufzugsysteme vorgestellt und die Anwendungsentwicklung mit Hilfe des Instanziierungswerkzeuges gezeigt.

Abschließend werden in Kapitel 9 die zentralen Konzepte zusammengefasst und weiterführende Arbeits- und Forschungsbereiche aufgezeigt.

2 Anwendungsentwicklung durch Wiederverwendung

In diesem Kapitel werden verschiedene Konzepte präsentiert, die zur Wiederverwendung bei der Anwendungsentwicklung dienen. Alle vorgestellten Konzepte sind in der aktuellen Programmierpraxis verankert und geben den Stand der Technik auf dem Gebiet der Wiederverwendung bei der Anwendungsentwicklung wieder. Besonders ausführlich wird die Framework-Technologie vorgestellt, die dem Anwendungsentwickler eine besonders weit reichende Unterstützung bietet und in der hier vorgestellten Arbeit eine zentrale Rolle spielt. Abschließend werden verschiedene Bewertungskriterien eingeführt und auf die vorgestellten Wiederverwendungskonzepte angewendet. Mit Hilfe dieser Kriterien werden die Konzepte miteinander verglichen und die Unterstützung des Programmierers bei der Anwendungsentwicklung beurteilt.

2.1 Bibliotheken

2.1.1 Anwendungsentwicklung mit Bibliotheken

In der Implementierungsphase einer Anwendung benötigt der Entwickler häufig allgemeine Funktionen, die nicht im Kern der verwendeten Programmiersprache angeboten werden, aber trotzdem unabhängig von der zu erstellenden Anwendung sind. Dazu gehören beispielsweise mathematische Funktionen (z.B. Sinus, Cosinus) oder Funktionen zur Bearbeitung von Zeichenketten. Dieser Bedarf an allgemeinen Funktionen wird in der Regel durch Softwarebibliotheken abgedeckt, die eine besonders verbreitete Form der einfachen Wiederverwendung darstellen.

Eine *Bibliothek* (eng. Library) ist eine organisierte Sammlung von Software-Einheiten (z.B. Unterprogramme, Module, Klassen), die voneinander größtenteils unabhängig sind und die der Entwickler nach Bedarf in seinem Programm verwenden kann [Balz96, RePo97]. Die einzelnen Elemente innerhalb der Bibliothek implementieren eine bestimmte Funktionalität, auf die der Entwickler zurückgreifen kann, ohne die entsprechende Funktionalität selbst implementieren zu müssen. Gemeinsames Merkmal aller Bibliothekselemente ist, dass der Kontrollfluss vom benutzenden Kontext ausgeht, d.h. die angebotene Funktionalität unabhängig vom Einsatzkontext ist. Der Anwendungsentwickler ist auch weiterhin dafür zuständig, den globalen Kontrollfluss und die Architektur seiner Anwendung festzulegen. Bibliotheken betonen somit die Wiederverwendung von Quellcode, ohne dadurch Entwurfs- oder Architekturmerkmale der Anwendung vorwegzunehmen. Im Gegensatz zu anderen Konzepten besitzen Bibliotheken eine sehr feine Granularität, da hauptsächlich einzelne Funktionen mit einer überschaubaren Anzahl von Quellcodezeilen wieder verwendet werden.

2.1.2 Einsatzbereiche

Alle modernen Sprachen sind mit passenden Standard-Bibliotheken ausgestattet, welche die Sprachen um allgemein benötigte Funktionen (z.B. Manipulation von Zeichenketten, mathematische Funktionen) erweitern und dadurch die Anwendungsentwicklung auf der Quellcode-Ebene erleichtern. Dies gilt unabhängig davon, welches Paradigma die Sprache unterstützt (z.B. prozedural, objektorientiert). Beispiele für solche Sprachen sind C [KeRi88], C++ [Strou97], Java [GJSB00] oder Tcl [Welc97]. Bibliotheken müssen nicht zwangsläufig als Quellcode vorliegen, sondern können auch als übersetzter Binärcode bereitgestellt werden. In objektorientierten Bibliotheken werden die Dienstleistungen oft zu *Application Programming Interfaces* (API) zusammengefasst, die sowohl die direkte Verwendung der enthaltenen Klassen oder die Redefinition der Funktionalität mit Hilfe von Vererbung ermöglichen. Abbildung 2.1 zeigt in Anlehnung an [Balz96] verschiedene typische Einsatzbereiche für Bibliotheken.

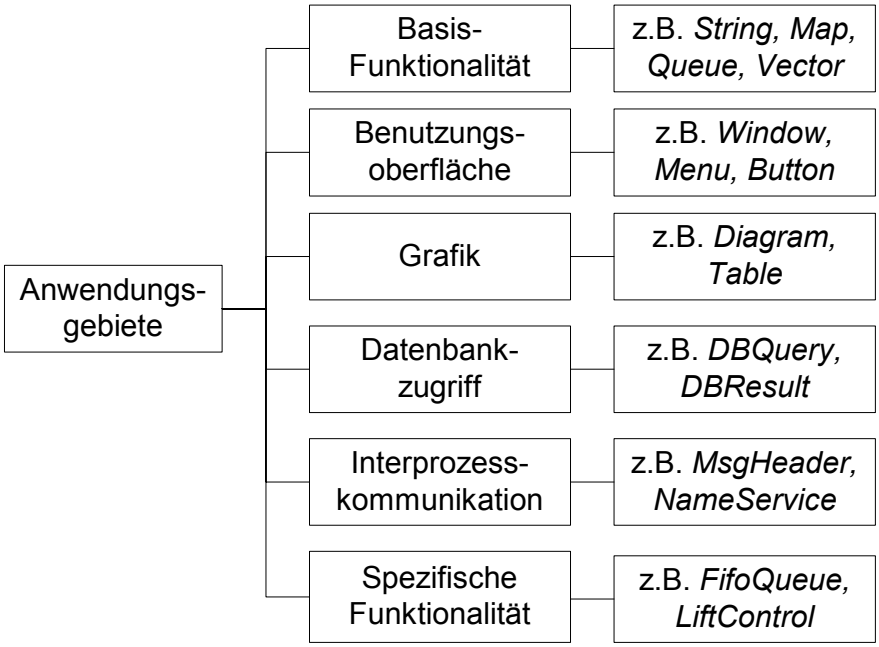

Abbildung 2.1: Verschiedene Einsatzbereiche für Bibliotheken

2.2 Muster

2.2.1 Anwendungsentwicklung mit Mustern

In etablierten Disziplinen wie dem Bauingenieurwesen oder dem Maschinenbau spielen Erfahrungswerte eine wichtige Rolle. Diese Erfahrungen repräsentieren erfolgreiche und erprobte Lösungen, die für bestimmte Problembereiche eingesetzt werden können [Alex80]. Mit Hilfe dieser Erfahrungswerte sind beispielsweise Lösungen möglich, für die es keine oder nur

sehr aufwändige numerische Berechnungen gibt. Auch im Software-Entwurf spielt die Wiederverwendung von Erfahrungen ein große Rolle, da die erneute Lösungsfindung für bereits gelöste Probleme vermieden werden soll. Ziel ist die systematische Entwicklung von Katalogen, die erfolgreiche Entwürfe dokumentieren und zur Wiederverwendung bereitstellen.

Unter einem *Muster* (eng. Pattern) versteht man im Bereich der Softwaretechnik eine abstrakte Lösungsvorschrift für ein typisches Problem, welches mehrfach auftreten kann [GHJV95, Coad95, BMR+96]. Das Muster stellt dabei die Problemlösung nicht als konkrete Implementierung bereit, sondern vermittelt das Erfahrungswissen, wie dieses Problem möglichst elegant zu lösen ist. Dieses Wissen wird mit Hilfe von Diagrammen, textuellen Erläuterungen und Beispielen vermittelt. Aus einem Muster können somit immer wieder neue, an konkrete Problemstellungen angepasste Lösungsvarianten entstehen. Muster sind ein Mittel, um Erfahrung zu dokumentieren, so dass andere Entwickler davon profitieren können. Im Gegensatz zu Bibliotheken wird nicht Quellcode, sondern die abstrakte Lösungsidee für ein bestimmtes Problem wieder verwendet.

Damit Muster systematisch gesammelt und katalogisiert werden können, ist ein Dokumentationsformat notwendig, welches das Auffinden und den Einsatz von Mustern möglichst gut unterstützt. Jedes Muster besteht aus vier wesentlichen Elementen, die sich in jeder Musterbeschreibung wieder finden:

- *Mustername*: Der Name des Musters soll den Kern des Problems in ein oder zwei Worten festhalten. Dieser Name wird zum Bestandteile des Entwurfsvokabulars und dient zur Kommunikation zwischen den Entwicklern.

- *Problembeschreibung*: Ziel der Problembeschreibung ist die Klärung der Frage, in welchem Kontext das Muster angewendet werden soll. Dieses Ziel kann entweder durch eine ausführliche Beschreibung des Problems oder durch Auflistung von Vorbedingungen erreicht werden, die als Voraussetzung für die Anwendung des Musters erfüllt sein müssen.

- *Problemlösung*: Die Problemlösung stellt die Lösungselemente vor und beschreibt die Beziehungen, die Zuständigkeiten und das Zusammenspiel der einzelnen Elemente. Dabei wird keine konkrete Implementierung vorgegeben, sondern ein abstrakter Lösungsweg, der in verschiedenen ähnlichen Situationen anwendbar ist.

- *Konsequenzen*: Die Verwendung eines Musters ist immer mit bestimmten Auswirkungen verbunden. Diese Auswirkungen müssen beschrieben werden, damit der Entwickler Alternativen erörtern kann oder den Nutzen des Musters im Verhältnis zu den Kosten (z.B. Laufzeit und Speicherbedarf) abwägen kann.

Die ersten Muster bzw. Mustersammlungen waren schwerpunktmäßig im Bereich der allgemeinen objektorientierten Softwareentwicklung angesiedelt. Inzwischen gibt es auch eine Reihe von Mustern für spezielle Anwendungsgebiete, wie z.B. für die Entwicklung von parallelen, verteilten Systemen [LaSc95, ABM96] und Echtzeitsystemen [Doug99a]. Im Nachfolgenden sollen die unterschiedlichen Typen von Mustern vorgestellt werden, die bei der Softwareentwicklung zum Einsatz kommen. Abbildung 2.2 zeigt die Zuordnung verschiedener Mustertypen zur jeweiligen Phase im Entwicklungsprozess.

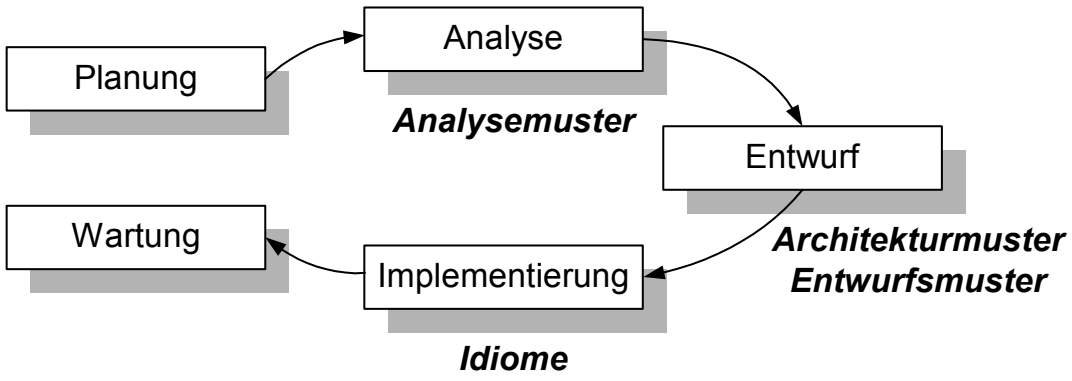

Abbildung 2.2: Mustertyp-Zuordnung im Entwicklungsprozess

2.2.2 Analysemuster

Analysemuster werden – wie der Name bereits vermuten lässt – in der Analysephase des Software-Entwicklungsprozesses eingesetzt. Sie nützen die Tatsache aus, dass in vielen Anwendungsbereichen die konkreten Aufgabenstellungen sehr ähnlich sind [Fowl97]. Analysemuster stellen vorgegebene Modelle dar, mit denen wiederkehrende Anforderungen an zu entwickelnde Systeme dargestellt werden können.

Analysemuster helfen bei der Erstellung von Analysemodellen ohne die Anwendungsarchitektur festzulegen. Sie stellen jedoch eine wichtige Basis für die Auswahl geeigneter Architekturmuster dar.

2.2.3 Architekturmuster

Der Systementwurf wird von Architektur- und Entwurfsmustern unterstützt. *Architekturmuster* dienen als fundamentales Organisationsschema zur Beschreibung von grobgranularen Software-Strukturen. Sie stellen eine Sammlung von vordefinierten Subsystemen bereit, legen deren Verantwortung fest und fügen Regeln und Richtlinien für die Organisation der Beziehungen zwischen den Subsystemen hinzu [BMR+96]. Jede spätere Verfeinerung des Entwurfs wird durch die vom gewählten Architekturmuster festgelegte Software-Struktur beeinflusst.

Das bekannteste Beispiel ist das Model-View-Controller Muster [KrPo88], das eine Struktur für interaktive Software-System mit einer klaren Trennung zwischen Datenhaltung (Model), Datendarstellung auf dem Bildschirm (View) und Datenmanipulation (Controller) vorsieht (siehe Abbildung 2.3).

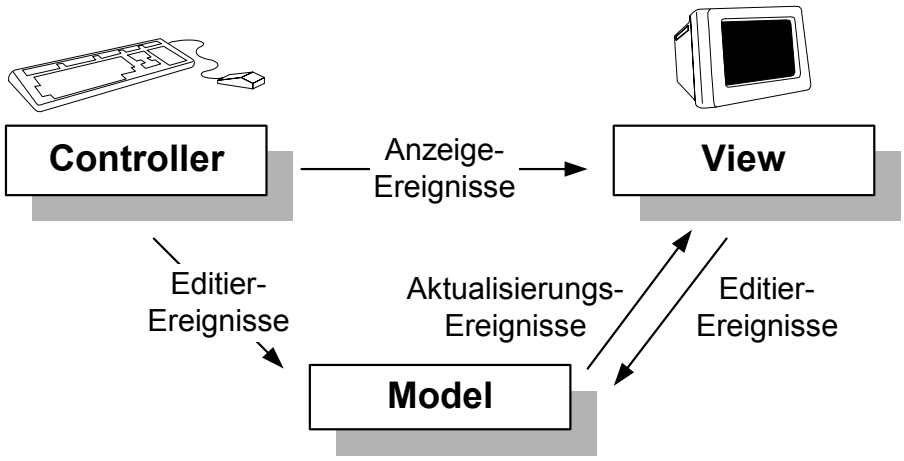

Abbildung 2.3: Model-View-Controller Architekturmuster

2.2.4 Entwurfsmuster

Entwurfsmuster werden für den Detailentwurf eines Systems verwendet. Sie verfeinern, erweitern und präzisieren die grundlegende Architektur einer Anwendung, wie z.B. die verwendeten Kommunikationsmechanismen zwischen Subsystemen oder den internen Aufbau von Subsystemen [GHJV95].

Der wesentliche Unterschied zu Architekturmustern besteht in der Konzentration auf Details des Anwendungsentwurfs. Während das Architekturmuster die grobe Systemstruktur vorgibt, müssen mehrere Entwurfsmuster eingesetzt werden, um diese Struktur zu präzisieren. Durch ihre Konzentration auf Entwurfsdetails sind Entwurfsmuster im Gegensatz zu Architekturmustern viel unabhängiger von einem bestimmten Anwendungsbereich. Wichtig ist lediglich, dass ein gewähltes Entwurfsmuster in den strukturellen Kontext passt und mit der Aufgabenstellung harmoniert.

2.2.5 Idiome

Idiome sind Codier-Muster, die während der Implementierungsphase eingesetzt werden und speziell auf eine Programmiersprache ausgerichtet sind. Sie beschreiben, wie einzelne Merkmale einer bestimmten Programmiersprache genutzt werden können, um qualitativ besseren Code zu erhalten [Cope92]. Idiome sind Codierrichtlinien, die den Anwendungsentwickler bei der Umsetzung des Entwurfs in den Quellcode unterstützen sollen.

Im Gegensatz zu Entwurfsmustern sind Idiome auf die Implementierungsphase beschränkt und immer spezifisch an eine Programmiersprache gebunden. Für die Programmiersprache C++ finden sich in [Cope92] zahlreiche Beispiele für C++-Idiome.

2.3 Komponenten

2.3.1 Anwendungsentwicklung mit Komponenten

In der Softwaretechnik bestehen seit Jahrzehnten Bestrebungen, Anwendungen nicht mehr individuell zu programmieren, sondern aus vorgefertigten, mehrfach verwendbaren Bausteinen herzustellen [McIl76]. In Analogie zur modernen Elektronik, wo Schaltungen industriell nicht mehr aus einzelnen Widerständen, Kondensatoren und Transistoren aufgebaut werden, sondern aus universellen Bausteinen (sog. Integrated Circuits, ICs) entstehen, so soll auch Anwendungssoftware nicht mehr aus einzelnen Quellcodezeilen sondern aus komplexeren Software-Komponenten erstellt werden. Die einzelnen Komponenten werden miteinander "verknüpft" und bilden auf diese Weise eine vollständige Anwendung. Abbildung Abbildung 2.4 veranschaulicht die komponentenbasierte Softwareentwicklung mit den vorgefertigten Komponenten auf der linken Seite und der fertigen Anwendung auf der rechten Seite, die durch Auswahl, Konfiguration und Verknüpfung von entsprechenden Komponenten entstanden ist.

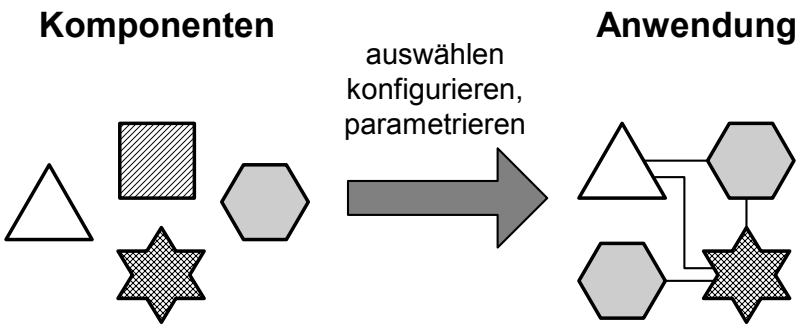

Abbildung 2.4: Komponentenbasierte Anwendungsentwicklung

Analog zum Einsatz von Komponenten bei der Hardwareentwicklung verspricht der Einsatz von Softwarekomponenten eine Reihe von Vorteilen bei der Softwareentwicklung:

- *Wiederverwendbarkeit*: einmal erstellte Komponenten können mehrfach in unterschiedlichsten Anwendungen verwendet werden.
- *Kostensenkung*: Durch die mehrfache Verwendung von Komponenten wird der Implementierungsaufwand gesenkt, was gleichzeitig eine Kostensenkung bewirkt.

- *Qualitätssteigerung*: Für die Mehrfachverwendung entwickelte Komponenten bieten eine höhere Qualität als individuell entwickelte Software, da sie mehrfach verwendet werden und somit einen höheren Entwicklungs- und Testaufwand rechtfertigen.

- *Austauschbarkeit*: Komponenten können einfach gegen andere Komponenten ausgetauscht werden, ohne dass die übrigen Bestandteile der Anwendung davon betroffen sind.

Neben den aufgeführten Vorteilen darf natürlich nicht verschwiegen werden, dass die Erstellung von mehrfach verwendbaren Softwarekomponenten einen erhöhten Aufwand im Vergleich zu individuell implementierter Software bedeutet. Eine Softwarekomponente, die in verschiedenen Anwendungsumgebungen funktionieren soll, muss mit großer Sorgfalt geplant und realisiert werden. Dieser Aufwand kompensiert sich jedoch schnell durch die Wiederverwendung der Komponente – bereits bei der dritten Verwendung sollte der erhöhte initiale Aufwand vollständig kompensiert sein [Bigg94, Trac95].

Auf dem Gebiet der allgemeinen Informationsverarbeitung wurden in den letzten Jahren eine Reihe unterschiedlicher Konzepte für die komponentenbasierte Entwicklung aufgestellt. Viele dieser Konzepte verstehen unter einer Komponente nicht zwangsläufig ausführbaren

Code – eine Komponente kann auch eine Beschreibung sein, aus der Code generiert werden kann [GuNa99] bzw. jedes wesentliche Ergebnis des Software-Entwicklungsprozesses, wie z.B. eine Anforderungsspezifikation oder Testfälle [Flei99]. In der Praxis sind jedoch vor allem die so genannten *Componentware*-Plattformen [Udel94a] relevant, die eine Softwarekomponente als unabhängige binäre Einheit mit definierter Funktionalität verstehen. Die Componentware-Plattform übernimmt dabei die Aufgabe, die Infrastruktur festzulegen und die Interaktion der einzelnen Komponenten untereinander zu ermöglichen. Im Folgenden werden die Komponentenmodelle der wichtigsten Componentware-Vertreter eingeführt und erläutert.

2.3.2 COM/DCOM

Das Komponentenmodell *COM* (Component Object Model) der Firma Microsoft stellt einen proprietären, sprachunabhängigen, binären Standard für die Implementierung von Komponenten zur Verfügung [Roge97]. Binärer Standard bedeutet, dass zwei unabhängig voneinander übersetzte Komponenten zur Laufzeit zuverlässig zusammenarbeiten können. Der COM-Standard betrachtet eine Komponente als eine Menge von Funktionsimplementierungen, die mit Hilfe von einer oder mehreren Schnittstellen (Interface) nach außen hin angeboten werden. Zusätzlich bietet COM die Möglichkeit, neue Komponenten aus bestehenden zusammenzusetzen und definiert, wie Komponenten auf Schnittstellen zugreifen, wie Fehlercodes mitgeteilt und Parameter übergeben werden. Mit Hilfe der *IUnknown*-Schnittstelle, die von jeder Komponente implementiert werden muss, können sogar zur Laufzeit die verfügbaren Schnittstellen einer Komponente ermittelt werden. Die Verwendung einer COM-Komponente

geschieht ausschließlich über ihre Schnittstellen, d.h. ein direkter Zugriff auf die Implementierung ist nicht vorgesehen.

Die ursprüngliche Version von COM war auf einen Rechner beschränkt und unterstützte keine verteilten Anwendungen. Durch die Erweiterung *Distributed COM* (DCOM) wurde diese Einschränkung aufgehoben, wodurch auch verteilte Anwendungen möglich sind [BrKi98]. Microsoft definiert COM/DCOM als fundamentales Komponentenmodell, auf das weiterführende Standards aufgesetzt werden. Beispiele hierfür sind OLE (Object Linking and Embedding) zum Einbetten von Komponenten in Dokumente, ActiveX als Komponenten-technologie für das Inter- bzw. Intranet [Chap96] und OPC (OLE for Process Control) zum Datenaustausch in der Automatisierungstechnik [BBH+99].

2.3.3 CORBA

CORBA (Common Object Request Broker Architecture) [Pope97] ist die Referenzarchitektur der *Object Management Group* (OMG), einer firmenübergreifenden Organisation, die sich die Definition und Verbreitung von Integrationstechniken für verteilte, heterogene Umgebungen zum Ziel gemacht hat. Die Grundidee von CORBA geht auf das Client-Server-Prinzip zurück, bei dem ein Anbieter (Server) Dienstleistungen für beliebige Kunden (Clients) zur Verfügung stellt [RePo97]. Diese Dienstleistungen können von Clients sprachen- und plattformunabhängig angefordert werden. Kern der Architektur ist der so genannte *Object Request Broker* (ORB), der die zentrale Vermittlungsstelle zwischen den Dienstaufrufen des Clients und den Rückantworten des Servers darstellt. Der ORB ist gleichzeitig für die Unabhängigkeit von Programmier-sprachen, Adressräumen, Betriebssystemen und Rechnerplattformen verantwortlich. Im Vergleich mit Feldbussystemen der Automatisierungstechnik erfüllt der ORB die Rolle eines „Softwarebusses“ für Komponenten.

CORBA-Anwendungen bestehen aus Komponenten, die über den ORB mit anderen Komponenten interagieren. Komponenten sind dabei identifizierbare, gekapselte Implementierungseinheiten, die ihre Dienstleistungen über Schnittstellen anbieten. Jede Komponente kann zur Erfüllung ihrer Aufgaben sowohl die Server- als auch die Client-Rolle einnehmen. Damit die Client- und Serverkomponenten unabhängig von ihrer Programmier-sprache miteinander kommunizieren können, werden die Schnittstellen mit Hilfe einer standardisierten Beschreibungssprache, der *Interface Definition Language* (IDL) [OrHa98], festgelegt. IDL ist an die Syntax von C++ angelehnt, hat jedoch einen rein deskriptiven Charakter, d.h. es stehen weder Kontrollstrukturen noch Variablen zur Verfügung. Interessant ist auch die Tatsache, dass Schnittstellen mit anderen Schnittstellen Vererbungsbeziehungen (auch mehrfach) eingehen können.

CORBA ist historisch als Componentware-Plattform für die allgemeine Datenverarbeitung entstanden. Der Vorteil, Anwendungen aus verteilten Komponenten aufzubauen, wird

größtenteils durch Speicherplatz- und Laufzeiteinbußen erkauft. Da die Vorteile von CORBA auch für die Automatisierungstechnik interessant sind, die damit verbundenen Nachteile aber nicht akzeptabel sind, wird mittlerweile an speziellen CORBA-Spezifikationen für die Automatisierungstechnik gearbeitet [ScKu00].

2.3.4 JavaBeans / Enterprise JavaBeans

Das *JavaBeans* Komponentenmodell auf der Clientseite [Haro98] und das *Enterprise JavaBeans* Modell auf der Serverseite [EJB20, Mons00] sind Technologien, die sich stark auf die Möglichkeiten der Programmiersprache Java [GJSB00] stützen. Beans sind eigenständige, mehrfach verwendbare Softwareeinheiten, die sowohl mit Hilfe von visuellen Applikationswerkzeugen als auch mit normalen Quellcode-Editoren zu vollständigen Anwendungen oder auch Applets[1] zusammengesetzt werden können. Dabei werden Beans zunächst ausgewählt und instanziiert. Danach werden die Parameter gesetzt, welche das Verhalten und das Aussehen der Komponenten modifizieren. In einem letzten Schritt werden die instanziierten Beans untereinander verbunden, was die möglichen Interaktionen zwischen den einzelnen Komponenten festlegt. Jedes Bean besteht aus drei grundlegenden Elementen: Eigenschaften, Methoden und Ereignissen. Abbildung 2.5 zeigt diese wesentlichen Bestandteile einer Komponente.

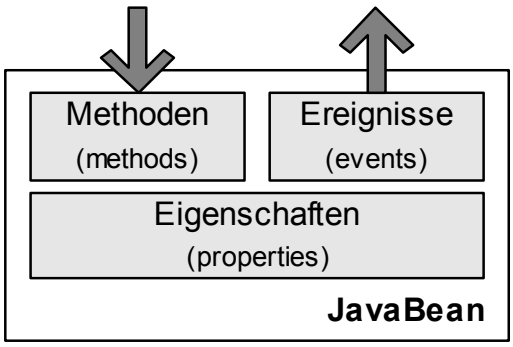

Abbildung 2.5: Grundlegende Elemente einer Java Komponente

Die Eigenschaften repräsentieren den aktuellen Zustand der Komponente. Damit die Eigenschaften verändert werden können, muss ein Bean passende `get`- bzw. `set`-Schnittstellen-Methoden bereitstellen, die von anderen Beans benutzt werden können (z.B. `getEigenschaftX()` und `setEigenschaftX()`). Ohne diese Methoden mit dem entsprechenden `get/set`-Namesprefix können andere Komponenten nicht auf die Eigenschaften der Bean zugreifen.

[1] kleine Programme, die in HTML-Seiten eingebunden werden und im Browserfenster ablaufen

Methoden sind die Funktionen der Komponente, die von außerhalb zugänglich sind („Bean Eingang“). Alle öffentlichen (public) Methoden sind normalerweise für andere Komponenten zugänglich. Beans verwenden Ereignisse, um mit anderen Komponenten zu kommunizieren („Bean Ausgang“). Eine Komponente, die ein Ereignis aussenden möchte (Quell-Komponente), konsultiert zunächst eine interne Liste von Komponenten, die am betreffenden Ereignis Interesse angemeldet haben (sog. Listener-Komponenten). Danach sendet die Quellkomponente das betreffende Ereignis an jede einzelne Listener-Komponente.

Wegen seiner konzeptionellen Klarheit und der einfachen Implementierung von Komponenten ist das JavaBeans Komponentenmodell sehr gut geeignet, um Software-Komponenten für verschiedene Anwendungsbereiche der Automatisierungstechnik zu implementieren. Dabei muss allerdings beachtet werden, dass Java als zugrunde liegende Programmiersprache nicht für alle Anwendungsbereiche geeignet ist. Insbesondere Bereiche mit harten Echtzeitanforderungen oder minimalen Hardware-Ressourcen müssen im Vorfeld genauestens auf ihre Eignung für das JavaBeans Modell untersucht werden. Trotz Fortschritten in diesem Problembereich [BoSu98, HBB00, Nils98] gibt es bisher keine alternativen Modelle, welche die genannten Einschränkungen aufheben und gleichzeitig dieselben Vorteile wie JavaBeans bieten.

2.4 Frameworks

2.4.1 Anwendungsentwicklung mit Frameworks

Viele Entwickler machen im Alltag die Erfahrung, dass große Teile der Infrastruktur verschiedener Anwendungen gleichartig sind. Diese Ähnlichkeiten werden sogar noch größer, wenn die betrachteten Anwendungen zu einem bestimmten Anwendungsbereich (Domäne) gehören [Doug99a]. Im Bereich von Steuerungsprogrammen für Lasten- und Personenaufzügen sind beispielsweise Regelungsalgorithmen, Zustandsmaschinen und nebenläufige Prozesse typische Merkmale des Anwendungsbereichs und damit keine speziellen Merkmale einer bestimmten Steuerungsanwendung [Dujm00a].

Betrachtet man verschiedene Anwendungen einer Domäne von einem abstrakten Standpunkt aus, so werden immer wieder dieselben Problemlösungen implementiert. Trotzdem ist es schwierig, die verschiedenen Anwendungen auf der Implementierungsebene durch einen einheitlichen, wiederverwendbaren Quellcode zu repräsentieren – die Detailabweichungen der einzelnen Aufgabenstellungen und die Verflechtung zwischen invarianten und varianten Programmelementen sind zu groß. Der klassische Software-Entwicklungsprozess, der die einzelne Anwendung in den Mittelpunkt stellt, ist ebenfalls nicht in der Lage, diese Gemeinsamkeiten zu nutzen und führt deshalb zur wiederholten Implementierung von ähnlichen oder sogar identischen Konzepten. Frameworks bieten für definierte Anwendungsbereiche ein

effizientes Wiederverwendungskonzept, welches weit über die Mehrfachverwendung von Quellcode hinausgeht.

Ein *Framework* ist ein „Anwendungsrahmen“, welcher die Architektur und den Kontrollfluss darauf aufbauender Anwendungen definiert [JoFo88]. Der Entwickler bekommt durch das Framework eine domänenspezifische, generische Lösung in die Hand, die er durch Parametrisierungen und Erweiterungen zu seiner speziellen Anwendung vervollständigen kann. Durch den Einsatz eines Frameworks sind viele Entwicklungsentscheidungen bereits vorweggenommen und die Möglichkeit des Entwicklers, Einfluss zu nehmen und dadurch Fehler zu machen, sind deutlich reduziert.

Frameworks können – vergleichbar mit Bibliotheken – sowohl prozedural als auch objektorientiert entwickelt werden [FSS86]. Die meisten Frameworks der aktuellen Literatur basieren auf dem Paradigma der Objektorientierung, es gibt jedoch auch einige wenige Beispiele für nicht-objektorientierte Frameworks (z.B. [PaPr99]). Frameworks sind – noch deutlicher als andere Konzepte – auf Eigenschaften wie Modularität und Wiederverwendbarkeit angewiesen, was durch die objektorientierte Vorgehensweise besonders gut unterstützt wird [FaSc97]. Prinzipiell können diese Eigenschaften aber auch mit nicht-objektorientierten Mitteln erreicht werden.

2.4.2 Merkmale und Eigenschaften

Domänenbindung

Die Entwicklung und der Einsatz von Frameworks ist immer dann sinnvoll, wenn mehrere Anwendungen für einen Anwendungsbereich (Domäne) erstellt werden müssen und die einzelnen Anwendungen eine starke Variabilität im Detail aufweisen. Wenn Programme zu einer definierten und klar abgegrenzten Domäne gehören, so ist die Wahrscheinlichkeit sehr groß, dass sich in den einzelnen Programmen sehr viele Gemeinsamkeiten finden werden. Durch die Entwicklung eines speziellen Frameworks für diesen Anwendungsbereich ist der Programmierer in der Lage, diese Gemeinsamkeiten in einem programmübergreifenden Entwurf auszudrücken und gleichzeitig in konkretem Programmcode zu fixieren. Frameworks in verschiedenen Anwendungsbereichen wie beispielsweise grafische Editoren [VlLi90], Gebäudeüberwachungssysteme [Moli97], Netzwerkprotokolle [HJE95], Maschinensteuerungen [Casa95, Schmi95], meteorologische Software [Kell97] oder Satellitensoftware [PaPr99]) veranschaulichen die vielfältigen Einsatzmöglichkeiten und das Wiederverwendungspotenzial der Framework-Technologie.

Abbildung 2.6 veranschaulicht grafisch den beschriebenen Sachverhalt. Die verschiedenen Applikationen eines Anwendungsbereichs (durch gestrichelte Rechtecke angedeutet) besitzen durch ihre Zugehörigkeit zu diesem Bereich eine Schnittmenge, welche die Gemeinsamkeiten

ausdrückt, die innerhalb aller Applikationen vorhanden sind. Ein Framework versucht diese Schnittstelle möglichst umfassend abzudecken, um auf diese Weise eine einheitliche Entwurfs- und Codebasis für bestehende und zukünftige Applikationen in diesem Problembereich bereitzustellen.

Ein Framework stellt somit eine Meta-Lösung für eine bestimmte Domäne dar, da es nicht auf eine spezielle Lösung in dieser Domäne fixiert ist, sondern einen Lösungsansatz für die Gesamtheit der Probleme des Bereichs bietet.

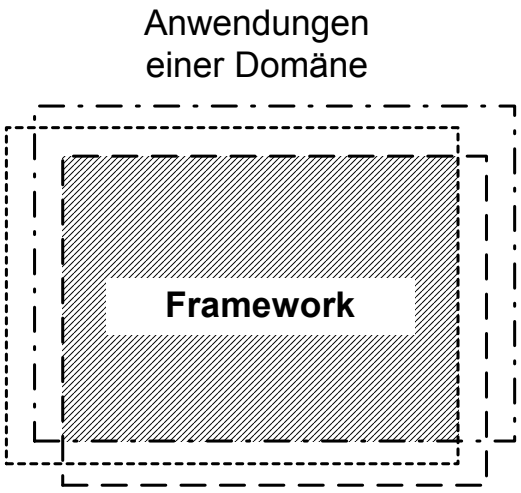

Abbildung 2.6: Framework als Schnittmenge aller Softwarevarianten eines Anwendungsbereichs

Hot/Frozen Spots

Jedes Framework besteht aus zwei unterschiedlichen Teilen. Der überwiegende Teil ist stabil und bringt die Gemeinsamkeiten des Anwendungsbereichs zum Ausdruck. Da sich dieser Teil in jeder Applikation unverändert wieder finden wird, spricht man bei den konstanten Framework-Bestandteilen von den so genannten *Frozen Spots* [Pree95]. Der andere Teil ist hingegen variabel und legt ausschließlich die benötigten Schnittstellen fest. Diese variablen Bestandteile werden [Hot Spots] genannt und sind für die Erzeugung verschiedener Applikationen aus einem Framework verantwortlich. Jeder einzelne Hot Spot steuert dabei die Flexibilisierung eines bestimmten Framework-Aspekts. Abbildung 2.7 veranschaulicht die invariablen und variablen Bestandteile eines Frameworks.

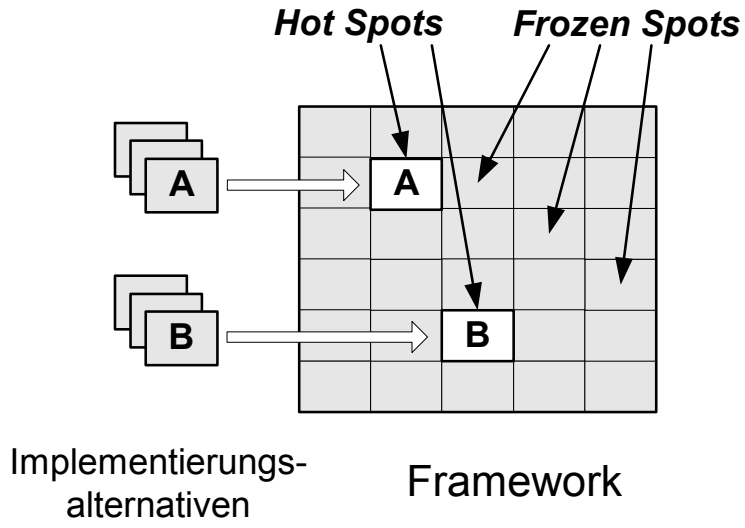

Abbildung 2.7: Hot bzw. Frozen Spots im Framework

Ein Framework ist dann gut entworfen, wenn die typischen Variabilitäten des Anwendungsbereichs in Form von Hot Spots modelliert sind. Die Entscheidung, was ein unveränderlicher oder ein variabler Aspekt sein soll, gehört zu den schwierigsten Entwurfsfragen im Entwicklungsprozess von Frameworks. Diese Entscheidungen setzen sehr viel Wissen über den Anwendungsbereich und seine zukünftige Entwicklung voraus. In [Pree95] wird deshalb sogar eine spezielle Entwicklungsmethode eingeführt, bei der die Fragen nach den Hot und Frozen Spots im Zentrum der Methodik stehen.

Invertierter Kontrollfluss

Die Effizienz von Frameworks als Wiederverwendungskonzept beruht vor allem darauf, dass sie einen Großteil der aus ihnen entstehenden Anwendungen im Voraus festlegen. Dieses Bestreben, möglichst viel von den zukünftigen Anwendungen vorwegzunehmen, beschränkt sich nicht nur auf die Funktionalität, sondern wird auch auf den Kontrollfluss ausgeweitet. Aus diesem Grund ist die Umkehr des Kontrollflusses ein wesentliches Merkmal von Frameworks – benutzerspezifischer, variabler Quellcode wird aus dem invariablen Frameworkteil heraus aufgerufen. In der Literatur wird diese Umkehrung des Kontrollflusses auch *Hollywood-Prinzip* („Don't call us, we'll call you“) genannt. Abbildung 2.8 zeigt die Inversion des Kontrollflusses bei Frameworks im Gegensatz zu gewöhnlichen Funktionsaufrufen bei Bibliotheken. Während bei Bibliotheken der Anwendungsentwickler jederzeit für den globalen Kontrollfluss seines Programms zuständig ist, wird bei Frameworks der Kontrollfluss größtenteils nicht durch den anwendungspezifischen Code, sondern den invariablen Framework-Code festgelegt.

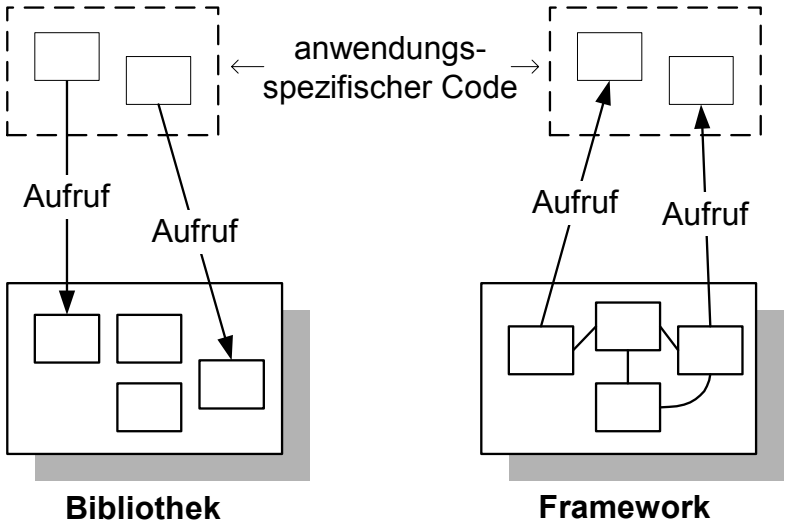

Abbildung 2.8: Invertierter Kontrollfluss im Framework

Charakteristische Vorteile

Frameworks bieten eine Reihe von Vorteilen bei der Anwendungsentwicklung. Viele dieser Vorteile sind charakteristisch für Frameworks oder in dieser Kombination nicht in anderen Wiederverwendungskonzepten vorhanden. Im Nachfolgenden sollen einige dieser besonderen Vorteile näher betrachtet werden (siehe auch [FaSc97]).

- *Weniger anwendungsspezifischer Quellcode*: Durch die Bereitstellung eines Entwurfs- und Implementierungsgerüsts sinkt der Anteil des Codes, der vom Anwendungsentwickler geschrieben werden muss. Der Anwendungsentwickler muss lediglich die anwendungsspezifischen Teile zum Framework hinzufügen, die für seine spezielle Anwendung benötigt werden und nicht bereits durch das Framework angeboten werden. Dieser anwendungsspezifische Quellcode wird im Normalfall nur einem Bruchteil des Codes entsprechen, der ohne den Einsatz des Frameworks für die Anwendung notwendig gewesen wäre.

- *Schnelleres Time-to-Market*: Der Framework-Benutzer kann mit geringem Aufwand Anwendungen mit umfangreicher Funktionalität erzeugen. Durch den Einsatz eines erprobten und bewährten Frameworks wird automatisch auch der Test- und Debugging-Aufwand verringert. Der Benutzer ist somit in der Lage, sehr schnell ein qualitativ hochwertiges Softwareprodukt auf den Markt zu bringen [SBF96].

- *Kapselung von Domänenwissen*: Durch die Ausrichtung des Frameworks auf einen bestimmten Anwendungsbereich ist das Wissen über diesen Bereich im Framework gekapselt. Der Programmierer kann sich auf das konzentrieren, was für seine Anwendungen wesentlich ist. Er muss sich nicht mit der Komplexität und den Eigenheiten des Anwendungsbereichs auseinander setzen, die bereits durch das Framework erfasst sind,

sondern kann sich vollständig auf die Lösung seines Problems fixieren. Die Kapselung des Domänenwissens löst beiläufig auch ein anderes Problem – das Wissen und die Erfahrungen wird aus den Köpfen der Experten explizit und dauerhaft in den Framework-Code transferiert. Auf diese Weise wird die Abhängigkeit von einzelnen Experten-Persönlichkeiten erheblich verringert.

- *Verbesserte Konsistenz und Wartbarkeit*: Das Framework fixiert die Gemeinsamkeiten des Anwendungsbereichs im Entwurf und in der Implementierung. Alle aus dem Framework entstehenden Anwendungen stützen sich konsequent auf diese Vorgaben und es kann keine inkonsistenten Lösungsansätze für die gleiche Aufgabenstellung geben. Durch die gemeinsame Entwurfs- und Codebasis vereinfacht sich auch die Wartung: es ist weniger aufwändig ein Framework und daraus entstehende Anwendungen, als mehrere eigenständige Anwendungen zu warten.

2.4.3 Vergleich mit anderen Konzepten

Die oben aufgezählten Vorteile von Frameworks werden teilweise auch von den anderen vorgestellten Wiederverwendungskonzepten beansprucht. Aus diesem Grund stellt sich die Frage, inwiefern sich Frameworks von den anderen Wiederverwendungskonzepten unterscheiden.

Bibliotheken und Frameworks

Bibliotheken bieten einzelne, eigenständige Einheiten zur Wiederverwendung, die größtenteils unabhängig voneinander sind. Weiterhin können die meisten Bibliotheken weitgehend unabhängig von der speziellen Domäne der gewünschten Anwendung eingesetzt werden. Frameworks bieten hingegen einen Entwurf für einen bestimmten Anwendungsbereich, der normalerweise aus einer Sammlung von zusammenhängenden Elementen besteht – die einzelnen Elemente sind weder unabhängig voneinander noch können sie in unterschiedlichen Anwendungsbereichen zum Einsatz kommen. Da Frameworks die Wiederverwendung von Entwurfsentscheidungen und Quellcode ermöglichen, ist mit ihnen bezüglich der gesamten Anwendung ein deutlich höherer Wiederverwendungsanteil als mit ausschließlich codeorientierten Bibliotheken zu erreichen.

Trotz dieser deutlichen Unterscheidungsmerkmale müssen sich Frameworks und Bibliotheken nicht gegenseitig ausschließen. Viele Frameworks sind evolutionär aus Bibliotheken entstanden, nachdem die Autoren den Anwendungsbereich besser verstanden hatten und programmübergreifende Gemeinsamkeiten identifizieren konnten [RoJo96]. Zusätzlich stößt man in der Praxis häufig auf Bibliotheken, die eng mit Frameworks gekoppelt sind und eine wesentliche Rolle im Implementierungscode des Frameworks spielen [Lewi95].

Muster und Frameworks

Muster konservieren Erfahrungen in verschiedenen Phasen des Software-Entwicklungsprozesses und fördern das ingenieurmäßige Vorgehen bei der Softwareentwicklung. Da ähnliche Absichten auch von Frameworks verfolgt werden, sind konzeptionelle Ähnlichkeiten nicht verwunderlich. Trotzdem unterscheiden sich Frameworks und Muster in drei wichtigen Aspekten:

- *Muster sind abstrakter als Frameworks*:
  Muster werden nur zur Veranschaulichung mit Codebeispielen ausgestattet, während fertiger Code ein elementarer und unverzichtbarer Bestandteil von Frameworks ist. Muster betonen vor allem die Wiederverwendung abstrakter und bewährter Lösungen, wohingegen Frameworks durch ausführbaren Programmcode die Wiederverwendung von Entwurf und Code unterstützen.

- *Frameworks sind umfangreicher als Muster*:
  Ein typisches Framework enthält normalerweise mehrere Muster, die Umkehrung ist jedoch nie gültig.

- *Frameworks sind spezialisierter als Muster*:
  Frameworks sind für einen definierten und deutlich begrenzten Anwendungsbereich konstruiert (z.B. GUI oder Maschinensteuerungen). Eine Verwendung außerhalb dieses Anwendungsbereichs ist nicht sehr lohnend, da meistens die gesamte Framework-Architektur aufgebrochen werden muss. Im Gegensatz dazu können die meisten Muster ohne Schwierigkeiten in verschiedenen Anwendungsbereichen eingesetzt werden.

Trotz dieser Unterschiede gibt es auch zwischen Mustern und Frameworks vielfältige Berührungspunkte, die in eine symbiotische Beziehung münden. Viele der bekannten Entwurfsmuster wurden während der Entwicklung oder dem nachträglichen Studium von Frameworks entdeckt. Fast alle Muster in [GHJV95] sind beispielsweise im ET++ Framework[2] [WGM89] verankert und haben dort ihre Nützlichkeit als allgemein gültiges Entwurfswissen unter Beweis gestellt. Jedes Framework ist auf Flexibilität an den Hot Spots angewiesen und genau diese Eigenschaft fördert die Entstehung und Entdeckung von Mustern. Aber auch Frameworks profitieren in mehrfacher Hinsicht von der Existenz bestehender Muster – sie werden während der Entwicklung von Frameworks zur Realisierung von Hot Spots eingesetzt [Pree97, Schm96] und dienen später dazu, die im Framework definierten Strukturen zu dokumentieren und somit anderen Entwicklern verständlicher zu machen [John92, MCK97, Zimm98].

---

[2] Framework zur Entwicklung von portablen Anwendungen mit grafischen Benutzungsoberflächen

Komponenten und Frameworks

Sowohl Componentware-Plattformen als auch Frameworks unterstützen die Wiederverwendung von Implementierungscode. Als Folge davon ist auch die Parametrierung während der Anwendungsentwicklung eine gemeinsame Eigenschaft – sowohl Komponenten als auch Frameworks werden parametriert, um die Eigenschaften bzw. das Verhalten innerhalb der Anwendung endgültig festzulegen. Im Gegensatz zu Komponenten sind Frameworks jedoch meistens monolithische Konstrukte, die eine Anwendung größtenteils abdecken. Komponenten hingegen betrachten nur Teilbereiche einer Anwendung und können deshalb auch in unterschiedlichen Anwendungsbereichen eingesetzt werden.

2.5 Bewertung der vorgestellten Ansätze

2.5.1 Kriterien und Bewertung

Jedes Wiederverwendungskonzept muss daran gemessen werden, wie viel Nutzen sein Einsatz bringt und mit welchen Kosten dieser Nutzen erkauft wird. Das beste Konzept wird sich nicht durchsetzen, wenn die damit verbundenen Kosten nicht deutlich unter einer vollständigen Neuprogrammierung liegen [Booc94]. Alle hier vorgestellten Konzepte der Wiederverwendung sollten deshalb mit Hilfe von Bewertungskriterien verglichen werden, die den Nutzen und den Aufwand für den *Anwendungsentwickler* hervorheben. Im Nachfolgenden werden die einzelnen Bewertungskriterien kurz erläutert und auf die verschiedenen Konzepte angewendet.

Granularität

Ein Vergleichskriterium ist die *Granularität* der Wiederverwendungseinheiten, die durch das Konzept ermöglicht werden. Dieses Kriterium bewertet, welcher Grad an Wiederverwendung mit dem Konzept erreicht werden kann. Während Wiederverwendung auf der Ebene einzelner, kleinerer Implementierungseinheiten (z.B. Funktionen, Module, Klassen) betrieben werden kann, sind für den Anwendungsentwickler vor allem solche Konzepte interessant, die eine Durchgängigkeit bis zu vollständigen Anwendungen bieten. Je gröber die Granularität des betrachteten Konzepts, desto besser können Implementierungsdetails vom Anwendungsentwickler verborgen werden.

Durch ihre Ausrichtung auf einzelne Funktionen unterstützen Bibliotheken lediglich eine feingranulare Wiederverwendung. Sie dienen ausschließlich dazu, bei der Anwendungsentwicklung die wiederholte Eingabe identischer oder ähnlicher Quellcodezeilen zu verhindern. Muster und Komponenten bieten hingegen eine Wiederverwendung auf unterschiedlichen Abstraktionsebenen – sie können sowohl einzelne Funktionen kapseln, als auch vollständige Anwendungen repräsentieren. Ihre häufigste Verwendung besteht jedoch darin, wesentliche Teilelemente zu beschreiben, die vom Entwickler noch zu einer Anwendung zusammengesetzt

werden müssen. Frameworks sind besonders grobgranular, da sie von Anfang an auf die Wiederverwendung von vollständigen Anwendungen ausgerichtet sind.

Verständlichkeit

Das Kriterium der *Verständlichkeit* soll bewerten, wie hoch der Lernaufwand und wie schwierig das Verständnis für einen Benutzer ist, der auf der Basis des Konzepts Anwendungen entwickeln möchte. Die Verständlichkeit entscheidet somit über den initialen Aufwand für den Benutzer, der die Anfangsphase der Verwendung des Konzepts kennzeichnet. Für den Entwickler ist ausschlaggebend, dass dieser Aufwand möglichst klein und überschaubar ist, um nicht bereits im Vorfeld die Verwendung des Konzepts zu vereiteln.

Muster und Frameworks sind durch einen besonders hohen anfänglichen Lernaufwand gekennzeichnet. Bei Mustern werden detaillierte Implementierungs- und Entwurfskenntnisse für das Verständnis vorausgesetzt, während bei Frameworks vor allem der zu bewältigende Informationsumfang durch die Vielzahl von Entwurfs- und Implementierungsentscheidungen ein Problem darstellt. Bibliotheken und Komponenten sind hingegen relativ einfach zu verstehen – hier stehen vor allem die angebotene Funktionalität und die Parametrierungsmöglichkeiten im Vordergrund.

Anwendbarkeit

Das Kriterium der *Anwendbarkeit* betrachtet, wie einfach oder schwierig das Konzept vom Anwendungsentwickler verwendet werden kann. Kann der Entwickler das Konzept direkt in seinem Projekt einsetzen oder muss er zunächst noch Transformationsschritte durchführen, um mit dem Konzept zu arbeiten. Die Anwendbarkeit ist somit der Aufwand für den Benutzer, der die Verwendungsphase des Konzepts nach der Lernphase kennzeichnet. Genau wie bei der Verständlichkeit muss bei diesem Kriterium ein geringer Aufwand für den Entwickler angestrebt werden, um den Erfolg des Wiederverwendungskonzepts sicher zu stellen.

Bezüglich der Anwendbarkeit schneiden Muster am schlechtesten ab. Da sie lediglich ein abstraktes Lösungsrezept bieten, muss der Anwendungsentwickler zunächst eine individuelle Transformation auf die konkrete Problemstellung durchführen. Bei Frameworks sind zwar keine Transformationen notwendig, es müssen jedoch viele Abhängigkeiten bei der Anwendungsentwicklung beachtet werden, die nicht immer offensichtlich sind. Bei Bibliotheken und Komponenten gestaltet sich die Anwendung besonders einfach, da eine direkte Verwendung vorgesehen ist.

Adaptierbarkeit

Das Kriterium der *Adaptierbarkeit* bewertet, inwiefern der Anwendungsentwickler die Eigenschaften und das Verhalten der jeweiligen Wiederverwendungeinheit beeinflussen kann. Ist die Wiederverwendungseinheit ein in sich geschlossener Baustein, der nur unverändert oder

mit geringen Einflussmöglichkeiten wieder verwendet werden kann, oder ist eine weit reichende Anpassung durch den Anwendungsentwickler vorgesehen. Für die Anwendungsentwicklung sind natürlich besonders solche Konzepte interessant, die eine hohe Flexibilität bieten, ohne dadurch die Integrität der einzelnen Wiederverwendungseinheit zu gefährden.

Die Adaptierbarkeit der einzelnen Konzepte verhält sich erfahrungsgemäß reziprok zum Aufwand beim Einsatz des Konzepts – je flexibler ein Konzept auf die konkreten Anforderungen der zu erstellenden Anwendung angepasst werden kann, desto aufwändiger ist der Einsatz dieses Konzepts. Bibliotheken und Komponenten bieten dem Anwendungsentwickler durch ihre Abgeschlossenheit nur geringfügige Eingriffsmöglichkeiten. Muster und Frameworks sind hingegen durch ihre Offenheit sehr flexibel an spezielle Anforderungen anpassbar.

Automatisierbarkeit

Das Kriterium der *Automatisierbarkeit* betrachtet, in welchem Umfang das Wiederverwendungskonzept Ansätze für eine Werkzeugunterstützung bietet. Quellcode-betonte Konzepte bieten im Gegensatz zu abstrakteren Konzepten meistens mehrere Ansatzpunkte für effiziente Werkzeuge. Der Anwendungsentwickler wünscht sich natürlich ein Werkzeug, das die Arbeit mit dem Konzept erleichtert und dadurch die erfolgreiche Wiederverwendung zusätzlich zu den konzeptionellen Vorzügen fördert.

Muster bieten die schlechtesten Voraussetzungen für eine umfassende Werkzeugunterstützung, da sie von einem individuellen, nicht vorhersehbaren Transformationsprozess bei der Anwendungsentwicklung ausgehen [EYG97]. Bei Bibliotheken ist prinzipiell zwar eine Werkzeugunterstützung möglich, wegen der geringfügigen Effizienzsteigerung bei der Anwendungsentwicklung aber nicht lohnenswert. Komponenten und Frameworks bieten konzeptionell verhältnismäßig viele Ansatzpunkte für Werkzeuge bei der Anwendungsentwicklung. Während für Komponenten bereits entsprechende Werkzeuge existieren (sog. Application Builder), wird das Potenzial bei Frameworks noch häufig unterschätzt.

2.5.2 Zusammenfassende Bewertung

In Tabelle 2.1 werden abschließend noch einmal alle betrachteten Wiederverwendungkonzepte gegenübergestellt und bezüglich der aufgestellten Kriterien zusammenfassend bewertet. Dabei wird für jedes Konzept bewertet, ob das zugrunde gelegte Kriterium gut (+), neutral (0) oder schlecht (-) erfüllt wird.

Aus der Bewertung in Tabelle 2.1 wird ersichtlich, dass sowohl Komponenten als auch Frameworks eine gute technologische Ausgangsbasis bilden, um eine umfassende Unterstützung bei der Anwendungsentwicklung zu verwirklichen. Frameworks unterstützen diese Forderung im Vergleich zu Komponenten sogar besonders weit reichend, da sie als einziges

Wiederverwendungskonzept die Möglichkeit bieten, sowohl Implementierungen als auch Entwurfsentscheidungen wieder zu verwenden. Bibliotheken und Muster scheiden aus, weil sie zu feingranular oder zu abstrakt für die Anwendungsentwicklung sind.

Tabelle 2.1: Wiederverwendungskonzepte im Vergleich

	Bibliotheken	**Muster**	**Komponenten**	**Frameworks**
Granularität	–	0	0	+
Verständlichkeit	+	–	+	–
Anwendbarkeit	+	–	+	0
Adaptierbarkeit	–	+	–	+
Automatisierbarkeit	0	–	+	+

Vergleicht man Komponenten und Frameworks aus der Anwendersicht, so stellt man zusätzlich fest, dass sich beide Konzepte gegenseitig gut ergänzen – Komponenten bieten Vorteile, wo Frameworks Schwächen haben und umgekehrt. Eine Vereinigung beider Konzepte unter Beibehaltung der jeweiligen Vorteile würde einen wesentlichen Fortschritt bei der Unterstützung der Anwendungsentwicklung ermöglichen.

In diesem Kapitel wurden verschiedene Wiederverwendungskonzepte präsentiert, die dem aktuellen Stand der Technik entsprechen. Die vorgestellten Konzepte wurden gegenübergestellt und anhand von definierten Kriterien bewertet, welche als Voraussetzung für die effiziente Unterstützung des Anwendungsentwicklers gelten. Diese Bewertung hat ergeben, dass die Framework-Technologie eine besonders gute Ausgangsposition bietet, um eine umfassende Unterstützung bei der Anwendungsentwicklung zu verwirklichen.

Im nächsten Kapitel wird zunächst die Framework-Technologie aus der Sicht des Anwendungsentwicklers detaillierter beleuchtet. Dabei werden besonders die Schwierigkeiten dargestellt, die normalerweise an den Einsatz der Framework-Technologie gekoppelt sind. Im Kapitel 4 wird dann auf den Ergebnissen dieser Betrachtungen ein neues Wiederverwendungs-konzept vorgestellt, welches die Vorteile der Framework- und Komponententechnologie in sich vereint.

3 Frameworks aus der Sicht des Anwendungsentwicklers

In diesem Kapitel werden die Probleme und Schwierigkeiten erläutert, die aus heutiger Sicht an die Framework-Technologie gekoppelt sind. Ein besonderes Augenmerk wird dabei auf die Bedürfnisse des Anwendungsentwicklers gelegt, der mit dem Framework individuelle Anwendungen erstellen möchte. Weiterhin werden die Ursachen für diese Schwierigkeiten analysiert, sowie bestehende Lösungsansätze aus der Literatur vorgestellt und bewertet. Abschließend werden auf der Grundlage dieser Betrachtungen Anforderungen aufgestellt, die für eine umfassende Unterstützung des Anwendungsentwicklers beim Einsatz der Framework-Technologie gelten müssen.

3.1 Framework-Problematik

Frameworks ermöglichen einen sehr hohen Grad an Wiederverwendung und eine beträchtliche Produktivitätssteigerung, wenn die zu erstellende Anwendung in der Domäne des Frameworks liegt [MoNi96, SBF96]. Ein passendes Framework kann in vielen Fällen einen großen Teil der gewünschten Anwendungsfunktionalität abdecken. Leider sind diese Vorzüge durch Schwierigkeiten belastet, die sich bei dem Einsatz von Frameworks offenbaren. In [BMM+99] werden vier grundlegende Problemkategorien identifiziert, die mit der Framework-Technologie gekoppelt sind:

- *Framework-Entwicklung*: Probleme der Entwicklungsphase umfassen beispielsweise die sinnvolle Abgrenzung der Domäne, die Identifikation von sinnvollen Framework-Bausteinen, die Definition spezieller Framework-Entwicklungsmethoden, sowie das Testen von Frameworks.

- *Anwendungsentwicklung mit Frameworks*: Der Anwendungsentwickler hat beispielsweise Schwierigkeiten bei der Entscheidung, ob ein Framework für seine geforderte Anwendung geeignet ist und mit welchem Entwicklungsaufwand gerechnet werden muss. Er muss das Framework ausreichend verstehen, um darauf basierende Anwendungen zu erstellen und Programmfehler zu bereinigen. Seine Aufgabe wird zusätzlich dadurch erschwert, dass keine Entwicklungsmethode für frameworkbasierte Anwendungen verfügbar ist und die meisten aktuellen Methoden und Notationen den Framework-Einsatz nicht angemessen berücksichtigen.

- *Framework Komposition*: Falls der gleichzeitige Einsatz von mehreren Frameworks innerhalb einer Anwendung erforderlich ist, kann der Entwickler auf erhebliche Integrationsprobleme stoßen. Dazu gehören beispielsweise strukturelle Unverträglichkeiten, die Überlappung von Funktionalität oder die Kollision von Kontrollflüssen. Die gleichzeitige Verwendung mehrerer Frameworks ist häufig ein schwieriges Unterfangen, da Frameworks nicht auf die Koexistenz mit anderen Frameworks ausgelegt sind.

- *Framework Evolution und Wartung*: Frameworks sind normalerweise langlebige Softwareprodukte, die kontinuierlich weiterentwickelt und gewartet werden müssen. Das Framework-Entwicklungsteam muss deshalb frühzeitig Strategien entwickeln, welche die Evolution und Wartung erleichtern.

Im Mittelpunkt der vorliegenden Arbeit steht die Unterstützung des Anwendungsentwicklers beim Einsatz von Frameworks. Aus diesem Grund werden im Nachfolgenden vor allem die Problembereiche unter Punkt 2 näher betrachtet, welche die Anwendungsentwicklung mit Frameworks – die so genannte *Framework-Instanziierung* – betreffen. Die zentrale Frage wird dabei sein: Wie kann der Anwendungsentwickler unterstützt werden, damit er bei minimalem Aufwand möglichst schnell und zielstrebig zu seiner spezifischen Anwendung kommt?

3.2 Schwierigkeiten bei der Framework-Instanziierung

3.2.1 Framework-Selektion

Eine der ersten Schwierigkeiten für den Anwendungsentwickler besteht darin, den Anwendungsbereich des Frameworks zu verstehen und mit dem Anwendungsbereich der geplanten Anwendung zu vergleichen. Der Entwickler muss im Voraus und mit möglichst hoher Zuverlässigkeit entscheiden, ob ein Framework für die gewünschte Anwendung geeignet ist, oder ob möglicherweise eine Anwendungsentwicklung ohne Framework angebrachter ist. Von einem abstrakten Standpunkt aus geht es um die Frage, inwiefern der Anwendungsbereich des Frameworks die gewünschte Anwendung abdeckt.

Diese scheinbar einfache Frage kann in der Praxis sehr komplex werden und mehrere Entscheidungsebenen umfassen. Eine besonders offensichtliche Ebene ist die funktionale Ebene, welche die Domäne des Frameworks definiert und festlegt, was zu den funktionalen Leistungen des Frameworks gehört. Weitere nicht so offensichtliche, aber sehr wichtige Entscheidungsebenen offenbaren sich in der Automatisierungstechnik – hier spielen neben den funktionalen Aspekten auch nichtfunktionale Aspekte eine entscheidende Rolle. Im Gegensatz zu funktionalen Aspekten ergeben sich nichtfunktionale Anforderungen nicht direkt aus der geforderten Funktionalität des Frameworks, sondern aus den Randbedingungen, die durch den Einsatz des Frameworks in einer bestimmten Hardware- und Softwareumgebung entstehen.

Dazu gehören beispielsweise Echtzeitanforderungen, Speicherplatzbegrenzungen oder Sicherheitseigenschaften. Diese nichtfunktionalen Aspekte können ein für die Anwendungsentwicklung funktional passendes Framework im Endergebnis als ungeeignet qualifizieren.

3.2.2 Aufwandsabschätzung

Eine weitere Schwierigkeit bei der frameworkbasierten Anwendungsentwicklung besteht darin, den Arbeitsaufwand für den Anwendungsentwickler realistisch einzuschätzen. Erfahrungen zeigen, dass auch hier die berühmte Gesetzmäßigkeit von Pareto (Pareto's Law) zum Tragen kommen kann: 20 % der Entwicklungszeit wird für 80 % der Anwendung aufgewendet, während die restlichen 80 % der Zeit für einen kleinen, komplizierten Teil der Anwendung (20 %) benötigt werden. Das Auftreten dieser Gesetzmäßigkeit hängt bei frameworkbasierten Anwendungen vor allem davon ab, ob wesentliche Merkmale der Anwendung vom Framework unterstützt werden. Ist die Unterstützung solcher Anwendungsmerkmale durch das Framework gewährleistet, so erfolgt die Implementierung schnell und zielstrebig. Falls andererseits wesentliche Merkmale nicht unterstützt werden, so kann die nachträgliche, frameworkkonforme Implementierung einen deutlichen Mehraufwand im Vergleich zur konventionellen Anwendungsentwicklung ohne Framework verursachen.

Um seinen Arbeitsaufwand realistisch einschätzen zu können, muss der Entwickler vor allem solche Anwendungsmerkmale im Auge behalten, die sich an der Grenze der Framework-Domäne bewegen. Eine unvorhergesehene Überschreitung dieser Grenze bei der Anwendungsentwicklung kann die Produktivitätsvorteile von Frameworks im Vergleich zur konventionellen Anwendungsentwicklung schnell zunichte machen.

3.2.3 Framework-Verständnis

Eine wesentliche Voraussetzung für den erfolgreichen Einsatz von Frameworks besteht darin, dass der Anwendungsentwickler das Framework ausreichend verstanden hat, um es bestimmungsgemäß einzusetzen. Er muss die globale Architektur und teilweise auch Implementierungsdetails kennen, um bei der Anwendungsentwicklung keine frameworkinternen Konventionen zu verletzen. Bei vielen objektorientierten Frameworks bedeutet diese Voraussetzung, dass der Anwendungsentwickler sich zumindest ausschnittsweise mit dem Quellcode des Frameworks vertraut machen und die mitgelieferten Beispielanwendungen nachvollziehen muss. Er muss somit mit einem beträchtlichen initialen Einarbeitungsaufwand rechnen, bevor er von den Vorteilen bei der frameworkbasierten Anwendungsentwicklung profitieren kann.

Der Anwendungsentwickler wird diesen anfänglichen Aufwand nur dann tolerieren, wenn Framework-Verständnis und -Anwendung deutlich weniger Zeit in Anspruch nehmen als das Erstellen einer vergleichbaren Anwendung ohne Framework. In [Booc94] wird diese Voraussetzung deutlich zum Ausdruck gebracht:

> „The most profoundly elegant framework will never be reused unless the *cost of understanding* it and then *using its abstractions* is lower than the programmer‘s perceived cost of writing them from scratch.”

Der erfolgreiche Einsatz von Frameworks setzt deshalb voraus, dass der Entwickler das Framework in kürzester Zeit ausreichend verstanden hat, um die gewünschte Anwendung zu realisieren. Aus diesem Grund sind die Qualitätsmerkmale *Verständlichkeit* und leichte *Anwendbarkeit* für Frameworks von besonderer Bedeutung. Da heutige Frameworks meistens umfangreiche und komplexe Softwareprodukte sind, können diese Qualitätsmerkmale nur durch besondere Anstrengungen erreicht werden.

3.2.4 Methoden und Werkzeuge für die Anwendungsentwicklung

Eine besondere Schwierigkeit bei der frameworkbasierten Anwendungsentwicklung ist die Frage nach einem passenden Entwicklungsprozess und einer darauf abgestimmten Entwicklungsmethode. Für die konventionelle Anwendungsentwicklung ohne Frameworks existieren verschiedene bewährte Methoden, die auf Wasserfall- oder Spiralmodellen basieren. Diese sind aber nur eingeschränkt für die Anwendungsentwicklung mit Frameworks geeignet, da sie die durch Frameworks zusätzlich hinzukommenden Herausforderungen größtenteils ignorieren [FPP00]. Der konsequente Einsatz dieser Methoden kann sogar dazu führen, dass wesentliche Elemente der frameworkbasierten Anwendungsentwicklung, wie z.B. die saubere Trennung zwischen anwendungs- und frameworkspezifischer Funktionalität, vollständig vernachlässigt wird. In der Praxis geschieht die Anwendungsentwicklung mit Frameworks deshalb größtenteils in einem explorativen, ad-hoc Vorgehen, welches weder definiert noch reproduzierbar ist.

Durch das Fehlen einer Methodik für die Anwendungsentwicklung, die Frameworks als Ausgangspunkt berücksichtigt, existieren auch keine passenden Werkzeuge, die den Anwendungsentwickler unterstützen und durch den Entwicklungsprozess führen.

3.2.5 Test- und Fehlerbereinigungsproblematik

Auch während dem eigentlichen Instanziierungsprozess – insbesondere den Test- und Fehlerbereinigungszyklen – muss der Anwendungsentwickler mit typischen Schwierigkeiten rechnen. Eine frameworkbasierte Anwendung besteht in der Regel aus zwei unterschiedlichen Teilen: einem *Frameworkanteil*, der durch das Framework festgelegt ist und sich in allen

Anwendungen unverändert wieder findet, und einem *anwendungsspezifischen Anteil*, der durch die besonderen Anforderungen der Anwendung bedingt ist. Falls innerhalb des anwendungsspezifischen Teils ein Fehler auftritt, so kommen prinzipiell zwei Ursachen dafür in Frage:

1. Der anwendungsspezifische Teil wurde bezüglich der Anforderungen fehlerhaft implementiert oder

2. der anwendungsspezifische Teil wurde zwar bezüglich der Anforderungen korrekt implementiert, sein Verhalten verletzt jedoch frameworkinterne Konventionen.

Der Anwendungsentwickler hat häufig nicht die Möglichkeit, zwischen diesen beiden Fällen zu unterscheiden, um geeignete Korrekturmaßnahmen zu ergreifen. Im Extremfall wird er einen Fehler im anwendungsspezifischen Teil suchen, der in Wirklichkeit durch die Interaktion zwischen Framework und anwendungsspezifischem Teil bedingt ist. Erschwerend kommt hinzu, dass konventionelle Werkzeuge zur Fehlerbereinigung (sog. Debugger) nicht auf den Framework-Einsatz abgestimmt sind und keinen Unterschied zwischen den beiden Anwendungsanteilen machen. Die richtige Zuordnung von Fehlern zu entsprechenden Codeteilen fällt dadurch wieder auf den Anwendungsentwickler zurück.

3.3 Problemursachen

Um die Ursachen für die oben beschriebene Instanziierungsproblematik zu verstehen, muss der Lebenszyklus von Frameworks betrachtet werden. Dieser Lebenszyklus besteht normalerweise aus zwei unabhängigen Entwicklungsschritten – der Entwicklung des Frameworks und den nachfolgenden Framework-Instanziierungen, bei denen mit Hilfe des Frameworks individuelle Anwendungen entwickelt werden. Die Entwicklung eines neuen Frameworks und die nachfolgenden Instanziierungen erfolgen normalerweise in unterschiedlichen Projekten, die voneinander vollkommen unabhängig sind. Man kann auch nicht davon ausgehen, dass dieselben Personen bei diesen Aktivitäten involviert sind.

Die Konsequenz dieser Tatsache ist, dass der Anwendungsentwickler nicht über dieselben Informationen wie der Frameworkentwickler verfügt. Der Informationsfluss von dem Frameworkentwickler zu dem Anwendungsentwickler ist durch einen *Informationsverlust* gekennzeichnet, der sich nachteilig auf den Instanziierungsprozess auswirkt und in Abbildung 3.1 skizziert ist.

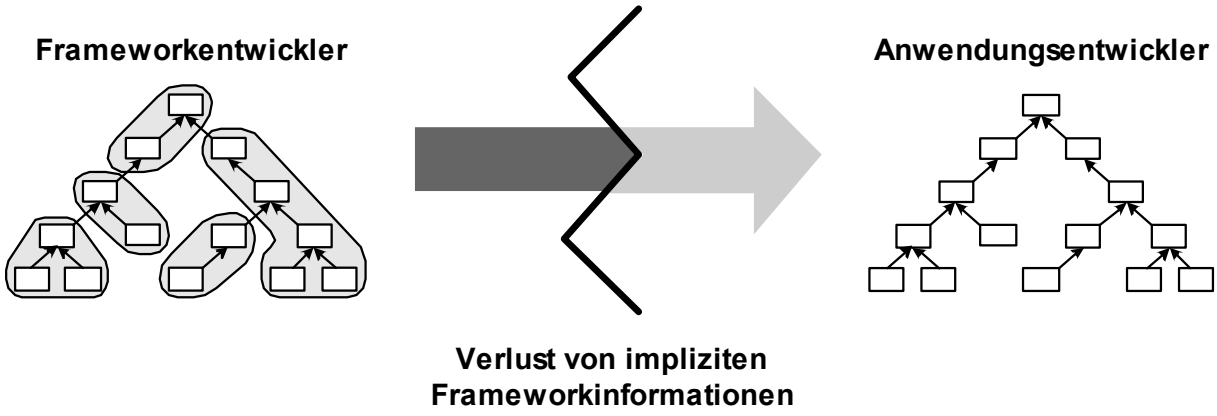

Abbildung 3.1: Informationsverlust zwischen Frameworkentwicklung und -Instanziierung

Auf der einen Seite steht der Frameworkentwickler, der ein umfassendes Wissen bezüglich des Framework-Entwurfs und der dazu gehörigen Implementierung besitzt. Er hat eine präzise Vorstellung vom Anwendungsbereich des Frameworks und weiß, mit welchem Aufwand neue Anwendungen auf der Basis des Frameworks erstellt werden können. Er kennt die Bestandteile des Frameworks, die in allen Anwendungsprojekten unverändert bleiben und jene, die von Anwendung zu Anwendung variieren (Frozen/Hot Spots [Pree95]). Er weiß auch, wie das Framework instanziiert werden muss, ohne dabei die frameworkinternen Konventionen zu verletzen. Er kennt die Implementierungsklassen, die semantisch zusammengehörende Bestandteile bilden, und weiß, wie diese Teile voneinander abhängen. All diese für die Instanziierung wesentlichen Informationen sind größtenteils implizit vorhanden und können nicht direkt aus dem Quellcode abgeleitet werden.

Auf der anderen Seite steht der Anwendungsentwickler, der lediglich den Framework-Quellcode oder sogar nur eine Sammlung von Schnittstellen mit den dazugehörigen binären Implementierungsdateien besitzt. Er kann bestenfalls das explizite Framework-Wissen rekonstruieren, wie z.B. einzelne Klassen oder Klassenhierarchien. Das gesamte implizite Wissen des Frameworkentwicklers (wie z.B. semantisch zusammengehörende Teile) kann jedoch nicht aus dem Quellcode rekonstruiert werden und steht somit bei der Anwendungsentwicklung nicht mehr zur Verfügung.

Zusammenfassend kann man festhalten, dass der Anwendungsentwickler für die Instanziierung einerseits

- *zu wenige Informationen* erhält, da die meisten impliziten und wichtigen Informationen des Frameworkentwicklers dem Anwendungsentwickler nicht mehr zugänglich sind. Andererseits erhält er

- *zu viele Informationen*, die für die Instanziierung irrelevant sind. Er wird sich beispielsweise nicht mit Entwurfs-und Implementierungsdetails beschäftigen, solange diese nicht für die Instanziierung benötigt werden.

Als Folge dieser Situation reicht es in keiner Weise aus, wenn der Frameworkentwickler ein funktionsfähiges und vollständig implementiertes Framework ausliefert. Solch ein Framework ist lediglich die technische Grundlage für den Instanziierungsprozess. Seine Arbeit ist erst dann abgeschlossen, wenn er zusätzlich gewährleistet, dass die Verständlichkeit und Benutzbarkeit für den Anwendungsentwickler gegeben ist. Dies muss vor allem dadurch erfolgen, dass der Anwendungsentwickler alle für die Instanziierung notwendigen Informationen erhält, ohne mit unnötigen Details überladen zu werden.

3.4 Bestehende Lösungsansätze

3.4.1 Dokumentation

Da der Informationsverlust zwischen Frameworkentwickler und -anwender eine der wichtigsten Ursachen für die Schwierigkeiten bei der Instanziierung von Frameworks ist, können verschiedene Vorschläge in der Literatur gefunden werden, die dem Informationsverlust entgegenwirken. Die meisten dieser Vorschläge konzentrieren sich auf die Bereitstellung von Dokumentation als ein Mittel zur Vereinfachung des Instanziierungsprozesses. Johnson [John94] beschreibt drei verschiedene Dokumentationsarten, die zur Klassifizierung herangezogen werden können:

- *Entwurfs- und Referenzdokumentation*: Diese Dokumentationsart ist für Benutzer gedacht, die auf tiefergehende Informationen angewiesen sind. Es werden alle Details beschrieben, die für die Erweiterung und Modifikation des Frameworks benötigt werden.

- *Benutzungsdokumentation*: Diese Dokumentationsart beschreibt typische Benutzungsszenarien und zeigt auf, wie aus dem Framework Applikationen entwickelt werden können.

- *Dokumentation zur Framework-Selektion*: Diese Dokumentationsart hilft bei der Entscheidung, ob das Framework für die beabsichtigte Applikation geeignet ist.

Die meisten Literaturvorschläge betreffen die erste Dokumentationsart und beschreiben verschiedene Ansätze zur Dokumentation von Informationen aus der Entwurfsphase von Frameworks. *Entwurfsmuster* sind ein bekanntes Beispiel für die Dokumentation von Entwurfsentscheidungen bei Frameworks [MCK97, Zimm98]. Muster machen Entwurfsinformationen explizit und stellen ein einheitliches Vokabular für Framework- und Anwendungsentwickler zur Verfügung. Obwohl Entwurfsmuster sehr wirkungsvoll zur

Dokumentation von Entwurfsentscheidungen eingesetzt werden können, bieten sie keine angemessene Unterstützung bei der Auswahl und Instanziierung von Frameworks.

Vergleichsweise wenige Vorschläge betreffen die zweite Dokumentationsart, die sich auf die typische Verwendung von Frameworks konzentriert. *Cookbooks* [KrPo88] sind die bekanntesten Vertreter dieser Dokumentationsart. Ein Cookbook unterteilt den ganzen Instanziierungsprozess in unabhängige Teilaufgaben, deren Lösungen als einzelne Schritte präsentiert werden und sukzessiv zu einer vollständigen Applikation führen. Obwohl Cookbooks eine sehr gute Unterstützung für die Instanziierungsproblematik aufzeigen, eröffnen sie keine Möglichkeiten für eine wirkungsvolle Werkzeugunterstützung – sie basieren auf textuellen Beschreibungen ohne vordefinierte Strukturen oder Formen.

Hilfe bei der Auswahl eines bestimmten Frameworks (dritte Dokumentationsart) wird dem Anwendungsentwickler lediglich in Form von informellen Texten und Beispielapplikationen gegeben. Es existieren keine etablierten Ansätze in der Literatur, welche die zur Framework-Selektion benötigten Informationen benennen und eine einheitliche, standardisierte Beschreibungsnotation vorschlagen. In Zukunft wird diese Problematik sicherlich stärker von den Ergebnissen und Erfahrungen des *Domain Engineering* [CzEi00, Jost00] profitieren, wo teilweise identische Fragestellungen betrachtet werden.

3.4.2 Instanziierungswerkzeuge

Orthogonal zu den oben beschriebenen dokumentationsorientierten Ansätzen zur Vermeidung des Informationsverlustes bei der Framework-Instanziierung kommen in der Praxis auch spezialisierte Instanziierungswerkzeuge zum Einsatz. Diese Werkzeuge sind spezialisierte Codegeneratoren (oft *Wizards* oder *Assistenten* genannt), die anhand von Parametrierungen dem Anwendungsentwickler die einfache Erstellung eines Anwendungsgerüsts auf der Basis eines Frameworks ermöglichen. Für die meisten kommerziellen Frameworks sind entsprechende Wizards verfügbar. Das in der Entwicklungsumgebung Visual C++ ® von Microsoft eingesetzte MFC-Framework besitzt beispielsweise einen sogenannten *MFC AppWizard*. Dieser Wizard ist in der Lage, mit wenigen Angaben des Anwendungsentwicklers eine lauffähige, MFC-basierte Anwendung zu generieren [Sche99]. Abbildung 3.2 zeigt ein typisches Dialogfenster des MFC AppWizards, welches zur Abfrage von Anwendungsinformationen dient.

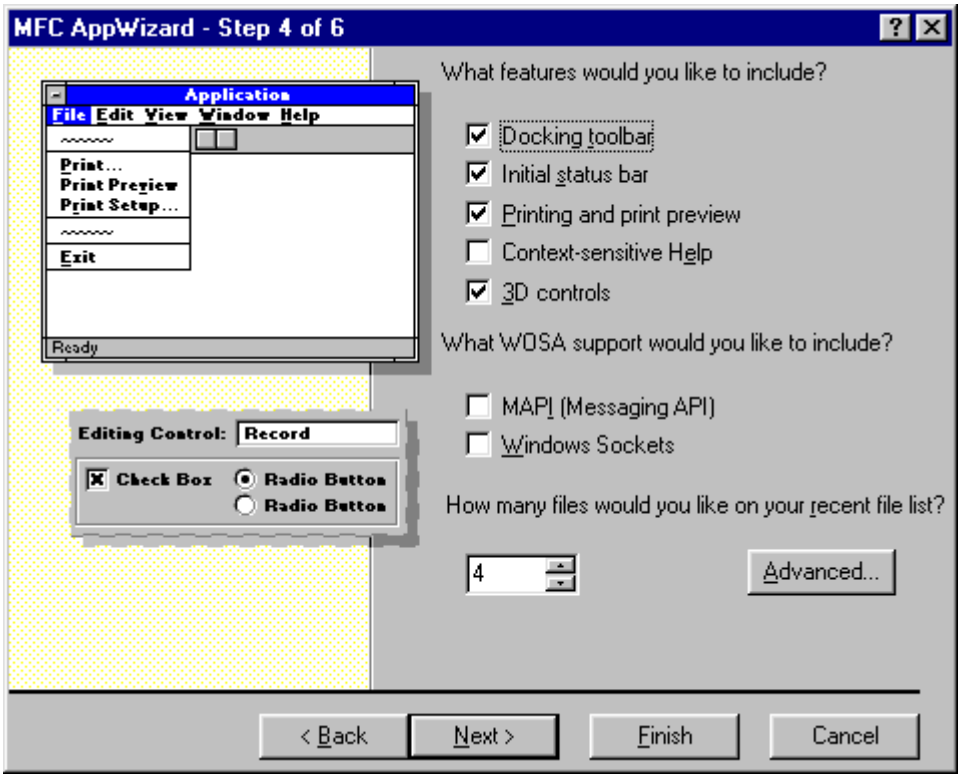

Abbildung 3.2: Dialogfenster des MFC AppWizards

Als größter Vorteil dieser Werkzeuge ist die einfache Erstellung von Quellcode zu nennen, die es sogar Nicht-Programmierern ermöglicht, ein frameworkbasiertes Anwendungsgerüst (sog. Startup-Code) zu erstellen. Um dieses Gerüst zu einer Anwendung zu vervollständigen, sind jedoch auch weiterhin fundierte Programmier- und Framework-Kenntnisse notwendig. Das Werkzeug unterstützt den Anwendungsentwickler somit lediglich bei der anfänglichen Anbindung seiner Anwendung an das Framework. Im restlichen Instanziierungsprozess, der sehr anwendungsspezifisch und aufwändig ist, bleibt der Anwendungsentwickler auch weiterhin ohne die benötigte Unterstützung. Zudem sind diese Werkzeuge streng an ein bestimmtes Framework gebunden – für jedes Framework muss ein spezieller Wizard erstellt werden, was für den Frameworkentwickler einen deutlichen Zusatzaufwand darstellt.

3.4.3 Bewertung der Ansätze

Während die oben beschriebenen Bemühungen zur Verbesserung der Unterstützung des Anwendungsentwicklers im Instanziierungsprozess interessante Lösungen für Teile der Gesamtproblematik bieten, so wird doch die Problematik im vollen Umfang nicht in Angriff genommen. Jeder der vorgeschlagenen Ansätze beschreibt die Lösung für ein eingeschränktes Teilproblem der Instanziierung, ohne den restlichen Teil zu berücksichtigen. Cookbooks zeigen beispielsweise anhand einer vordefinierten Anwendung die Vorteile einer schrittweisen Instanziierung auf, ohne Ansatzpunkte für eine Werkzeugunterstützung zu geben. Wizards hingegen zeigen einen guten Ansatz für die Werkzeugunterstützung auf, verlassen sich jedoch

auf die Programmierkenntnisse des Entwicklers zur Vervollständigung der Anwendung. Der Anwendungsentwickler benötigt jedoch einen integrativen Ansatz, der den gesamten Instanziierungprozess abdeckt und sowohl Dokumentations- als auch Werkzeugaspekte umfasst.

3.5 Anforderungen an die Instanziierung

3.5.1 Anforderungen an die Unterstützung des Benutzers

Aus den oben gemachten Ausführungen folgt, dass bestehende Ansätze immer nur Lösungen für Teilbereiche des gesamten Instanziierungsprozesses bieten. Zudem sind nicht alle Teilbereiche durch entsprechende Lösungsansätze abgedeckt. Es stellt sich somit die Frage nach einem umfassenden Konzept, das die Vorteile bestehender Ansätze integriert und gleichzeitig den gesamten Instanziierungsprozess abdeckt. Aus den oben beschriebenen Vorteilen und Problemen aktueller Ansätze leiten sich folgende Anforderungen für ein umfassendes Gesamtkonzept ab, welches die Unterstützung des Anwendungsentwicklers gewährleistet.

- *Allgemeine Unterstützung für die Framework-Instanziierung*: Das Instanziierungskonzept und das darauf aufbauende Werkzeug sollen nicht an ein bestimmtes Framework gebunden sein. Es soll eine Unterstützung unterschiedlicher Frameworks ermöglicht werden, ohne umfangreiche Werkzeugmodifikationen vorauszusetzen.

- *Schrittweise Instanziierung*: Der Entwickler soll vom Framework zu seiner speziellen Anwendung in kleinen, überschaubaren und nachvollziehbaren Schritten gelangen. Die Framework-Instanziierung ist für den Entwickler ein iterativer Prozess, bei dem am Ende die spezifische Anwendung entsteht.

- *Aktive Benutzerführung*: Der Entwickler wird aktiv durch den Instanziierungsprozess geleitet. Er bleibt zu keinem Zeitpunkt der Instanziierung sich selbst überlassen und muss sich keine Gedanken machen, ob wichtige Schritte vergessen wurden.

- *Unterstützung von Domänen-Experten*: Das Konzept soll die Framework-Instanziierung für Personengruppen ermöglichen, die zwar ausreichende Domänenkenntnisse, aber nur wenige oder sogar keine Programmierkenntnisse besitzen. Die Fragen des Domänen-Experten („Was soll entwickelt werden?“), und nicht die Fragen des Programmier-Experten („Wie soll es entwickelt werden?“) sollen bei der Anwendungsentwicklung in den Vordergrund treten.

- *Kopplung von Werkzeugunterstützung und Dokumentation*: Das Lesen von Benutzungsdokumentation für die Framework-Instanziierung und der eigentliche Instanziierungsprozess sollen eine Einheit bilden. Die Anwendungsentwicklung sollte erfolgen, *während* der Entwickler die dazu benötigte Dokumentation liest.

- *System- und Plattformunabhängigkeit*: Das zu entwickelnde Konzept soll weitestgehend von speziellen Randbedingungen (z.B. Betriebssystem, Programmiersprache, Datenformate, u.s.w.) unabhängig sein.

3.5.2 Anforderungen der Automatisierungstechnik

Die Automatisierungstechnik umfasst ein breites Gebiet unterschiedlichster Ausprägungen und Anforderungen, was sich zwangsläufig auch in der dazu gehörigen Automatisierungssoftware widerspiegelt. Das Spektrum reicht von Softwaresystemen für komplexe Produktionsanlagen mit Hunderten von Sensoren und Aktoren bis hin zu eingebetteten Systemen, wo der Software-Einsatz nicht offensichtlich ist (z.B. Kühlschrank, Waschmaschine). Aus diesem Grund gibt es auch keinen einheitlichen Satz von Anforderungen, der für alle Automatisierungssysteme gleichermaßen gilt. Es gibt jedoch eine Reihe von Anforderungen, die im Zusammenhang mit Automatisierungssoftware besonders häufig auftauchen:

- *Echtzeit-Eigenschaften*: Die Korrektheit der Informationsverarbeitung ist nicht nur vom Resultat der Berechnung, sondern auch davon abhängig, *wann* dieses Resultat bereitgestellt wird [LaGö98]. Interessant ist dabei die Tatsache, dass nicht die globale Ausführungsgeschwindigkeit der Software im Vordergrund steht, sondern die Reaktion innerhalb definierter und vorhersehbarer Zeitschranken.

- *Effizienter Umgang mit Ressourcen*: Die Hardwareausstattung des Rechnersystems (z.B. Prozessor, Speicherplatz) wird in der Automatisierungstechnik oft minimal gehalten, um die Gesamtkosten des Systems bei hohen Stückzahlen niedrig zu halten. Deshalb ist der effiziente und sparsame Umgang der Software mit den Hardware-Ressourcen erforderlich.

- *Sicherheit und Zuverlässigkeit*: Während Sicherheit die Verhinderung von Gefahrensituationen beinhaltet, betrachtet die Zuverlässigkeit vor allem die Verhinderung von Ausfällen. Sicherheits- und Zuverlässigkeitsanforderungen sind für Automatisierungs-systeme bedeutsam, bei denen ein Fehlverhalten große materielle Verluste oder sogar die Gefährdung von Menschenleben bedeuten kann.

Die genannten Anforderungen kann man als *nichtfunktional* betrachten, da sie nicht direkt aus der gewünschten Funktionalität des Systems resultieren, sondern eine Folge der Randbedingungen der Ausführungsumgebung und der Interaktion mit dem technischen Prozess sind. Weiterhin können diese nichtfunktionalen Anforderungen unabhängig voneinander in nahezu beliebigen Kombinationen auftreten. Ein Automatisierungssystem kann beispielsweise

gleichzeitig echtzeit-, ressourcen- und sicherheitsbezogene Anforderungen enthalten. Es sind jedoch auch Automatisierungssysteme möglich, die keine dieser Anforderungen enthalten.

Außer den allgemeinen Anforderungen an ein Konzept für die Unterstützung der Framework-Instanziierung (Abschnitt 3.5.1), die natürlich auch für die Automatisierungstechnik gültig sind, müssen zusätzliche Anforderungen für die Anwendungsentwicklung beachten werden, die speziell durch das Anwendungsgebiet der Automatisierungstechnik hinzukommen.

- *Spezifikation und Dokumentation nichtfunktionaler Eigenschaften*: Der Frameworkentwickler muss die Möglichkeit erhalten, nichtfunktionale Eigenschaften seines Frameworks (wie z.B. Laufzeit, Speicherplatz) und damit verbundene Einschränkungen für die Anwendungsentwicklung explizit auszudrücken.

- *Nichtfunktionale Garantien*: Der Anwendungsentwickler möchte nicht erst nach der abgeschlossenen Instanziierung erfahren, dass seine Anwendung im gewünschten Zielsystem nicht lauffähig ist, weil beispielsweise Zeitschranken nicht eingehalten werden können. Bereits im Instanziierungsprozess muss der Anwendungsentwickler auf potenzielle Verletzung von nichtfunktionalen Eigenschaften hingewiesen werden. Der Frameworkentwickler kann bestimmte nichtfunktionale Eigenschaften einer auf seinem Framework basierenden Anwendung zusichern, wenn gleichzeitig die von ihm geforderten Voraussetzungen eingehalten werden.

In diesem Kapitel wurden die Schwierigkeiten untersucht, die aus der Sicht des Anwendungsentwicklers an die Framework-Technologie gekoppelt sind. Es wurden auch bestehende Lösungsansätze aus der Literatur vorgestellt und bewertet. Auf der Grundlage dieser Betrachtungen wurde deutlich, dass bestehende Lösungsansätze immer nur Teilaspekte der Unterstützung des Anwendungsentwicklers bei der Instanziierung von Frameworks aufgreifen. Aus diesem Grund wird ein neues Lösungskonzept benötigt, welches die Instanziierungsproblematik in ihrer Gesamtheit betrachtet. Im folgenden Kapitel werden zunächst *Komponenten-Frameworks* als eine neue Framework-Art vorgestellt, die im Vergleich zu den verbreiteten objektorientierten Frameworks bessere Voraussetzung für die Unterstützung des Anwendungsentwicklers mitbringt. Danach wird in Kapitel 5 ein neues Verfahren vorgestellt, dass die effiziente, werkzeuggestützte Anwendungsentwicklung auf der Grundlage von Komponenten-Frameworks ermöglicht.

4 Grundlagen der Komponenten-Frameworks

In diesem Kapitel wird das Konzept der Komponenten-Frameworks eingeführt und definiert. Es wird erklärt, warum das bloße Bereitstellen von Komponenten in entsprechenden Bibliotheken nicht ausreichend für eine effiziente Unterstützung des Anwendungsentwicklers ist. Es wir auch erläutert, warum ein übergeordnetes Wiederverwendungskonzept in Form von Komponenten-Frameworks notwendig ist und welche Merkmale dieses Wiederverwendungskonzept kennzeichnen. Abschließend werden Komponenten-Frameworks mit den verbreiteten objektorientierten Frameworks verglichen und die Unterschiede beider Wiederverwendungs-konzepte herausgearbeitet.

4.1 Wiederverwendung mit Komponenten-Bibliotheken

Nachdem im vorangehenden Kapitel geschildert wurde, mit welchen Schwierigkeiten bei der frameworkbasierten Anwendungsentwicklung gerechnet werden muss, drängt sich die Frage nach alternativen Wiederverwendungskonzepten auf. Komponenten, als abgeschlossene binäre Einheiten beliebiger Granularität, bieten sich als nahe liegende Alternative bei der Anwendungsentwicklung an. Mit Hilfe von Komponenten-Bibliotheken und umfangreicher Dokumentation könnte der Anwendungsentwickler ebenfalls benutzerspezifische Anwendungen zusammenbauen, ohne einen großen Programmieraufwand in Kauf nehmen zu müssen. Leider ist auch dieses naheliegende Vorgehen, welches ausschließlich auf dem Zusammenbau von einzelnen, voneinander unabhängigen Komponenten beruht, durch praktische Schwierigkeiten belastet. In [Szyp98] wird dieser Sachverhalt deutlich zum Ausdruck gebracht:

> „It is far too simplistic to assume that components are simply selected from catalogs, thrown together, and magic happens. In reality, the disciplined interplay of components is one of the hardest problems of software engineering today.“

Folglich ist die bloße Bereitstellung von geeigneten Komponenten für eine erfolgreiche komponentenbasierte Anwendungsentwicklung nicht ausreichend. Falls der Anwendungsentwickler die Komponenten nicht selbst implementiert hat oder Komponenten unterschiedlicher Hersteller kombinieren möchte, sind meistens Schnittstellenprobleme die Folge [GAO95]. Die Semantik einer Komponenten-Schnittstelle ist nicht offensichtlich und kann auch nicht innerhalb der Komponente formal spezifiziert werden. Der Anwendungsentwickler benötigt somit eine ausführliche Dokumentation, welche die Funktionalität, die Schnittstellen und die Voraussetzungen für den erfolgreichen Komponenteneinsatz präzise dokumentiert. Mit Hilfe dieser Dokumentation muss er in die Lage versetzt werden, typische Instanziierungsfragen zu beantworten, wie z.B.:

- Welche Komponenten werden für die gewünschte Anwendung benötigt?
- Welche Verbindungen müssen zwischen den Komponenten existieren?
- Welche Komponenten Ein- bzw. Ausgänge müssen für diese Verbindungen verwendet werden?
- Welche Abhängigkeiten ergeben sich durch die Auswahl bestimmter Optionen?
- Welche Komponenten-Kombinationen sind nicht erlaubt?
- Wie müssen die einzelnen Komponenten parametriert werden?
- Wo findet man Informationen zu einzelnen, detaillierten Instanziierungsfragen?

Diese unvollständige Liste von Fragen macht deutlich, dass selbst umfangreiche und sorgfältige Dokumentation von Komponenten nicht alle Fragen ausreichend abdecken kann. Werden Komponenten jedoch nicht als unabhängiges Wiederverwendungskonzept betrachtet, sondern in einen Framework-Kontext eingebettet, so können die meisten der oben genannten Probleme vermieden werden. In diesem Fall gehen Komponenten und Frameworks eine wechselseitige Beziehung ein – Das Framework stellt die Semantik zwischen den Komponenten zur Verfügung und die Komponenten repräsentieren die funktionalen Bestandteile des Frameworks. Diese Kernidee ist die Grundlage für die so genannten *Komponenten-Frameworks*, einer neuen Framework-Art, die in dieser Arbeit eine zentrale Rolle spielt und im nachfolgenden Abschnitt beschrieben wird.

4.2 Definition von Komponenten-Frameworks

Komponenten-Frameworks sind – in Anlehnung an die Philosophie von objektorientierten Frameworks – eine generische Lösung für einen bestimmten Anwendungsbereich. Durch Komponenten-Frameworks wird eine Architektur für einen bestimmten Problembereich definiert und somit ein Rahmen für das Zusammenspiel von Komponenten festgelegt. Jedes vollständige Komponenten-Framework ist durch zwei wichtige obligatorische Bestandteile gekennzeichnet:

- *Komponenten*, welche die Funktionalität für die frameworkbasierten Anwendungen implementieren und einer
- *Frameworkbeschreibung*, welche die Summe aller möglichen Anwendungen beschreibt, die durch Selektion, Konfiguration und Parametrierung aus den Komponenten erstellt werden können.

Komponenten sind Software-Einheiten, die auf der Basis von bestehenden Komponentenmodellen wie z.B. COM oder JavaBeans realisiert werden. Einzelne Komponenten können parametriert oder durch alternative Implementierungen ausgetauscht werden. Sie können jedoch nicht bezüglich ihres Implementierungscodes verändert werden. Jede Komponente realisiert einen wesentlichen funktionalen Aspekt des Anwendungsbereichs, für den das Framework erstellt wurde. Die Granularität kann dabei vom Komponenten-Entwickler beliebig gewählt werden. In der Automatisierungstechnik könnte eine Komponente beispielsweise einzelne Hardware-Bauteile wie Sensoren und Aktoren repräsentieren und den Zugriff auf diese Bauteile kapseln (z.B. Treiber-Komponente). Eine einzige Komponente könnte aber auch eine komplexe Steuerung repräsentieren, die Hunderte von einzelnen Hardware-Bauteilen umfasst.

Die Frameworkbeschreibung erfüllt gleichzeitig zwei wesentliche Aufgaben. Sie spezifiziert einerseits alle möglichen Anwendungen, die aus dem Framework erstellt werden können. Auf diese Weise wird explizit und präzise dokumentiert, welche Produkte aus dem Framework entstehen können. Die Frameworkbeschreibung ermöglicht andererseits auch die Anwendungsentwicklung in einer automatisierten Werkzeugumgebung. Sie ist die Eingangsinformation für das Instanziierungswerkzeug, mit dem der Anwendungsentwickler auf bequeme Weise seine spezifische Anwendung erstellen kann.

Abbildung 4.1 veranschaulicht noch einmal grafisch die Frameworkbeschreibung und die dazu gehörigen Komponenten als Bestandteile von Komponenten-Frameworks. Im Instanziierungsprozess erzeugt der Anwendungsentwickler werkzeuggestützt durch Auswahl und Parametrierung seine gewünschte Anwendung, die als eine Anwendungsvariante in der Frameworkbeschreibung definiert ist. Im Gegensatz zur gewöhnlichen komponentenbasierten Softwareentwicklung bedeuten Komponenten-Frameworks eine Einschränkung, da Variabilitäten im Voraus festgelegt werden und nicht alle denkbaren Komponentenkombinationen erlaubt sind. Dafür erhält der Anwendungsentwickler jedoch einen definierten und bequemen Instanziierungsprozess, der letztendlich eine funktionstüchtige Anwendung gewährleistet. Dadurch kann auch ein Domänenexperte ohne umfangreiche Programmierkenntnisse individuelle Anwendungen erstellen, ohne an Implementierungsdetails und Schnittstellenproblemen zu scheitern.

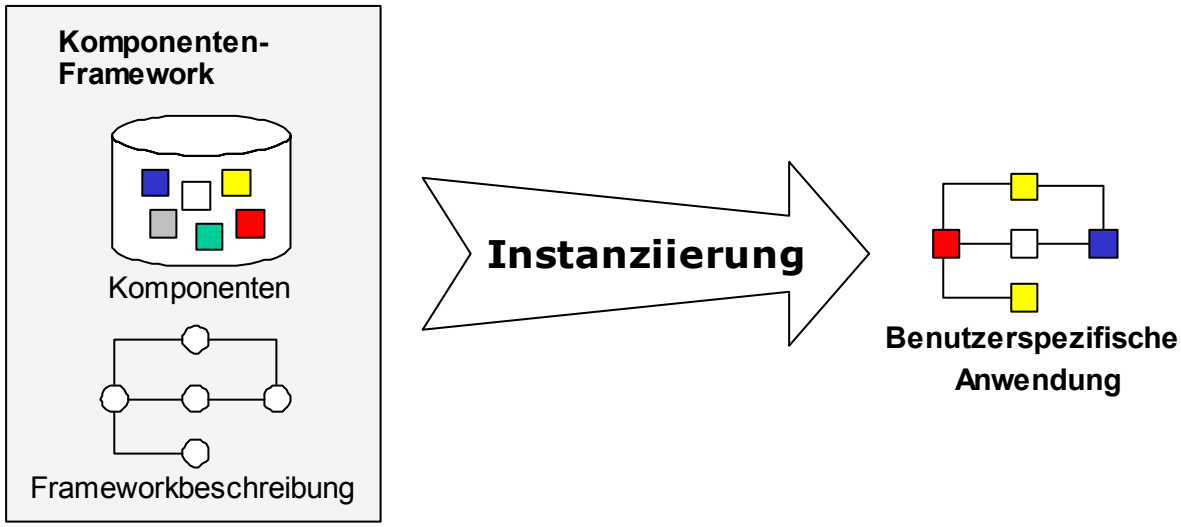

Abbildung 4.1: Instanziierung von Komponenten-Frameworks

Wegen der zentralen Rolle, die der Frameworkbeschreibung bei der Definition von Komponenten-Frameworks zukommt, sollen im nachfolgenden Abschnitt die charakteristischen Eigenschaften von Frameworkbeschreibungen detaillierter erläutert werden. In Kapitel 6 wird dann ausführlich die vorgeschlagene Spezifikationssprache beschrieben, die zur Erstellung von Frameworkbeschreibungen dient.

4.3 Eigenschaften der Frameworkbeschreibung

4.3.1 Variabilitätsebenen

Jede Frameworkbeschreibung eines Komponenten-Frameworks muss in der Lage sein, sowohl invariable als auch variable Elemente zukünftiger Anwendungen auszudrücken. Während die invariablen Elemente sich in jeder resultierenden Anwendung unverändert wieder finden, sind die variablen Elemente von besonderem Interesse. Sie ermöglichen die Flexibilität bei der Anwendungsentwicklung, die für Frameworks charakteristisch ist. Bei komponentenbasierten Anwendungen sind drei verschiedene Ebenen der Variabilität zu finden, die in der Frameworkbeschreibung berücksichtigt werden müssen und im Nachfolgenden erläutert sind.

Architekturvariabilität

Komponentenbasierte Anwendungen sind entscheidend durch die Verbindungen zwischen den einzelnen Komponenten charakterisiert. Diese Verbindungsstrukturen repräsentieren zusammen mit den Komponenten die *Architektur* einer komponentenbasierten Anwendung [ShGa96]. Da ein Komponenten-Framework nicht nur eine einzige, sondern eine Vielzahl von komponentenbasierten Anwendungen beschreibt, muss die entsprechende Framework-beschreibung ebenfalls die Spezifikation von *Architekturvariabilitäten* ermöglichen. Es muss

also möglich sein, mit Hilfe der Frameworkbeschreibung eine beliebige Anzahl von zusammenhängenden Anwendungsarchitekturen festzulegen, die alle zu einem definierten Anwendungsbereich gehören.

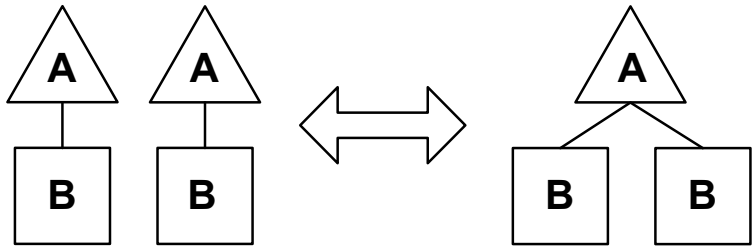

Abbildung 4.2: Variabilitäten auf der Architekturebene

Abbildung 4.2 zeigt ein Beispiel für eine Architekturvariabilität: Auf der linken Seite ist jede Komponente vom Typ B mit einer entsprechenden Komponente vom Typ A verbunden. Auf der rechten Seite hingegen sind alle Komponenten vom Typ B mit nur einer einzigen Komponente vom Typ A verbunden. Überträgt man dieses Beispiel auf den Anwendungsbereich der Automatisierungstechnik, so könnte die in der Abbildung 4.2 gezeigte Architekturvariabilität beispielsweise bedeuten, dass einzelne Aktoren-Komponenten (Typ B) entweder alle mit einer einzigen Steuerungskomponente (Typ A) verbunden sind, oder dass jede Aktor-Komponente ihre eigene Steuerungskomponente besitzt.

Zur Kategorie der Architekturvariabilitäten gehören auch mehrfache Instanziierungen von Komponenten und optionale Komponenten. Bei mehrfachen Instanziierungen wird dieselbe Komponente beliebig oft instanziiert, wobei die Verbindungen zu den anderen Komponenten bei jeder Instanz identisch sind. Optionale Komponenten sind solche, die nur auf Wunsch des Entwicklers in der resultierenden Anwendung erscheinen. Zu dieser Gruppe könnte beispielsweise eine Watchdog-Komponente[3] gehören, die nur auf ausdrücklichen Wunsch des Entwicklers in die Anwendung miteinbezogen wird.

Komponentenvariabilität

Ein weiterer wichtiger Aspekt in der Frameworkbeschreibung ist die Forderung nach Austauschbarkeit von Komponenten. Hierbei wird die Absicht verfolgt, semantisch ähnliche Komponenten mit unterschiedlichen Implementierungseigenschaften in einer Architektur gegeneinander austauschen zu können. Diese Eigenschaft wird beispielsweise benötigt, wenn in einer Steuerungssoftware unterschiedliche Berechnungsalgorithmen als Komponenten zur Verfügung stehen sollen, ohne die restliche Software zu beeinflussen. Eine Komponente könnte die Berechnung besonders präzise durchführen, dafür aber mehr Speicherplatz und Rechenzeit

[3] Komponente zur Überwachung der korrekten Funktionsweise anderer Komponenten

beanspruchen. Eine andere Komponente rechnet weniger präzise, besitzt dafür aber ein zeitlich deterministisches Verhalten. Beide Komponenten erfüllen dieselbe Aufgabe, wobei der Anwendungsentwickler sich in Abhängigkeit von weiteren Randbedingungen für eine der beiden Komponenten entscheiden wird.

Dieser genannte Aspekt der Frameworkbeschreibung wird als *Komponentenvariabilität* bezeichnet. Abbildung 4.3 veranschaulicht grafisch den beschriebenen Sachverhalt: An den in der Frameworkbeschreibung vorgesehenen Platz für die Komponente von Typ A kann sowohl die Komponenten-Implementierung A_1 als auch A_2 eingesetzt werden. Da beide Implementierungen an der Schnittstelle identisch sind, werden weder die Anwendungsarchitektur noch andere Komponenten durch diese Variabilität beeinflusst.

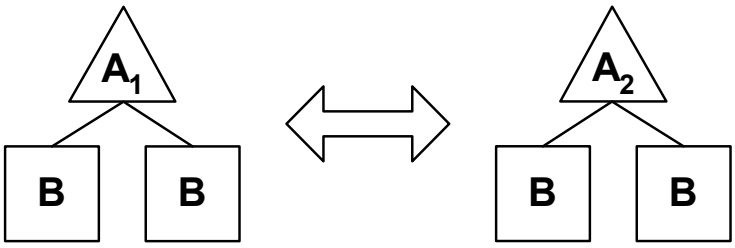

Abbildung 4.3: Variabilitäten auf der Ebene von Komponenten

Instanzenvariabilität

Nachdem die bisher beschriebenen Variabilitätsebenen der Frameworkbeschreibung nur ein Vordringen bis zur Komponentenschnittstelle ermöglichen, müssen auch Variabilitäten innerhalb der Komponente berücksichtigt werden. Da eine Komponente verschiedene Parameter besitzen kann, die ihr Verhalten und ihr Erscheinungsbild beeinflussen, müssen auch diese Parameter von der Frameworkbeschreibung berücksichtigt werden. Da diese Parametrierungsmöglichkeiten erst auf der Ebene von Komponenten-Instanzen angeboten werden, wird dieser Aspekt als *Instanzenvariabilität* bezeichnet.

Abbildung Abbildung 4.4 stellt zwei Systemausschnitte gegenüber, die bezüglich der Verbindungen und der Komponenten vollkommen identisch sind. Lediglich die Eigenschaft „Farbe“ der Komponenten-Instanzen von A_1 ist auf der linken Seite mit dem Wert „weiss“ und auf der rechten mit dem Wert „grau“ parametriert.

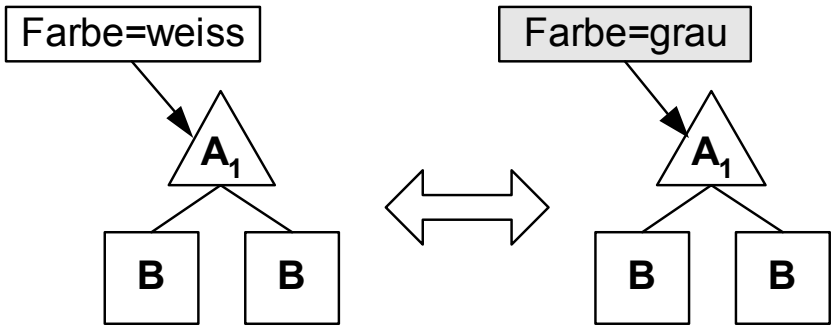

Abbildung 4.4: Variabilitäten auf der Ebene von Instanzen

4.3.2 Benutzerspezifische Komponenten

Die bisher vorgestellten Eigenschaften von Frameworkbeschreibungen ermöglichen eine beliebige Anzahl von Anwendungsvarianten, so lange diese vom Frameworkentwickler eingeplant sind. Möchte der Anwendungsentwickler jedoch die vom Frameworkentwickler vorgesehenen Instanziierungspfade verlassen, kann er mit keinerlei Unterstützung rechnen. Diese Abgeschlossenheit kann in der Praxis zu Problemen führen, da der Frameworkentwickler im Normalfall nie alle Forderungen des Anwendungsentwicklers vorhersehen kann. Ein starres Framework kann somit wegen kleinster Abweichungen von der ursprünglichen Absicht des Frameworkentwicklers für den Anwendungsentwickler vollkommen wertlos sein. Aus diesem Grund bietet die Frameworkbeschreibung die Möglichkeit, an bestimmten Stellen *benutzerspezifische Komponenten* miteinzubeziehen, ohne dadurch die Vorteile von Komponenten-Frameworks preiszugeben.

Abbildung 4.5 veranschaulicht die Verwendung von benutzerspezifischen Komponenten in der Frameworkbeschreibung. Innerhalb der Frameworkbeschreibung wird eine Schnittstelle von Typ B definiert, die für alle Komponenten verbindlich ist und die als Platzhalter in der Architektur fungiert. Gleichzeitig wird explizit vereinbart, dass dieser Platzhalter auch durch benutzerspezifische Komponenten ersetzt werden darf. Auf der linken Seite wird der Platzhalter durch die Komponente B_1 ersetzt, die vom Frameworkentwickler dafür vorgesehen ist. Auf der rechten Seite hingegen wird der Platzhalter durch die externe, benutzerspezifische Komponente B_{ext} ersetzt, die nicht im Lieferumfang des Komponenten-Frameworks vorhanden war. Diese nachträgliche Erweiterbarkeit könnte beispielsweise dazu genutzt werden, um ein neues Hardware-Bauteil anzusteuern, für das im ursprünglichen Framework keine Treiber-Komponente vorhanden war.

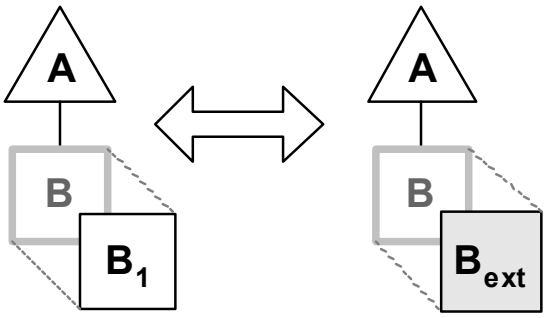

Abbildung 4.5: Benutzerspezifische Komponenten-Implementierungen

4.4 Unterschiede zu objektorientierten Frameworks

Komponenten-Frameworks haben mit ihren objektorientierten Pendant viele Gemeinsamkeiten bezüglich der Absicht, die sie verfolgen: Sie bieten beide eine vordefinierte Lösung für einen bestimmten Problembereich, in dem sie Anwendungs- und Interaktionsstrukturen festlegen und dadurch eine Vielzahl von Entwurfsentscheidungen vorwegnehmen. Beide möchten anwendungsübergreifende Gemeinsamkeiten für die Entwicklung neuer Anwendungen bereitstellen, damit der Entwickler sich vollständig auf die spezifischen Merkmale seiner Anwendung konzentrieren kann.

Unterhalb dieser abstrakten Ebene gemeinsamer Ziele und Absichten gibt es jedoch klare Unterschiede bezüglich der Weise, wie diese Absichten verfolgt werden. Besonders deutlich treten die Unterschiede ans Tageslicht, wenn die beiden Aspekte *Implementierung des Frameworks* und *Anwendungsentwicklung mit dem Framework* gegenüber gestellt werden.

Unterschiede bei der Implementierung

Ein objektorientiertes Framework wird normalerweise in einer objektorientierten Programmiersprache implementiert, während dieses Kriterium nicht zwingend für ein Komponenten-Framework gelten muss. Prinzipiell könnten alternative Komponenten auch in unterschiedlichen Programmiersprachen realisiert werden, solange das korrekte Zusammenspiel im Framework sichergestellt ist.

Das Komponenten-Framework kapselt alle Implementierungsdetails innerhalb der Komponente, d.h. jede Komponente repräsentiert ein wesentliches Element der Domäne. Bei objektorientierten Frameworks ist die Trennung in unterstützende und domänenrelevante Implementierungselemente nicht vorhanden. Hier können beispielsweise unterstützende Hilfsklassen (z.B. Liste, Ausgabeformatierung) aus der Implementierungssicht nicht von wesentlichen Domänenklassen (z.B. Sensor, Steuerung) unterschieden werden.

Jedes Komponenten-Framework besitzt eine saubere Trennung zwischen der Implementierung der Funktionalität (in Form von Komponenten) und der Architektur, die in der Frameworkbeschreibung spezifiziert wird. Bei objektorientierten Frameworks ist diese Trennung bestenfalls implizit vorhanden, da sowohl funktionelle als auch strukturelle Aspekte durch Klassen ausgedrückt werden.

Unterschiede bei der Anwendungsentwicklung

Aus einem objektorientierten Framework entstehen Anwendungen, indem bestimmte Klassen abgeleitet und dafür vorgesehene Methoden überschrieben werden. Bei Komponenten-Frameworks steht die Auswahl von Komponenten und die Parametrierung im Vordergrund. Als Folge der verwendeten Vererbungstechnik wird bei objektorientierten Frameworks meistens auf der Quellcode-Ebene gearbeitet: der Anwendungsentwickler arbeitet in der Programmiersprache, in der auch das Framework implementiert wurde. Bei Komponenten-Frameworks muss der Anwendungsentwickler nicht zwangsläufig mit Programmiersprachen arbeiten. Er kann zur Instanziierung auf verschiedene Skriptsprachen oder auf geeignete Instanziierungswerkzeuge zurückgreifen. Das Komponenten-Framework unterschiedet deutlich zwischen der Rolle des Framework- und des Anwendungsentwicklers, da beide auf unterschiedlichen Abstraktionsebenen arbeiten (siehe Abbildung 4.6).

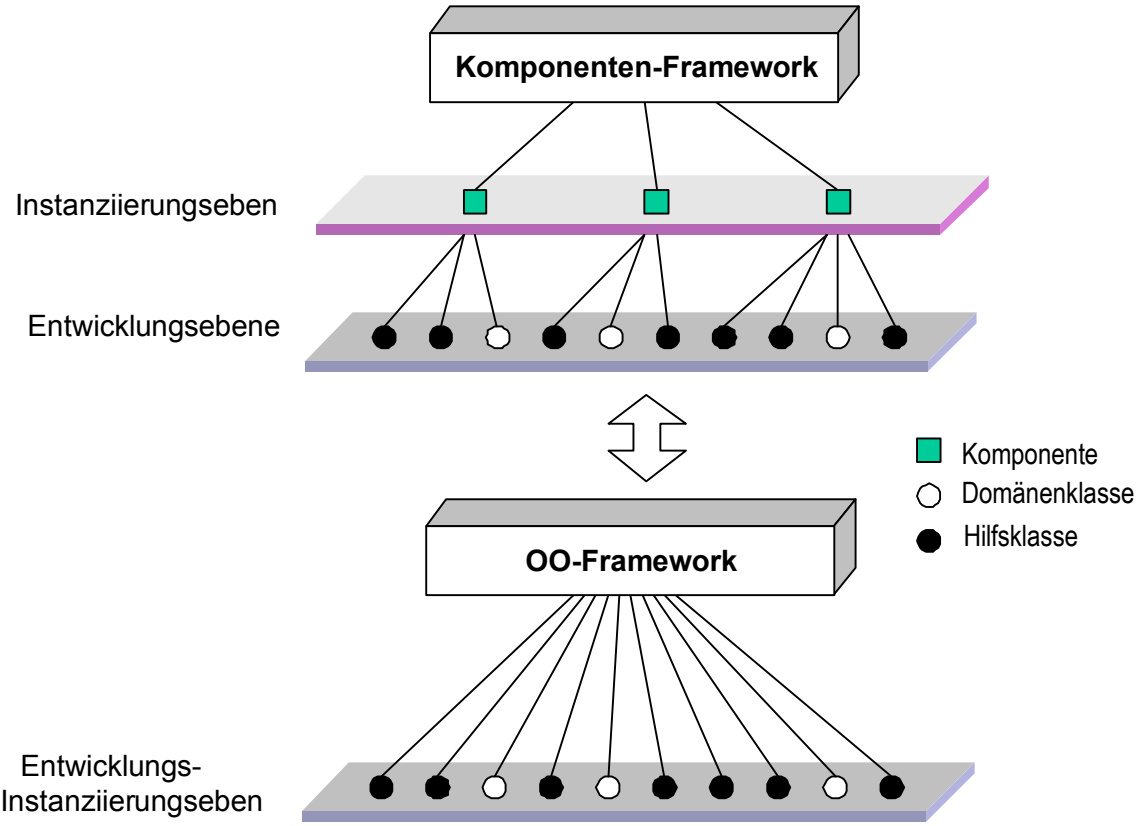

Abbildung 4.6: Anwendungsentwicklung mit Komponenten-Frameworks und objektorientierten Frameworks

Ein weiterer wichtiger Unterschied bezieht sich auf das Ergebnis der Framework-Instanziierung: Objektorientierte Frameworks sind darauf ausgelegt, zu einer einzelnen, monolithischen Anwendung vervollständigt zu werden. Komponenten-Frameworks können zwar auch eigenständige Anwendungen erzeugen, sie können jedoch auch neue Komponenten erzeugen, die als benutzerspezifische Komponenten in einem übergeordneten Anwendungskontext zur Verfügung stehen. Auf diese Weise wird eine Schachtelung von Frameworks möglich – Eine frameworkbasierte Anwendung kann somit auch aus Komponenten bestehen, die selbst das Ergebnis einer Framework-Instanziierung sind. Mit dieser Technik könnte beispielsweise eine komplexe Steuerungskomponente für eine Fertigungsmaschine bereitgestellt werden, die der Anwendungsentwickler durch eine individuelle grafische Benutzungsoberfläche zu einer Anwendung vervollständigt.

Tabelle 4.1 zeigt noch einmal zusammenfassend die Gemeinsamkeiten und Unterschiede zwischen objektorientierten Frameworks und Komponenten-Frameworks.

Tabelle 4.1: Vergleich von Komponenten-Frameworks und objektorientierten Frameworks

	Komponenten-Framework	**Objektorientiertes Framework**
Domänenbindung	Problemlösung für definierte Domänen	
Anwendungs- und Interaktionsstrukturen	Festlegung von Entwurfs- und Implementierungs-entscheidungen	
Trennung von Domänen- und Hilfselementen	Komponenten kapseln Hilfselemente	Keine Trennung zwischen Domänen- und Hilfselementen
Implementierungssprache für das Framework	Freie Sprachauswahl	Normalerweise objektorientierte Implementierungssprache
Anwendungsentwicklung	Auswahl, Konfiguration und Parametrierung von Komponenten	Vererbung, Spezialisierung von Implementierungsklassen
Trennung zwischen Framework- und Anwendungsentwickler	Framework- und Anwendungs-entwickler arbeiten auf unterschied-lichen Abstraktionsebenen	Framework- und Anwendungs-entwickler arbeiten auf gleicher Abstraktionsebene
Instanziierungsergebnis	Anwendung oder komplexe Komponente	Monolithische Anwendung

In diesem Kapitel wurde das Konzept der Komponenten-Frameworks vorgestellt und erläutert. Komponenten-Frameworks ermöglichen eine besonders effiziente Anwendungsentwicklung, da sowohl Implementierungscode als auch Anwendungsstrukturen wiederverwendet werden. In der Praxis können diese Vorteile jedoch nur verwirklicht werden, wenn das Komponenten-Framework zusammen mit einem geeigneten Verfahren zur Anwendungsentwicklung verwendet wird. Im folgenden Kapitel wird ein Verfahren vorgestellt, das auf Komponenten-Frameworks basiert und eine effiziente, werkzeuggestützte Anwendungsentwicklung ermöglicht.

5 Verfahren zur Anwendungsentwicklung mit Komponenten-Frameworks

Im vorangegangenen Kapitel wurde erläutert, dass Komponenten-Frameworks einen Fortschritt in Bezug auf die Instanziierungsproblematik bedeuten. Sie ermöglichen die Beschreibung aller möglichen Instanziierungsprodukte, die aus dem Framework entstehen können. Komponenten-Frameworks sind somit eine notwendige Voraussetzung für eine effiziente Unterstützung des Anwendungsentwicklers. Komponenten-Frameworks alleine sind jedoch noch nicht in der Lage, alle Anforderungen zu erfüllen, die in Abschnitt 3.5 für eine umfassende Unterstützung des Anwendungsentwicklers gefordert wurden. Sie stellen die Basis-Technologie dar, auf die weiterführende Konzepte aufsetzen müssen.

In diesem Kapitel wird ein Verfahren vorgestellt, das auf Komponenten-Frameworks aufbaut und den Kreis zur benutzungsfreundlichen, frameworkbasierten Anwendungsentwicklung schließen. Dazu werden zunächst die speziellen Rahmenbedingungen diskutiert, die bei der Instanziierung vom Komponenten-Frameworks beachtet werden müssen. Anschließend werden die Grundlagen des Instanziierungsverfahrens präsentiert, die aus einer Spezifikationssprache für Frameworkbeschreibungen und einem passenden Instanziierungswerkzeug bestehen. Abschließend werden konzeptionelle Alternativen vorgestellt und mit dem vorgeschlagenen Verfahren zur Instanziierung von Komponenten-Frameworks verglichen.

5.1 Instanziierung von Komponenten-Frameworks

5.1.1 Abhängigkeiten zwischen Variabilitätsebenen

Frameworks spezifizieren eine Reihe von Anwendungen für eine bestimmte Domäne, wobei sowohl variable als auch invariable Anwendungselemente (Hot bzw. Frozen Spots) betrachtet werden. Während die invariablen Elemente feste Bestandteile jeder frameworkbasierten Anwendung sind, müssen die variablen Bestandteile nicht zwangsläufig in jeder Anwendung vertreten sein. Nur wenn der Anwendungsentwickler das betreffende variable Element ausgewählt und konfiguriert hat, wird es sich in der neuen Anwendung wieder finden.

Durch die Variabilitätsebenen in der Frameworkbeschreibung (siehe Abschnitt 4.3.1) arbeitet der Anwendungsentwickler während der Instanziierung mit drei verschiedenen Arten von Auswahlmöglichlichkeiten:

1. *Architekturalternativen*: Der Anwendungsentwickler muss zunächst festlegen, welche Architektur seine gewünschte Anwendung besitzen soll. Dazu gehört die Auswahl verschiedener Kommunikationsstrukturen, die Festlegung der Anzahl von mehrfach instanziierbaren Komponenten und die Auswahl von optionalen Komponenten.

2. *Komponentenalternativen*: Im Anschluss an die Auswahl der Architekturalternativen wählt der Entwickler die konkreten Komponenten-Implementierungen, falls mehrere Alternativen zur Auswahl stehen. Dazu gehört auch die Auswahl von benutzerspezifischen Implementierungen, die nicht im ursprünglichen Lieferumfang des Frameworks enthalten sind.

3. *Instanzenalternativen*: Abschließend müssen die Eigenschaften der einzelnen Komponenten parametriert werden, um das endgültige Verhalten der Anwendung festzulegen.

Der Anwendungsentwickler bewegt sich somit während der Instanziierung auf drei verschiedenen Entscheidungsebenen, die untereinander starke Abhängigkeiten aufweisen. Jede übergeordnete Ebene beeinflusst die Entscheidungen auf den darunter liegenden Ebenen. Abbildung 5.1 veranschaulicht die Abhängigkeiten zwischen den drei Entscheidungsebenen während der Instanziierung. Die zur Verfügung stehenden Komponentenalternativen sind beispielsweise von der Wahl der Architektur abhängig – Ist eine bestimmte Architektur gewählt, so werden im weiteren Verlauf nur noch die Komponenten berücksichtigt, die in dieser Architekturvariante verwendet werden. Ähnliches gilt auch für die Parametrierung der Komponenten – Es werden nur die Komponenten-Eigenschaften zur Parametrierung angeboten, die in der gewählten Architektur und der vorliegenden Komponenten-Implementierung vorhanden sind. Auf diese Weise wird sichergestellt, dass der Anwendungsentwickler einerseits nur notwendige und sinnvolle Entscheidungen treffen muss und dass andererseits in der Anwendung kein überflüssiger Code auftaucht, der für die Funktionalität der aktuellen Anwendung irrelevant ist.

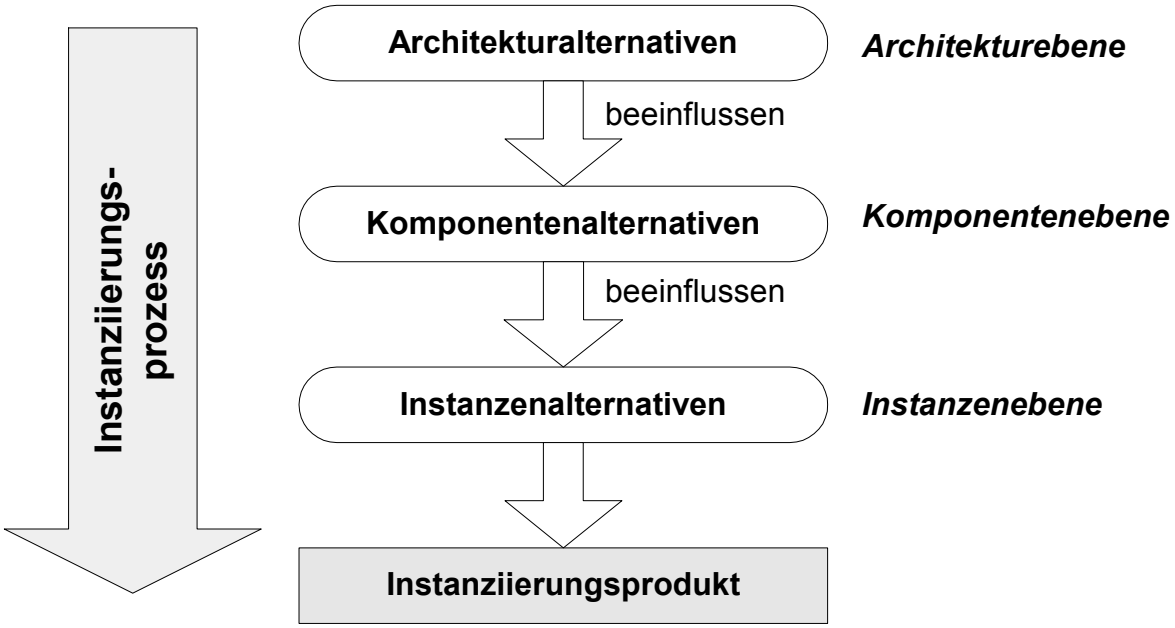

Abbildung 5.1: Abhängigkeiten während der Instanziierung

5.1.2 Verlauf der Instanziierung

Durch die drei Entscheidungsebenen und ihre Abhängigkeiten bei der Anwendungsentwicklung wird ein Instanziierungsverlauf definiert, der in der Frameworkbeschreibung jedes Komponenten-Frameworks verankert ist. Der gesamte Verlauf der Instanziierung ist in kleine und überschaubare Einzelschritte gegliedert, die jeweils zur Architektur-, zur Komponenten- oder zur Instanzenebene gehören. Jede Ebene kann dabei in beliebig viele Einzelschritte unterteilt werden. Bei jedem Instanziierungsschritt werden vom Anwendungsentwickler Entscheidungen gefordert, die den nächsten Schritt bestimmen und damit den weiteren Verlauf der Instanziierung beeinflussen. Die Instanziierung des Komponenten-Frameworks ist dann abgeschlossen, wenn der Anwendungsentwickler alle notwendigen Entscheidungen getroffen hat, die unter Berücksichtigung der Abhängigkeiten für die vollständige Spezifikation einer frameworkbasierten Anwendung notwendig sind.

An dieser Stelle muss betont werden, dass der Anwendungsentwickler bei der Erstellung unterschiedlicher frameworkbasierter Anwendungen zwar unterschiedliche Instanziierungswege durch die Frameworkbeschreibung erfährt, jedoch nicht für die Steuerung des Ablaufs zuständig ist. Er muss weder die Abhängigkeiten zwischen den einzelnen Schritten kennen, noch den nächsten notwendigen Schritt bestimmen. Diese wichtige Aufgabe wird direkt vom Instanziierungswerkzeug übernommen, welches in Abschnitt 5.3 vorgestellt wird.

5.1.3 Anforderungen an die Frameworkbeschreibung

Aus den besonderen Eigenschaften eines Komponenten-Frameworks und dem beschriebenen Verlauf der frameworkbasierten Anwendungsentwicklung ergeben sich zusätzliche Anforderungen, welche die allgemeinen Anforderungen aus Abschnitt 3.5 ergänzen bzw. erweitern:

- *Unterstützung variabler und invariabler Framework-Elemente*: Ein Framework besteht aus einer beliebigen Kombination von variablen und invariablen Elementen. Die Frameworkbeschreibung muss in der Lage sein, diese Kombinationsmöglichkeiten vollständig und umfassend zu beschreiben. Sie muss sogar die Beschreibung vollständiger Anwendungen ohne Variabilitäten zulassen.

- *Berücksichtigung der drei Variabilitätsebenen*: Die Frameworkbeschreibung muss die Beschreibung von Architektur-, Komponenten- und Instanzenvariabilitäten ermöglichen, um den gesamten Instanziierungsprozess bis hin zur fertigen Anwendung zu umfassen.

- *Definition diskreter Instanziierungsschritte*: Die Frameworkbeschreibung muss die Aufteilung des gesamten Instanziierungsprozesses in kleine und überschaubare Instanziierungsschritten unterstützen. Dadurch kann der Anwendungsentwickler auf einem nachvollziehbaren Weg zu seinem gewünschten Endprodukt geführt werden.

- *Zuordnung von Instanziierungsinformationen*: Zu jedem Instanziierungsschritt muss die Frameworkbeschreibung die Zuordnung von passenden Informationstexten ermöglichen. Auf diese Weise wird die Instanziierungsinformation ein integraler Bestandteil der Instanziierung, wodurch das aufwändige Suchen nach passenden Erklärungstexten während der Instanziierung vermieden wird.

- *Abhängigkeiten zwischen Instanziierungsentscheidungen*: Die Konsequenzen einzelner Entscheidungen auf den nachfolgenden Verlauf der Instanziierung müssen in der Frameworkbeschreibung definiert werden können. Dadurch können die im Komponenten-Framework vorhandenen Abhängigkeiten während der Anwendungsentwicklung berücksichtigt werden.

- *Anbindung an ein Instanziierungswerkzeug*: Die Frameworkbeschreibung muss in einer formalisierten Form vorliegen, damit ein Instanziierungswerkzeug die entsprechenden Informationen verarbeiten kann. Eine informale Beschreibung (z.B. Cookbooks) ist zwar für den Anwendungsentwickler leichter zu lesen, erschwert aber die effiziente Werkzeugunterstützung.

5.1.4 Anforderungen an das Instanziierungswerkzeug

Für die Werkzeugunterstützung von Komponenten-Frameworks können ebenfalls weitere Anforderungen abgeleitet werden, welche die Anforderungen aus Abschnitt 3.5 ergänzen und erweitern:

- *Verarbeitung von Frameworkbeschreibungen*: Das Werkzeug soll durch Einlesen und Interpretieren von Frameworkbeschreibungen eine komfortable, frameworkbasierte Anwendungsentwicklung ermöglichen. Außer der Frameworkbeschreibung sollen keine zusätzlichen Informationen für die Instanziierung erforderlich sein.

- *Interaktion mit dem Anwendungsentwickler*: Das Werkzeug soll die Informationen in der Frameworkbeschreibung interpretieren und den Benutzer durch den Instanziierungsprozess führen, d.h. der Benutzer muss sich nicht unmittelbar mit der Frameworkbeschreibung auseinander setzen. Für den Anwendungsentwickler ist der Instanziierungsprozess eine Folge von Schritten, bei denen Informationen zum Framework präsentiert und einfache Entscheidungen getroffen werden müssen.

- *Generierung von Instanziierungsprodukten*: Das Werkzeug soll in der Lage sein, aus dem Komponenten-Framework und den Eingaben des Benutzers das fertige Instanziierungsprodukt zu erzeugen, das vom Anwendungsentwickler gewünscht wird. Dies kann entweder eine vollständige, lauffähige Anwendung sein oder eine komplexe Komponente, die in einen anderen Anwendungskontext eingebettet wird.

5.2 Realisierung der Frameworkbeschreibung

5.2.1 Spezifikationssprache für Frameworkbeschreibungen

Zu den wichtigsten Aufgaben einer Frameworkbeschreibung als Bestandteil eines Komponenten-Frameworks gehört die Spezifikation aller möglicher Instanziierungsprodukte, die Kopplung der Instanziierung mit der Dokumentation und die Ermöglichung einer werkzeuggestützten Anwendungsentwicklung. Im bisherigen Verlauf dieses Dokuments wurde erklärt, welche bedeutsame Rolle die Frameworkbeschreibung für ein Komponenten-Framework spielt und welche Anforderungen an die Beschreibung gestellt werden. Es wurde allerdings noch nicht geklärt, in welcher Form diese Beschreibung vorliegt. In diesem Abschnitt steht die Form von Frameworkbeschreibungen im Mittelpunkt der Betrachtungen.

Für die Realisierung der Frameworkbeschreibung auf der Grundlage der oben definierten Anforderungen wurde die Form einer eigenständigen Spezifikationssprache gewählt. Da in der Literatur keine adäquate Sprache bekannt ist, wurde zu diesem Zweck die *Framework Markup Language* (FML) neu entwickelt. FML ist eine auf der Metasprache XML (eXtensible Markup

Language) [BeMi00] basierende Sprache, welche als formalisierte Notation für Frameworkbeschreibungen eingesetzt wird. Durch die enge Beziehung zu XML profitiert FML von allen Vorteilen, die von XML bekannt sind:

- *Standardisierte Sprache*: XML ist bereits heute ein verbreiteter Industriestandard, der dank seiner einfachen Syntax in vielen Anwendungsbereichen und Produkten eingesetzt wird.

- *Unabhängigkeit von Systemplattformen*: Da XML-Informationen in Textdateien gespeichert werden, können die Informationen verhältnismäßig einfach auf verschiedenen Rechner-, Betriebssystem- und Anwendungsplattformen verarbeitet werden.

- *Flexible Strukturierungsregeln*: XML ermöglicht die Definition von Strukturierungsregeln innerhalb von FML-Beschreibungsdateien, die eine nachträgliche Erweiterung des Sprachumfangs erlauben.

- *Breite Werkzeugunterstützung*: Für XML ist eine große Auswahl unterschiedlicher Werkzeuge verfügbar (z.B. Editoren, Parser, Generatoren), die teilweise sogar kostenlos verfügbar sind.

Außer den XML-gebundenen Eigenschaften besitzt FML auch eine Reihe von Merkmalen, die nicht an die zugrunde liegende Metasprache sondern direkt an die Spezifikationssprache FML gekoppelt sind:

- *Unabhängigkeit von Programmiersprachen und Komponentenmodellen*: FML ist nicht an ein Komponentenmodell oder eine Programmiersprache gebunden. Dadurch kann FML prinzipiell verschiedene Programmiersprachen und Komponentenmodelle mit derselben Frameworkbeschreibung unterstützen.

- *Trennung von Architektur und Komponenten*: FML beschränkt sich strikt auf die Spezifikation von architekturrelevanten Instanziierungsdaten. Alle Implementierungsdetails (z.B. Funktionsaufrufe und -definitionen) müssen innerhalb von Komponenten gekapselt werden, da FML hierfür keine Sprachkonstrukte anbietet.

- *Trennung zwischen Frameworkentwicklung und -Anwendung*: Der Frameworkentwickler ist für die Erstellung von FML-Spezifikationen zuständig. Der Framework-Anwender verwendet diese Spezifikationen, ohne sie modifizieren zu müssen. Auf diese Weise ist bezüglich der FML-Spezifikation eine klare Trennung zwischen dem Frameworkentwickler und dem -Anwender vorgesehen.

- *Interpretation von Framework- und Anwendungsbeschreibungen*: Mit FML können sowohl Frameworkbeschreibungen mit Variabilititäten auf verschiedenen Abstraktionsebenen, als auch Anwendungsbeschreibungen ohne Variabilitäten spezifiziert werden. Abbildung 5.2 veranschaulicht grafisch den Sprachumfang von FML.

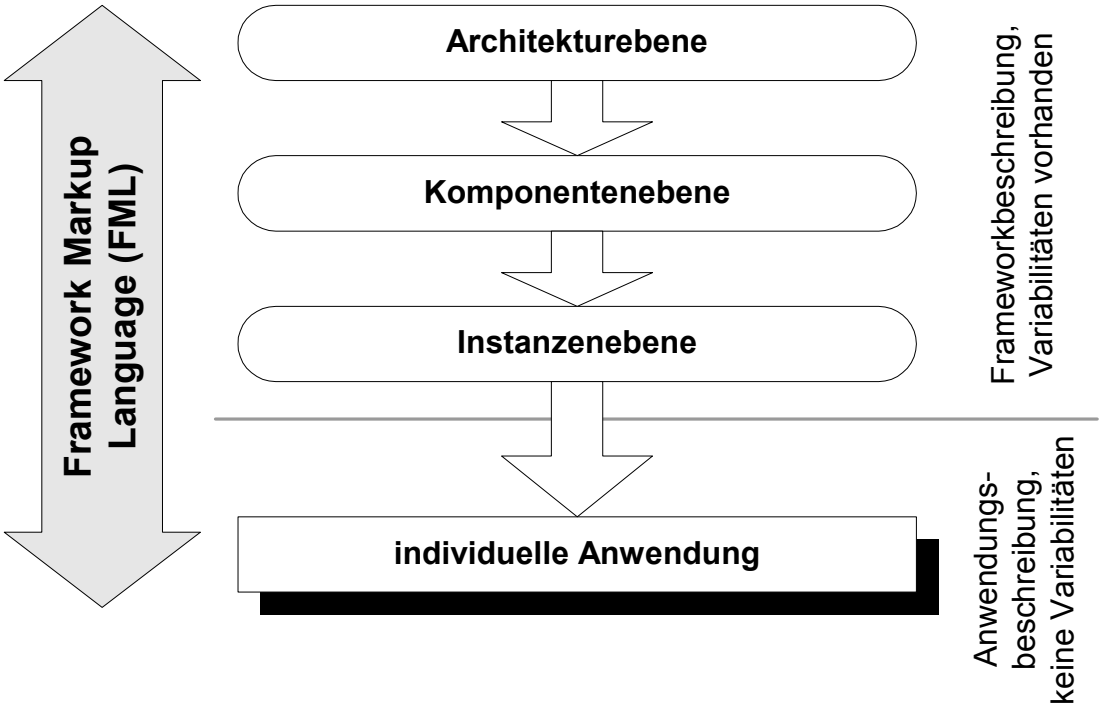

Abbildung 5.2: Sprachumfang der FML Spezifikationssprache

FML fungiert als Bindeglied zwischen dem Komponenten-Framework und dem Instanziierungswerkzeug – das Werkzeug kennt lediglich die Spezifikationssprache, hat jedoch keine weiteren Informationen über das zu instanziierende Framework. Jedes in FML beschriebene Framework kann deshalb mit dem gleichen Werkzeug instanziiert werden, solange dieses Werkzeug die FML-Beschreibungen interpretieren kann. Im nachfolgenden Kapitel 6 werden die einzelnen Sprachelemente ausführlich beschrieben und durch anschauliche Beispiele verdeutlicht.

5.3 Realisierung des Instanziierungswerkzeugs

5.3.1 Werkzeug-Konzept

Das Instanziierungswerkzeug hat die Aufgabe, den Anwendungsentwickler durch den Instanziierungsprozess zu leiten und ihm die einfache Erstellung von individuellen, frameworkbasierten Anwendungen zu erleichtern. Aus der Literatur sind bereits Werkzeuge bekannt, die Teilaspekte dieser Aufgabe unterstützen (siehe Abschnitt 3.4). Es ist jedoch kein Werkzeug bekannt, welches eine umfassende Unterstützung für den Anwendungsentwickler bietet und gleichzeitig allen oben definierten Anforderungen gerecht wird. Aus diesem Grund wurde in dieser Arbeit ein neues Werkzeug namens *Framework Instantiation Tool* (FIT) konzipiert und realisiert.

Voraussetzung für die Arbeit mit FIT ist ein Komponenten-Framework, das aus einer Sammlung verschiedener Komponenten und einer dazu passenden Frameworkbeschreibung besteht. Zusätzlich können weitere Dokumentationstexte hinzukommen, welche beispielsweise Frameworkdetails beschreiben und während der Instanziierung mit FIT eingeblendet werden. Damit FIT die Frameworkbeschreibung korrekt einlesen und interpretieren kann, muss sie in der Spezifikationssprache FML vorliegen. FML ermöglicht die Anwendungsentwicklung mit dem FIT-Werkzeug, ohne das Werkzeug an ein bestimmtes Komponenten-Framework zu koppeln, d.h. jedes mit FML spezifizierte Komponenten-Framework wird in vollem Umfang von FIT unterstützt. Abbildung 5.3 veranschaulicht die Anwendungsentwicklung mit dem FIT-Werkzeug, was im nachfolgenden Abschnitt näher erläutert wird. Eine detaillierte Beschreibung des Werkzeuges FIT ist in Kapitel 7 zu finden.

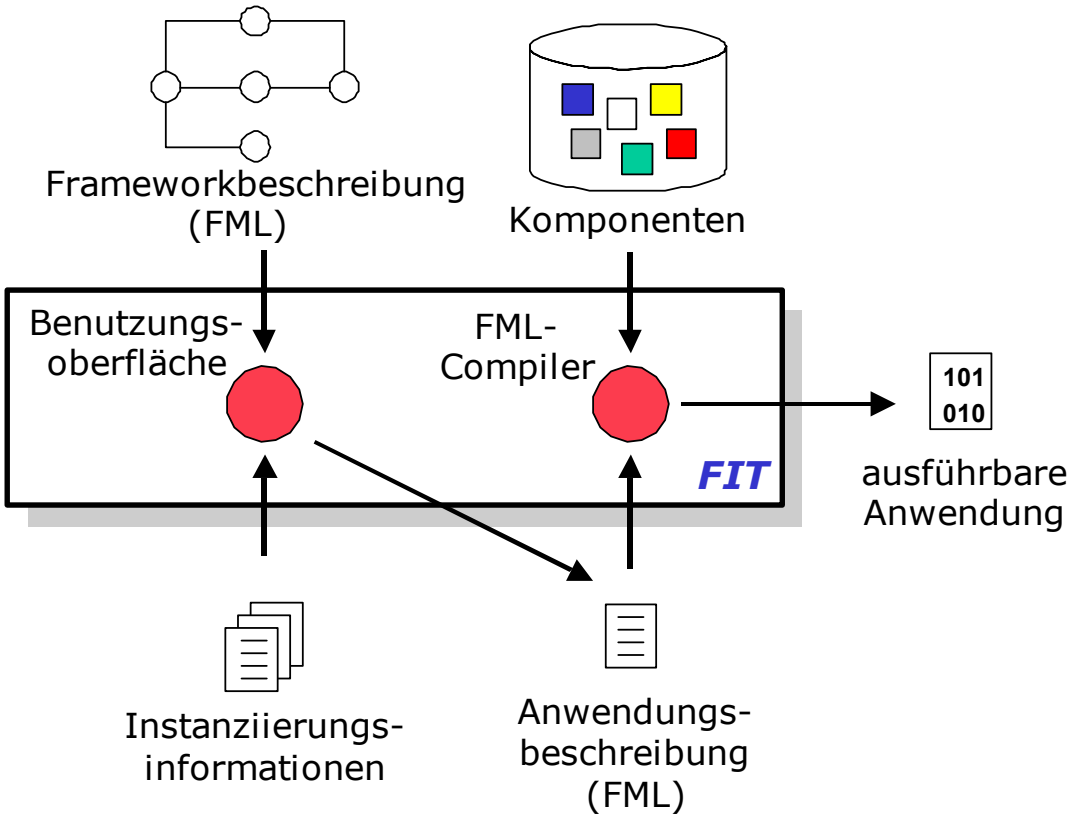

Abbildung 5.3: Anwendungsentwicklung mit FML und FIT

5.3.2 Werkzeuggestützte Anwendungsentwicklung

Wenn der Anwendungsentwickler eine neue frameworkbasierte Anwendung erstellen möchte, so muss er zunächst die Frameworkbeschreibung mit dem FIT-Werkzeug öffnen. FIT interpretiert die darin enthaltenen Informationen und führt den Entwickler schrittweise durch den Instanziierungsprozess. Für den Anwendungsentwickler stellt sich die Instanziierung als ein interaktiver Prozess dar, bei dem er Informationen (Instanziierungsinformationen) und Fragen

erhält und durch seine Antworten und Entscheidungen den weiteren Verlauf der Instanziierung beeinflusst. Dieser gesamte Vorgang spielt sich in der komfortablen grafischen Benutzungsoberfläche von FIT ab, ohne dass der Anwendungsentwickler direkt mit den einzelnen Quelldokumenten (Frameworkbeschreibung und Instanziierungsinformationen) arbeiten muss.

Hat der Anwendungsentwickler schließlich alle Entscheidungen getroffen, die für die Erstellung einer neuen Anwendung notwendig sind, so werden diese Informationen in einer neuen Datei gespeichert. Diese Datei ist die so genannte Anwendungsbeschreibung, die ebenfalls in FML vorliegt und zusammen mit der Frameworkbeschreibung eine neue frameworkbasierte Anwendung repräsentiert. Bis zu diesem Zeitpunkt des Instanziierungsprozesses wird die neue Anwendung lediglich spezifiziert, d.h. es wird noch kein ausführbarer Code generiert. Erst wenn die Anwendungsbeschreibung vollständig vorliegt, kann mit Hilfe der Komponenten eine neue Anwendung oder Komponente generiert werden. Diese Aufgabe wird vom FML-Compiler des Werkzeuges FIT übernommen und geschieht ebenfalls ohne den manuellen Eingriff des Anwendungsentwicklers.

Die Anwendungsentwicklung mit dem Werkzeug FIT ist somit in zwei wesentliche Schritte unterteilt: Zunächst wird im ersten Schritt auf der Grundlage der Entscheidungen des Entwicklers eine Anwendungsbeschreibung generiert, welche die gewünschte Anwendung vollständig spezifiziert. Erst dann wird im zweiten Schritt aus der Anwendungsbeschreibung eine ausführbare Anwendung generiert, welche das Endprodukt des Instanziierungprozesses darstellt. Obwohl dieser zweigeteilte Instanziierungsprozess auf den ersten Blick nicht notwendig ist, so birgt er doch eine Reihe von interessanten Möglichkeiten:

- *Anwendungsentwicklung ohne ausführbaren Code*: Für die Erstellung der Anwendungsbeschreibung werden die Komponenten-Implementierungen nicht benötigt. Jeder potenzielle Anwendungsentwickler könnte unverbindlich prüfen, ob seine Anforderungen durch das Framework ausreichend abgedeckt sind. Erst wenn diese Versuche erfolgreich waren, könnte das Framework in vollem Umfang eingesetzt werden.

- *Abspeichern von Teil-Instanziierungen*: FIT bietet die Möglichkeit, jeden beliebigen Stand der Instanziierung in eine Datei abzuspeichern. Auf diese Weise kann ein umfangreicher Instanziierungsprozess mehrfach unterbrochen und wieder aufgenommen werden. Es besteht weiterhin die Möglichkeit, einen abgespeicherten Zwischenstand wiederholt aufzunehmen und zu verschiedenen Anwendungsbeschreibungen zu vervollständigen.

- *Unterstützung verschiedener Programmiersprachen und Komponentenmodelle*: Die Frameworkbeschreibung ist unabhängig von der Framework-Implementierung. Prinzipiell entsteht dadurch die Möglichkeit, mit derselben Frameworkbeschreibung auch andere Programmiersprachen und Komponentenmodelle zu unterstützen. Das Vorhandensein

solcher alternativen Implementierungen in verschiedenen Programmiersprachen und Komponentenmodellen würde sogar die Option eröffnen, die Programmiersprache oder das Komponentenmodell als Option in den Instanziierungsprozess einzufügen.

5.4 Konzeptionelle Alternativen

Vor dem Hintergrund der gestellten Anforderungen ist das hier beschriebene Instanziierungskonzept nicht das einzige, welches zur Unterstützung des Anwendungsentwicklers bei der Instanziierung von Frameworks gewählt werden kann. Außer der Spezifikationssprache FML und dem dazu gehörenden Werkzeug FIT sind prinzipiell auch andere Lösungswege denkbar, um die gestellten Anforderungen zu erfüllen. In diesem Abschnitt sollen zwei Realisierungsalternativen vorgestellt werden, die zwar einen Großteil der gestellten Anforderungen erfüllen, aber – im Gegensatz zu dem hier favorisierten Konzept – mit wesentlichen Einschränkungen behaftet sind.

Realisierung eines frameworkspezifischen Werkzeuges

Eine Möglichkeit wäre beispielsweise die Realisierung eines frameworkspezifischen Werkzeuges, welches das geforderte Wissen über das Komponenten-Framework kapselt und den Anwendungsentwickler bis hin zur fertigen Anwendung führt. Mit Hilfe dieses Werkzeuges könnte der Entwickler frameworkbasierte Anwendungen erstellen, ohne sich mit den Implementierungsdetails des Frameworks auseinander zu setzen. Obwohl mit diesem Vorgehen bei der Realisierung der Frameworkbeschreibung viele Anforderungen erfüllt werden können, müssen auch entscheidende Nachteile berücksichtigt werden. Die Vereinigung von Frameworkbeschreibung und Instanziierungswerkzeug bedeutet einen enormen zusätzlichen Aufwand für den Frameworkentwickler. Er muss für jedes Komponenten-Framework ein individuelles Werkzeug implementieren, welches dem Anwendungsentwickler die Arbeit mit dem Framework erleichtert. Dieses Vorgehen würde die Framework-Entwicklung durch erhebliche Zusatzkosten belasten und die Akzeptanz des Verfahrens mindern. Im Gegensatz dazu kann das Werkzeug FIT unverändert mit beliebigen Frameworks verwendet werden, wenn eine FML-basierte Frameworkbeschreibung verfügbar ist.

Realisierung einer universellen parametrierbaren Anwendung

Eine weitere Möglichkeit für die Realisierung eines Instanziierungskonzeptes besteht in der Implementierung einer Anwendung, welche alle möglichen Variabilitäten des Frameworks enthält und über umfangreiche Parametrierungsdialoge an die Bedürfnisse des Anwenders angepasst werden kann. Diese konfigurierbare Anwendung würde gleichzeitig das Endprodukt der Instanziierung als auch das zugrunde liegende Framework repräsentieren. Der Anwendungsentwickler kann durch Parametrierung die Anwendungsbestandteile ausschalten, die er nicht benötigt.

Auch dieses Vorgehen bei der Realisierung der Frameworkbeschreibung hat einen entscheidenden Nachteil, welcher den Nutzen des Vorgehens in Frage stellt: In jeder Anwendung sind alle Variabilitäten des Frameworks enthalten, d.h. alle Anwendungen besitzen den identischen Implementierungscode und unterscheiden sich lediglich durch die zur Laufzeit ausgeführten Codeteile. Auf diese Weise ist in jeder Anwendung ein erheblicher Anteil unnötigen Codes enthalten, der sich nachteilig auf den Speicherplatzbedarf und die Laufzeit der Anwendung auswirkt. Da das Verfahren in der Automatisierungstechnik eingesetzt werden soll, wo nichtfunktionale Anforderungen (z.B. Laufzeit, Speicherplatz, Zuverlässigkeit) eine entscheidende Rolle spielen, sind solche konzeptionellen Einschränkungen nicht akzeptabel. Das in dieser Arbeit favorisierte Konzept vermeidet diesen Overhead, da erst nach der vollständigen Spezifikation der gewünschten Anwendung der entsprechende Code generiert wird. Dieser optimierte Code enthält nur die Komponenten und Verbindungen, die für die individuelle Anwendung unabdingbar sind.

In diesem Kapitel wurde ein Verfahren vorgestellt, dass auf der Grundlage von Komponenten-Frameworks eine effiziente und werkzeuggestützte Anwendungsentwicklung ermöglicht. Zentrale Elemente dieses Verfahrens sind die FML Spezifikationssprache, die zur Erstellung vom Frameworkbeschreibungen dient, und das Werkzeug FIT, welches den Anwendungs-entwickler durch den Instanziierungsprozess führt, ohne ihn mit unnötigen Details zu belasten. Im folgenden Kapitel wird zunächst die Spezifikationssprache detaillierter erläutert und anhand von Beispielen veranschaulicht. In Kapitel 7 wird schließlich das Werkzeug FIT vorgestellt, das den Anwendungsentwickler bei der Arbeit mit Komponenten-Frameworks unterstützt.

6 Spezifikationssprache für Framework-beschreibungen

Im vorangehenden Kapitel wurde die Framework Markup Language (FML) als Spezifikationssprache für Frameworkbeschreibungen vorgestellt. Da FML als Sprache für Frameworkbeschreibungen eine zentrale Rolle in dieser Arbeit und dem darin vorgeschlagenen Instanziierungskonzept spielt, soll in diesem Kapitel die Spezifikationssprache FML anhand von illustrierenden Beispielen ausführlich erläutert werden. Dazu werden zunächst die Sprachelemente vorgestellt, die zur Spezifikation komponentenbasierter Anwendungen ohne Variabilitäten notwendig sind. Darauf aufbauend werden dann die Framework-relevanten Sprachelemente eingeführt, die zur Beschreibung von Variabilitäten benötigt werden. Abschließend wird die FML mit ähnlichen Spezifikationssprachen verglichen, die im Bereich von Architekturen, Komponenten und Anwendungen eingesetzt werden.

6.1 Grundlagen der Framework Markup Language

Die Spezifikationssprache FML ist die Brücke zwischen dem Framework- und dem Anwendungsentwickler, die zur Kommunikation von elementaren Informationen und zur umfassenden Unterstützung bei der Instanziierung dient. Ziel von FML ist die Nutzung von Produktivitätsvorteilen der Framework-Technologie unter Vermeidung der Nachteile, die normalerweise an das Framework-Konzept gekoppelt sind. Der Anwendungsentwickler soll in die Lage versetzt werden, aus dem Framework seine spezifische Anwendung zu erstellen, ohne sich mit unnötigen Entwurfs- und Implementierungsdetails auseinander setzen zu müssen.

FML erfüllt dabei zwei wesentliche Aufgaben, die im Nachfolgenden näher beschrieben sind:

- Spezifikation und Dokumentation von Komponenten-Frameworks
- Grundlage für die Werkzeugunterstützung

Die Sprache erfüllt einerseits die Aufgabe, das Komponenten-Framework einschließlich aller darauf basierenden Anwendungsvarianten explizit zu beschreiben. Dadurch wird sichergestellt, dass jede auf der Frameworkbeschreibung basierende Anwendung konform mit den Absichten des Frameworkentwicklers ist. Der Anwendungsentwickler kann also davon ausgehen, dass seine Anwendung nicht mit frameworkinternen Konventionen kollidieren wird. FML erfüllt andererseits die Aufgabe, das Fundament für die Werkzeugunterstützung zu legen. Die Formalisierung der Sprache erlaubt die Verarbeitung durch spezialisierte Werkzeuge, die den Benutzer durch den Instanziierungsprozess führen und abschließend eine ablauffähige Anwendung generieren.

In diesem Kapitel werden die wichtigsten Sprachelemente von FML vorgestellt und erläutert, die für das konzeptionelle Verständnis der Spezifikationssprache notwendig sind. Zugunsten der Übersichtlichkeit wird bewusst darauf verzichtet, *alle* Sprachelemente detailliert und vollständig zu erläutern. Für eine tiefergehende Betrachtung der FML einschließlich aller Beschreibungselemente sei auf die FML Benutzungsdokumentation [FML01] verwiesen.

6.2 Grundlegende Beschreibungselemente

6.2.1 Invariante Anwendungselemente

Grundstruktur der FML-Beschreibung

FML ist in der Lage, sowohl Frameworks mit allen Variabilitäten als auch frameworkbasierte Anwendungen ohne Variabilitäten zu beschreiben. In diesem Abschnitt werden zunächst die Sprachmittel vorgestellt, die zur Beschreibung von invariablen Anwendungselementen benötigt werden. Diese Sprachmittel werden im Verlauf der Instanziierung nicht modifiziert und später direkt in den ausführbaren Code übersetzt. Abbildung 6.1 zeigt das Grundgerüst einer FML-Beschreibung, die in jeder FML-Datei zu finden ist.

```
1:    <?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE fml-file
2:     SYSTEM "fml.dtd">
3:
4:    <fml-file author="Du" company="ias" date="6.2.01" version="1.0">
5:     <project id="Bsp-Projekt" main-fw="Bsp-FW">
6:     <fw-spec id="Bsp-FW">
7:
8:     <!-- Hier erscheint die FW-Beschreibung-->
9:
10:    </fw-spec>
11:    </project>
12:   </fml-file>
```

Abbildung 6.1: Grundstruktur einer FML-Datei

Die ersten zwei Zeilen sind typisch für alle FML-Dateien: In der ersten Zeile wird die Datei als XML-Datei identifiziert und in der zweiten Zeile wird spezifiziert, nach welchen Strukturierungsregeln die Datei aufgebaut ist. Alle FML-Dateien folgen den Regeln der FML-DTD (Document Type Definition), die in der Datei (fml.dtd) festgelegt sind und von jeder FML-Datei explizit referenziert werden [BeMi00]. Da die ersten zwei Zeilen bei allen FML-Beschreibungen identisch sind, werden sie im Nachfolgenden nicht mehr explizit dargestellt.

In Zeile 4 beginnt der FML-spezifische Teil der Datei. Zunächst wird jede Datei mit dem Auszeichnungselement[4] `<fml-file>` begonnen. Dieses Element leitet jede FML-Datei ein und beinhaltet als Attribute Informationen wie den Autor, die Firma, das Datum und die Version. Das nächste Element `<project>` legt die Zugehörigkeit der nachfolgenden Beschreibung zu einem Projekt fest. In FML werden grundsätzlich alle Beschreibungen zu entsprechenden Projekten zugeordnet, da Beschreibungen auf mehrere Dateien verteilt werden können. Jedes Projekt erhält als Attribut eine eindeutige ID (Identifier) und eine Angabe zum Namen des Hauptframeworks (`main-fw`). FML fordert für alle wesentlichen Beschreibungselemente eine eindeutige ID, damit das betreffende Element an anderen Stellen flexibel referenziert werden kann. Die Angabe des Hauptframeworks ist deshalb notwendig, weil in einem Projekt mehrere Frameworkbeschreibungen verwendet werden können, wodurch die projektübergreifende Wiederverwendung ermöglicht wird. Auf diese Weise können Frameworkbeschreibungen oder Teile davon in einem anderen Projekt verwendet werden. Das Attribut `main-fw` legt den Namen des Frameworks fest, der unter anderem im `<project>`-Tag durch das gleichnamige Attribut referenziert wird. Mit dem Element `<fw-spec>` wird der Kern der eigentlichen Framework-Spezifikation eingeleitet. Innerhalb von `<fw-spec>` erfolgt die eigentliche Frameworkbeschreibung, die im Nachfolgenden detaillierter erläutert wird.

Juggler-Anwendung

Um die Frameworkbeschreibung anschaulicher zu erläutern, soll ein kleines Beispiel verwendet werden, das den Einsatz von FML für das Komponentenmodell JavaBeans konkretisiert. An dieser Stelle soll jedoch betont werden, dass FML konzeptionell nicht an ein bestimmtes Komponentenmodell gebunden ist. Prinzipiell könnten auch andere Modelle (z.B. COM, CORBA) verwendet werden. Lediglich die Generierungswerkzeuge, die aus FML-Beschreibungen ausführbaren Quellcode erzeugen, sind natürlich an die Gegebenheiten des Komponentenmodells gebunden. Das Beispiel besteht aus einer einfachen *Juggler-Anwendung* (siehe Abbildung 6.2), die nur aus drei verschiedenen Komponenten (JavaBeans) besteht.

[4] englisch: Tag

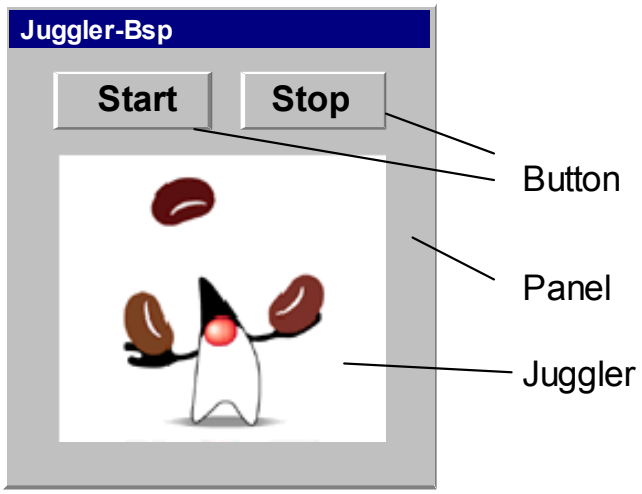

Abbildung 6.2: Juggler-Anwendung aus drei unterschiedlichen JavaBeans

Die wichtigste Komponente ist der so genannte *Juggler*, der eine kleine Animation in Gestalt einer jonglierenden Figur darstellt. Das Verhalten des Jugglers wird durch zwei Knöpfe (*Button*) gesteuert, welche die Animation starten bzw. unterbrechen. Der Juggler und die zwei Knöpfe sind schließlich in ein so genanntes *Panel* eingebettet, welches als Container für die Komponenten dient. Diese Juggler-Anwendung besitzt keinerlei Variabilitäten, zeigt aber sehr anschaulich die Ausdrucksmöglichkeiten der FML, die für jede Anwendung benötigt werden. Abbildung 6.3 zeigt in gekürzter Form den FML-Code, der die Juggler-Anwendung repräsentiert.

```
1:    <fml-file author="Du" company="ias" date="6.2.01" version="1.0">
2:     <project id="Juggler-Projekt" main-fw="Juggler-FW">
3:     <fw-spec id="Juggler-FW">
4:
5:     <!-- Anwendungsrahmen festlegen -->
6:     <start-in class="java.awt.Frame" id="Hauptfenster"/>
7:     <class-name value="Juggler-Bsp" package="fml.demo"/>
8:
9:     <!-- verwendete Komponenten instanziieren -->
10:    <component class="sunw.demo.juggler.Juggler" id="Juggler"/>
11:    <component class="java.awt.Panel" id="Panel"/>
12:    <component class="java.awt.Button" id="StartButton"/>
13:    <component class="java.awt.Button" id="StopButton"/>
14:
15:    <!-- Komponenten verbinden -->
16:    <component source="StartButton">
17:     <event-binding name="action">
18:     <script>
19:     <call-method target="Juggler" name="start"/>
20:     </script>
21:     </event-binding>
22:    </component>
23:    ...
```

```
24:
25:     <!-- Komponenten-Eigenschaften festlegen -->
26:     <component source="StartButton">
27:      <property name="label" value="Start"/>
28:     </component>
29:     ...
30:
31:     <!-- Komponenten in Panel-Container einfuegen -->
32:     <component source="Panel">
33:      <add>
34:      <component source="StartButton"/>
35:      </add>
36:     </component>
37:     ...
38:
39:     </fw-spec>
40:     </project>
41:    </fml-file>
```

Abbildung 6.3: FML-Beschreibung der Juggler-Anwendung

In der FML-Beschreibung der Juggler-Anwendung finden sich zunächst die bereits erläuterten Tags `<fml-file>`, `<project>` und `<fw-spec>` wieder (Zeile 1 bis 3). In den Zeilen 6 und 7 wird der Kontext für die Juggler-Anwendung festgelegt. Das Tag `<start-in>` legt fest, dass die Anwendung als gewöhnliche Java-Applikation starten soll. Alternativ könnte die Beschreibung auch ein Java-Applet oder eine neue Java-Komponente als Endprodukt der Instanziierung definieren. Das Tag `<class-name>` legt mit dem Attribut `name` den Klassennamen für die zu generierende Implementierung der Juggler-Anwendung fest. Optional kann im Attribut `package` ein Java-Package für die Implementierungsklasse festgelegt werden.

In den Zeilen 10 bis 13 werden alle in der Juggler-Anwendung benötigten Komponenten mit Hilfe des `<component>`-Tags instanziiert und erneut mit einer eindeutigen ID versehen. Die Eindeutigkeit ist an dieser Stelle von besonderer Bedeutung, da die Button-Komponente zweifach instanziiert wird. Nur anhand der ID kann zwischen dem Start- und dem Stop-Button unterschieden werden, da die Komponenten-Implementierung für beide Instanzen identisch ist. Mit dem Attribut `class` wird die Implementierungsklasse der jeweiligen Komponente festgelegt.

Nach der Instanziierung der benötigten Komponenten können die Instanzen miteinander verbunden werden. Die Zeilen 16 bis 22 zeigen exemplarisch, wie die Verbindung zwischen dem StartButton und dem Juggler definiert wird. Diese FML-Zeilen legen fest, dass bei jeder Betätigung des StartButtons der Juggler mit der Animation beginnen soll. Da im JavaBeans Komponentenmodell Verbindungen zwischen Komponenten ausschließlich über das Versenden von Ereignissen zustande kommen, werden Verbindungen als gerichtete Kommunikationskanäle

definiert. Zunächst wird die Quelle des Ereignisses festgelegt (Zeile 16), was im vorliegenden Beispiel der StartButton ist. Da jede Komponente prinzipiell beliebige Ereignisse aussenden kann, muss weiterhin definiert werden, welches konkrete Ereignis der Senderkomponente für die Verbindung relevant ist. Im Beispiel ist nur das „action“-Ereignis des StartButtons interessant, welches beim Betätigen des Buttons ausgelöst wird (Zeile 17). Nachdem die Senderseite vollständig beschrieben wurde, kann die Verbindung durch die Angabe der Empfängerkomponente und ihrer Reaktionsmethode vervollständigt werden. Im Beispiel soll der Juggler das Ereignis des StartButtons empfangen und an seine Schnittstellen-Methode „start“ zur Bearbeitung weiterleiten (Zeile 19). Um das Anhalten der Animation zu ermöglichen, müsste der StopButton auf ähnliche Weise mit dem Juggler verbunden werden, was in der vorliegenden FML-Beschreibung jedoch nicht dargestellt wird.

Außer der Instanziierung von Komponenten und der Festlegung von Kommunikationsverbindungen können auch spezielle Eigenschaften für einzelne Komponenten parametriert werden (Zeile 26 bis 28). Im Beispiel wird der StartButton mit einer Aufschrift (sog. Button-Label) versehen. Dazu wird zunächst wieder die betreffende Komponenten-Instanz durch das `<component>`-Tag referenziert. Mit dem nachfolgenden `<property>`-Tag kann durch das `name`-Attribut ein Parameter herausgegriffen und mit dem `value`-Attribut ein gewünschter Wert zugeordnet werden. In Zeile 27 wird beispielsweise dem StartButton als Eigenschaft für die Button-Aufschrift (label) die Zeichenkette „Start“ zugewiesen. Auf diese Weise können bei allen Komponenten-Instanzen die verfügbaren Parameter durch individuelle Parameterwerte besetzt werden.

In Zeile 32 bis 36 wird schließlich dargestellt, wie mit Hilfe von FML einzelne Komponenten in einen übergeordneten Container eingebettet werden. Im Beispiel wird der StartButton in das Panel-Element eingebettet. Dies ist ein wesentliches Merkmal für die Hierarchisierung von Komponenten, da einzelne Komponenten zu übergeordneten Komponenten zusammengefasst werden können.

An dieser Stelle soll erwähnt werden, dass der im Beispiel gewählte Aufbau (Instanziieren – Verbinden – Parametrieren) nicht zwingend eingehalten werden muss, obwohl ein entsprechendes Vorgehen zur Strukturierung der FML-Beschreibung sinnvoll ist. In der vorliegenden FML-Beschreibung könnten beispielsweise zunächst die einzelnen Parametrierungen vorgenommen und danach die Komponentenverbindungen hergestellt werden. Wichtig ist dabei nur, dass logische Abhängigkeiten beachtet werden – es kann beispielsweise keine Komponente parametriert werden, die in der Beschreibung noch nicht deklariert wurde.

6.2.2 Steuerungselemente

Nachdem in den vorangehenden Beispielen ausschließlich statische Anwendungselemente der FML beschrieben wurden, die keine Ausdrucksmöglichkeiten für Variabilitäten bieten, soll

diese Einschränkung in den nachfolgenden Abschnitten schrittweise aufgehoben werden. Zu diesem Zweck wird die invariable Juggler-Anwendung allmählich zu einem vollwertigen Framework mit verschiedenen Variabilitäten ausgebaut. In diesem Abschnitt werden zunächst die Steuerungselemente der FML vorgestellt, die eine Interaktion mit dem Anwendungsentwickler und eine Beeinflussung des Instanziierungsprozesses ermöglichen.

<info>-Tag

Während der Instanziierung ist es manchmal notwendig, Meldungen und Nachrichten an den Anwendungsentwickler auszugeben, ohne dass konkrete Entscheidungen von ihm benötigt werden. Das `<info>`-Tag erfüllt genau diesen Zweck. Dieses Beschreibungselement könnte beispielsweise an den Anfang des Juggler-Beispiels gesetzt werden, um den Anwender zu begrüßen und ihn auf die nachfolgenden Schritte vorzubereiten. Abbildung 6.4 zeigt eine entsprechende Erweiterung für das Juggler-Beispiel.

```
1:   <fw-spec id="Bsp-FW">
2:     <info id="willkommen-info" help="c:\juggler\help\willkommen.html">
3:       Herzlich willkommen zum Juggler-Framework! In den nachfolgenden
4:       Schritten können Sie mit diesem Framework ...
5:     </info>
6:   ...
```

Abbildung 6.4: Verwendung des <info>-Tags

Da FML-Beschreibungen sequenziell abgearbeitet werden, wird der Entwickler die zwischen den `<info>`-Tags eingefügte Nachricht sofort nach der Ausführung der FML-Beschreibung im Instanziierungswerkzeug sehen. Das optionale `help`-Attribut wird dazu verwendet, zusätzliche Hilfedateien (HTML- oder Textdateien) an dieses Beschreibungselement zu binden. Auf diese Weise kann jeder Bearbeitungsschritt durch zusätzliche Informationen ergänzt werden, die dem Benutzer bei Bedarf zur Verfügung stehen.

<question>-Tag

Wichtigstes Element für die Interaktion mit dem Benutzer im Instanziierungswerkzeug ist das `<question>`-Tag. Dieses Element wird verwendet, um Eingaben des Benutzers abzufragen, die den weiteren Verlauf der Instanziierung beeinflussen. Mit Hilfe dieses Steuerungselements ist es beispielsweise möglich, die Aufschrift des StartButton nicht fest vorzugeben, sondern während der Instanziierung vom Benutzer abzufragen. Abbildung 6.5 zeigt dieses Verhalten in der FML-Beschreibung.

```
1:    <component source="StartButton">
2:     <property name="label">
3:      <question type="string" id="startLabel"
4:       help="c:\juggler\help\StartButton.html" default="Start">
5:        Bitte geben Sie die Aufschrift für den StartButton ein!
6:      </question>
7:     </property>
8:    </component>
```

Abbildung 6.5: Verwendung des <question>-Tags

Der Text für die Beschriftung des StartButtons wird im Beispielcode nicht fest vorgegeben, sondern als Frage vom Benutzer angefordert. Dabei sieht der Benutzer im Instanziierungswerkzeug den Text, der zwischen den `<question>`-Tags eingefügt ist. Das Attribut `type` legt den erwarteten Typ des Eingabewertes fest, wobei außer Zeichenketten (string) auch verschiedene Zahlen (int, float)[5], boolsche Werte (boolean), Dateinamen (file) oder Verzeichnispfade (path) möglich wären. Das `id`-Attribut spielt eine wichtige Rolle bei der Abspeicherung der eingegebenen Daten. Der Benutzer kann zu jedem Zeitpunkt im Instanziierungsprozess bereits gemachte Angaben in einer Datei speichern und diese zu einem späteren Zeitpunkt wieder laden. Um die abgespeicherten Informationen korrekt den entsprechenden Fragen zuordnen zu können, sind eindeutige IDs für jede Frage notwendig.

Das bereits bekannte `help`-Attribut ermöglicht die Kopplung von zusätzlichen Informationen und Hilfstexten an die Fragestellung. Damit kann dem Benutzer geholfen werden, falls er mit der ihm gestellten Frage nicht zurecht kommt oder weitergehende Informationen wünscht. Mit dem optionalen `default`-Attribut können zusätzlich Vorgaben gemacht werden, die dem Benutzer im Eingabefeld als Vorschlag unterbreitet werden.

<variable>-Tag

Ein weiteres wichtiges FML-Beschreibungselement im Zusammenhang mit der Steuerung des Instanziierungsprozesses ist das `<variable>`-Tag. Es wird verwendet, um innerhalb der Framework-Spezifikation Variablen zu deklarieren, Variablen Werte zuzuweisen, Variablenwerte zu manipulieren oder den Wert der Variablen in anderen Beschreibungselementen zu verwenden. Alle mit diesem Tag definierten Variablen sind innerhalb einer Framework-Spezifikation global definiert.

Im vorangehenden FML-Beispiel zum `<question>`-Tag wurde gezeigt, wie Benutzereingaben direkt einer Komponenten-Eigenschaft (Aufschrift des StartButtons) zugewiesen werden. Häufig kann in Framework-Spezifikationen die Benutzereingabe jedoch nicht direkt an eine

Komponenten-Eigenschaft gekoppelt werden oder eine Eingabe muss an mehreren Stellen verwendet werden. In solchen Fällen möchte man die Eingabe in einer Variablen zwischenspeichern und zu einem späteren Zeitpunkt einer Komponenten-Eigenschaft zuweisen. Abbildung 6.6 zeigt, wie Benutzereingaben mit Hilfe von Variablen zwischengespeichert und an anderer Stelle wieder verwendet werden.

```
1:    <!-- Variablendefinition -->
2:    <variable name="LabelStartButton">
3:     <question type="string" id="StartLabelFrage"
4:      help="c:\juggler\help\StartButton.html" default="Start">
5:        Bitte geben Sie die Aufschrift für den StartButton ein!
6:     </question>
7:    </variable>
8:    ...
9:    <!-- Verwendung der Variablen -->
10:   <component source="StartButton">
11:    <property name="label" value="var:LabelStartButton"/>
12:   </component>
```

Abbildung 6.6: Definition und Verwendung von Variablen

In diesem Beispiel wird zunächst die Variable `LabelStartButton` definiert (Zeile 2) und mit der Benutzereingabe auf die Frage (Zeile 3 bis 6) initialisiert. Danach steht die Eingabe innerhalb der Framework-Spezifikation unter dem Variablennamen `LabelStartButton` global zur Verfügung. In Zeile 11 wird die Variable schließlich verwendet, um den StartButton mit einem Beschriftungstext zu versehen. Im Vergleich zur statischen Parametrierung ändert sich lediglich die Angabe des Parameterwertes – statt einer konstanten Zeichenkette wird der Wert der Variablen `LabelStartButton` zugewiesen.

Bei der Arbeit mit Variablen muss beachtet werden, dass die Variable *vor* der Verwendung einen definierten Wert besitzen muss. Variablen ohne zugewiesenen Wert sind mit der Zeichenkette "notdefined" vorbesetzt, was immer ein Hinweis auf eine fehlerhafte FML-Beschreibung ist.

Verzweigungen

Um den Instanziierungsprozess zu steuern, stehen verschiedene FML-Beschreibungselemente zur Verfügung, die eine Verzweigung in der Ablauflogik ermöglichen. Mit Hilfe dieser Elemente können Bedingungen überprüft und in Abhängigkeit vom Ergebnis verschiedene Beschreibungsteile aufgeführt oder übersprungen werden. Eines dieser FML-Steuerungselemente für Verzweigungen sind die `<if>`- und `<else>`-Tags. Sie bieten in

---

[5] Bei Zahlen ist mit dem optionalen range-Attribut auch eine Einschränkung des erlaubten Wertebereichs möglich, z.B. range="1..3" für Ganzzahlen von 1 bis 3 oder range="1,3,5,7" für ungerade Ganzzahlen von 1 bis 7.

Anlehnung an gleichnamige Konstrukte vieler Programmiersprachen die Möglichkeit, Bedingungen für die Ausführung bestimmter Beschreibungsteile zu formulieren. Das `<if>`-Tag umschließt dabei den FML-Block, der nur dann ausgeführt wird, wenn die im `condition`-Attribut angegebene Bedingung erfüllt ist. Mit dem optionalen `<else>`-Tag kann ein Block definiert werden, der nur dann ausgeführt wird, wenn der dazu gehörige if-Block nicht ausgeführt wurde.

Um die Verwendung des `<if>`-Tags zu demonstrieren, soll das bereits bekannte Juggler-Beispiel durch eine Architekturalternative erweitert werden. Im Gegensatz zum Standard-Juggler kann der Anwendungsentwickler mit der Alternative einen Juggler auswählen, der nur einen StartButton ohne den entsprechenden StopButton besitzt. Die Animation des Jugglers wird mit dem StartButton ausgelöst und automatisch nach einer festgelegten Zeitspanne angehalten. An Stelle des StopButtons wird deshalb ein Timer benötigt, der das Beenden der Animation veranlasst. Abbildung 6.7 zeigt die beschriebene Architekturalternative im FML-Code.

```
1:    <!-- Benutzereingabe -->
2:    <variable name="EinButtonJuggler">
3:     <question type="boolean" id="AnzahlButton"
4:      help="c:\juggler\help\ButtonZahl.html">
5:        Wollen Sie eine Juggler-Anwendung mit einem StartButton,
6:        die automatisch nach einer bestimmten Zeit anhält?
7:     </question>
8:    </variable>
9:    ...
10:   <!-- Definition der Architekturalternativen -->
11:   <if condition="! var:EinButtonJuggler">
12:   <!-- Standard-Juggler mit zwei Knoepfen -->
13:    <component class="java.awt.Button" id="StopButton"/>
14:   </if>
15:
16:   <else>
17:   <!-- Ein-Button-Juggler mit Timer -->
18:    <component class="javax.swing.Timer" id="Timer"/>
19:   </else>
20:   ...
```

Abbildung 6.7: Juggler-Alternative mit dem if-Tag

Im Beispiel wird zunächst eine Frage formuliert, bei welcher der Anwendungsentwickler sich für oder gegen den Ein-Button-Juggler entscheiden muss (Zeile 3 bis 7). Die Antwort wird der Variablen `EinButtonJuggler` zugeordnet und steht für den nachfolgenden Gebrauch zu Verfügung. In Zeile 11 kommt schließlich die if-Abfrage zum Einsatz, welche die Architekturalternative implementiert. Im `condition`-Attribut des Tags wird zunächst

überprüft, ob der Benutzer sich gegen den Ein-Button-Juggler entschieden hat (Negation der boolschen Variable `EinButtonJuggler`). Falls dieser Fall zutrifft, wünscht der Benutzer einen Standard-Juggler und es wird ein zweiter Button für die Stopp-Funktion instanziiert (Zeile 13). Falls dies nicht zutrifft, so wird der else-Zweig ausgeführt, der den benötigten Timer instanziiert (Zeile 18). Die gleichzeitige Verwendung von StopButton und Timer ist ausgeschlossen, da sich beide in gegenseitig ausschließenden Zweigen der Ablauflogik befinden. Natürlich müssten in der gleichen Weise auch andere Konsequenzen der Architekturalternativen (z.B. Verbindungen, Parametrierungen) berücksichtigt werden, was im Beispiel jedoch nicht ausgeführt ist.

Außer den vorgestellten `<if>`- und `<else>`-Tags stehen auch `<switch>`- und `<case>`-Tags für Verzweigungen in der Ablauflogik zu Verfügung. Die Verwendung dieser Tags bietet sich vor allem an, wenn eine Variable mehrere unterschiedliche Werte annehmen kann und in Abhängigkeit von diesen Werten verschiedene unabhängige Beschreibungsblöcke ausgeführt werden sollen. Für den Juggler wäre beispielsweise zusätzlich zur Standard- und Ein-Button-Version eine weitere Alternative denkbar, die permanent animiert ist und ohne Buttons auskommt. Falls der Benutzer sich anhand einer Frage zwischen diesen drei Alternativen entscheiden muss, so ist eine Realisierung mit dem `<switch>`-Element sinnvoll.

Schleifen

Zusätzlich zu den bisher behandelten Beschreibungselementen für die Verzweigung in der Ablauflogik besitzt FML auch Steuerungselemente, die zur Realisierung von Schleifen verwendet werden können. Um einzelne Beschreibungsteile mehrfach zu durchlaufen, kann eine `<for>`-Schleife verwendet werden. Dieses Schleifenkonstrukt bietet sich an, wenn die Anzahl der Schleifendurchläufe bereits vor Eintritt in die Schleife bekannt ist.

Um die Verwendung der if-Schleife zu demonstrieren, soll eine Komponente mehrfach instanziiert werden. Im Rahmen des Juggler-Beispiels könnte eine Anwendung beschrieben werden, die aus mehreren identischen Juggler-Instanzen besteht. Alle Juggler werden durch dieselben Buttons gestartet und gestoppt. Abbildung 6.8 zeigt diese auf Mehrfachinstanziierungen basierende Juggler-Variante für zwei Juggler.

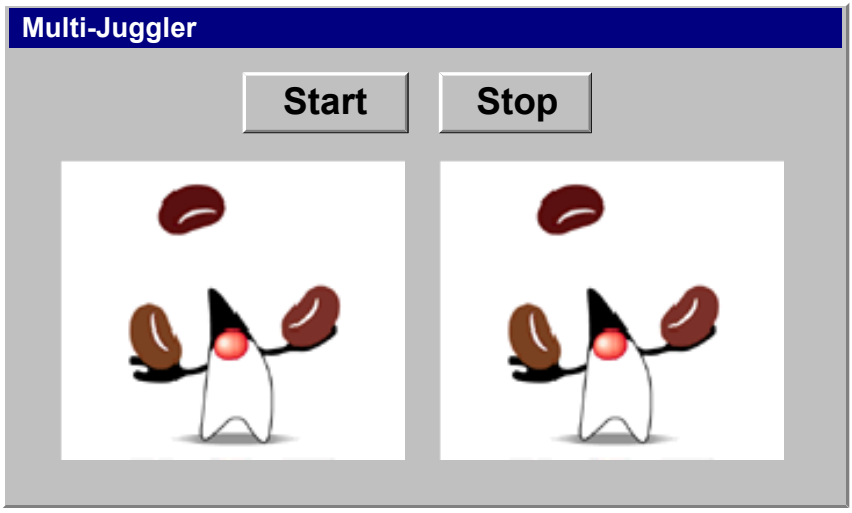

Abbildung 6.8: Anwendung mit zwei Juggler-Instanzen

```
1:    ...
2:    <!-- Definition der verwendeten Komponenten -->
3:    <for variable="i" start-value="1"
4:     condition="var:i &lt;= 2" counter="+">
5:     <!-- Fuer jede Instanz unterschiedliche ID -->
6:     <variable name="id">
7:      <variable name="nameStr" value="Juggler" operation="+">
8:       <variable name="i"/>
9:      </variable>
10:    </variable>
11:
12:    <component class="sunw.demo.juggler.Juggler" id="var:id"/>
13:
14:   </for>
15:   ...
```

Abbildung 6.9: Doppel-Juggler im FML-Code

Abbildung 6.9 zeigt ausschnittsweise die Realisierung des beschriebenen Doppel-Jugglers im entsprechenden FML-Code. Zeile 3 zeigt die Verwendung der for-Schleife für die zweifache Instanziierung des Jugglers. Das `<for>`-Tag wird durch folgende Attribute gesteuert:

- `variable` gibt den Namen der Zählvariablen an und entspricht dem `name`-Attribut des Elements `<variable>`.
- `start-value` definiert den Startwert der Zählvariablen. Statt der 1 (wie im Beispiel) können beliebige Integerwerte verwendet werden.

- `condition` definiert die Abbruchbedingung für die Schleife. Im Beispiel wird die Schleifenausführung unterbrochen, wenn die Variable `i` nicht mehr kleiner gleich 2 ist. Da das Zeichen „<“ als Tag-Begrenzung eine besondere Bedeutung in XML-Dateien besitzt, muss es durch die Entität „`<`“ (less than) ersetzt werden. Das Gleiche gilt für die Zeichen „>“ und „`&`“, die als „`>`“ (greater than) und „`&`“ (&) notiert werden.

- `counter` ermöglicht zusätzlich die Festlegung einer Zählrichtung, d.h. man kann angeben, ob die Zählvariable inkrementiert („+“) oder dekrementiert („−“) werden soll.

Da in FML jede Komponente eine eindeutige ID benötigt, muss sichergestellt werden, dass bei jedem Schleifendurchgang und für jede Juggler-Instanziierung eine andere ID vergeben wird. Dies geschieht mit Hilfe der Zeilen 6 bis 10. In diesen Zeilen wird der Wert der Variablen `id` berechnet, der sich aus der konstanten Zeichenkette „Juggler“ und dem aktuellen Wert der Zählvariablen `i` zusammensetzt („Juggler1“, „Juggler2“). Erst in Zeile 12 wird der Juggler instanziiert und mit der bereitgestellten ID versehen.

Zusätzlich zum `<for>`-Tag stellt das `<while>`-Tag eine weitere Möglichkeit zur Verfügung, um Schleifen zu implementieren. Das Schleifenende hängt dabei von einer beliebigen Bedingung ab, die im `condition`-Attribut nach den bereits bekannten Regeln formuliert wird. Zur Vermeidung von Endlosschleifen muss innerhalb der Schleife ein Zustand erreicht werden können, der die Beendigung der Schleifendurchläufe auslöst.

6.2.3 Komponentenalternativen

Bei der Verwendung von Komponenten in Framework-Spezifikationen können Situationen auftreten, in denen verschiedene Komponenten für dieselbe Aufgabe zur Verfügung stehen. Beispielsweise könnte statt der gewohnten Juggler-Komponente eine Alternative mit unterschiedlichem grafischen Aussehen angeboten werden (siehe Abbildung Abbildung 6.10). Beide Varianten erfüllen sinngemäß dieselbe Aufgabe, können aber vollkommen unterschiedliche Eigenschaften besitzen.

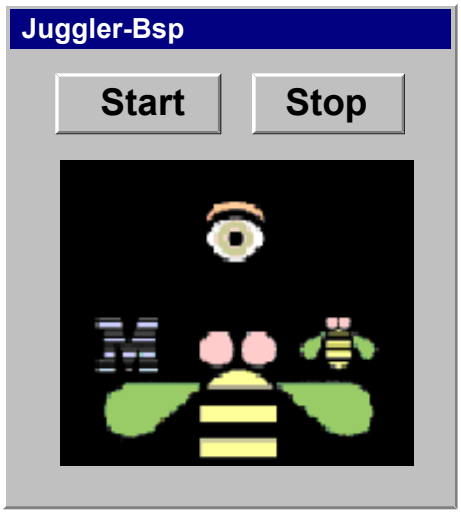

Abbildung 6.10: Juggler-Alternative

Als weiteres Szenario wäre auch denkbar, dass nach der Fertigstellung des Frameworks neue Komponenten entwickelt werden sollen. Diese sollen dem Benutzer dann zur Auswahl stehen, ohne dass die ursprüngliche Framework-Spezifikation modifiziert werden muss. Dabei kann es auch vorkommen, dass die nachträglich hinzugekommenen Komponenten andere Schnittstellen als die ursprünglichen Komponenten besitzen. Um diese Möglichkeiten angemessen zu berücksichtigen, führt FML das Konzept der *Komponentenalternativen* ein. Innerhalb eines Frameworks sind Komponentenalternativen redundante Komponenten mit identischen Schnittstellen und ähnlichen Aufgaben, die jedoch unterschiedliche Implementierungseigenschaften aufweisen. Im Nachfolgenden soll zunächst beschrieben werden, wie Komponentenalternativen vereinbart werden. Im darauf folgenden Abschnitt wird dann ihre Verwendung innerhalb von Framework-Spezifikationen gezeigt.

Beschreibung von Komponentenalternativen

Um Komponentenalternativen zu verwenden, müssen sie zunächst beschrieben werden. Zu diesem Zweck steht das `<comp-alternatives>`-Tag zur Verfügung. Dieses Tag steht auf der gleichen Hierarchieebene wie die Framework-Spezifikation (`<fw-spec>`), d.h. die Komponentenalternativen werden hierarchisch direkt nach dem `<project>`-Tag innerhalb der FML-Datei beschrieben. Innerhalb des `<comp-alternatives>`-Tags können beliebig viele Alternativen beschrieben werden, die jeweils durch `<comp-implementation>`-Tags voneinander abgegrenzt sind. Wird innerhalb des `<comp-alternatives>`-Tags nur eine einzige Komponentenalternative (nur ein `<comp-implementation>`-Tag) beschrieben, so wird dem Benutzer der Framework-Spezifikation im Instanziierungsprozess keine Auswahlmöglichkeit gegeben, sondern automatisch diese einzige Alternative verwendet. Auch in solchen Situationen kann die Definition einer Komponentenalternativen-Beschreibung sinnvoll sein, um spätere Erweiterungen mit zusätzlichen Komponenten zu erleichtern.

Abbildung 6.11 zeigt anhand des Juggler-Beispiels den benötigten FML-Code für eine Komponentenalternative, die aus zwei Komponenten (Standard Sun Juggler und IBM Juggler) besteht.

```
1:    <project id="Juggler-Projekt" main-fw="Juggler-FW">
2:     <fw-spec id="Juggler-Framework">
3:     ...
4:     </fw-spec>
5:
6:     <comp-alternatives id="JugglerCompAlt"
7:      help="c:\juggler\help\JugglerAlt.html">
8:
9:     <!-- erste Alternative: Standard Sun Juggler -->
10:    <comp-implementation class="sinw.demo.juggler.Juggler"
11:     id="SunJuggler" help="c:\juggler\help\SunJuggler.html">
12:    <!-- Eingabeschnittstellen abbilden -->
13:    <in name="starten">
14:     <script>
15:      <call-method name="start" target="this" />
16:     </script>
17:    </in>
18:    <in name="anhalten">
19:     <script>
20:      <call-method name="stop" target="this" />
21:     </script>
22:    </in>
23:    </comp-implementation>
24:
25:    <!-- zweite Alternative: IBM Juggler -->
26:    <comp-implementation class="demos.juggler.Juggler" id="IBMJuggler"
27:     help="c:\juggler\help\IBMJuggler.html">
28:    <!-- Eingabeschnittstellen abbilden -->
29:    ...
30:    </comp-implementation>
31:
32:    </comp-alternatives>
33:   </project>
```

Abbildung 6.11: Beschreibung von Komponentenalternativen

Wie bereits erläutert, wird die Definition von Komponentenalternativen mit dem Tag `<comp-alternatives>` eingeleitet (Zeile 6). Das `id`-Attribut gibt jeder Komponentenalternative eine eindeutige Kennzeichnung, die für die spätere Verwendung innerhalb der Framework-Spezifikation wichtig ist. In der dargestellten FML-Beschreibung sind zwei alternative Juggler-Komponenten vorhanden, die jeweils durch entsprechende `<comp-implementation>`-Beschreibungselemente eingeleitet werden (Zeile 10 und 26). Während das `class`-Attribut des

Elements auf die dazu gehörige Komponenten-Implementierung verweist, wird das angegebene `id`-Attribut während der Instanziierung als Beschreibungstext für den Auswahldialog verwendet.

Der innere Aufbau des `<comp-implementation>`-Tags kann im Beispiel einfach nachvollzogen werden. Zunächst wird die in der Framework-Spezifikation verwendete einheitliche Schnittstelle für alle Komponentenalternativen auf die konkreten Schnittstellen der einzelnen Komponenten abgebildet. Dies ist insofern wichtig, da die einzelnen realen Komponenten durchaus unterschiedliche Schnittstellen besitzen dürfen, innerhalb der Framework-Spezifikation jedoch durch eine einheitliche und für alle Alternativkomponenten verbindliche Schnittstelle angesprochen werden. Die Abbildung der Eingangsschnittstellen geschieht mit Hilfe des `<in>`-Tags[6]. Das `name`-Attribut dieses Tags gibt den in der Framework-Spezifikation verwendeten Funktionsnamen (z.B. „starten“ in Zeile 13) an. Innerhalb dieses Tags erfolgt die Zuordnung zur Schnittstellen-Funktion der vorliegenden Komponente. Das `<call-method>`-Tag gibt an, welche Funktion der vorliegenden Komponente (Attribut `name`) aufgerufen wird, wenn die entsprechende Framework-Schnittstelle angefordert wird. Da zur Spezifikationszeit die aufrufende Komponente nicht bekannt ist, wird mit Hilfe des Attributs `target` eine Zuordnung zum Zeitpunkt der Instanziierung erzwungen. Der Attributwert `this` bezieht sich somit auf die Komponente, von der die betreffende Komponentenalternative aufgerufen wird.

Komponentenalternativen in der Framework-Spezifikation

Nachdem im vorangehenden Abschnitt erläutert wurde, wie Komponentenalternativen beschrieben werden, steht in diesem Abschnitt ihre Verwendung innerhalb von Framework-Spezifikationen im Vordergrund. Bisher diente das `<component>`-Tag dazu, konkrete Komponenten mit Hilfe des `class`-Attributs zu instanziieren. Mit Hilfe des neu eingeführten `alternatives`-Attributs wird dasselbe Tag erweitert, um auch Komponentenalternativen ansprechen zu können. Dazu muss lediglich das `alternatives`-Attribut mit dem Wert belegt werden, der in der gewünschten `<comp-alternatives>`-Vereinbarung als `id`-Attribut spezifiziert wurde. Abbildung 6.12 zeigt beispielhaft die Verwendung der Komponentenalternative, die in Abbildung 6.11 vereinbart wurde.

Trifft das Instanziierungswerkzeug auf ein Beschreibungselement wie in Zeile 4, so wird nach der dazu gehörenden Vereinbarung für die Komponentenalternativen gesucht und – falls mehrere Alternativen vorhanden sind – dem Anwendungsentwickler die Auswahl einer konkreten Implementierung angeboten. Hat der Entwickler seine Wahl getroffen, so steht die gewählte Komponente unter der `id` „Juggler“ zur Verfügung und kann innerhalb der

Framework-Spezifikation wie gewohnt mit anderen Komponenten verbunden und parametriert werden.

```
1:    <project id="Juggler-Projekt" main-fw="Juggler-FW">
2:     ...
3:     <!-- Verwendung der Juggler-Alternativen -->
4:     <component alternatives="JugglerCompAlt" id="Juggler"/>
5:     ...
6:    </project>
```

Abbildung 6.12: Verwendung von Komponentenalternativen

6.3 Weiterführende FML Spracheigenschaften

6.3.1 Internationalisierung

Im Zuge der Globalisierung wird moderne Software häufig für verschiedene Länder und Sprachen angepasst. Diese so genannte *Internationalisierung* gilt natürlich auch für die Anwendungsentwicklung mit Frameworks – der Anwendungsentwickler muss die Möglichkeit haben, die Instanziierung eines Frameworks in seiner Sprache vorzunehmen, um potenzielle Schwierigkeiten durch fehlende Fremdsprachen-Kenntnisse zu vermeiden. Der Frameworkentwickler muss andererseits die Möglichkeit haben, das Framework einfach und ggf. auch nach Abschluss der Framework-Spezifikation an neue Sprachen anzupassen. FML bietet eine umfassende Unterstützung für die Internationalisierung, die sowohl dem Framework- als auch dem Anwendungsentwickler weitestgehend entgegenkommt.

Die grundsätzliche Idee zur Realisierung einer überschaubaren und leicht erweiterbaren Sprachunterstützung beruht darauf, dass alle sprachspezifischen Teile aus der eigentlichen Framework-Spezifikation in spezielle Blöcke ausgelagert werden. In der Framework-Spezifikation werden die einzelnen Elemente dieser Blöcke durch für alle Sprachen identische IDs referenziert. Alle Sprachblöcke werden durch `<lang-reference>`-Tags abgegrenzt, welche auf der gleichen Hierarchieebene wie die Framework-Spezifikation (`<fw-spec>`) oder die Komponentenalternativen (`<comp-alternatives>`) stehen. Für jede Sprache wird ein eigener `<lang-reference>`-Block vereinbart, wobei die Sprache durch das Attribut `xml:lang` definiert wird. Die Festlegung der Sprachen und Länder erfolgt nach den Standardkonventionen von XML, wonach „de-DE“ beispielsweise für „deutsch-Deutschland“ steht. Abbildung 6.13 zeigt ausschnittsweise, wie ein Sprachblock für das Juggler-Beispiel aussehen könnte.

---

[6] Es existiert auch ein <out>-Tag zur Abbildung von Ausgangsschnittstellen, was jedoch hier nicht weiter erläutert wird.

```
1:    <project id="Juggler-Projekt" main-fw="Juggler-FW">
2:     <lang-reference xml:lang="de-DE">
3:
4:     <!-- sprachabhängige Hilfe-Dateien -->
5:     <file id="WillkommenHilfe"
6:      target="c:\Juggler\help\Willkommen.html"/>
7:
8:     <!-- Textmeldungen und Fragen -->
9:     <message id="Willkommen"> Willkommen zum Beispielframework!
10:    </message>
11:
12:    <!-- Auswahlmöglichkeiten -->
13:    <choice id="ButtonAuswahl">
14:      <choice-element display="Je ein Button für Start und Stopp"
15:       value="2"/>
16:      <choice-element display="Nur ein Startbutton" value="1"/>
17:    </choice>
18:
19:    </lang-reference>
20:   </project>
```

Abbildung 6.13: Internationalisierung mit FML

Innerhalb eines Sprachblocks können drei unterschiedliche Arten von Sprachelementen erfasst werden. Zunächst werden sprachabhängige Hilfedateien angegeben (Zeile 5 und 6), die umfangreiche Hilfstexte und Erläuterungen für die Instanziierung beinhalten. Danach folgen Textmeldungen und Fragetexte (Zeile 9 und 10), die im `<info>`- bzw. `<question>`-Tag verwendet werden. Abschließend folgen noch sprachabhängige Auswahltexte (Zeile 13 bis 17), die für Auswahllisten mit mehreren Optionen benötigt werden. Jedes der genannten Elemente hat ein `id`-Attribut, um später für die Framework-Spezifikation referenzierbar zu sein. Diese `id` muss pro Sprache und innerhalb eines Projektes einmalig sein. Gleiche Elemente in unterschiedlichen Sprachen besitzen natürlich die gleiche `id`, da sie inhaltlich gleichwertig sind.

Nachdem die sprachabhängigen Teile in einem separatem Block beschrieben wurden, können sie innerhalb der Framework-Spezifikation verwendet werden. Um beispielsweise während der Instanziierung eine sprachabhängige Nachricht an den Anwendungsentwickler auszugeben, ist folgende FML-Zeile notwendig:

```
<info message="Willkommen" id="Willkommen"
 help="WillkommenHilfe"/>
```

Dieses `<info>`-Tag unterscheidet sich von den bisher gezeigten Tags (siehe Abbildung 6.4) durch das neu hinzugekommene `message`-Attribut, welches der `id` des `<message>`-Tags in der Sprachblock-Vereinbarung entspricht. Die direkte Angabe des Nachrichtentextes wird somit durch eine Referenz auf einen Text im Sprachblock ersetzt. Weiterhin wird auch im `help`-

Attribut keine konkrete Hilfedatei vereinbart sondern eine entsprechende Datei im Sprachblock referenziert. Mit Hilfe dieser Referenzierungen können Framework-Spezifikationen erstellt werden, die vollkommen frei von sprachabhängigen Elementen sind, was die Anpassung an verschiedene Sprachen natürlich wesentlich erleichtert.

6.3.2 Projektaufteilung in mehrere Dateien

Die bisherige Vorstellung der FML-Beschreibungselemente geschah unter der impliziten Annahme, dass die gesamte Frameworkbeschreibung in einer großen Datei gespeichert werden muss. Eine solche Einschränkung kann selbst bei kleineren Projekten schnell zur Unübersichtlichkeit führen. Für größere Projekte, bei denen mehrere Mitarbeiter parallel an der Frameworkbeschreibung arbeiten, ist eine solche Einschränkung im Voraus mit vielen Schwierigkeiten verbunden. Aus diesem Grund unterstützt FML die flexible Verteilung der Frameworkbeschreibung auf mehrere Dateien, die unabhängig voneinander entwickelt und gepflegt werden können.

Grundsätzlich können alle Beschreibungselemente, die direkt innerhalb eines `<project>`-Tags auftreten können, in eine separate Datei ausgelagert werden. Dies betrifft von den bisher vorgestellten Elementen vor allem die Framework-Spezifikation (`<fw-spec>`), die Vereinbarung von Komponentenalternativen (`<comp-alternatives>`) und die sprachabhängigen Informationen (`<lang-reference>`). Um die auf einzelne Dateien verteilten Beschreibungsteile wieder zusammenzuführen, muss eine Projektdatei erstellt werden. Diese Projektdatei referenziert alle zusammengehörenden Dateien und erzeugt auf diese Weise wieder eine zusammenhängende Frameworkbeschreibung. Abbildung 6.14 zeigt den Aufbau einer Projektdatei.

```
1:    <fml-file author="Du" company="ias" date="6.2.01" version="1.0">
2:     <project id="Bsp-Projekt" main-fw="Bsp-FW">
3:
4:     <file-reference>
5:     <path value="c:\juggler\">
6:       <file target="Juggler-Framework-Spec.fml"/>
7:       <file target="Juggler-Alternativen.fml"/>
8:       <file target="Juggler-Sprachunterstuetzung_de_DE.fml"/>
9:     </path>
10:    </file-reference>
11:
12:    </project>
13:   </fml-file>
```

Abbildung 6.14: Aufbau einer Projektdatei

Wie das Beispiel veranschaulicht, enthält die Projektdatei keine Beschreibungsdaten, sondern definiert lediglich mit Hilfe des `<file-reference>`-Tags (Zeile 4), welche Dateien zum Projekt gehören. Das `<path>`-Element in Zeile 5 erleichtert zusätzlich die Referenzierung der einzelnen Dateien (Zeile 6 bis 8), da nur noch der Dateiname und nicht der vollständige Verzeichnispfad angegeben werden muss.

6.3.3 Komponenten-Schnittstelle für Frameworks

In vielen Fällen werden Frameworkbeschreibungen dazu verwendet, eine Anwendung als Endprodukt der Instanziierung zu spezifizieren. Es kann jedoch auch Fälle geben, bei denen der Anwendungsentwickler keine fertige Anwendung wünscht, sondern eine komplexe Komponente, die er beliebig in seinen Anwendungskontext einbetten kann. Dieser Fall könnte beispielsweise auftreten, wenn ein Hersteller zwar die Steuerungssoftware für seine Maschinenkonfigurationen flexibel halten muss, dem Anwender jedoch nicht die grafische Benutzungsoberfläche für die Software vorschreiben möchte. Deshalb ist es zweckmäßig, als Endprodukt der Instanziierung nicht nur vollständige Anwendungen, sondern auch neue, vom Benutzer konfigurierte Komponenten zuzulassen. Diese Komponenten bestehen aus beliebigen Sub-Komponenten und Kommunikationsbeziehungen, die der Anwendungsentwickler während der Instanziierung speziell an seine Bedürfnisse anpasst.

Um eine Komponente als Endprodukt der Framework-Instanziierung zu ermöglichen, muss eine Schnittstelle in der Frameworkbeschreibung vereinbart werden, die das Verhalten der Komponente nach außen hin spezifiziert. Dabei werden natürlich nur solche Merkmale und Eigenschaften nach außen offen gelegt, die für die weitere Verwendung der Komponente notwendig sind. Alle anderen internen Details bleiben innerhalb der Komponente gekapselt und sind für den Benutzer der Komponente nicht sichtbar. Außer der Beschreibung der äußeren Schnittstelle müssen die einzelnen Schnittstellen-Elemente auf die entsprechenden Elemente innerhalb der Komponente abgebildet werden. Dieser Vorgang ist vergleichbar mit der Verwendung von Komponentenalternativen (siehe 6.2.3), wo ebenfalls einzelnen Schnittstellenelementen konkrete Komponenten-Implementierungen zugeordnet werden. Abbildung 6.15 zeigt anhand des Juggler-Beispiels die Vereinbarung einer Framework-Schnittstelle, welche die Erzeugung einer erweiterten Juggler-Komponente (Juggler, Start- und Stopp-Button) aus der Frameworkbeschreibung ermöglicht.

```
1:    <fml-file author="Du" company="ias" date="6.2.01" version="1.0">
2:     <comp-interface id="Juggler-FW-CompInterface">
3:
4:     <!-- Eingabeschnittstellen definieren -->
5:     <in name="start" received-event="java.awt.event.ActionEvent"/>
6:     <in name="stop" received-event="java.awt.event.ActionEvent"/>
7:
```

```
8:      <!-- auszusendende Events -->
9:      <out event-listener="java.awt.event.ActionListener"
10:      thrown-event="java.awt.event.ActionEvent"/>
11:
12:     <!-- Komponenten-Eigenschaften -->
13:     <property name="sichtbar">
14:       <property-definition type="boolean"/>
15:     </property>
16:
17:     </comp-interface>
18:    </fml-file>
```

Abbildung 6.15: Framework-Schnittstelle

Die Vereinbarung einer externen Schnittstelle für das Framework erfolgt mit Hilfe des `<comp-interface>`-Tags. Diese Schnittstellen-Definition kann entweder in einem extra Block innerhalb eines bestehenden FML-Projektes oder (wie im obigen Beispiel) in einer separaten Datei vereinbart werden, die von der Projektdatei referenziert wird. Im letzteren Fall dient das `id`-Attribut zur Referenzierung innerhalb der Projektdatei. Innerhalb des Blocks für die Schnittstellenbeschreibung werden nacheinander alle Eingänge (`<in>`), Ausgänge (`<out>`) und Eigenschaften (`<property>`) definiert. Im Beispielcode besitzt die Juggler-Komponente zwei Eingangsmethoden mit dem Namen `start` und `stop` (Zeile 5 bis 6), die Ereignisse vom Typ `ActionEvent` empfangen. Die von der Komponente ausgehenden Ereignisse werden durch die für den Empfänger notwendige Empfangsmethode (`ActionListener`) und den Ereignistyp (`ActionEvent`) spezifiziert (Zeile 9 bis 10). Schließlich können noch Eigenschaften festgelegt werden, die bei der Verwendung der aus dem Framework resultierenden Komponente parametrierbar sind (Zeile 13 bis 15). Alle in der Schnittstellen-Definition vereinbarten Merkmale stehen somit für die Benutzung innerhalb eines beliebigen Anwendungskontextes zur Verfügung.

6.3.4 Benutzerdefinierte Komponentenalternativen

Der bisher beschriebene Sprachumfang von FML erlaubt dem Anwendungsentwickler die Benutzung aller Optionen, die vom Frameworkentwickler explizit vorgesehen sind. Es sind aber auch Situationen denkbar, in denen der Anwender die Frameworkbeschreibung unverändert verwenden und gleichzeitig eigene Komponenten-Implementierungen einsetzen möchte. Eine solche Situation wäre beispielsweise gegeben, wenn in der Komponentenbibliothek für eine neue Hardwarevariante keine passende Treiberkomponente vorhanden ist, obwohl die vorhandene Frameworkbeschreibung die gewünschte Anwendung vollständig abdeckt. Für diese Situationen bietet FML die Definition von benutzerdefinierten Komponentenalternativen an, wodurch die Frameworkbeschreibung für Implementierungen außerhalb der vom Frameworkentwickler vorgesehenen Komponentenbibliothek geöffnet wird. Abbildung 6.16

zeigt, wie in FML benutzerdefinierte Komponentenalternativen am Beispiel einer zusätzlichen, vom Anwendungsentwickler implementierten Juggler-Komponente (`OwnJuggler`) vereinbart werden.

```
1:     <comp-alternatives id="JugglerCompAlt"
2:      help="c:\juggler\help\JugglerAlt.html">
3:
4:     <!-- bisherige Alternativen in der Bib: Sun und IBM Juggler -->
5:     <comp-implementation class="sinw.demo.juggler.Juggler"
6:      id="SunJuggler/>"
7:     <comp-implementation class="demos.juggler.Juggler"
8:      id="IBMJuggler"/>
9:
10:    <!-- offene Komponentenalternative ausserhalb der Bibliothek -->
11:    <comp-implementation id="OwnJuggler"
12:     help="c:\juggler\help\OwnJuggler.html">
13:     <!-- Eingabeschnittstellen deklarieren -->
14:     <in name="starten"/>
15:     <in name="anhalten"/>
16:    </comp-implementation>
17:
18:    </comp-alternatives>
```

Abbildung 6.16: Benutzerdefinierte Komponentenalternativen

Um eine benutzerdefinierte Komponentenalternative zu vereinbaren, wird die Liste der bereits vorhandenen Alternativen um ein weiteres `<comp-implementation>`-Tag erweitert (siehe Zeile 11). Im Gegensatz zu den anderen Alternativen fehlt bei dieser Erweiterung das `class`-Attribut, wodurch die Zuordnung der Alternative zu einer konkreten Komponenten-Implementierung offen gelassen wird. Weiterhin bleiben auch die einheitlichen Schnittstellenfunktionen (Zeile 14 bis 15) ohne entsprechende Zuordnungen. Aufgrund dieser fehlenden Zuordnungen wird der Benutzer während der Instanziierung bei Auswahl dieser Alternative aufgefordert, eine Implementierung zuzuordnen. Danach muss er die einheitlichen Schnittstellenfunktionen den konkreten Funktionen der gewählten Komponente zuordnen. Auf diese Weise kann der Benutzer eine eigene Juggler-Implementierung verwenden, ohne die Framework-Spezifikation zu modifizieren.

6.3.5 Nichtfunktionale Anforderungen

In vielen Bereichen der Automatisierungstechnik spielen nichtfunktionale Anforderungen eine wichtige Rolle für die Funktionsfähigkeit des Gesamtsystems. Dazu gehören besonders Echtzeitanforderungen, Speicherplatzeffizienz und besondere Anforderungen an die Sicherheit und Zuverlässigkeit von Automatisierungssystemen (siehe Abschnitt 3.5.2). Der Framework-entwickler muss in die Lage versetzt werden, diese nichtfunktionalen Anforderungen seines

Frameworks zu dokumentieren und damit verbundene Einschränkungen für die Anwendungsentwicklung explizit auszudrücken. Auf der Seite des Anwendungsentwickler muss gewährleistet sein, dass eine neue, frameworkbasierte Anwendung im Zielsystem lauffähig ist, ohne beispielsweise bestehende Zeitschranken zu verletzen. Der Anwendungsentwickler möchte nicht erst nach der abgeschlossenen Instanziierung erfahren, dass seine neue Anwendung wesentliche nichtfunktionale Anforderungen nicht einhalten kann.

FML bietet dem Frameworkentwickler in der Frameworkbeschreibung die Möglichkeit, nichtfunktionale Einschränkungen und Bedingungen zu formulieren, die den Anwendungsentwickler während der Instanziierung auf potenzielle Verletzungen von nichtfunktionalen Anforderungen hinweisen. Die Durchsetzung von nichtfunktionalen Einschränkungen während der Instanziierung des Frameworks bleibt in der Verantwortung des Anwendungsentwicklers, d.h. falls entsprechende Hinweise während der Instanziierung ignoriert werden, können trotzdem Anwendungen entstehen, welche die geforderten nichtfunktionalen Anforderungen nicht erfüllen. Der Frameworkentwickler kann bestimmte nichtfunktionale Eigenschaften einer frameworkbasierten Anwendung nur dann zusichern, wenn gleichzeitig die von ihm geforderten Voraussetzungen eingehalten werden.

Für die nichtfunktionalen Anforderungen stehen alle Ausdrucksmöglichkeiten der FML zur Verfügung, die bisher vorgestellt wurden. Besonders geeignet für diese Aufgabe sind `<info>`-Tags, welche die Voraussetzungen für nichtfunktionale Eigenschaften in Form von Dialogfenstern präsentieren oder `<question>`-Tags, welche die Voraussetzungen direkt innerhalb einer Instanziierungsfrage darstellen (siehe Abschnitt 6.2.2). Abbildung 6.17 zeigt ein Dialogfenster für die Juggler-Komponente, welche den möglichen Wertebereich für die Animationsrate bei langsameren Prozessoren zugunsten einer flüssigen Animation einschränkt.

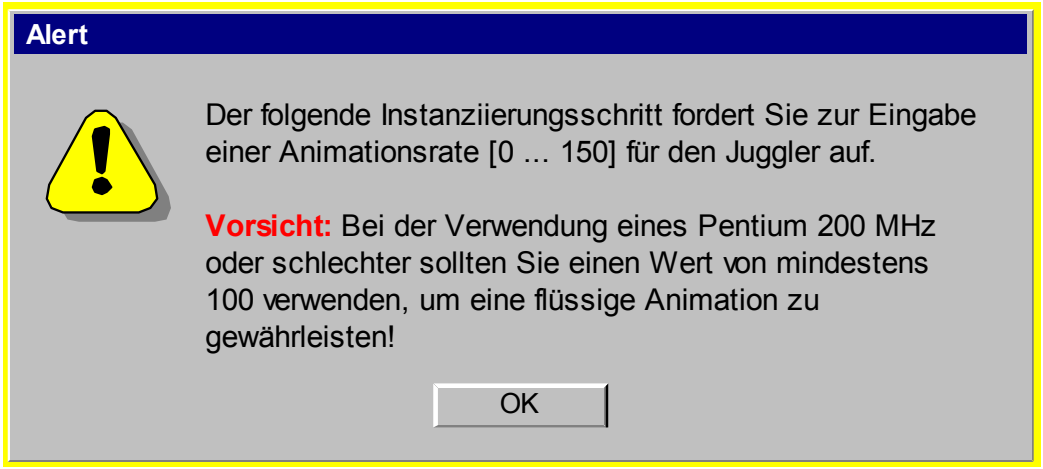

Abbildung 6.17: Nichtfunktionale Eigenschaften in der Frameworkbeschreibung

6.3.6 Vorparametrierungen

Bei der Auslieferung der Frameworkbeschreibung an verschiedene Anwendungsentwickler ist es wünschenswert, die identische Beschreibung optimal an die Anforderungen und Wünsche der einzelnen Anwender anpassen zu können. Denkbar wäre beispielsweise eine komplexe und umfangreiche Frameworkbeschreibung mit Hunderten von Auswahlmöglichkeiten, die für bestimmte Anwender auf wenige, aber wesentliche Auswahlmöglichkeiten reduziert wird. In einem solchen Fall bietet FML die Möglichkeit zur Vorparametrierung von Frameworkbeschreibungen. Dabei können die Auswahl- und Einstellungsmöglichkeiten beliebig eingeschränkt und die Beschreibung für den Anwender auf einen optimalen Umfang reduziert werden.

Die Vorparametrierung erfolgt in einer externen FML-Datei außerhalb der eigentlichen Frameworkbeschreibungen. Diese Datei wird einfach als zusätzliche Datei in der Projektdatei referenziert. Der Aufbau und Inhalt dieser Datei entspricht exakt den vom Instanziierungswerkzeug generierten Dateien, welche die Benutzereingaben speichern. Für den Frameworkentwickler ergibt sich damit eine effiziente Art der Vorparametrierung seiner Frameworkbeschreibung – er schlüpft einfach in die Rolle des Anwendungsentwicklers und belegt alle nicht benötigten Auswahlmöglichkeiten durch entsprechende Vorgaben. Alle für den Endanwender gewünschten Auswahlmöglichkeiten werden einfach übersprungen und bleiben dadurch offen. Am Ende werden die gemachten Vorgaben abgespeichert und als zusätzliche Projektdatei referenziert. Auf diese Weise kann der Frameworkentwickler gezielt die Bedürfnisse einzelner Anwender und Anwendergruppen ansprechen, ohne den FML-Code der Framework-Spezifikation zu ändern.

6.4 Vergleich von FML mit ähnlichen Spezifikationssprachen

6.4.1 Architekturbeschreibungssprachen

Für die globale Beschreibung eines Softwaresystems verwenden Entwickler häufig die *Architektur* eines Systems. Die Architektur eines Systems besteht dabei aus einzelnen Komponenten und den Verbindungen, welche die stattfindenden Interaktionen zwischen den Komponenten spezifizieren [BCK98, GaSh94]. Obwohl die explizite Beschreibung von Architekturentscheidungen eine wesentliche Rolle für das Verständnis von komplexen Softwaresystemen spielt, wird die Architektur häufig nur informell beschrieben und dargestellt. Dieser Sachverhalt resultiert in einer Reihe von Schwierigkeiten:

- Architekturentscheidungen sind nicht nachvollziehbar und entziehen sich einer formalen Analyse oder Simulation.

- Der Architekturentwurf kann sich – mangels entsprechender Ausdrucksmittel – nicht als Ingenieurstätigkeit etablieren.
- Architekturbedingte Einschränkungen können während der Systemevolution nicht kontrolliert und durchgesetzt werden.
- Die Arbeit des Systemarchitekten kann nicht durch dedizierte Werkzeuge unterstützt werden.

Um diese Schwierigkeiten in den Griff zu bekommen, wurden formalisierte Sprachen entwickelt, welche die präzise Spezifikation von Softwarearchitekturen ermöglichen. Die so genannten *Architekturbeschreibungssprachen* (eng. Architecture Description Language, ADL) dienen der expliziten und formalisierten Beschreibung von Softwarearchitekturen mit der Absicht, das Verständnis und die Wiederverwendbarkeit von Architekturentscheidungen zu fördern. ADLs werden hauptsächlich in der Analysephase des Software-Entwicklungsprozesses eingesetzt und sollen dabei helfen, die funktionalen und nichtfunktionalen Auswirkungen (z.B. Zeiteigenschaften, Sicherheit, Zuverlässigkeit) von bestimmten Architekturentscheidungen auf das zukünftige System im Voraus zu ermitteln. Typische Vertreter für ADLs sind *Rapide* [LKA+95], *Wright* [AlGa96] und *UniCon* [SDK+95] für allgemeine Softwaresysteme, sowie *MetaH* [BEJV96] für eingebettete Echtzeitanwendungen im Bereich der Luft- und Raumfahrt.

Bezüglich der expliziten Darstellung und Dokumentation von Architekturentscheidungen mit dem Ziel der erhöhten Verständlichkeit und Wiederverwendbarkeit verfolgt FML identische Absichten wie die genannten ADLs. Auch die Verfolgung von funktionalen und nichtfunktionalen Auswirkungen einzelner Entscheidungen auf die zukünftige Anwendung ist eine Gemeinsamkeit von beiden Ansätzen. Auffälligster Unterschied ist die Möglichkeit zur Vereinbarung von Variabilitäten – während ADLs immer auf eine bestimmte Architekturausprägung fixiert sind, erlaubt FML die Beschreibung von Variabilitäten auf verschiedenen Abstraktionsebenen. Im Vergleich zu ADL-Spezifikationen kann eine FML-Beschreibung somit als eine Meta-Spezifikation angesehen werden, die eine ganze Reihe von konkreten Architekturen und Anwendungen umfasst. Hauptproblem der ADLs ist die fehlende Kontinuität im Entwicklungsprozess – die Produkte und Ergebnisse der Architekturanalyse können nicht unmittelbar in entsprechenden Implementierungscode überführt werden. Nach der Verwendung der ADL muss der Anwendungsentwickler auf eine Implementierungssprache zurückgreifen und die gewählte Architektur von Grund auf neu implementieren. Durch die fehlende Kopplung zwischen der ADL und der Implementierungssprache ist nicht gewährleistet, dass die implementierte Systemarchitektur der ADL-spezifizierten Architektur entspricht. FML vermeidet diese Problematik, da jede FML-Beschreibung automatisch auch die Grundlage für die werkzeuggestützte Anwendungsentwicklung ist. Eine Diskrepanz zwischen der FML-spezifizierten Architektur und der realen Anwendungsarchitektur ist somit nicht möglich.

6.4.2 Komponentenbeschreibungssprachen

Beim Einsatz von Komponenten in der Anwendungsentwicklung entsteht der Bedarf nach einer passenden Beschreibung für die Komponenten, die sowohl implementierungsrelevante Informationen (z.B. Klassennamen, Typen, Methoden) als auch anwendungsnahe Information (z.B. Funktionsumfang der Komponente, wichtige Einschränkungen) beinhaltet. Eine solche Komponentenbeschreibungssprache muss also sowohl den Anwendungsentwickler bei der Auswahlentscheidung unterstützen als auch die automatisierte Verwendung der Komponente innerhalb einer Werkzeugumgebungen ermöglichen. Beispiele für solche XML-basierten Beschreibungssprachen für Komponenten (sog. Deskriptoren) sind die Deployment-Deskriptoren der *Enterprise JavaBeans* [EJB00] und die Deskriptoren der *CORBA Komponenten* [OMG99].

Vergleicht man Deskriptoren mit FML so ergeben sich erwartungsgemäß Gemeinsamkeiten. Beide Konzepte dienen sowohl zur Dokumentation von wieder verwendbaren Komponenten als auch zur werkzeuggestützten Instanziierung derselbigen. Die Beschreibungsmöglichkeiten für die Interaktion zwischen Komponenten und Variabilitäten sind bei Deskriptoren natürlich nicht vorhanden, da sie – im Gegensatz zur FML – die einzelne Komponente und nicht das Zusammenwirken von Komponenten in den Mittelpunkt stellen. Aus diesem Grund kann alles, was auf der Abstraktionsebene überhalb von einzelnen Komponenten steht (wie z.B. Architekturen und vollständige Anwendungen), mit Deskriptoren nicht beschrieben werden. In diesem Zusammenhang darf auch nicht vergessen werden, dass die Unterstützung des Anwendungsentwicklers durch beide Konzepte vollkommen unterschiedlich aussieht. Während der Entwickler mit Deskriptoren auf einer relativ implementierungsnahen Ebene arbeitet (Komponenten instanziieren, Kommunikationsverbindungen festlegen, Parameter setzen), ermöglicht die FML innerhalb der Werkzeugumgebung einen komfortablen, interaktiven Instanziierungsprozess, der sich an den Bedürfnissen des Domänenexperten ohne tiefergehende Programmierkenntnisse orientiert.

6.4.3 Anwendungsbeschreibungssprachen

Die *Bean Markup Language* (BML) [WeDu99] der Firma IBM ist eine XML-basierte Sprache zur Beschreibung von komponentenbasierten Anwendungen. Mit BML können Anwendungen beschrieben werden, die aus JavaBeans-Komponenten und den Kommunikationsbeziehungen zwischen diesen Komponenten bestehen. Aus den BML-Beschreibungen können mit Hilfe eines BML-Compilers ablauffähige Java-Anwendungen erzeugt werden. Damit entspricht die BML in etwa dem Funktionsumfang, den FML zur Beschreibung von statischen Anwendungsstrukturen bietet. Diese Ähnlichkeiten sind insofern nicht verwunderlich, da der Leistungsumfang von BML als Ausgangspunkt für die Entwicklung von FML verwendet wurde.

Im Gegensatz zu FML bietet die BML jedoch keine Merkmale, die über die Beschreibung von statischen, komponentenbasierten Anwendungen hinausgehen. Es ist beispielsweise nicht möglich, Variabilitäten oder Interaktionen mit dem Anwendungsentwickler zu vereinbaren. Weiterhin erlaubt BML nicht, Komponenten als Endprodukte der Instanziierung zu vereinbaren oder bestehende BML-Beschreibungen innerhalb von neuen Beschreibungen wieder zu verwenden. Trotz vorhandener Gemeinsamkeiten zwischen FML und BML im Bereich der Beschreibung von statischen Anwendungen ist BML somit nicht in der Lage, als Beschreibungssprache für Komponenten-Frameworks eingesetzt zu werden.

In diesem Kapitel wurde die Spezifikationssprache FML vorgestellt, die zur Erstellung von Frameworkbeschreibungen dient. Es wurde gezeigt, dass die FML sowohl zur Beschreibung von komponentenbasierten Anwendungen ohne Variabilitäten als auch zur Spezifikation von Komponenten-Frameworks mit einer Vielzahl unterschiedlicher Variabilitäten eingesetzt werden kann. Der Anwendungsentwickler wird normalerweise keine FML-Spezifikationen lesen, sondern über das Instanziierungswerkzeug FIT auf die in der Frameworkbeschreibung enthaltenen Informationen zugreifen. Aus diesem Grund wird im nächsten Kapitel das Werkzeug FIT ausführlich beschrieben, das die Anwendungsentwicklung auf der Grundlage von Komponenten-Frameworks wesentlich erleichtert.

7 Realisierung des Instanziierungswerkzeugs

In diesem Kapitel steht die Realisierung des Instanziierungswerkzeuges FIT (Framework Instantiation Tool) im Mittelpunkt der Betrachtungen. Dazu wird zunächst die grafische Benutzungsoberfläche präsentiert, die eine komfortable Anwendungsentwicklung auf der Grundlage von Komponenten-Frameworks ermöglicht. Danach werden sowohl die Software-Architektur als auch die einzelnen Bestandteile des Werkzeugs vorgestellt. Am Ende dieses Kapitels wird das Werkzeug FIT mit ähnlichen Werkzeugen für die Anwendungsentwicklung verglichen, die in der Praxis eingesetzt werden.

7.1 Grafische Benutzungsoberfläche des Framework Instantiation Tools

Hauptaufgabe von FIT ist die Unterstützung des Anwendungsentwicklers bei der Instanziierung von Komponenten-Frameworks, für die eine Spezifikation in FML vorliegt. FIT ermöglicht dem Anwendungsexperten die Erstellung verschiedener Anwendungen und komplexer Komponenten, ohne dabei tief gehende Programmierkenntnisse vorauszusetzen. Weitere wichtige Anforderungen, die bei der Entwicklung von FIT zugrunde gelegt wurden sind:

- Intuitive Bedienbarkeit, minimaler Einarbeitungsaufwand
- Plattformunabhängigkeit des Werkzeugs
- Framework-Unabhängigkeit
- Kopplung von Dokumentation und Instanziierung
- Schrittweise Instanziierung bei aktiver Benutzerführung

Bevor im nachfolgenden Abschnitt die Werkzeug-Architektur und die Implementierungsdetails von FIT erläutert werden, soll zunächst die Benutzungsoberfläche gezeigt werden, die dem Anwendungsentwickler präsentiert wird. Die FIT-Benutzungsoberfläche ist bewusst einfach und überschaubar gehalten, um dem Anwendungsentwickler ohne großen Einarbeitungsaufwand ein effizientes Arbeiten zu ermöglichen. Abbildung 7.1 zeigt die wesentlichen Elemente der Benutzungsoberfläche, die zur Steuerung des Werkzeuges und zur Instanziierung von Komponenten-Frameworks benötigt werden.

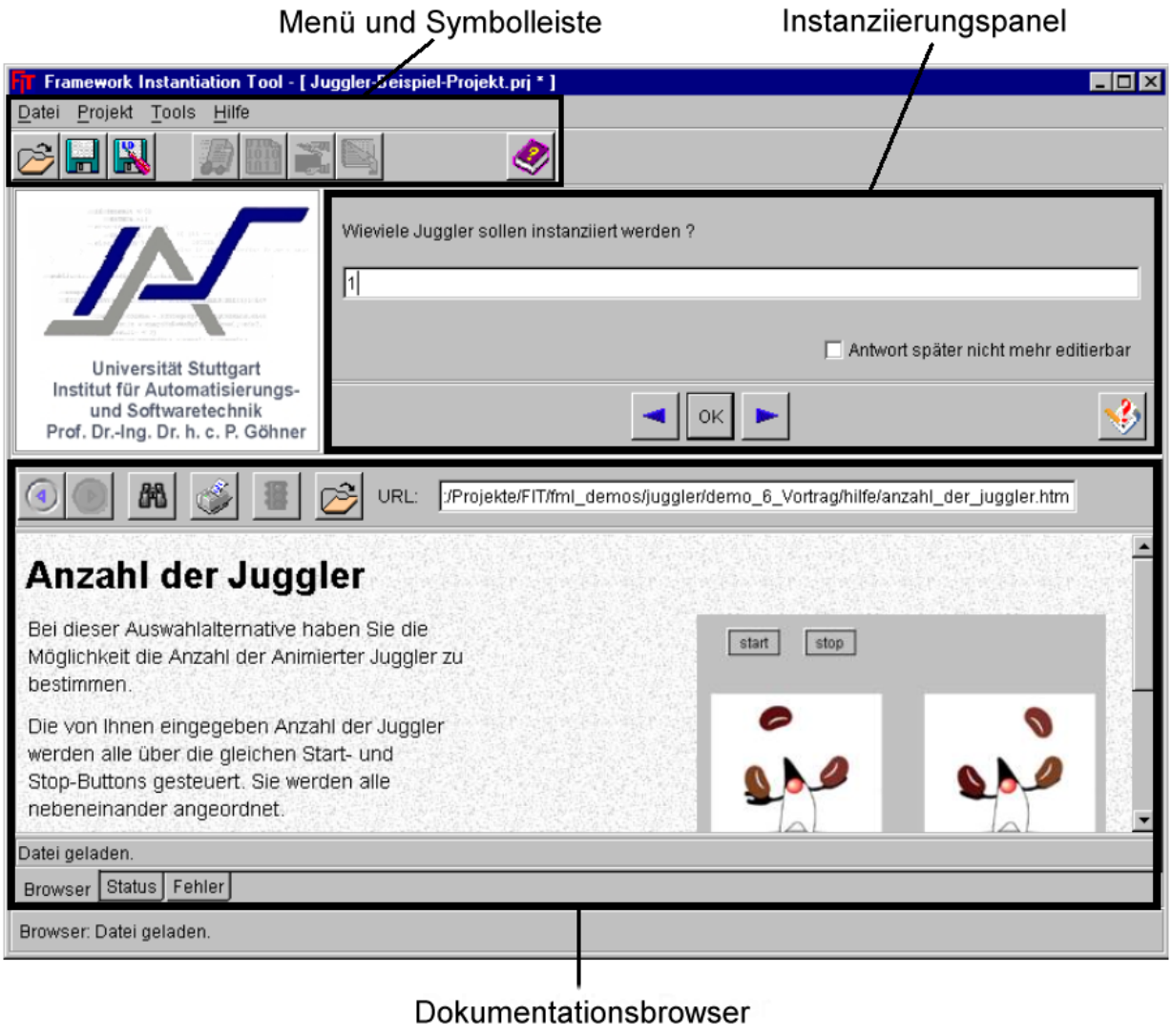

Abbildung 7.1: FIT-Benutzungsoberfläche

Das Werkzeug besitzt ein Menü und eine Symbolleiste, die einen schnellen Zugriff auf die wichtigsten Funktionen des Werkzeuges ermöglichen. Diese beiden Elemente der Benutzungsoberfläche dienen ausschließlich der Bedienung von FIT, wie z.B. dem Öffnen von neuen Frameworkbeschreibungen, dem Speichern von Eingabedaten des Anwendungsentwicklers, der Einstellung von Werkzeugoptionen, dem Aufruf von Hilfsseiten zur Werkzeugbedienung oder der Generierung von lauffähigen Anwendungen am Ende des Instanziierungsprozesses.

Während dem Instanziierungsprozess muss der Anwendungsentwickler sich lediglich auf zwei Bereiche innerhalb der Benutzungsoberfläche von FIT konzentrieren:

- *Instanziierungspanel* (eng. InstantiationPanel): In diesem Bereich werden Instanziierungsinformationen dargestellt oder Fragen präsentiert, auf die der Benutzer durch entsprechende Eingaben reagieren muss. Da der gesamte Instanziierungsvorgang in kleine Schritte

unterteilt ist, entspricht jede Frage im Panel einem Schritt im gesamten Instanziierungspfad. Durch die Buttons im unteren Teil des Panels kann der Entwickler Eingaben bestätigen (OK-Button) oder ohne Eingabe auf dem Instanziierungspfad vor-und zurücknavigieren.

- *Dokumentationsbrowser*: Dieser Bereich dient zur Darstellung von zusätzlichen Instanziierungs- oder Hilfsseiten, die als einfache Textdateien oder als HTML-formatierte Dokumente vorliegen. Durch die Auswahl einer entsprechenden Registrierkarte in der linken unteren Ecke des Browsers können auch Status- bzw. Fehlermeldungen angezeigt werden.

7.2 FIT-Softwarearchitektur

7.2.1 Übersicht

Der Entwurf der FIT-Softwarearchitektur wurde unter Berücksichtigung der Anforderungen vorgenommen, die bereits im Vorfeld erläutert wurden. Zusätzlich zu diesen Anforderungen wurden weitere Entscheidungen gefällt, die sich nicht zwangsläufig aus den Anforderungen ergeben, aber im gegebenen Kontext nahe liegend oder zumindest günstig sind. Im Nachfolgenden sollen diese Entscheidungen genannt und erläutert werden.

Aufgrund positiver persönlicher Erfahrungen in vorangegangenen Projekten wurde die Programmiersprache Java als Implementierungssprache und JavaBeans als Komponentenmodell für Komponenten-Frameworks gewählt. Aus diesem Grund wurde Java auch als Implementierungssprache für das FIT-Werkzeug beibehalten, obwohl auch andere Sprachen möglich gewesen wären. Da FIT für den Einsatz mit Komponenten-Frameworks konzipiert ist, wurde auch beim Entwurf von FIT auf eine komponentenbasierte Architektur Wert gelegt. Das Werkzeug besteht aus einer überschaubaren Anzahl von JavaBeans-Komponenten, die jede für sich eine abgeschlossene Funktionalität kapseln und untereinander durch Ereignisse kommunizieren. Abbildung 7.2 zeigt die Softwarearchitektur von FIT mit den unterschiedlichen Komponenten und den Kommunikationsbeziehungen zwischen den einzelnen Komponenten.

Wie aus der FIT-Architektur ersichtlich ist, besteht das Werkzeug aus 6 verschiedenen Komponenten, die untereinander durch Ereignisse bidirektional kommunizieren. Die Java GUI-Komponenten nehmen eine Sonderstellung ein, da es sich um gewöhnliche Komponenten für die Benutzungsoberfläche (Fenster, Buttons, Menüs, usw.) handelt, die durch die Java Standardbibliothek bereitgestellt werden. Alle anderen Komponenten sind speziell für das FIT-Werkzeug entworfene und implementierte Komponenten, die jeweils für einen bestimmten Aufgabenbereich zuständig sind. In den folgenden Abschnitten werden diese Komponenten und ihre Aufgaben genauer beschrieben.

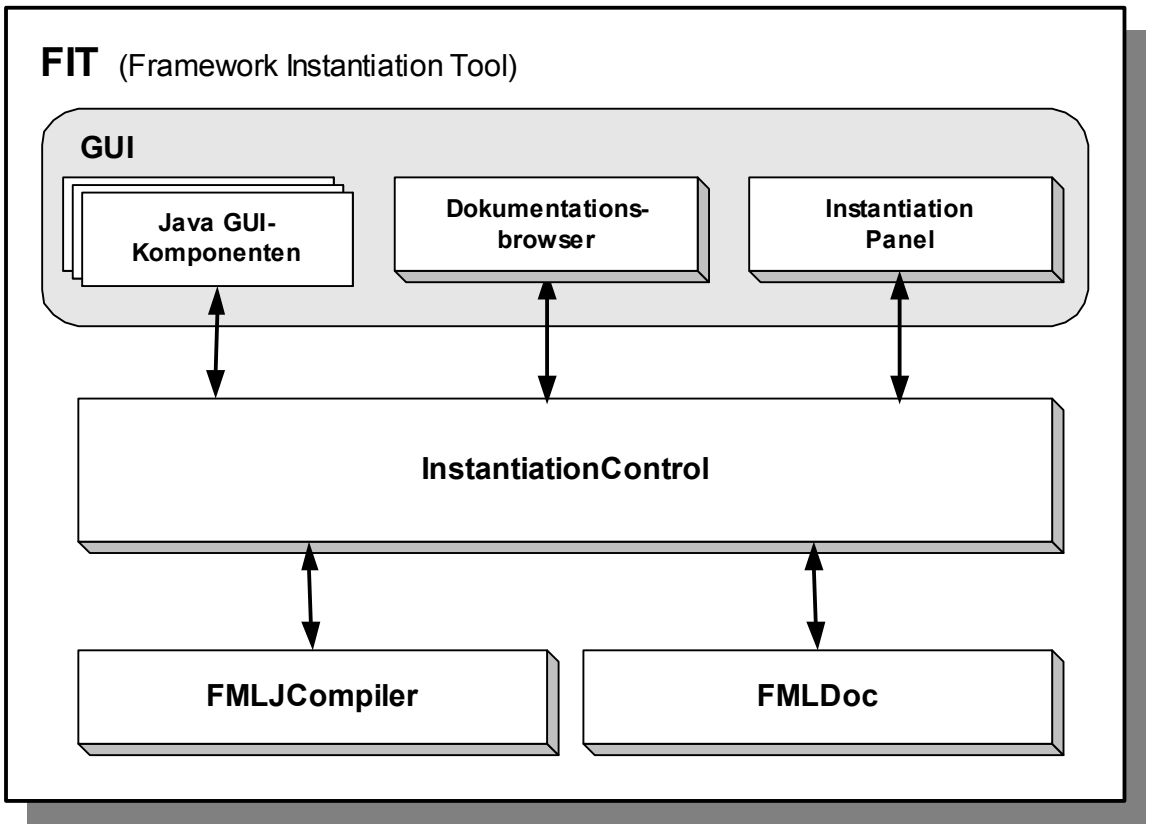

Abbildung 7.2: FIT-Softwarearchitektur

7.2.2 FIT-Komponenten

Dokumentationsbrowser

Eine wesentliche Anforderung für das Instanziierungswerkzeug war die Kopplung von vorhandener Framework-Dokumentation mit dem eigentlichen Instanziierungsvorgang. Der Anwendungsentwickler soll die benötigte Dokumentationstexte zu den einzelnen Instanziierungsschritten direkt innerhalb des Werkzeuges erhalten. Das Werkzeug soll dabei automatisch jeden Instanziierungsschritt mit der dazu verfügbaren Information in der Gesamtdokumentation synchronisieren.

Um die von FML ermöglichte Kopplung der Frameworkbeschreibung mit textueller Dokumentation auch in FIT zu unterstützen, wurde ein kompletter Dokumentationsbrowser in die grafischen Benutzungsoberfläche des Werkzeuges integriert. Dieser Dokumentationsbrowser wird durch die visuelle Komponente *Browser* repräsentiert. Der Browser besitzt den Funktionsumfang eines einfachen WWW-Browsers und kann sowohl HTML- als auch einfache Textdateien darstellen. Durch die Unterstützung von HTML können attraktive Dokumentations- und Hilfstexte geschaffen werden, die den Anwendungsentwickler nicht nur inhaltlich, sondern auch optisch ansprechen. Dabei spielt es keine Rolle, ob die Dokumentation im lokalen Dateisystem oder auf einem zentralen HTTP-Server abgelegt ist.

InstantiationPanel

Die schrittweise Instanziierung des Frameworks soll dem Anwendungsentwickler die Gelegenheit geben, in kleinen und überschaubaren Schritten zu seiner speziellen Anwendung zu gelangen. Die Framework-Instanziierung soll für den Benutzer ein iterativer Dialog mit dem Werkzeug sein, bei dem einfache Fragen beantwortet werden.

In der Benutzungsoberfläche von FIT ist die Komponente *InstantiationPanel* für die Umsetzung dieser Forderung zuständig. InstantiationPanel ist eine visuelle Komponente, die zur Interaktion mit dem Anwendungsentwickler dient. Sie präsentiert dem Benutzer einerseits Fragen und Informationen und nimmt andererseits seine Eingaben und Entscheidungen entgegen. Um die Darstellung der Fragen möglichst optimal an den Inhalt der Frage anzupassen, stehen unterschiedliche Formular- und Meldungsarten zur Verfügung:

- *Informationsmeldung*: Anzeige von Text ohne Eingabemöglichkeit
- *Textformular*: Anzeige einer Frage mit Eingabezeile für Zeichenketten und Zahlenwerte
- *Auswahlformular*: Anzeige einer Frage mit vorgegebenen Auswahlmöglichkeiten (siehe Beispiel in Abbildung 7.3)
- *Optionsformular*: Anzeige einer Frage mit der Eingabemöglichkeit, eine Option zu selektieren
- *Datei-Formular*: Anzeige einer Frage mit der Möglichkeit, in der Eingabezeile einen Dateinamen einzugeben. Ein zusätzlicher Button öffnet einen Dialog zur Auswahl einer Datei, deren Name anschließend direkt in die Eingabezeile übernommen wird.
- *Pfad-Formular*: Anzeige einer Frage und Eingabezeile für einen Dateipfad. Ein zusätzlicher Button öffnet einen Dialog zur Auswahl eines Ordners, dessen vollständiger Pfad in die Eingabezeile übernommen wird.

Abbildung 7.3 zeigt die Komponente InstantiationPanel mit einem Auswahlformular für verschiedene Komponentenalternativen. Außer der Instanziierungsfrage und dem Auswahlformular sind weitere wichtige Bedienelemente der Komponente sichtbar. Die Dokumentation-Buttons dienen zur Anforderung von zusätzlicher Dokumentation, die im Fenster der Browser-Komponente angezeigt wird. Der untere Button präsentiert die Dokumentationstexte, die zum aktuellen Instanziierungsschritt verfügbar sind, während der obere Button zusätzliche Informationen zu den einzelnen Komponentenalternativen bereitstellt. Mit den Navigation-Buttons (vor, zurück) kann der Anwendungsentwickler zu vorangehenden Instanziierungsschritten zurückspringen oder die aktuelle Frage unbeantwortet überspringen, um sich die nachfolgenden Fragen anzuschauen. Der OK-Button übernimmt die aktuelle Antwort des Entwicklers und fährt mit der nächsten Frage fort. Während der Benutzer sich zwischen den

einzelnen Instanziierungsschritten vor- und zurückbewegt, achtet FIT automatisch darauf, dass keine Inkonsistenzen auftreten oder inhärente Abhängigkeiten unberücksichtigt bleiben.

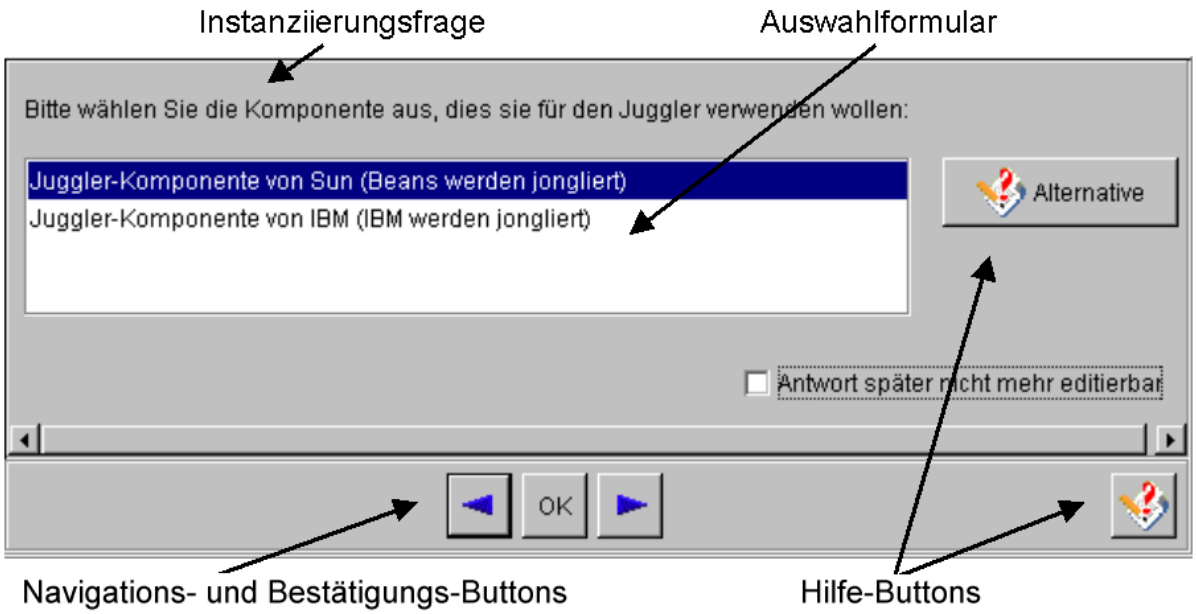

Abbildung 7.3: Komponente InstantiationPanel mit Auswahlformular

FMLJCompiler

Eine zentrale Anforderung an das Instanziierungswerkzeug ist die intuitive Anwendbarkeit für den Entwickler. Er soll möglichst ohne Programmierkenntnisse zu einer vollständigen Anwendung gelangen. Sämtliche Framework- und Anwendungsbeschreibungen liegen aber als Programmiersprachen-unabhängige FML-Spezifikationen vor, was eine direkte Ausführung verhindert. Aus diesem Grund wird eine Komponente benötigt, welche die deskriptiven FML-Beschreibungen in ausführbaren Code einer Programmiersprache übersetzt. Diese Aufgabe wird für die Programmiersprache Java und das JavaBeans Komponentenmodell von der Komponente *FMLJCompiler* übernommen.

Die Komponente FMLJCompiler erzeugt aus der Frameworkbeschreibung und den benutzerspezifischen Parametrierungen (Anwendungsbeschreibung) den notwendigen Code, der die einzelnen Framework-Bausteinen in der Bibliothek zu einer ausführbaren Anwendung vervollständigt. Der Compiler kommt deshalb erst zum Einsatz, wenn der Anwendungsentwickler die in der Frameworkbeschreibung vorhandenen Variabilitäten durch konkrete und konsistente Entscheidungen ersetzt hat und somit seine Anwendung vollständig parametriert hat. Der Generierungsvorgang wird gestartet, indem der Compiler die

Frameworkbeschreibung und die benutzerspezifischen Parametrierungen als Baumstruktur (sog. DOM: Document Object Model[7]) entgegen nimmt. Der Compiler arbeitet sich rekursiv von der Baumwurzel bis in alle Zweige durch und ersetzt dabei alle Variabilitäten durch konkrete Parametrierungsdaten aus der Anwendungsbeschreibung des Entwicklers. Die entstehenden Informationen werden nicht direkt als Quellcode ausgegeben, sondern zunächst in einen linearen Puffer geschrieben. Auf diese Weise können Abhängigkeiten zwischen Zweig-Information berücksichtigt werden, die durch das rekursive Vorgehen nicht unmittelbar erfasst werden. Der letzte Schritt des Übersetzungsvorgangs besteht in der Generierung von Quellcode aus den Informationen im Puffer. Falls gewünscht, kann der FMLJCompiler auch die Übersetzung des Java-Quellcodes und die Ausführung der resultierenden Anwendung veranlassen.

Im Gegensatz zu den bisher eingeführten FIT-Komponenten ist FMLJCompiler keine visuelle Komponente und besitzt deshalb auch keine grafische Benutzungsoberfläche. Der Compiler ist als eigenständige Komponente realisiert, die über Ereignisse mit anderen FIT-Komponenten kommuniziert. Durch die definierte Schnittstelle zwischen dem Compiler und den restlichen Komponenten können verhältnismäßig einfach zusätzliche Programmiersprachen und Komponentenmodelle unterstützt werden – FMLJCompiler muss lediglich gegen eine alternative Compiler-Komponente ausgetauscht werden, welche die benötigte Unterstützung realisiert. Eine Unterstützung für zusätzliche Sprachen und Modelle wurde zwar in der prototypischen Realisierung des FIT-Werkzeuges nicht vorgenommen, stellt aber eine sehr interessante Möglichkeit für zukünftige Erweiterungen dar.

FMLDoc

Im Verlauf des Instanziierungsprozesses trifft der Anwendungsentwickler verschiedene Entscheidungen, welche die Variabilitäten der Frameworkbeschreibung durch benutzerspezifische Parametrierungen ersetzen. Am Ende wird aus diesen Daten eine ablauffähige Anwendung erzeugt, welche die Anforderungen des Anwenders widerspiegelt. Die einzelnen Entscheidungen in Form von Parameterwerten, die zur individuellen Anwendung geführt haben, sind jedoch für den Benutzer nicht mehr offen verfügbar. Trotzdem kann es wünschenswert oder sogar notwendig sein, die zur Anwendung führende Entscheidungskette zu dokumentieren und der Anwendung hinzuzufügen. Eine solche Situation könnte beispielsweise auftreten, wenn bestehende Anwendungen gewartet oder modifiziert werden müssen, oder wenn neue Anwendungsbenutzer die Entscheidungen des Anwendungsentwicklers nachvollziehen müssen.

Die Komponente *FMLDoc* bietet eine Antwort auf diese Problematik, da sie die explizite Dokumentation sämtlicher Entscheidungen des Anwendungsentwicklers ermöglicht. FMLDoc erstellt aus den benutzerspezifischen Parametrierungen eine Projektdokumentation im HTML-

[7] Darstellung hierarchischer XML-Baumstrukturen im Hauptspeicher

Format, welche exakt die Einstellungen des Anwendungsentwicklers während der Instanziierung dokumentiert. Mit Hilfe dieser HTML-Dokumentation kann jeder Außenstehende den Entstehungsprozess einer existierenden Anwendung exakt nachvollziehen und gegebenenfalls auch selbst wiederholen (siehe Abbildung 7.4). Auf diese Weise wird durch FIT nicht nur der eigentliche Instanziierungsprozess, sondern auch die Dokumentation des Prozesses unterstützt.

Abbildung 7.4: Projektdokumentation mit FMLDoc

InstantiationControl

Jede der bisher beschriebenen FIT-Komponenten erfüllt eine dedizierte Aufgabe, die für die Gesamtfunktionalität von FIT bedeutend ist. Dabei konzentrieren sich die bisherigen Komponenten streng auf ihre Teilaufgabe innerhalb von FIT und kümmern sich nicht um globale Zusammenhänge, die außerhalb ihres Aufgabenbereiches wichtig sind. Sie besitzen weder Informationen, noch machen sie Annahmen über die strukturellen Beziehungen des Gesamtwerkzeuges, was sie modular und dadurch besonders einfach austauschbar macht. Damit diese Einzelkomponenten eine funktionsfähige Anwendung bilden, ist eine koordinierende FIT-Komponenten notwendig, welche den Ablauf des gesamten Instanziierungsprozesses und die Kommunikation zwischen den Komponenten steuert. Diese zentrale Aufgabe wird von der Komponente *InstantiationControl* übernommen.

Die Komponente InstantiationControl bestimmt den Ablauf des gesamten Instanziierungsprozesses und ist somit das Herzstück des FIT-Werkzeuges. Sie unterhält Kommunikationsverbindungen zu allen anderen Komponenten (siehe Abbildung 7.2), d.h. sie nimmt Anforderungen von anderen Komponenten entgegen, bearbeitet oder delegiert diese Anforderungen und leitet Ergebnisse und Meldungen an andere Komponenten weiter. Wenn beispielsweise der Anwendungsentwickler zu einem Instanziierungsschritt weitere Informationen anfordern will, so klickt er auf den Hilfe-Knopf der Komponente InstantiationPanel. Diese Anforderung wird zunächst an die zentrale Komponente InstantiationControl weitergeleitet und von dort mit der Angabe der entsprechenden Dokumentationsseite an die Browser-Komponente geschickt, die letztendlich die Anzeige der gewünschten Informationen vornimmt. Auf diese Weise wird ein organisatorischer Zusammenhalt geschaffen, in den die anderen Komponenten mit ihrer Funktionalität eingebunden werden, ohne selbst strukturelle Informationen zu besitzen.

InstantiationControl ist jedoch nicht nur für die Koordinierung zwischen den Komponenten zuständig, sondern übernimmt auch selbst eine Reihe von wichtigen Aufgaben für das Gesamtwerkzeug. Zu diesen Aufgaben gehört beispielsweise das Öffnen und Parsen von Frameworkbeschreibungen oder das Speichern und Verarbeiten von Benutzereingaben, die während des Instanziierungsprozesses gemacht werden. Weitere Aufgaben der Komponente sind die Steuerung des Compilers und des Projekt-Dokumentationsgenerators, sowie die Verarbeitung von Fehlern und Warnungen. Die Gesamtheit dieser Aufgaben macht die Komponente InstantiationControl zur zentralen FIT-Komponente, die das Rückgrat des gesamten Instanziierungswerkzeuges darstellt. Im Gegensatz zu anderen Komponenten kann InstantiationControl nicht einfach gegen andere Komponenten ausgetauscht werden, da sie die Ablauflogik von FIT und die Verarbeitung von FML-Projekten kapselt.

7.3 Vergleich mit ähnlichen Werkzeugen

7.3.1 Generatoren

Ein *Generator* ist ein Software-Werkzeug, welches aus Spezifikationen Quellcode oder vollständige Anwendungen erzeugt. Die Vorgehensweise ähnelt eines Compilers [ASU86] bis auf den Unterschied, dass Generatoren für spezialisierte Anwendungsbereiche eingesetzt werden und deshalb mit spezifischen textuellen oder grafischen Beschreibungssprachen zusammen arbeiten. Bekannte Beispiele für Generatoren sind Lex und Yacc im Compilerbau [Appe98], StateMate zur Spezifikation reaktiver Systeme [HLN+90] und interaktive Werkzeuge zur Erstellung grafischer Benutzungsoberflächen [VaBa97].

Als größter Vorteil von Generatoren ist die einfache Erzeugung von benutzerspezifischem Quellcode zu nennen, die es sogar Nicht-Programmierern mit entsprechenden Domänen-Kenntnissen möglich macht, am Entwurfsprozess einer neuen Anwendung teilzuhaben. Dieser Vorteil wird jedoch dadurch erkauft, dass die Realisierung von Generatoren im Allgemeinen schwierig ist und nur für definierte und überschaubare Anwendungsbereiche möglich ist.

Vergleicht man FIT mit den genannten Generatoren, so kann man feststellen, dass FIT mit der Komponente FMLJCompiler einen vollständigen Generator enthält. FMLJCompiler generiert aus deskriptiven Anwendungsspezifikationen Java-Quellcode, der mit jeder Java-Entwicklungsumgebung weiter verarbeitet werden kann. Der Schwerpunkt von FIT liegt allerdings auf der Unterstützung des Anwenders bei der Erstellung von Anwendungs-spezifikationen, was von den klassischen Generatoren entweder überhaupt nicht oder nur durch eine spezielle Spezifikationsumgebung unterstützt wird. FIT nutzt die Vorteile der Generator-Technologie, geht aber mit seiner Anwenderunterstützung weit über die Möglichkeiten von Generatoren hinaus.

7.3.2 Assistenten und Wizards

Ein Assistent – oft auch *Wizard* genannt – ist ein Softwarewerkzeug, welches den Benutzer bei der Erstellung komplexer Software-Produkte unterstützt. Bekannte Beispiele sind die Wizards bekannter Frameworks und Klassenbibliotheken (z.B. MFC AppWizard der Microsoft Entwicklungsumgebung Visual C++®, siehe Abbildung 3.2) oder Assistenten zur Erstellung von Dokumenten (z.B. MS-Word®-Dokumente oder MS-Powerpoint®-Präsentationen). Diese Softwarewerkzeuge fragen den Anwender einige Einstellungsoptionen ab und erstellen daraus ein schematisches Ausgangsprodukt, das vom Anwender weiter bearbeitet werden muss.

Prinzipiell teilt FIT sehr viele Merkmale mit diesen Software-Assistenten. Sowohl FIT als auch Assistenten versuchen die Komplexität der jeweiligen Produkte vor dem Anwender zu verbergen und die Entwicklung dieser Produkte effizienter zu gestalten. Im Gegensatz zu Assistenten hört FIT jedoch nicht bei einem initialen Ausgangsprodukt auf, sondern begleitet den Anwender bis zum fertigen Endprodukt. Auf diese Weise muss sich der FIT-Anwender zu keinem Zeitpunkt mit den Implementierungsdetails des geplanten Produktes auseinander setzen. Ein weiterer wesentlicher Unterschied ist die Offenheit von FIT bezüglich verschiedener Anwendungsbereiche und Produkte. Assistenten sind hingegen fest an ein bestimmtes Produkt gebunden und können nicht für andere Produkte oder sogar Anwendungsbereiche eingesetzt werden.

In diesem Kapitel wurde das Werkzeug FIT vorgestellt, das dem Anwendungsentwickler die komfortable Arbeit mit Komponenten-Frameworks ermöglicht. Der Enwickler muss sich somit weder mit dem Quellcode der Komponenten noch mit der FML-Frameworkbeschreibung auseinandersetzen. Er arbeitet normalerweise ausschließlich mit der grafischen Benutzungsoberfläche von FIT, die alle unnötigen Details vom ihm abschirmt. Im nächsten Kapitel wird an einem Fallbeispiel aus der Automatisierungstechnik gezeigt, wie dieser Einsatz von FIT und FML in der Praxis aussieht. Dazu wird der Anwendungsbereich der Aufzugsteuerungen gewählt, der sowohl durch domänentypische Gemeinsamkeiten, aber auch viele Variablitäten im Detail gekennzeichnet ist.

8 Anwendungsbeispiel: Aufzugsteuerungen

Um das in den vorangegangenen Kapiteln vorgestellte Verfahren der Framework-Spezifikation mit FML und der Instanziierung mit FIT an einem realen Anwendungsbeispiel der Automatisierungstechnik zu erproben, wurde die Domäne der Aufzugsysteme gewählt. Es wurde ein Komponenten-Framework für die Steuerungssoftware von Aufzugsystemen entwickelt und die resultierenden Steuerungsanwendungen an einem realen Aufzugmodell getestet, das am Institut für Automatisierungs- und Softwaretechnik (IAS) zur Verfügung steht.

Im folgenden Abschnitt wird anhand des verwendeten Aufzugmodells zunächst die Domäne der Aufzugsysteme eingeführt und typische Variabilitäten diskutiert. Danach wird die Realisierung des Komponenten-Frameworks für Aufzugsysteme erläutert. Abschließend werden der Instanziierungsprozess aus der Sicht des Anwendungsentwicklers und die mit dem Framework gesammelten Erfahrungen geschildert.

8.1 Anwendungsbereich Aufzugsysteme

8.1.1 Hardwaremodell

Das Aufzugmodell besitzt zwei Schächte, die gemeinsam vier Stockwerke bedienen. An jedem Einstiegspunkt sind für die möglichen Fahrtrichtungen Tasten angebracht, um den Aufzug anzufordern. Diese Tasten können außerdem beleuchtet werden, um die Anforderung zu quittieren. Zusätzlich besitzt jedes Stockwerk für jeden Fahrkorb Signale, welche die aktuelle Fahrtrichtung des Fahrkorbs anzeigen. Im obersten und untersten Stockwerk stehen natürlich nur die Anforderungstasten und Kontrollsignale zur Verfügung, in deren Richtung eine Weiterfahrt möglich ist. Jede Einstiegsmöglichkeit in den Aufzug ist durch eine Tür gesichert, deren Zustand durch einen Sicherheitsschalter überwacht wird. Die Bedienelemente, die normalerweise in den Fahrkörben vorhanden sind, wurden zur Vereinfachung der Bedienung an der Frontseite des Aufzugmodells angebracht. Dabei steht für jedes Stockwerk eine Anforderungstaste zur Verfügung, die den Fahrtwunsch durch Beleuchtung quittiert. Weiterhin ist ein Not-Aus-Schalter vorhanden, um den Fahrbetrieb im Notfall sofort anhalten zu können.

Abbildung 8.1 zeigt die Frontseite des beschriebenen Aufzugmodells. Die aktuelle Position und Bewegung des Fahrkorbs wird mit Hilfe von vier Sensoren je Haltestelle erfasst, die innerhalb des Schachts angebracht sind. Zwei Sensoren sind dabei jeder Fahrtrichtung zugeordnet - der Erste ist für das rechtzeitige Abbremsen des Fahrkorbs zuständig, falls im betreffenden Stockwerk angehalten werden soll, und der Zweite signalisiert die exakte Bündigposition, an welcher der Fahrkorb angehalten werden muss.

Abbildung 8.1: Frontseite des Aufzugmodells

Die Überwachung und Steuerung der gesamten Aufzugshardware wird von einem PC übernommen, auf dem die frameworkbasierte Steuerungsanwendung ausgeführt wird. Der PC ist über einen CAN Feldbus (Controller Area Network [Etsc94]) mit der Aufzugshardware verbunden. Da analoge Hardware wie Lampen, Schalter und Motoren nicht direkt an den Feldbus angeschlossen werden kann, wurde diese Verbindung mit Hilfe von SLIO[8]-Modulen verwirklicht. Die Anbindung des PC an den Feldbus wurde mit einer kommerziellen CAN-Einsteckkarte und der dazugehörigen Treiberbibliothek vorgenommen.

8.1.2 Variabilitäten

Im Folgenden werden die Flexibilitätsanforderungen beschrieben, die im Framework als mögliche Variabilitäten (Hot Spots) berücksichtigt sind:

- *Anzahl der Stockwerke und Schächte*: Der offensichtlichste Unterschied verschiedener Aufzugsysteme ist die Anzahl der Stockwerke und die Anzahl der Schächte. Während

[8] Serial Link Input Output

kleinere Wohneinheiten mit ein oder zwei Aufzugschächten auskommen, können Hochhäuser vier und mehr Schächte besitzen, welche die gleichen Stockwerke bedienen. Die Steuerungssoftware sollte mit diesen Unterschieden mühelos zurechtkommen.

- *Steuerungsalgorithmen*: Die Aufzugsteuerung ist abhängig vom erwarteten Vertikalverkehr und muss deshalb an die Art der Gebäudenutzung angepasst werden. Die Aufzugsteuerung eines Wohngebäudes verwendet beispielsweise einen anderen Algorithmus als die Steuerung eines Parkhauses.

- *Einzel- oder Gruppensteuerung*: Wenn mehrere Aufzugschächte nebeneinander bzw. gegenüber angeordnet sind, so können die Aufzüge als Gruppe gesteuert werden (Gruppensteuerung), statt für jeden Aufzug eine eigene Steuerung bereitzustellen. Durch eine Gruppensteuerung wird die Position und Fahrtrichtung mehrerer Fahrkörbe registriert und die Steuerung kann flexibel entscheiden, welcher Fahrkorb den Fahrtwunsch eines Benutzers erfüllt. Eine Gruppensteuerung kann dabei helfen, die Förderkapazität zu erhöhen und die Gesamtleistung des Aufzugsystems zu optimieren.

- *Feldbussysteme*: Ein Aufzugsystem besteht aus vielen einzelnen Hardwarekomponenten, die zur Erfüllung der globalen Aufgabe miteinander kommunizieren müssen. Das Feldbussystem bietet diese Kommunikationsplattform und sorgt gleichzeitig dafür, dass der Installationsaufwand überschaubar bleibt. Außer dem hier eingesetzten CAN-Bus existieren noch weitere Feldbussysteme, die für den Einsatz in Aufzugsystemen geeignet sind.

- *Fernüberwachung*: Die Beobachtung eines Aufzugsystem über das Internet eröffnet nicht nur für den Aufzugshersteller, sondern auch für Service- und Wartungsfirmen interessante Möglichkeiten. Fehlfunktionen können unverzüglich wahrgenommen werden und der Aufzugsmechaniker kann mit der genauen Kenntnis der Ursache zur Fehlerbehebung antreten. Diese Funktionalität kann dabei helfen, die Kosten für den Aufzugsbetreiber zu senken und gleichzeitig den Komfort für die Aufzugsbenutzer zu steigern.

8.2 Entwicklung des Aufzug-Frameworks

8.2.1 Framework-Realisierung

Für die oben beschriebene Domäne der Aufzugsysteme, einschließlich der aufgezählten Variabilitäten, wurde ein Komponenten-Framework verwirklicht, das aus 6 verschiedenen JavaBeans Komponenten (*Schacht, Steuerung, Statusverarbeitung, Fehlerverarbeitung, Ereignisumsetzung* und *Busanbindung* [Bloc99]) und einer umfangreichen Framework-beschreibung besteht. Jede einzelne Komponente verkörpert ein wesentliches Element des Anwendungsbereichs Aufzugsteuerungen. Die geforderten Variabilitäten wurden entweder als

eigenständige Komponenten (Architektur- oder Komponentenebene) oder als Eigenschaftsvariablen einer vorhandenen Komponente (Instanzenebene) realisiert.

Abbildung 8.2 zeigt exemplarisch die Architektur einer Steuerungsanwendung, die aus dem Framework instanziiert wurde und die alle verfügbaren Komponenten des Aufzug-Frameworks verwendet. Die Anwendung repräsentiert eine Steuerung für ein Aufzugsystem mit einem Schacht. Die Abbildung zeigt auch die Kommunikationsbeziehungen zwischen den Instanzen, die durch gerichtete Pfeile ausgedrückt sind. Im Nachfolgenden werden die einzelnen Komponenten des Aufzug-Frameworks, ihre Aufgaben und ihre Beziehungen untereinander näher erläutert.

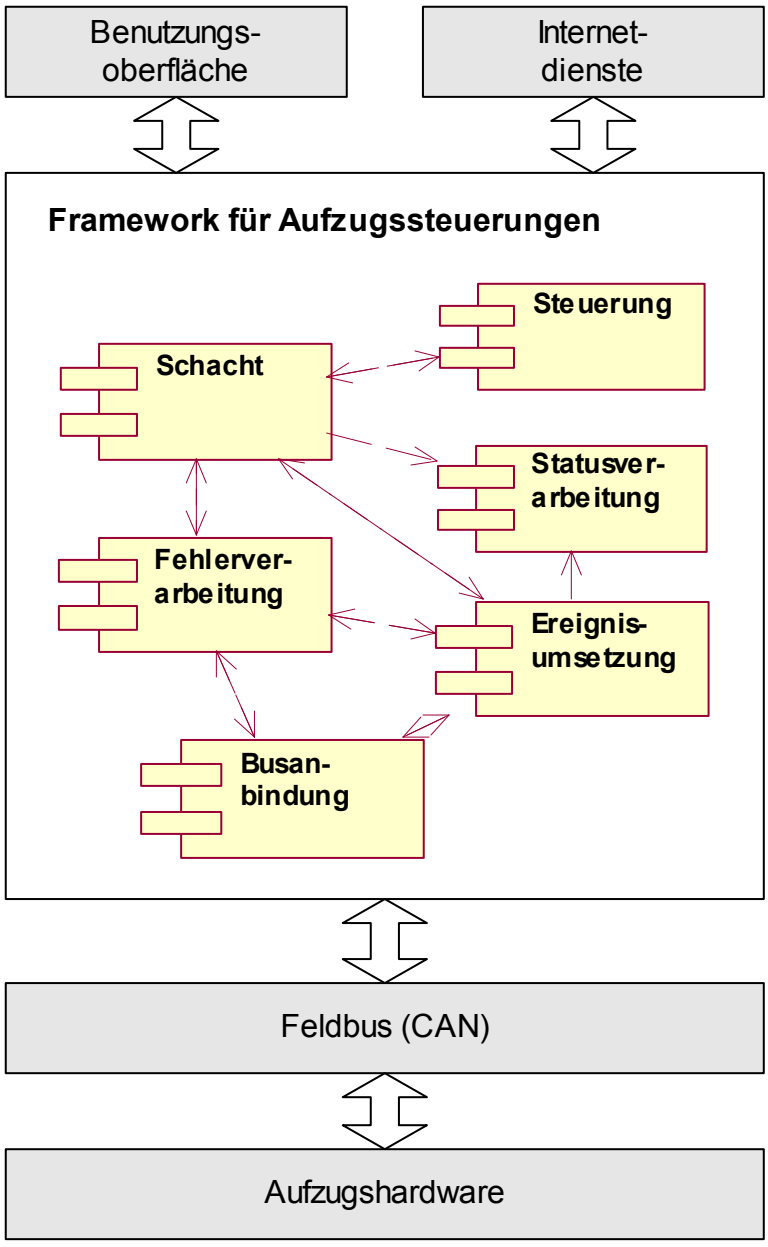

Abbildung 8.2: Architektur einer frameworkbasierten Anwendung

8.2.2 Systemkomponenten und Interaktionsbeziehungen

Komponente Busanbindung

Die Komponente *Busanbindung* ist für die Initialisierung der Feldbusgeräte (z.B. PC Einsteckkarte) und das Abwickeln der Kommunikation über den Bus zuständig. Da im vorliegenden Fall der CAN-Bus zum Einsatz kommt, besitzt die Komponente CAN-spezifische Funktionen für die Initialisierung der CAN-Einsteckkarte und für das Senden bzw. Empfangen von CAN-Nachrichten über den Bus. Falls während der Initialisierung oder im normalen Betrieb Fehler erkannt werden, so schickt die Komponente *Busanbindung* ein spezielles Fehler-Ereignis an die Komponente *Fehlerverarbeitung* und wartet auf die Antwort dieser Komponente. Um andere Feldbussysteme zu unterstützen, muss die Implementierung der Komponente vollständig durch eine andere Busimplementierung (Komponentenalternative) ersetzt werden. Ungeachtet des verwendeten Feldbussystems muss jedoch jede Steuerungsanwendung genau eine Komponente *Busanbindung* enthalten.

Komponente Ereignisumsetzung

Alle CAN-Nachrichten der Komponente *Busanbindung* werden an die Komponente *Ereignisumsetzung* geschickt. Diese Komponente ist in der Lage, die Ereignisse vorzufiltern und an geeignete Empfängerkomponenten weiterzuleiten. *Ereignisumsetzung* spielt eine wichtige Vermittlerrolle, da es die busnahen Binärdaten einer CAN-Nachricht in entsprechende höherwertige Ereignisse übersetzen und auch in umgekehrter Richtung zurück übersetzen kann. Alle Komponenten oberhalb der *Ereignisumsetzung* arbeiten mit höherwertigen Ereignissen, die lediglich die Informationen für die entsprechende Komponente beinhalten. Diese bidirektionale Übersetzung wird unter Zuhilfenahme einer relationalen Datenbank vorgenommen, die allen verfügbaren Sensoren und Aktoren entsprechende CAN-Nachrichten zuordnet. Eine weitere wichtige Funktionalität der Komponente *Ereignisumsetzung* besteht in der Eliminierung von doppelten oder bedeutungslosen Nachrichten, die beispielsweise durch prellende Hardware[9] entstehen. Auf diese Weise wird das Ereignisaufkommen zwischen den Komponenten reduziert und somit unnötige Leistungseinbußen verhindert. Jede Aufzugsanwendung enthält genau eine Komponente *Ereignisumsetzung*.

Komponente Fehlerverarbeitung

Alle Fehlerereignisse, die andere Komponenten detektieren, werden an die Komponente *Fehlerverarbeitung* geschickt. Diese Komponente kapselt das Wissen, wie in bestimmten Fehlerfällen zu reagieren ist. In Abhängigkeit von Fehlertyp und -schwere kann entweder eine einfache Bestätigung durch den Softwarebenutzer den Normalbetrieb wieder herstellen oder bei

---

[9] Beispiel mechanischer Schalter: eine Schalterbetätigung löst nicht eine einzige CAN-Nachricht, sondern eine Sequenz von Nachrichten aus.

besonders schwer wiegenden Systemfehlern ein vollständiger Neustart der Steuerung gefordert werden. *Fehlerverarbeitung* gehört ebenfalls zu den zwingenden Bestandteilen jeder Aufzugsanwendung.

Komponente Statusverarbeitung

Die Komponente *Statusverarbeitung* beobachtet den Ereignisverkehr zwischen den Komponenten und kann jederzeit über den aktuellen Zustand des Aufzugsystems Auskunft geben. Wenn man diese Komponente mit einer beliebigen Java Ausgabeklasse wie z.B. einem Textfeld verbindet, so kann der Benutzer den Systemzustand am Bildschirm beobachten. Für die Anbindung der Aufzugsteuerung an das Internet kann *Statusverarbeitung* auch an entsprechende TCP/IP-Klassen aus der Java Standardbibliothek gekoppelt werden. In Ergänzung zu der bereits genannten Funktionalität besitzt die Komponente auch eine parametrierbare Filterfunktion, mit der für den Benutzer uninteressante Information unterdrückt werden können. Die Komponente *Statusverarbeitung* gehört zur Kategorie der optionalen Komponenten, d.h. nur wenn der Benutzer die entsprechende Funktionalität wünscht, taucht die Komponente in der resultierenden Steuerungsanwendung auf.

Komponente Steuerung

Die Komponente *Steuerung* sammelt alle Fahrtwünsche und entscheidet, welches Stockwerk als Nächstes vom Fahrkorb angesteuert wird. Die zugrunde liegende Entscheidungsstrategie kann durch die Auswahl verschiedener Steuerungsalgorithmen beeinflusst werden. Für das Framework wurden exemplarisch zwei verschiedene Algorithmen implementiert und zur Auswahl gestellt: ein einfacher FIFO-Algorithmus und eine komplexere Gruppensteuerung. Der FIFO-Algorithmus stellt alle Fahrtwünsche sequenziell in eine Warteschlange und arbeitet sie in der gleichen Reihenfolge ab, wobei mit dem ersten Wunsch begonnen wird. Die Gruppensteuerung arbeitet mit mehreren Fahrkörben und versucht immer, einen Fahrtwunsch mit dem nächstgelegenen Fahrkorb zu bearbeiten, der sich bereits in die geforderte Fahrtrichtung bewegt. Auch die Komponente *Steuerung* ist ein zwingender Bestandteil jeder Anwendung, wobei das Verhalten jedoch stark durch entsprechende Parameter beeinflusst werden kann.

Komponente Schacht

Der *Schacht* ist die funktional wichtigste und vom Codeumfang größte Komponente des Frameworks. Sie repräsentiert einen vollständigen Aufzugschacht einschließlich Fahrkorb, allen Anforderungsknöpfen und Signalen innerhalb und außerhalb des Schachts sowie den verschiedenen Stockwerken, die man als Parameter der Komponente einstellen kann. Jeder *Schacht* ist direkt und bidirektional mit der Komponente *Ereignisumsetzung* verbunden: Die Komponente *Schacht* empfängt Nachrichten vom Feldbus (z.B. Anforderungsknopf gedrückt)

und sendet Befehle an die Aktoren (z.B. starte schnelle Motorbewegung nach unten). Auf ähnliche Weise ist der *Schacht* auch mit der Komponente *Steuerung* verbunden, indem er sie über die aktuelle Fahrkorbposition informiert und von ihr stockwerkbezogene Fahrzielbefehle entgegen nimmt. Wie die meisten anderen Komponenten sendet auch der *Schacht* im Fehlerfall entsprechende Ereignisse an die *Fehlerverarbeitung* und wartet auf die Antwort.

Die Komponente *Schacht* spielt auch eine zentrale Rolle in der Startphase der Steuerungsanwendung, da sie andere Komponenten mit wichtigen Initialisierungsdaten versorgt. Zur Vermeidung von Dateninkonsistenzen ist jeder einzelne Parameter innerhalb des Frameworks einmalig und exklusiv einer Komponente zugeordnet. Falls eine Komponente einen bestimmten Parameterwert benötigt und ihn nicht selbst besitzt, so muss sie zu Beginn mit diesem Wert initialisiert werden. Der *Schacht* liest beispielsweise in der Startphase einen Wert für seinen Parameter Stockwerkanzahl ein und teilt diesen danach der Komponente *Steuerung* mit, die keinen Parameter für die Anzahl der Stockwerke besitzt, aber ohne diesen Wert nicht sinnvoll arbeiten kann. Im Gegensatz zu den meisten anderen Komponenten darf die Komponente *Schacht* in Abhängigkeit von den vorhandenen Aufzugschächten auch mehrfach instanziiert werden, wobei die Eigenschaften jeder Instanz individuell parametriert werden können.

8.2.3 Frameworkbeschreibung

Die Frameworkbeschreibung des Aufzug-Frameworks erfüllt zwei wesentliche Aufgaben. Ihre erste Aufgabe besteht in der Spezifikation aller Steuerungsanwendungen, die aus dem Aufzug-Framework instanziiert werden können. Auf diese Weise wird explizit und präzise dokumentiert, welche Elemente bei allen resultierenden Steuerungsanwendungen identisch sind und welche Elemente von Anwendung zu Anwendung variieren können. Die zweite wichtige Aufgabe ist an den eigentlichen Instanziierungsprozess des Aufzug-Frameworks gekoppelt – die Frameworkbeschreibung legt fest, wie die schrittweise Instanziierung innerhalb des Werkzeuges FIT abläuft. Durch die Erfüllung dieser zwei Aufgaben wird sichergestellt, dass der Anwendungsentwickler komfortabel verschiedene individuelle Aufzugsteuerungen erstellen kann, ohne sich um die Details der benötigten Komponenten oder um den Ablauf des Instanziierungsprozesses zu kümmern.

Zur besseren Übersichtlichkeit und Wartbarkeit der Frameworkbeschreibung für Aufzugsteuerungen wurden die FML-Spezifikationen auf mehrere Dateien aufgeteilt. Diese einzelnen Spezifikationsdateien, sowie die benötigten Komponenten werden von der Projektdatei "Aufzug-Projekt.prj" referenziert (siehe Abbildung 8.3).

```
1:    <fml-file author="Du" company="IAS" date="27.09.2001" version="1.1">
2:      <project id="Aufzug-Projekt" main-fw="Aufzug-Steuerung-Framework">
3:        <file-reference>
4:          <path>
5:            <path value="libs\">
6:              <file target="Aufzug.jar"/>
7:            </path>
8:            <file target="Aufzug-Steuerung-Framework.fml"/>
9:            <file target="Aufzug-Sprachunterstuetzung.fml"/>
10:           <file target="Schacht-Alternativen.fml"/>
11:           <file target="Can-Framework.fml"/>
12:           <file target="Schacht-Alt-Control-Interface.fml"/>
13:           <file target="Steuerung-Framework.fml"/>
14:           <file target="Steuerung-Alternativen.fml"/>
15:         </path>
16:       </file-reference>
17:     </project>
18:   </fml-file>
```

Abbildung 8.3: Projektdatei des Aufzug-Frameworks

In Zeile 6 der Projektdatei wird die Bibliotheksdatei "Aufzug.jar" referenziert, welche alle zur Anwendungsentwicklung benötigten Komponenten des Aufzug-Frameworks in binärer Form enthält. Durch diese Referenz wird die Frameworkbeschreibung mit den Implementierungen der Komponenten verbunden, was für die Erstellung von lauffähigen Anwendungen notwendig ist. In den Zeilen 8 bis 14 werden die sieben FML-Spezifikationsdateien aufgelistet, welche die eigentliche Frameworkbeschreibung des Aufzug-Frameworks enthalten. Innerhalb der Projektdatei sind somit keine Spezifikationsdaten enthalten. Es wird lediglich festgelegt, welche Dateien (Komponenten und Frameworkbeschreibung) zum Aufzug-Framework gehören.

8.3 Anwendungsentwicklung mit dem Aufzug-Framework

8.3.1 Instanziierungsprozess

Nachdem im vorangehenden Abschnitt das Komponenten-Framework beschrieben wurde, soll in diesem Abschnitt der Instanziierungsprozess aus der Sicht des Anwendungsentwicklers betrachtet werden. Um neue Steuerungsanwendungen für Aufzüge zu erstellen, arbeitet der Anwendungsentwickler ausschließlich mit der Benutzungsoberfläche des FIT-Werkzeuges (siehe Abbildung 7.1). Er öffnet mit Hilfe von FIT die Projektdatei für das Aufzug-Framework und wird automatisch durch den Instanziierungsprozess bis zur ausführbaren Steuerungsanwendung geleitet. Aus seiner Sicht besteht die Instanziierung aus einer Kette von Entscheidungen, die er in Form von textuellen Eingaben oder Auswahldialogen treffen muss. Dabei werden dem Anwendungsentwickler nur die Informationen und Fragen präsentiert, die für den

aktuellen Fortschritt des Prozesses notwendig sind. Das Werkzeug kümmert sich um alle anderen Aspekte, wie z.B. die Vollständigkeit der Entscheidungen und die Berücksichtigung von Abhängigkeiten zwischen den einzelnen Entscheidungsstufen. Abbildung 8.4 zeigt einen kleinen Ausschnitt aus dem Instanziierungsablauf des Aufzugs-Frameworks.

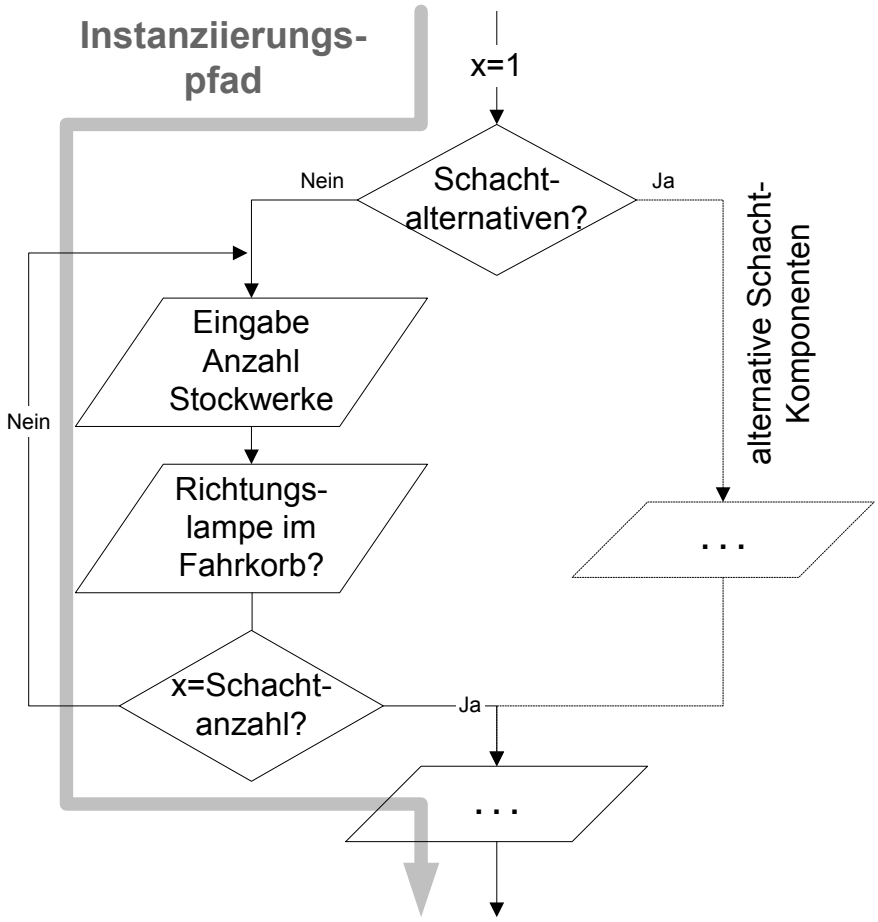

Abbildung 8.4: Ausschnitt des Instanziierungsablaufes

Im gezeigten Ablauf muss der Anwendungsentwickler zunächst wählen, ob er eigene Schacht-Komponenten verwenden möchte. Im Beispiel wird der Standard-Schacht gewählt (linke Seite), wodurch die Instanziierungsschritte im rechten Zweig nicht weiter beachtet werden. Der Benutzer „sieht“ im Werkzeug nur noch den linken Zweig, wo er weitere Entscheidungen treffen muss. Auf diese Weise durchläuft der Anwendungsentwickler immer nur den Pfad in der Frameworkbeschreibung (Instanziierungspfad), der sich durch seine Eingaben und Entscheidungen ergibt. Alle nicht benötigten Informationen und Entscheidungen werden ausgeblendet, wodurch bei der Anwendungsentwicklung eine hohe Flexibilität bei minimalem Aufwand möglich wird.

Während der Instanziierung des Aufzug-Frameworks wird der Anwendungsentwickler auch mit nichtfunktionalen Einschränkungen für die zu entwickelnde Steuerungsanwendung konfrontiert. Eine wichtige Einschränkung ergibt sich aus der Anzahl der gleichzeitig zu steuernden Aufzugschächte. Möchte der Anwendungsentwickler mehr als einen Aufzugschacht steuern, so muss der Prozessor des Steuerungsrechners eine Taktfrequenz von mehr als 500 MHz aufweisen. Unterhalb dieser Leistungsgrenze ist ein rechtzeitiges Halten der Fahrkörbe in den einzelnen Stockwerken nicht mehr sichergestellt. Abbildung 8.5 zeigt die Meldung, mit welcher der Entwickler auf diese nichtfunktionale Einschränkung aufmerksam gemacht wird.

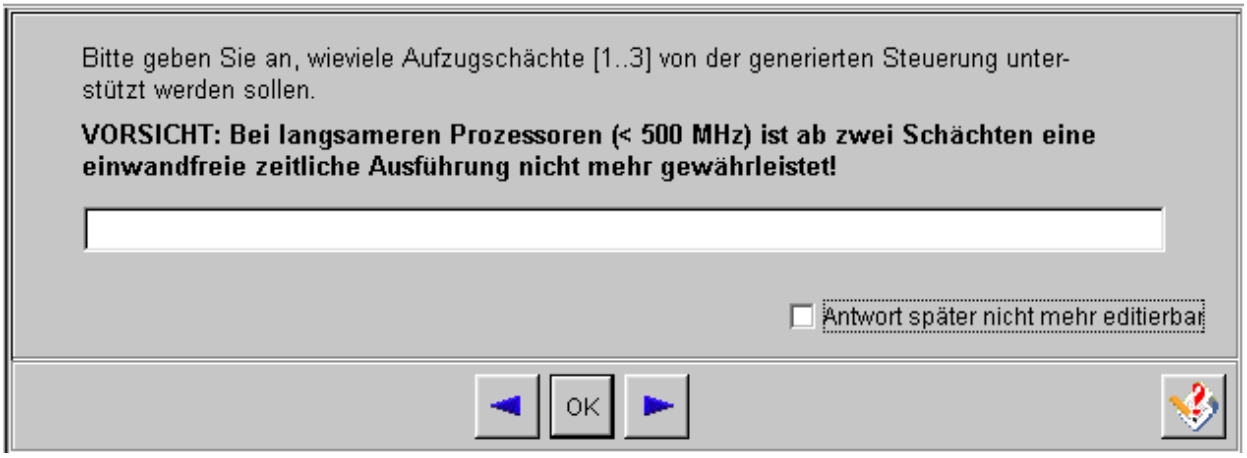

Abbildung 8.5: Nichtfunktionale Einschränkungen für die Steuerungsanwendungen

8.3.2 Instanziierungsprodukte

In Abhängigkeit von den während der Instanziierung getroffenen Entscheidungen können aus dem Aufzug-Framework unterschiedliche Instanziierungsprodukte resultieren. Die Frameworkbeschreibung erlaubt die Generierung von folgenden Produkten:

- *Steuerungskomponente*: Als Instanziierungsprodukt entsteht eine anwendungsspezifische Steuerungskomponente für Aufzugsysteme, die vom Anwendungsentwickler in eine übergeordnete Anwendung eingebettet werden kann. Auf diese Weise können beispielsweise Steuerungskomponenten für Aufzüge bereitgestellt werden, die der Entwickler durch beliebige Benutzungsoberflächen zu lauffähigen Anwendungen vervollständigen kann.

- *Eigenständige Anwendung*: Als Ergebnis der Instanziierung entsteht eine eigenständige Anwendung, die eine vorgegebene Benutzungsoberfläche mit verhältnismäßig wenig Gestaltungsmöglichkeiten besitzt. Im Gegensatz zu der ersten Option der anwendungsspezifischen Steuerungskomponente erhält der Anwendungsentwickler sofort eine lauffähige Steuerungsanwendung, die ohne weitere Entwicklungsschritte einsatzbereit ist. Dafür kann die resultierende Steuerungsanwendung jedoch nicht in übergeordnete Anwendungen eingebettet werden.

Abbildung 8.6 zeigt die Benutzungsoberfläche einer eigenständigen Steuerungsanwendung, die direkt mit dem Werkzeug FIT aus dem Aufzug-Framework erstellt wurde. Da das Aufzugmodell über die Anforderungstasten innerhalb und außerhalb des Fahrkorbs gesteuert wird, ist die Benutzungsoberfläche ausschließlich dazu bestimmt, die Aufzugsteuerung zu starten und den laufenden Betrieb zu überwachen. Im oberen Fenster werden alle Statusmeldungen angezeigt, die in der Betriebsphase des Aufzugs empfangen werden. Dazu gehören beispielsweise die Initialisierung der Steuerung, die verschiedenen Fahrtwünsche der Aufzugsbenutzer und die Positionsmeldungen des Fahrkorbs.

Im unteren Fenster werden alle Fehlermeldungen angezeigt, die während dem Aufzugsbetrieb detektiert werden. Dazu gehören beispielsweise die Meldung defekter Hardware oder das Auftreten eines gefährlichen Zustandes (z.B. Aufzugstür wird geöffnet während der Fahrkorb noch in Bewegung ist). In solchen Fällen werden alle Fahrkorb-Bewegungen gestoppt, eine textuelle Fehlermeldung in der Benutzungsoberfläche ausgegeben und eine entsprechende Fehlerbehebung gefordert. Nach der Beseitigung der Fehlerursache kann durch das Drücken der Taste "Fehler behoben" der normale Fahrbetrieb wieder aufgenommen werden.

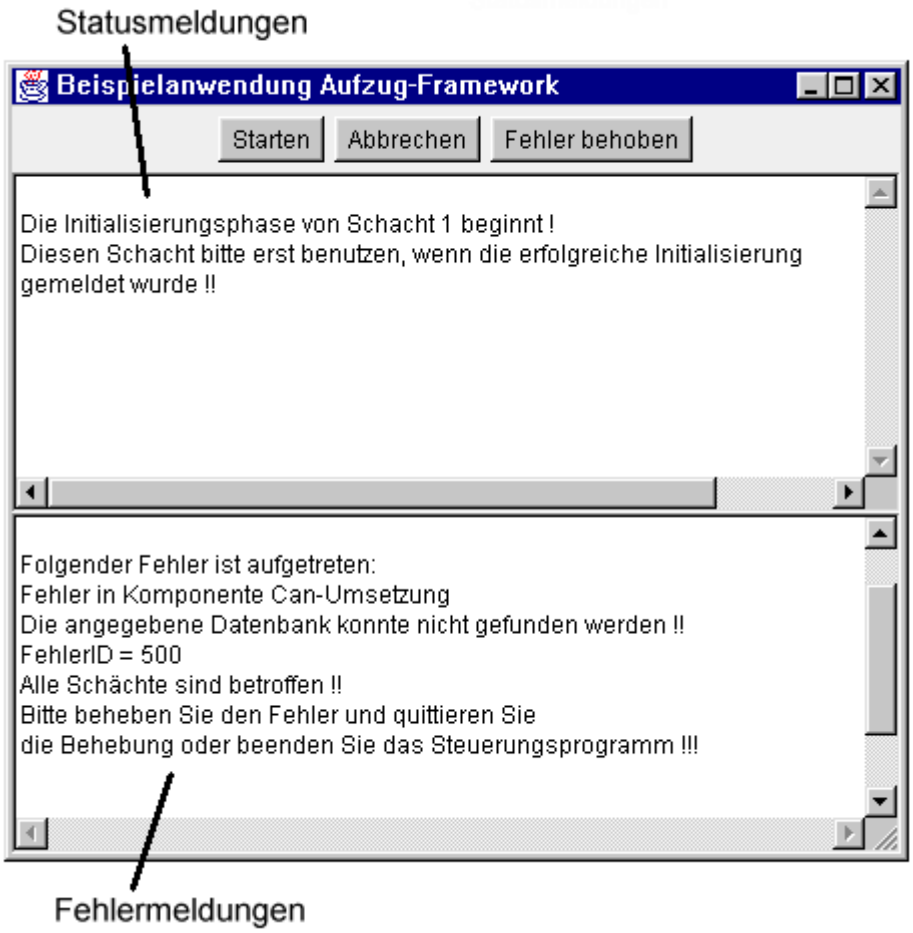

Abbildung 8.6: Benutzungsoberfläche einer frameworkbasierten Steuerungsanwendung

8.4 Ergebnisse und Erfahrungen

Nachdem das Komponenten-Framework für Aufzugsysteme beschrieben wurde und der Instanziierungsprozess aus der Sicht des Anwendungsentwicklers betrachtet wurde, sollen in diesem Abschnitt die Erfahrungen geschildert werden, die mit der Frameworkbeschreibung für Aufzugsysteme gemacht wurden.

Entwicklung des Aufzug-Frameworks

Auf der Implementierungsseite besteht das gesamte Framework aus 65 Klassen oder aus ungefähr 20000 Zeilen Java-Code (einschließlich Kommentaren). Tabelle 8.1 veranschaulicht, wie die Implementierungsklassen innerhalb des Komponenten-Frameworks verwendet werden. Dabei ist besonders interessant, dass von den 65 Klassen nur 17 Klassen für die Realisierung von domänenrelevanten Elementen zuständig sind. Alle anderen 48 Klassen spielen in der Domäne Aufzugsysteme keine Rolle und sind lediglich für die Implementierung von Steuerungsanwendungen notwendig. Die Komponente *Schacht* besitzt die mit Abstand größte Framework-Klasse „‚Schacht.java“’ mit einer Größe von 88 KByte.

Tabelle 8.1: Implementierungsklassen des Komponenten-Frameworks für Aufzugsteuerungen

Anzahl Klassen	Verwendungszweck	Beispiele
5	Elemente der Benutzungsoberfläche	Hauptfenster, Dialoge, Fehler-meldungen
9	Framework-interne Kommunikation	Steuerungsnachricht, Fehlernachricht
17	Domänenrelevante Elemente	Schacht, Steuerung, Fehlerver-arbeitung, Busanbindung
34	Hilfselemente	Exceptionklassen, Datenstrukturen, Ereignis-Listener

Bereits diese Gegenüberstellung von Zahlen macht einen Vorteil von Komponenten-Frameworks deutlich – statt sich mit 65 Implementierungsklassen auseinander zu setzen, muss der Anwendungsentwickler lediglich mit 6 Komponenten arbeiten, die zudem noch werkzeuggestützt zu ausführbaren Anwendungen verbunden werden.

Die Erstellung einer Frameworkbeschreibung in FML bedeutet einen zusätzlichen Aufwand, der bei der Projektplanung für die Framework-Entwicklung berücksichtigt werden muss. Bei dem vorgestellten Framework für Aufzugsteuerungen betrug dieser zusätzliche Arbeitsaufwand 2 Personenwochen, wobei die Hälfte dieses Aufwands nicht auf die Erstellung der eigentlichen FML-Beschreibungen, sondern auf die integrierte textuelle Dokumentation entfiel. Berücksichtigt man, dass diese Dokumentation auch ohne Frameworkbeschreibung notwendig ist und stellt man diesen zusätzlichen Spezifikationsaufwand dem gesamten Entwicklungs-

aufwand für das Aufzugs-Framework von 5 Personenmonaten gegenüber, so ergibt die Frameworkbeschreibung einen moderaten und überschaubaren Mehraufwand.

Eine der größten praktischen Schwierigkeiten bei der Erstellung der Frameworkbeschreibung war die korrekte Referenzierung der verschiedenen Beschreibungsteile. Diese Referenzierung erfolgt mit Hilfe von IDs, die innerhalb eines FML-Projektes eindeutig sein müssen. Bei der Erstellung von FML-Spezifikationen mit gewöhnlichen Text- oder XML-Editoren sind die meisten Fehler auf falsche IDs zurückzuführen. Eine automatische Verwaltung dieser IDs in einem speziellen FML-Editor würde den Frameworkentwickler deutlich entlasten und den Erstellungsaufwand für FML-Beschreibungen senken.

Anwendungsentwicklung mit dem Aufzug-Framework

Im Gegensatz zum Aufwand, den der Frameworkentwickler zur Beschreibung des Frameworks zusätzlich erbringen muss, stehen die Vorteile, die der Anwender eines solchen Frameworks hat. Auf der Anwenderseite reduziert sich sowohl der Einarbeitungsaufwand, als auch der eigentliche Instanziierungsaufwand drastisch. Die Instanziierung des Frameworks für Aufzugsteuerungen mit Hilfe des FIT-Werkzeuges dauert sogar bei einem FIT-unerfahrenen Anwender mit grundlegenden Kenntnissen des Anwendungsbereichs nicht länger als zwei Stunden. Bei einer Vertrautheit mit dem Werkzeug und dem Aufzugs-Framework können benutzerspezifische Anwendungen natürlich noch schneller erstellt werden. Hinzu kommt, dass der Anwendungsentwickler zu keinem Zeitpunkt der Instanziierung mit Quellcode oder Implementierungsdetails in Berührung kommt. Ähnliche Erfahrungen wurden auch in anderen Projekten gemacht, in denen das vorgeschlagene Verfahren eingesetzt wurde [Benh01].

In diesem Kapitel wurde am Beispiel eines Komponenten-Frameworks für die Steuerung von Aufzugsystemen gezeigt, wie die Arbeit mit der Spezifikationssprache FML und dem Instanziierungswerkzeug FIT in der Praxis aussieht. Dabei wurde noch einmal deutlich, dass der Einsatz des in dieser Arbeit vorgeschlagenen Konzeptes wesentliche Vorteile für den Anwendungsentwickler im Vergleich zur konventionellen Vorgehensweise bietet. Im nächsten Kapitel werden abschließend das vorgeschlagene Verfahren bewertet, die Grenzen des Verfahrens aufgezeigt und ein Ausblick auf mögliche Fortsetzungen der Arbeit gegeben.

9 Bewertung und Ausblick

9.1 Bewertung des vorgestellten Verfahrens

In der vorliegenden Arbeit wurde ein neues Konzept vorgestellt, das zur Unterstützung der Wiederverwendung in der Automatisierungstechnik dient und die einfache und effiziente frameworkbasierte Anwendungsentwicklung ermöglicht. Das Konzept stützt sich dabei auf zwei wesentliche Eckpfeiler:

- *Komponenten-Frameworks* als neue Framework-Art, welche sich durch die explizite Beschreibung von Architekturen und Variabilitäten innerhalb der Grenzen eines definierten Anwendungsbereichs auszeichnet und einer umfassenden
- *Werkzeugunterstützung für die Framework-Instanziierung*, die sich nach den Bedürfnissen des Domänenexperten und nicht nach den Kenntnissen des Programmierexperten richtet.

An zwei größeren Beispielen aus unterschiedlichen Bereichen der Automatisierungstechnik (Markiersysteme und Aufzugsysteme) und an kleineren, experimentellen Beispielen wurde das vorgeschlagene Verfahren in der Praxis evaluiert.

Anwendbarkeit für den industriellen Anwender

Als markantester Vorteil des in dieser Arbeit vorgeschlagenen Instanziierungsverfahrens ist die einfache und intuitive Anwendungsentwicklung mit Komponenten-Frameworks hervorzuheben. Mit Komponenten-Frameworks und dem Werkzeug FIT kann eine frameworkbasierte Anwendungsentwicklung verwirklicht werden, die sich weitgehend an den funktionalen Anforderungen des Anwendungsbereichs und nicht an den eingesetzten Implementierungstechnologien orientiert. Dadurch werden Frameworks auch für Personengruppen zugänglich, die bisher an den geforderten Programmierkenntnissen und dem hohen Einarbeitungsaufwand gescheitert sind.

Ein weiterer wichtiger Vorteil des Verfahrens ist die Kopplung von Werkzeugunterstützung und Dokumentation in einer Benutzungsoberfläche. Auf diese Weise ist eine selektive Präsentation von Dokumentation möglich, die zum aktuellen Instanziierungsschritt benötigt wird. Dadurch entfällt die aufwändige Suche in Handbüchern und Hilfe-Dateien, die den Anwendungsentwickler auf Umwegen zu der gewünschten Information führt.

Verträglichkeit mit bestehenden Technologien

Das vorgeschlagene Verfahren wurde anhand der Programmiersprache Java und des JavaBeans Komponentenmodells evaluiert. Dabei wurde jedoch darauf geachtet, dass die zugrunde liegenden Konzepte nicht von den verwendeten Technologien abhängig sind. Sowohl die Spezifikationssprache FML als auch das Instanziierungswerkzeug FIT könnte auch mit anderen Programmiersprachen und Komponentenmodellen eingesetzt werden. Bei FIT müsste lediglich der FMLCompiler angepasst werden, der aus den abstrakten Spezifikationsdaten konkrete Anwendungen erzeugen kann. Durch diese Flexibilität kann das Verfahren sehr gut in bestehende Entwicklungsprozesse integriert werden, ohne die Abkehr von bestehenden Technologien vorauszusetzen.

Unterstützung nichtfunktionaler Anforderungen

Ein weiterer Vorteil ist die Unterstützung von nichtfunktionalen Anforderungen, die sich aus Randbedingungen der Umgebung ergeben und besonders in der Automatisierungstechnik eine herausragende Rolle spielen. Der Frameworkentwickler kann mit seinen detaillierten Framework-Kenntnissen festlegen, welche Voraussetzungen bei der Anwendungsentwicklung eingehalten werden müssen, damit bestimmte nichtfunktionale Eigenschaften (z.B. Laufzeit) zugesichert werden können. Der Anwendungsentwickler wird zum Zeitpunkt der Instanziierung mit diesen Kombinationen aus Voraussetzungen und Zusicherungen konfrontiert und kann entscheiden, ob er die zugesicherten Eigenschaften benötigt oder darauf verzichten kann.

9.2 Grenzen des Verfahrens

Das vorgeschlagene Verfahren setzt voraus, dass für den gewünschten Anwendungsbereich ein Komponenten-Framework vorliegt oder zumindest eine klare Aufteilung in Funktionalität (Komponenten) und Architektur (Frameworkbeschreibung) möglich ist. Viele der heute in der Automatisierungstechnik eingesetzten Frameworks basieren auf dem objektorientierten Paradigma, wodurch eine Transformation zu Komponenten-Frameworks notwendig wird. Erst nach einer Transformation von einem objektorientierten zu einem Komponenten-Framework kann das vorgestellte Verfahren eingesetzt werden. Bei einem sauberen, objektorientierten Framework-Entwurf und bei entsprechenden Kenntnissen des Anwendungsbereichs stellt diese Transformation lediglich einen weiteren Optimierungsschritt dar [Benh01, RoJo96].

9.3 Ausblick

Im Verlauf der Arbeit und auf der Grundlage der gewonnenen Erfahrungen wurden einige Punkte identifiziert, die Ansätze für eine Fortsetzung der Arbeit bieten. In diesem Abschnitt sollen die wichtigsten Punkte vorgestellt und erläutert werden. Als besonders interessant wurden die folgenden drei Punkte erachtet:

- *Grafischer Editor für FML Beschreibungsdateien*: In dieser Arbeit wurden sämtliche FML-Spezifikationen mit einfachen Text- oder XML-Editoren erstellt. Dieses Vorgehen ist besonders bei größeren Projekten mühsam und fehleranfällig. Der Erstellungsaufwand für den Frameworkentwickler könnte mit speziellen, grafischen FML-Editoren deutlich gesenkt werden. Insbesondere die Unterstützung der korrekten Referenzierung verschiedener Projektteile durch eine automatische ID-Verwaltung und die grafische Eingabe von Spezifikationsdaten würden eine wesentliche Erleichterung bedeuten.

- *Fortschrittsanzeige für das Instanziierungswerkzeug*: Mit Hilfe des FIT-Werkzeuges wird der Anwendungsentwickler während der Instanziierung schrittweise bis zur vollständigen Anwendung geführt. Er hat allerdings bei der Arbeit mit neuen Frameworkbeschreibungen keine Möglichkeit, die Anzahl der benötigten Instanziierungsschritte abzuschätzen. Deshalb wäre im FIT-Werkzeug eine grafische Fortschrittsanzeige interessant, welche die Anzahl der insgesamt benötigten Instanziierungsschritte zum aktuellen Stand der Instanziierung ins Verhältnis setzt.

- *Unterstützung weiterer Programmiersprachen und Komponentenmodelle*: Die vorliegende, prototypische Implementierung des Instanziierungswerkzeuges FIT unterstützt die Programmiersprache Java und das darauf basierende JavaBeans Komponentenmodell. Konzeptionell gibt es jedoch keinen Zwang zu dieser Einschränkung. Aus diesem Grund wäre die Erweiterung des Werkzeuges für andere Programmiersprachen und Komponentenmodelle eine interessante Option. Hierbei könnten vor allem spezialisierte Komponentenmodelle berücksichtigt werden, die für bestimmte Einsatzbereiche der Automatisierungstechnik interessant sind.

Literaturverzeichnis

[ABM96] Amund Aarsten, Davide Brugali, Giuseppe Menga: Designing Concurrent and Distributed Control Systems. Communications of the ACM, 39(10):51–58, Oktober 1996.

[AlGa96] Robert Allen, David Garlan: The Wright Architectural Specification Language. Technischer Report CMU-CS-96-TB, School of Computer Science, Carnegie Mellon University, Pittsburgh, September 1996.

[Alex80] Christopher Alexander: The Timeless Way of Building. Oxford University Press, New York, 1980.

[Appe98] Andrew W. Appel: Modern Compiler Implementation in Java. Cambridge University Press, Cambridge, UK, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Compilers – Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, USA, 1986.

[Balz96] Helmut Balzert: Lehrbuch der Software-Technik: Teil 1: Software-Entwicklung. Spektrum Akademischer Verlag, Heidelberg, Berlin, 1996.

[BBE95] Andi Birrer, Walter R. Bischofberger, Thomas Eggenschwiler: Wiederverwendung durch Framework-Technik – Vom Mythos Zur Realität. OBJEKTspektrum, Seiten 18–26, 5/1995.

[BBH+99] Dieter Barelmann, Christof Bürger, Steffen Himstedt, R. Jamal: OPC in der Praxis. VDE Verlag, 1999.

[BCK98] Len Bass, Paul Clements, Rick Kazman: Software Architecture in Practice. Addison-Wesley, 1998.

[BEJV96] Pam Binns, Matt Englehart, Mike Jackson, Steve Vestal: Domain-Specific Software Architectures for Guidance, Navigation and Control. International Journal of Software Engineering and Knowledge Engineering, 6(2), 1996.

[Benh01] Anis Ben~Hamidene: Entwicklung eines Komponenten-Frameworks für Markiermaschinen auf der Basis von Java und XML. Diplomarbeit DA1778, Institut für Automatisierungs- und Softwaretechnik, Universität Stuttgart, 2001.

[Bigg94] T.~J. Biggerstaff: Is Technology a Second Order Term in Reuse's Success Equation? In Proceedings of International Conference on Software Reuse. IEEE Computer Society Press, November 1994.

[BrKi98] Nat Brown, Charlie Kindel: Distributed Component Object Model Protocol: DCOM 1.0. Microsoft Corporation, Redmond, WA, 1998.

[Bloc99] Jochen Blocher: Komponentenbasierte Steuerungssoftware für ein Aufzugmodell mit Hilfe von JavaBeans. Studienarbeit, Institut für Automatisierungs- und Softwaretechnik, Universität Stuttgart, Oktober 1999.

[BeMi00] Henning Behme, Stefan Mintert: XML in der Praxis – Professionelles Web-Publishing mit der Extensible Markup Language. Addison-Wesley, München, 2. Auflage, 2000.

[BMM+99] Jan Bosch, Peter Molin, Michael Mattsson, PerOlof Bengtsson, Mohamed E. Fayad: Building Application Frameworks: Object Oriented Foundations of Framework Design, Seiten 55–82. Wiley & Sons, 1999.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: Pattern-Oriented Software Architecture: A System of Patterns. Wiley & Sons, 1996.

[Booc94] Grady Booch: Designing An Application Framework. Dr. Dobb's Journal of Software Tools, 19(2):24–30, Februar 1994.

[BoSu98] Frederic Boussinot, Jean-Ferdy Susini: The SugarCubes Tool Box: A Reactive Java Framework. Software – Practice and Experience, 28(14):1531–1550, Dezember 1998.

[Casa95] Eduardo Casais: An Experiment in Framework Development - Issues an Results. Technical Report, FZI-Publication, Karlsruhe, 1995.

[CzEi00] Krzysztof Czarnecki, Ulrich W. Eisenecker: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading, MA, 2000.

[Chapp96] David Chappell: Understanding ActiveX and OLE. Microsoft Press, Redmond, WA, 1996.

[CNM95] Peter Coad, Daid North, Mark Mayfield: Object Models: Strategies, Patterns, & Applications. Prentice-Hall, 1995.

[Cope92] James Coplien: Advanced C++ Programming Styles and Idioms. Addison-Wesley, Reading, MA, USA, 1992.

[DaKn96] Per Dagermo, Jonas Knutsson: Developement of an Object-Oriented Framework for Vessel Control Systems. Technical report, ESPRIT III/ESSI/DOVER #10496, 1996.

[Doug99a] Bruce Powel Douglass: Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Object Technology Series. Addison-Wesley, 1999.

[Doug99b] Bruce Powel Douglass: Real-Time UML. Addison-Wesley, Reading, MA, 2. Auflage, 1999.

[Dujm00] Stjepan Dujmović: An Understandable and Configurable Domain-Specific Framework. In Proceedings of 33rd International Conference on Technology of Object-Oriented Langauges and Systems, Seiten 348–358. IEEE Computer Society, Juni 2000.

[Etsc94] Konrad Etschberger: Controller-Area-Network: Grundlagen, Protokolle, Bausteine, Anwendungen. Carl Hanser Verlag, München, 1994.

[EYG97] A. Eden, A. Yehudai, J. Gil: Precise Specification and Automatic Application of Design Patterns. In Proceedings of International Conference on Automated Software Engineering, Seiten 143–152. IEEE Computer Society Press, 1997.

[FaJo99] Mohamed E. Fayad, Ralph E. Johnson: Domain-Specific Application Frameworks. Wiley & Sons, New York, 1999.

[Flei99] Wolfgang Fleisch: Applying Use Cases for the Requirements Validation of Component-Based Real-Time Software. In Proceedings of 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Saint Malo, France, Seiten 75–84, 1999.

[FML01] Jochen Blocher: Framework Markup Language – Benutzerdokumentation. Internes Dokument, Institut für Automatisierungs- und Softwaretechnik, Universität Stuttgart, 2001.

[Fowl97] Martin Fowler: Analysis Patterns: Reusable Objects Models. Addison-Wesley, 1997.

[FPR00] Marcus Fontoura, Wolfgang Pree, Bernhard Rumpe: UML-F: A Modeling Language for Object-Oriented Frameworks. In Proceedings of ECOOP 2000, Seiten 63–82. Springer, 2000.

[FaSc97] Mohamed E. Fayad, Douglas Schmidt: Object-Oriented Application Frameworks. Communications of the ACM, 40(10):32–38, Oktober 1997.

[GAO95] D. Garlan, R. Allen, J. Ockerbloom: Architectural Mismatch or Why it's Hard to Build Systems out of Existing Parts. In Proceedings of the 17th International Conference on Software Engineering, Seiten 179–185, April 1995.

[GDH00] Peter Göhner, Stjepan Dujmović, Carsten Huber: Einsatz von Frameworks bei der Automatisierung technischer Produkte. Automatisierungstechnische Praxis (atp), 42(6):50–57, Juni 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, MA, 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: The Java Language Specification. Addison-Wesley, Boston, MA, 2. Auflage, 2000.

[GuNa99] Michael Gunzert, Andreas Nägele: Component-Based Development and Verification of Safety-Critical Software for a Break-by-Wire System with Synchronous Software Components. In Proceedings of Int. Symposium on Parallel and Distributed Systems Engineering, Los Angeles, CA, USA, Seiten 134–145, 1999.

[Göhn98] Peter Göhner: Komponentenbasierte Entwicklung von Automatisierungs-systemen. In GMA-Kongress 1998, Seiten 513–521. VDI-Berichte Nr. 1397, 1998.

[GaSh94] David Garlan, Mary Shaw: An Introduction to Software Architecture. Technischer Report CS-94-166, Carnegie Mellon University, School of Computer Science, Januar 1994.

[Haro98] Elliotte Rusty Harold: JavaBeans. IDG Books, Forster City, CA, 1998.

[HBB00] Gerald Hilderink, Andre Bakkers, Jan Broenink: A Distributed Real-Time Java System Based on CSP. In Proceedings of 3rd International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000), Newport Beach, CA, Seiten 400–407, März 2000.

[HJE95] Hermann Hüni, Ralph E. Johnson, Robert Engel: A Framework for Network Protocol Software. In Proceedings OOPSLA '95, ACM SIGPLAN Notices, Seiten 358–369. ACM Press, 1995.

[HLN+90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, Mark Trakhtenbrot: STATEMATE: A Working Environment for the Development of Complex Reactive Systems. IEEE Transactions on Software Engineering, 16(4):403–414, April 1990.

[HoRi99] Minh Son Hoang, Peter Rieger: Komponentenbasierte Automatisierungssoftware. Carl Hanser Verlag, München, 1999.

[JoFo88] Ralph E. Johnson, Brian Foote: Designing Reusable Classes. Journal of Object-Oriented Programming, 1(2):22–35, 1988.

[John92] Ralph E. Johnson: Documenting Frameworks using Patterns. ACM SIGPLAN Notices, 27(10):63–76, Oktober 1992.

[John94] Ralph E. Johnson: Documenting Frameworks. Framework Digest, 1 (13), ftp://st.cs.uiuc.edu/pub/FWList/v1n13, Oktober 1994.

[Jost00] Pascal Jost: Identifizierung von Komponenten für Domänen in der Automatisierungstechnik. In Tagungsband: Verteilte Automatisierung – Modelle und Methoden für Entwurf, Verifikation, Engineering und Instrumentierung, Magdeburg, März 2000.

[Kell97] Sean Kelly: An Object-Oriented Framework for Local Extensions to the WFO-Advanced Forecaster Display Workstation, D2D. In Proceedings of 13th International Conference on Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology, Long Beach, CA, Februar 1997.

[KrPo88] G. E. Krasner, S. T. Pope: A cookbook for using the model-view-controller user interface Paradigm in Smalltalk-80. Journal of Object Oriented Programming, 1(3):26–49, August/September 1988.

[KeRi88] B. W. Kernighan, D. M. Ritchie: The C Programming Language. Prentice-Hall, Englewood Cliffs, NJ, USA, 2. Auflage, 1988.

[Lewi95] Ted Lewis: Object-Oriented Application Frameworks. Manning Publications, Greenwich, USA, 1995.

[LaGö98] Rudolf Lauber, Peter Göhner: Prozessautomatisierung 1. Springer-Verlag, Berlin, 1998.

[LKA+95] David C. Luckham, John J. Kennedy, Larry M. Augustin, James Vera, Doug Bryan, Walter Mann: Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering, 21(4):336–354, April 1995.

[LaSc95] R. Greg Lavender, Douglas C. Schmidt: Active Object: an Object Behavioral Pattern for Concurrent Programming. Proc. Pattern Languages of Programs, September 1995.

[MAMY99] Ali Mili, Edward Addy, Hafedh Mili, Sherif Yacoub: Toward an Engineering Discipline of Software Reuse. IEEE Software, 16(5):22–31, 1999.

[MBF99] Michael Mattsson, Jan Bosch, Mohamed E. Fayad: Framework Integration Problems, Causes, Solutions. Communications of the ACM, 42(10):80–87, Oktober 1999.

[McIl76] M. D. McIlroy: Mass-Produced Software Components. In Software Engineering: Report on a Conference by the NATO Science Committee, Garmisch, Germany, Seiten 88–98, Oktober 1976.

[MCK97] Matthias Meusel, Krzysztof Czarnecki, Wolfgang Köpf: A Model for Structuring User Documentation of Object-Oriented Frameworks using Patterns and Hypertext. In Proceedings of ECOOP '97 – Object-Oriented Programming 11th European Conference, Jyväskylä, Finland, Lecture Notes in Computer Science, Seiten 496–510. Springer-Verlag, New York, NY, Juni 1997.

[Mons00] Richard Monson-Haefel: Enterprise JavaBeans. O'Reilly, UK, 2. Auflage, 2000.

[EJB00] Sun Microsystems: Enterprise JavaBeans Specification (Version 2.0). Proposed Final Draft, Sun Microsystems, Palo Alto, CA, http://www.javasoft.com/products/ejb, Oktober 2000.

[MoNi96] Simon Moser, Oscar Nierstrasz: The Effect of Object-Oriented Frameworks on Developer Productivity. IEEE Computer, Seiten 45–51, September 1996.

[Moli96] Peter Molin: Applying the Object-Oriented Framework Technique to a Family of Embedded Systems. Research Report 19/1996, Dept. of Computer Science, University of Karlskrona/Ronneby, November 1996.

[Nils98] Kelvin Nilsen: Adding Real-Time Capabilities to the Java Programming Language. Communications of the ACM, 41(6):49–56, Juni 1998.

[OrHa98] Robert Orfali, Dan Harkey: Client/Server Programming with Java and CORBA. Wiley & Sons, 2. Auflage, 1998.

[OMG99] OMG: CORBA Components. OMG-Dokument orbos/99-02-05, Object Management Group, März 1999.

[Pope97] Alan Pope: The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture. Addison-Wesley, Reading, MA, USA, Dezember 1997.

[PaPr99] A. Pasetti, Wolfgang Pree: A Component Framework for Satellite On-Board Software. In Proceedings of 18th Digital Avionics Systems Conference, Saint Louis, Missouri. IEEE, Oktober 1999.

[Pree95] Wolfgang Pree: Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.

[Pree97] Wolfgang Pree: Komponenten-basierte Softwareentwicklung mit Frameworks. dpunkt.verlag, Heidelberg, 1997.

[PSW99] D. Plakosh, D. Smith, K. Wallnau: Builder's Guide for WaterBeans Components. Technical Report CMU/SEI-99-TR-023, Carnegie Mellon University, Software Engineering Institute, 1999.

[PTH97] G. Pritschow, T. Tran, J. Hohenadel: Standalone PC-Controller on an Open Platform. In Proceeding of the 30th International Symposium on Automotive Technology & Automation, Florenz, Juni 1997.

[RoJo96] Don Roberts, Ralph E. Johnson: Evolve Frameworks Into Domain-Specific Languages. In Proceedings of the 3rd International Conference on Pattern Languages for Programming, Monticelli, IL, USA, 1996.

[Roge97a] G. Rogers: Framework-based software development in C++. Prentice-Hall, New Jersey, 1997.

[Roge97b] Dale Rogerson: Inside COM: Microsoft's Component Object Model. Microsoft Press, 1997.

[RePo97] Peter Rechenberg, Gustav Pomberger: Informatik Handbuch. Hanser Verlag, München, 1997.

[SBF96] Steve Sparks, Kevin Benner, Chris Faris: Managing Object-Oriented Framework Reuse. Computer, 29(9):52–61, September 1996.

[Sche99] Hans-Jürgen Scheibl: Visual C++ 6.0 für Einsteiger und Fortgeschrittene. Hanser Verlag, München, 1999

[Schm95] Hans Albrecht Schmid: Entwurf eines objektorientierten Baukastens zur Steuerung von Fertigungsanlagen. Informatik-Spektrum, 18(4):211–221, August 1995.

[Schm96] Hans Albrecht Schmid: Creating applications from components: a manufacturing framework design. IEEE Software, 13(6):67–75, November 1996.

[SCSP95] Stan Schneider, Vincent Chen, Jay Steele, Gerardo Pardo-Castellote: The ControlShell component-based real-time programming system, and its application to the Marsokhod Martian Rover. ACM SIGPLAN Notices, 30(11):146–155, November 1995.

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, Gregory Zelesnik: Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, 21(4):314–335, April 1995.

[ShGa96] Mary Shaw, David Garlan: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.

[ScKu00] Douglas C. Schmidt, Fred Kuhns: An Overview of the Real-Time CORBA Specification. IEEE Computer, 33(6):56–63, Juni 2000.

[SSF86] E. Seidewitz, M. Start, D. Firesmith: Object-Oriented Programming without an Object-Oriented Language. In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Jgg. 21, New York, NY, November 1986. ACM Press.

[Stro97] Bjarne Stroustrup: The C++ Programming Language. Addison-Wesley, Reading, MA, 3. Auflage, 1997.

[Szyp98] Clemens Szyperski: Component Software: Beyond Object-Oriented Programming. ACM Press and Addison-Wesley, New York, NY, 1998.

[Thal97] Georg Erwin Thaller: Software Engineering für Echtzeit und Embedded Systems. bhv Verlag, Kaarst, 1997.

[Trac95] Will Tracz: Confessions of a Used-Program Salesman: Lessons Learned. In Proceedings of the ACM SIGSOFT Symposium on Software Reusability SSR '95, Seattle, Seiten 11–13, April 1995.

[Udel94] Jon Udell: Componentware. Byte, 19(5):46–56, Mai 1994.

[VaBa97] Laura A. Valaer, Robert G. Babb: Choosing a User Interface Development Tool. IEEE Software, 14(4):29–39, Juli/August 1997.

[VlLi90] John M. Vlissides, Mark A. Linton: Unidraw: A Framework for Building Domain-Specific Graphical Editors. ACM Transactions on Information Systems, 8(3):237–268, Juli 1990.

[WeDu99] Sanjiva Weerawarana, Matthew J. Duftler: Bean Markup Language (Version 2.3) User's Guide. IBM TJ Watson Research Center, Hawthorne, NY, September 1999.

[Welc97] Brent B. Welch: Practical Programming in Tcl & Tk. Prentice Hall, New Jersey, 2. Auflage, 1997.

[WGM88] Andre Weinand, Erich Gamma, Rudolf Marty: ET++ – an Object Oriented Application Framework in C++. ACM SIGPLAN Notices, 23(11):46–57, November 1988.

[Zimm97] Walter Zimmer: Frameworks und Entwurfsmuster. Dissertation, Fakultät Informatik, Universität Karlsruhe, 1997.

Lebenslauf

Persönliche Daten:

15.09.1969	geboren in Zagreb / Kroatien

Schulbildung:

1976 – 1980	Grundschule
1980 – 1985	Realschule
1985 – 1989	Gymnasium, Abschluss Abitur

Studium:

1989 – 1996	Studium der Elektrotechnik an der Universität Stuttgart Studienrichtung: Theoretische Nachrichtentechnik Fachpraktikum bei ANT Nachrichtentechnik, Backnang
31.03.1996	Abschluss als Diplom-Ingenieur

Berufstätigkeit:

1996 – 2001	Wissenschaftlicher Assistent am Institut für Automatisierungs- und Softwaretechnik der Universität Stuttgart