Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

Untersuchung der Anwendung erweiterter Programmierparadigmen für die Programmierung eingebetteter Systeme

Verfasser:

Jörg Liebig

28.11.2008

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake (ITI),
Dipl.-Wirtsch.-Inf. Thomas Leich (ITI),
Dipl.-Inform. Michael Schulze (IVS)

Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg, Germany

Liebig, Jörg:
Untersuchung der Anwendung erweiterter Programmierparadigmen für die Programmierung eingebetteter Systeme
Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, 2008.

Danksagung

An erster Stelle möchte ich mich bei Prof. Dr. rer. nat. habil. Gunter Saake, Dipl.-Wirtsch.-Inf. Thomas Leich und Dipl.-Inform. Michael Schulze für die Betreuung meiner Diplomarbeit bedanken. Zahlreiche Diskussionen haben wesentlich zu dieser Arbeit beigetragen. Darüber hinaus gilt mein Dank Sven Apel und Christian Kästner, die mich mit kritischen Anmerkungen zu Form und Inhalt unterstützt haben. Ferner bin ich Dietmar Fehse und Silke Liebig für Vorschläge zu Rechtschreibung und Grammatik dankbar. Gleichzeitig möchte ich mich bei all jenen bedanken, die mich auf die eine oder andere Weise unterstützt haben, jedoch hier nicht namentlich aufgeführt sind.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AOP Aspekt-Orientierte Programmierung

API Application Programming Interface

CAN Controller Area Network

DBMS Datenbank Management System

EC++ Embedded C++

EEPROM Electrically Erasable Programmable Read Only Memory

FODA Feature-Orientierte Domänenanalyse

FOP Feature-Orientierte Programmierung

HPL Hardware-Produktlinie

ISR Interrupt-Service-Routine

LCD Liquid Crystal Display

LED Light Emitting Diode

LOC Lines of Code

OOP Objekt-Orientierte Programmierung

SPL Software-Produktlinie

SQL Structured Query Language

STL Standard Template Library

USART Universal Synchronous and Asynchronous Receiver Transmitter

VDI Verein Deutscher Ingenieure

Kapitel 1

Einleitung

1.1 Motivation

Stellenwert eingebetteter Systeme

Mehr als 98% der im Jahr 2000 weltweit produzierten Prozessoren waren Bestandteil eingebetteter Systeme [Ten00]. Der hohe Prozentsatz ist ein Anzeichen für die große Vielfalt eingebetteter Systeme, die in nahezu jedem technischen Gerät zu finden sind. Ihre Anwendungsgebiete liegen in den Bereichen Unterhaltungs- und Haushaltselektronik, Telekommunikation, Automotive, Industrieautomatisierung u. a. Jährliche Wachstumsraten im zweistelligen Bereich sorgten bis heute dafür, dass sich der Anteil an Prozessoren in eingebetteten Systemen gleichbleibend auf dem Niveau des Jahres 2000 hielt [Tur02, Kri05, Sch05, Tur08]. Weitergehend zeigen entsprechende Veröffentlichungen zu diesem Thema, dass eingebettete Systeme in Zukunft wahrscheinlich maßgeblich zur Wertschöpfung von Produkten beitragen [BKR+05, HKM+05].

Probleme Software-Entwicklung

Obwohl die Kosten für Hardware-Komponenten gering ausfallen, führt der vermehrte Einsatz eingebetteter Systeme zu einem erheblichen Anteil von ca. 50 % an den Produktionskosten in den zuvor genannten Anwendungsgebieten [HKM+05]. Einen entscheidenden Beitrag leistet hierbei die Software der Systeme. Laut Bollow et al. [BHK02] und Krishan [Kri05] nimmt ihr Umfang und ihre Komplexität fortwährend zu. Der Grund liegt in den zusätzlichen Anforderungen, die eingebettete Systeme mittlerweile erfüllen müssen und deren Umsetzung vermehrt in Software erfolgt. Bestehende Ressourcenbeschränkungen der eingesetzten Hardware führen in diesem Zusammenhang häufig zu aufwendigen Neuentwicklungen oder Anpassungen, die die Kosten zusätzlich steigern.

Lösungsansatz Produktlinie

Aus der Warenproduktion ist seit längerem die Entwicklungsform der *Hardware-Produktlinie (HPL)* bekannt. Anhand standardisierter Bausteine können individuelle Ausprägungen je nach Kundenwunsch gefertigt werden. Ein Beispiel der erfolgreichen Umsetzung einer HPL sind die 8-Bit Mikroprozessoren (AVR) für eingebettete Systeme des Herstellers Atmel.[1] Die einzelnen Modelle dieser Produktlinie unterscheiden sich in der Ausstattung an Programm- und Arbeitsspeicher, IO-Anschlüssen und integrierten

[1] http://www.atmel.com/products/avr/

Komponenten.

Mit Programmfamilien und *Software-Produktlinien (SPL)* existieren zwei Konzepte, die die Methodik der Produktlinientechnologie auch in der Software-Entwicklung umsetzen. Ziel dieser Vorgehensweisen ist die Modularisierung und Wiederverwendung von Software-Komponenten, für deren Umsetzung unterschiedliche Programmierparadigmen existieren.

OOP

Innerhalb der Anwendungsprogrammierung für Personalcomputer ist *Objekt-Orientierte Programmierung (OOP)* zur Strukturierung von Aufgabenstellungen weit verbreitet. Wie [EAK+01, Pre97, TOHSMS99] zeigen, reicht OOP zur Modularisierung von Software nicht aus. Unterschiedliche Sichten und Anforderungen lassen sich durch einfache Dekompositionen, wie sie mit OOP möglich sind, nicht abbilden. Zwei Techniken die diese Problematik adressieren sind *Aspekt-Orientierte Programmierung (AOP)* und *Feature-Orientierte Programmierung (FOP)*. Beide Programmierparadigmen erlauben eine über die OOP hinausgehende Wiederverwendbarkeit von Software-Modulen.

Vorbehalte gegenüber modernen Programmier-paradigmen

Die Anwendung moderner Programmierparadigmen zur Umsetzung von Software-Projekten für eingebettete Systeme wird in der Literatur kaum thematisiert. Dabei könnte die Software-Entwicklung eingebetteter Systeme von den Fortschritten in der Anwendungsprogrammierung profitieren. Aus Performancegründen wird häufig auf den Einsatz moderner Programmierparadigmen verzichtet. Vielfach wird diese Entscheidung mit dem Verweis auf Overhead (Programmgröße und -laufzeit) begründet, der im Zusammenhang mit neuen Software-Techniken stehen soll. Dies bezieht sich in erster Linie noch auf den Einsatz von OOP innerhalb von Software-Projekten.

Dennoch gibt es bereits einige Projekte in denen moderne Programmierparadigmen erfolgreich eingesetzt werden. In Schröder-Preikschat et al. [SPLSS07] wird die Entwicklung einer SPL für Betriebssysteme auf der Basis von AOP beschrieben (Projekt CiAO).[2] Der familienbasierte Ansatz von CiAO führt zur Kapselung verschiedener Betriebssystemfunktionalitäten, wie z. B. Strategien zur Verarbeitung von Threads oder Interruptfunktionalität.

In Lohmann et al. [LSSP06] wird die Entwicklung einer eingebetteten Wetterstation mit Hilfe verschiedener Implementierungen — darunter C, C++ und AspectC++ — untersucht. Alle Implementierungen folgen dem Ansatz der Software-Produktlinienentwicklung mit dem Fokus auf den eingesetzten Hardware-Komponenten (HPL). Die Ergebnisse zeigen, dass der entstehende Overhead der AspectC++-Implementierung im Vergleich zu einer effizienten C-Implementierung weniger als 3 % ausmacht. Darüber hinaus konnten leichte Geschwindigkeitsvorteile bei der Umsetzung mittels AspectC++ im Vergleich zu den anderen Implementierungen festgestellt werden.

[2] http://www4.informatik.uni-erlangen.de/z/Research/CiAO/

1.2 Zielstellung

Ziel dieser Arbeit ist es Empfehlungen für den Einsatz moderner Programmierparadigmen für die Umsetzung von Software-Projekten auf tief eingebetteten Systemen zu geben. Im Feld eingebetteter Systeme zeichnen sich tief eingebettete Systeme besonders durch ihre geringe Leistungsfähigkeit und Speicherausstattung aus. Neben den Empfehlungen werden zudem Interaktionspunkte zwischen HPL und SPL aufgezeigt.

Die Untersuchungen des Einsatzes erweiterter Programmierparadigmen erfolgt in dieser Arbeit an zwei Beispielen: Entwicklung von Gerätetreibern und Realisierung eines *Datenbank Management System (DBMS)* für die HPL eines tief eingebetteten Systems. Der Schwerpunkt wird auf die Umsetzung verschiedener Konfigurationen und Varianten mit Rücksicht auf das Zielsystem gelegt. Zur Realisierung werden die beiden Programmierparadigmen AOP und FOP herangezogen. Die Einschränkung auf diese beiden Paradigmen trägt dem Umstand Rechnung, dass in der vorliegenden Arbeit keine vollständige Untersuchung auf dem Gebiet moderner Programmierparadigmen erfolgen kann.[3] Die konkrete Untersuchung erfolgt für AOP und FOP anhand der beiden Erweiterungen AspectC++[4] und FeatureC++[5] für C++.

1.3 Gliederung der Arbeit

Alle für das Verständnis dieser Arbeit notwendigen Grundlagen werden in Kapitel 2 vermittelt. Das schließt eingebettete Systeme und deren Programmierung sowie Fachwissen aus dem Bereich moderner Programmierparadigmen ein. In Kapitel 3 folgt eine Analyse zum Einsatz des Präprozessors cpp bei der Umsetzung von Software-Projekten für tief eingebettete Systeme. Kapitel 4 greift die Ergebnisse des vorhergehenden Kapitels auf und diskutiert den Einsatz erweiterter Programmierparadigmen am Beispiel der Entwicklung von Gerätetreibern. Im Anschluss folgt eine Domänenanalyse für Datenmanagement auf tief eingebetteten Systemen. In diesem Zusammenhang werden existierende DBMS vorgestellt und ihre Eignung für den Einsatz auf tief eingebetteten Systemen analysiert. In Kapitel 6 wird der Aufbau und die Implementierung eines DBMS für tief eingebettete Systeme diskutiert. Einen Schwerpunkt dabei bildet die Bewertung der eingesetzten Paradigmen. Kapitel 7 fasst die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf weitere Arbeiten.

[3] Weitere Paradigmen sind Intentionale Programmierung, Subjekt-Orientierte Programmierung, Generative Programmierung u. a.

[4] http://www.aspectc.org/

[5] http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/fop/featurec/

Kapitel 2

Grundlagen

Mit dem Mikroprozessor Modell 4004 von Intel aus dem Jahr 1971 kam die Entwicklung von eingebetteten Systemen auf [Bar99]. Die zum damaligen Zeitpunkt vorherrschenden Ressourcenbeschränkungen prägen in ähnlicher Form bis heute den Einsatz und die Programmierung dieser Systeme. Im folgenden Kapitel wird u. a. ein Einblick in die Thematik eingebetteter Systeme gegeben. Dazu gehört neben der Vorstellung der für die Arbeit eingesetzten eingebetteten Systeme auch der aktuelle Stand der Programmierung.

Auf die Erläuterungen zur Programmierung eingebetteter Systeme folgt eine Einführung in die Methode der Domänenanalyse am Beispiel der *Feature-Orientierten Domänenanalyse (FODA)*. Das geschilderte Vorgehen wird später zur Bestimmung der Anforderungen von Datenmanagementsystemen für tief eingebettete Systeme verwendet (Kapitel 6, S. 73). Die Ergebnisse der Analyse bilden die Grundlage zur Entwicklung der *Software-Produktlinie (SPL)* des Datenmanagementsystems *RobbyDBMS*. Zum besseren Verständnis wird der Begriff SPL im Abschnitt 2.5 erörtert. Zwei Konzepte zu deren Umsetzung werden mit den Programmierparadigmen AOP und FOP im Anschluss vorgestellt. Mit AspectC++ und FeatureC++ wird jeweils eine Programmiersprache als Vertreter eines Paradigmas eingeführt.

2.1 Eingebettete Systeme

Eingebettete Systeme werden nach Scholz [Sch05] wie folgt definiert:

> *Eingebettete Systeme sind Computersysteme, die aus Hardware und Software bestehen und die in komplexe technische Umgebungen eingebettet sind.*

tief eingebettete Systeme

Tief eingebettete Systeme zeichnen sich darüber hinaus durch extreme Ressourcenbeschränkungen aus. Einschränkungen beziehen sich dabei auf die Leistungsfähigkeit des Mikroprozessors, den Speicher und den Stromverbrauch [SSPSS98]. In der Praxis bedeutet dies, dass der Prozessor mit einer geringen Taktfrequenz arbeitet und die Speicherausstattung bei nur wenigen Kilobyte liegt. Die Begrenzung auf minimale

Ressourcen ist eine Folge von Kostenvorteilen, die sich bei sehr hohen Stückzahlen erzielen lassen. Gleichzeitig führt der Kostendruck zur Entkopplung der Entwicklung tief eingebetteter Systeme von der Speicher- und Mikroprozessorentwicklung im Computerbereich [ABP06]. In Abschnitt 2.2 (S. 7) werden die für die Realisierung dieser Arbeit eingesetzten Systeme als Beispiel tief eingebetteter Systeme vorgestellt.

schematische Arbeitsweise

Eingebettete Systeme finden sich in vielen Anwendungsbereichen wieder und müssen unterschiedliche Anforderungen erledigen. Grundlegend erfüllen sie Steuerungs- und Regelungsaufgaben z. B. in Kraftfahrzeugen, Konsumelektronik. Die Abbildung 2.1 stellt schematisch ihre Arbeitsweise dar.

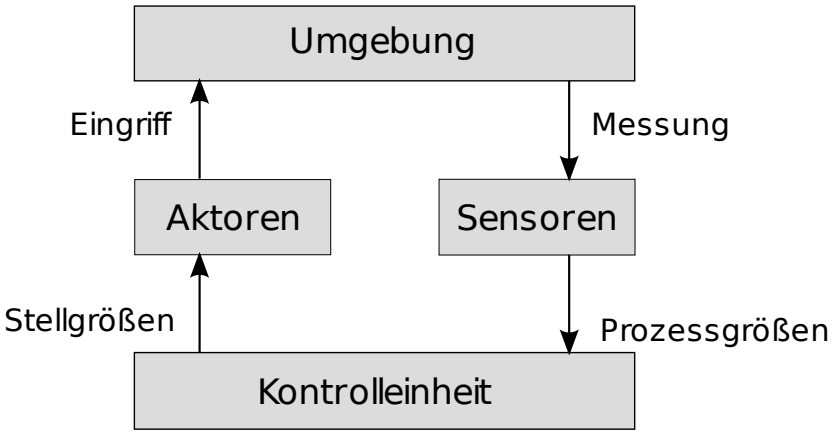


Abbildung 2.1: Wirkungskette System/Umgebung [Sch05]

Die Darstellung zeigt eingebettete Systeme als Sensor-Aktor Systeme. Zur Abarbeitung von Aufgabenstellungen stehen verschiedene Sensoren zur Verfügung, z. B. Druck, Temperatur, Beschleunigung. Anhand der Eingabewerte dieser Sensoren wird die Umgebung modellhaft erfasst und in der Kontrolleinheit verarbeitet. Diese generiert daraus Stellgrößen für Aktuatoren zur Rückwirkung auf die Umgebung. Je nach Nutzung oder Aufgabenbereich sind eingebettete Systeme noch mit weiteren Funktionen ausgestattet: Modul für drahtlose Kommunikation, Bussystem für drahtgebundene Übertragungen, LCD-Anzeige u. a.

Für verschiedene Aufgabenstellungen ist es notwendig, die mit den Sensoren gemessenen Daten für einen längeren Zeitraum vorzuhalten. Ein paar Beispiele für anfallende Daten sind: aufgetretene Fehler- oder Systemzustände, Konfigurationsdaten und Analysedaten, die während der Systementwicklung anfallen. Der Einsatz von Datenmanagementsystemen für die Datenhaltung ist bereits auf dieser Ebene sinnvoll. Eine genauere Erläuterung dazu erfolgt in den Abschnitten 5 und 6 (S. 59 bzw. 73).

Die Einbettung in reale Systeme und die damit verbundene Interaktion mit der Umwelt erfordert ein hohes Maß an Zuverlässigkeit. Im Einzelnen gehören dazu (Attribute der Zuverlässigkeit [Sch79]): Verfügbarkeit, Betriebsbereitschaft, Betriebssicherheit, Wartungsfreundlichkeit, Vertraulichkeit, Integrität und Ausfallsicherheit. Diese Attribute vergrößern die Anzahl der Anforderungen an ein eingebettetes System. Da nicht

alle Attribute aufgrund von Zielkonflikten gleichermaßen erfüllt werden können, sind Konfigurationen zur Festlegung des Funktionsumfangs eines Zielsystems erforderlich.

Hardware-Produktlinie

Die in technischen Produkten weit verbreiteten Produktlinien kommen auch hier zum Einsatz. Eine Reihe von eingebetteten Systemen unterscheidet sich mitunter nur in einzelnen Merkmalen oder in der Ausstattung vorhandener Komponenten. Dazu gehören die zuvor genannten Sensoren, aber auch die Leistungsfähigkeit und Größe des Arbeits-, Daten- und Programmspeichers. Beispiele verschiedener eingebetteter Systeme werden nachfolgend vorgestellt.

2.2 Atmel AVR Hardware-Produktlinie

In dieser Arbeit werden verschiedene eingebettete Systeme auf der Basis der Prozessorfamilie AVR der Firma Atmel eingesetzt. Es handelt sich dabei um 8-Bit Prozessoren mit integriertem Arbeits-, Daten- und Programmspeicher. Die Prozessoren selbst bilden eine Produktlinie. Durch vorhandene Komponentenausstattungen und optionale Schnittstellen entstehen verschiedene Variationen. Zu den grundlegenden Ausstattungsmerkmalen gehören:

- 1 - 8 MHz Taktrate des Prozessors,[1]
- 32 oder 128 KB Programmspeicher,
- 1 oder 4 KB Arbeitsspeicher und
- 1 oder 4 KB Datenspeicher (EEPROM).

Optionale Schnittstellen sind bspw. *Controller Area Network (CAN)* und *Universal Synchronous and Asynchronous Receiver Transmitter (USART)*. Eingebettete Systeme, deren Stromversorgung auf Batterien oder Akkumulatoren basiert, werden zusätzlich noch als autonom bezeichnet.

Produktlinie eingebettetes System

Auf der Basis der Prozessorfamilie AVR lassen sich unterschiedliche eingebettete Systeme kreieren. Zur Anzahl unterschiedlicher Varianten (Hardware-Konfigurationen) in der Produktlinie tragen vor allem externe Erweiterungen wie z. B. Sensoren bei. Darüber hinaus lassen sich weitere Varianten durch ein LCD-Display und Motoren aufbauen. Zwei Beispiele unterschiedlicher autonomer eingebetteter Systeme zeigt die Abbildung 2.2.

2.3 Programmierung eingebetteter Systeme

Im folgenden Abschnitt wird ein Überblick über Sprachen und Methoden zur Programmierung eingebetteter Systeme gegeben. Dazu werden einzelne Programmiersprachen erläutert, die u. a. in der vorliegenden Arbeit eingesetzt werden.[2] Diese Arbeit gibt keine Einführung in die genannten Programmiersprachen, sondern stellt deren Bedeutung

[1] Die angegebenen Taktraten gehören zur Werkskonfiguration, können jedoch angepasst werden.
[2] Weitere Sprachen sind Ada, Basic, Forth u. a.

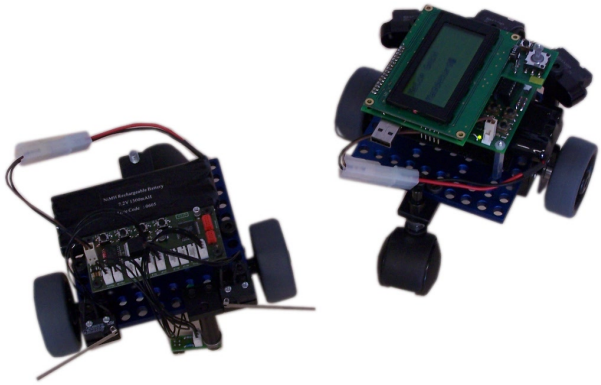


Abbildung 2.2: Zwei Varianten der eingesetzten Hardware-Produktlinie

für die Programmierung eingebetteter Systeme heraus. Beispiele zu den vorgestellten Programmiersprachen und eine Diskussion ihres Einsatzes zur Programmierung tief eingebetteter Systeme folgen in Kapitel 4 (S. 35).

2.3.1 Assembler

Assembler ist eine maschinennahe Programmiersprache, die sich auf eine konkrete Prozessorarchitektur bezieht. Assembler-Befehle werden häufig eins zu eins in Maschinenbefehle des Prozessors übersetzt und bilden damit die Grundlage für sehr effiziente Programme.

Nachteile Assembler

Heutzutage wird häufig auf die Verwendung von Assembler bei der Programmierung von eingebetteten Systemen verzichtet. Das liegt zum einen an der Verbesserung der Compiler die Quelltext aus einer Hochsprache in Assembler transformieren und zum anderen daran, dass die Portabilität und Produktivität hinter Programmen, die in Hochsprachen geschrieben werden, zurückliegt [Bar99]. Bestätigt wird diese Einschätzung durch die VDI[3]-Richtlinie 2422 [VDI2422] (Tabelle 2.1). In der Richtlinie wird beschrieben, dass Assembler zur Umsetzung komplexer Steuerungen ungeeignet ist. Dies wird mit dem Verweis auf die geringe Problemabstraktion begründet. Entsprechend der Angaben aus der Tabelle 2.1 eignen sich für komplexere Steuerungen nur Hochsprachen wie C, PL/M und Pascal. Dabei schafft es die Programmiersprache C als einzige unter den genannten Hochsprachen auch die Anforderung zeitkritischer Teilfunktionen zu erfüllen. Demnach kann C als Ersatz für Assembler betrachtet werden.

Obwohl die Prozessoren der AVR HPL einer Modellreihe angehören, unterscheiden sich ihre Assembler-Befehlssätze geringfügig. Je nach Modell fehlen einige Anweisungen,

[3] *Verein Deutscher Ingenieure (VDI)*

Programmiersprache \ Verwendung für	einfache Steuerung	komplexe Steuerung	zeitkritische Teilfunktion
ASSEMBLER	x		x
BASIC	x		
FORTRAN	x		
C	x	x	x
PL/M	x	x	
PASCAL	x	x	

Tabelle 2.1: Beispiele für Programmiersprachen und Hinweise zur Auswahl der Programmiersprache für die Umsetzung von Software-Projekten [VDI2422]

wodurch die Portierung von Programmen bereits innerhalb der Produktlinie erschwert wird.

2.3.2 Strukturierte Programmierung in C

„C has become the language for embedded programmers“ [Bar99]. Im Jahr 2000 wurden ca. 80 % aller Software-Projekte für eingebettete Systeme mit der Programmiersprache C umgesetzt. Als Gründe werden in der Literatur häufig der Bezug zur Hardware-nahen Programmierung und die hohe Effizienz übersetzter Programme angeführt [Bar99, Zur00]. Dabei bezieht sich die Effizienz sowohl auf die Programmgröße als auch auf die Ausführungszeit. C ist als Abstraktion von Assembler entworfen worden. Das Sprachdesign zielt auf die Portabilität von Programmen beim Wechsel der Prozessorarchitektur ab. Darüber hinaus soll die Effizienz übersetzter Programme beibehalten werden.

Wie in Abschnitt 2.1 (S. 5) bereits beschrieben, existieren von eingebetteten Systemen viele Varianten. Aus diesem Grund wird die Software für eingebettete Systeme in Module unterteilt. Innerhalb der Programmierung mit C erfolgt eine Strukturierung in erweiterte Datentypen, Methoden und Dateien. Zur Umsetzung von Konfigurationen und Alternativen, die sich nicht mit den genannten Strukturmitteln umsetzen lassen, wird auf Präprozessoren zurückgegriffen. Ein weit verbreiteter Präprozessor für C/C++ ist das Programm cpp[4].

2.3.3 Präprozessor cpp

Der Präprozessor cpp ist direkter Bestandteil bei der Programmierung in C/C++. Er wird während der Programmübersetzung als erstes vom Compiler aufgerufen. Alle im Quelltext enthaltenden Präprozessordirektiven (Makros oder Präprozessoranweisungen) werden entsprechend ihrer Definition vom Präprozessor verarbeitet. Dabei leitet das #-Zeichen eine Direktive ein. Der Präprozessor cpp stellt unterschiedliche Direktiven zur

[4] cpp - The C Preprocessor. Er ist Bestandteil der gcc-Compiler-Sammlung.

Verfügung mit deren Hilfe Textersetzungen in der aufgerufenen Datei vollzogen werden. Eine grobe Einteilung der zur Verfügung stehenden Makros mitsamt ihrer Bedeutung listet die Tabelle 2.2 auf. Mit Hilfe des *#include* Makros werden Definitionen anderer Module eingebunden. Durch *#define* werden bspw. Konstanten und Ausdrücke, die im nachfolgenden Programmtext ersetzt werden sollen, definiert. Für die Umsetzung verschiedener Konfigurationen kommen *#if, #ifdef* u. a. Makros zum Einsatz. Diese erlauben die bedingte Übersetzung des Programmtextes. Dazu werden i. d. R. Konfigurationsparameter, die zuvor mittels *#define* festgelegt wurden, abgefragt. Im Weiteren können durch die logischen Operatoren *&&,* || und *!* verschiedene Parameter zur Umsetzung von Konfigurationen in Beziehung gesetzt werden. Da der Präprozessor nur Text-basiert arbeitet, kann er für die Bearbeitung beliebigen Quelltextes eingesetzt werden und ist nicht allein auf C/C++-Quelltext beschränkt.

Direktive	**Bedeutung**
`#include`	Einfügen einer Datei
`#define`	Definition literaler Ausdrücke
`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`	bedingte Übersetzung

Tabelle 2.2: Direktiven und ihre Bedeutung

2.3.4 Objekt-Orientierte Programmierung mit C++

Laut Barr [Bar06] verdoppelt sich der Umfang von Software für eingebettete Systeme alle 10 - 24 Monate. Um der damit einhergehenden Komplexität und den steigenden Nutzeranforderungen zu begegnen, wird der Einsatz von OOP[5] mittels C++ vorgeschlagen [Bar99, Sch05, BHK02]. Die Autoren argumentieren, dass durch das Klassenkonzept eine Modularisierungstechnik zur Verfügung steht, die eine höhere Wart- und Wiederverwendbarkeit des geschriebenen Quellcodes erlaubt. Dies betrifft besonders die Entwicklung von Systemen für eine vorgegebene Domäne. Darüber hinaus motivieren die Autoren den Einsatz von C++ mit der besseren Eignung in Software-Projekten, die verteilt abgewickelt werden.

Nachteile einzelner Spracheigenschaften

In der Literatur wird der Einsatz von C++ zur Programmierung eingebetteter Systeme kontrovers diskutiert [Bar99, Bar06, BHK02, Pla97, Sch05]. Die Autoren verweisen auf den entstehenden Overhead, der sich bei Nutzung zusätzlicher Spracheigenschaften wie virtuelle Funktionen, Templates, Laufzeittypenidentifikation u. a. einstellen soll. Dabei gehen die Meinungen dahingehend auseinander, welche der genannten Sprachmittel für die Programmierung eingebetteter Systeme geeignet sind und welche nicht. Während Templates in den meisten Arbeiten als ungeeignet angesehen werden, spricht laut [Gol04, Str04, Sch02] nichts gegen ihren Einsatz zur Umsetzung statischer Konfigurationen. Weiterer Gegenstand zahlreicher Diskussionen sind virtuelle Funktionen, die

[5] Für Grundlagen zu OOP und C++ sei auf [Str00] verwiesen.

im Zusammenhang mit Polymorphie innerhalb von Vererbungshierarchien auftauchen (Laufzeitadaption). In Barr et al. [Bar06] wird herausgestellt, dass virtuelle Funktionen ein zumutbares Kosten-Nutzen-Verhältnis aufweisen und nur ein geringer Overhead in Kauf genommen werden muss. Andererseits sprechen bspw. die Ergebnisse der Studie Lohmann et al. [LSSP06] gegen die Verwendung virtueller Funktionen. Die Autoren konnten darin zeigen, dass durch den Einsatz von Laufzeitadaptionen die Programmgröße gegenüber statischen Konfigurationen um bis zu 78 % ansteigt.

In Stroustrup [Str02] stellt der Autor heraus, dass entstehender Overhead durch C++ eher auf die Benutzung der Sprache als auf genutzte Sprachmittel zurückzuführen ist. Dabei sieht er nur die Spracheigenschaften Ausnahmebehandlungen, Laufzeittypenüberprüfung und Speicherverwaltung als Bestandteil von C++ im Umfeld eingebetteter Systeme als kritisch an [Str07]. Eine eigene Untersuchung von C++ für die Programmierung tief eingebetteter Systeme in dieser Arbeit erfolgt in Abschnitt 4.3 (S. 39).

EC++

Die eingangs erwähnten Diskussionen bzgl. einzelner Spracheigenschaften von C++ haben u. a. zur Entwicklung von *Embedded C++ (EC++)* geführt. EC++, von einem Industriekonsortium speziell für den Einsatz auf eingebetteten Systemen entwickelt, ist ein Derivat von C++ und im Sprachumfang um jene Eigenschaften reduziert, die als nachteilig in Bezug auf entstehenden Overhead gelten. Zu den Abstrichen gehören das Entfernen von [Pla97]:

- Mehrfachvererbung und virtuelle Basisklassen,
- Typenerkennung zur Laufzeit,
- Ausnahmebehandlungen,
- Templates,
- Namensräume und
- Typenumwandlungskonstrukte.

Diese Spracheigenschaften sollen neben der längeren Ausführungszeit auch zur Vergrößerung des übersetzten Zielprogramms beitragen. Diese Einschätzung ist fragwürdig, da Templates, Namensräume und Typenumwandlungskonstrukte zur Übersetzungszeit ausgewertet werden und folglich kein Overhead zu erwarten ist.[6] Obwohl sich EC++ an eingebettete Systeme richtet, ist es ausschließlich für 32-Bit Mikroprozessoren spezifiziert. Deshalb kann vorhandener Quellcode nicht für 4- bis 16-Bit Mikroprozessoren eingebetteter Systeme übersetzt werden. Im Bereich tief eingebetteter Systeme sind diese Bit-breiten jedoch noch weit verbreitet.

2.4 Feature-Orientierte Domänenanalyse

Eine etablierte Methode zur Entwicklung von SPL ist die *Feature-Orientierte Domänenanalyse (FODA)*. Dieser Ansatz geht auf den durch das Software En-

[6] Die Aussage trifft für Templates nur eingeschränkt zu. Eine genaue Diskussion erfolgt in Abschnitt 4.3.3 (S. 47).

gineering Institute[7] vorgeschlagenen Prozess zur Domänenanalyse zurück. Neben FODA existieren weitere Vorgehensweisen zur Entwicklung von Domänen, die in Czarnecki et al. [CE00] vorgestellt werden.

Domänenanalyse allgemein

Die Domänenanalyse beschreibt einen Prozess zur Erfassung von Anforderungen einer Produktlinie. Das während des Prozesses erzeugte Domänenmodell besitzt generische Eigenschaften und lässt die Generierung einzelner Domänenmitglieder anhand von Parametrisierungen zu. Zur Entwicklung des generischen Modells werden im Laufe des Entwicklungsvorgangs Unterschiede zwischen den einzelnen Domänenmitgliedern herausgearbeitet und nachfolgend abstrahiert. Durch Abstraktion von den Differenzen ergeben sich die Gemeinsamkeiten der Domänenmitglieder.

FODA Operationen

FODA nutzt für die Bestimmung von Abstraktionen die Operationen Vereinigung/-Dekomposition und Generalisierung/Spezialisierung. Während die Vereinigung das Zusammenfassen von Bestandteilen zu einer Komponente beschreibt, drückt die Dekomposition die Zergliederung einer Komponente in seine Bestandteile aus. Die Abstraktion von Unterschieden innerhalb einer Gruppe führt zur Generalisierung. Generalisierte Einheiten können im Gegenzug durch Hinzunahme von einzelnen Details spezialisiert werden. Alle genannten Operationen treten in FODA einzeln und/oder in Kombination auf.

FODA Aufbau

Die Abbildung 2.3 zeigt die Bestandteile der Domänenanalyse, die nachfolgend für FODA erläutert werden [KCH+90].

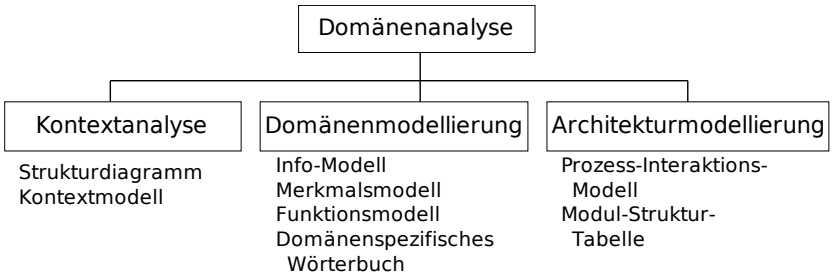


Abbildung 2.3: Phasen der Domänenanalyse

Der allgemeine Prozess der FODA teilt sich in drei Phasen auf: Kontextanalyse, Domänenmodellierung und Architekturmodellierung. Jeder Phase werden bestimmte Ergebnisse (Produkte) und zu durchlaufende Aktivitäten zugeordnet. Grundlage für alle im Prozess von FODA gemachten Analysen sind folgende Informationsquellen:

- Bücher,
- Standards,
- existierende Anwendungen und
- Experten auf dem Fachgebiet.

Unter den genannten Informationsquellen kommt die größte Bedeutung den existierenden Anwendungen zu.

[7] http://www.sei.cmu.edu/

2.4.1 Kontextanalyse

In der Kontextanalyse wird in erster Linie die betrachtete Domäne spezifiziert. Die Eingrenzung ist notwendig, um nutzbare Anwendungen erstellen zu können. Als Ergebnis entsteht ein Kontextmodell, dass aus einem oder mehreren Struktur- und Datenflussdiagrammen besteht. In die Erzeugung des Kontextmodells fließen externe Einflüsse und Bedingungen ein, die sich aus der Interaktion der Domäne mit der Umgebung ergeben. Die Kontextanalyse ist ein iterativer Prozess, in dem ausgehend von der Analyse vorhandener Informationen schrittweise das Kontextmodell erzeugt wird. Das Vorgehen schließt eine ständige Überprüfung der Zielanwendungen in Bezug auf die gesteckte Domäne ein. Eine genaue Übersicht aller in dieser Phase zu durchlaufenden Aktivitäten gibt Kang et al. [KCH+90].

2.4.2 Domänenmodellierung

Während in der Kontextanalyse vor allem technische Sachlagen der Domäne betrachtet wurden, geht es in der zweiten Phase (Domänenmodellierung) um die zu erzeugenden Zielanwendungen. Dazu wird dieser Abschnitt in drei Aktivitäten aufgeteilt:

Feature-Analyse Die Feature-Analyse dient der Bestimmung von Fähigkeiten eines Programms aus der Sicht von Anwendern. Dazu gehören bspw. die vorausgesetzten und angebotenen Dienste, die Leistungsfähigkeit der Anwendung, die benötigte Hardware und die Kosten. Die genannten Fähigkeiten stehen im direkten Kontakt zu den Anwendern und verbergen zugrunde liegende technische Zusammenhänge. Als Ergebnis werden Feature-Modelle erstellt. Diese werden durch Feature-Diagramme repräsentiert, die später noch vorgestellt werden.

Entity-Relationship-Modellierung In der Entity-Relationship-Modellierung (ER-Modellierung) wird das domänenspezifische Wissen, das für die Anwendungsentwicklung notwendig ist, zusammengefasst. Grundlage bilden ER-Modelle, die eine graphische Repräsentation aller für die Domäne wichtigen Einheiten sind. Dazu gehören u. a. die Eigenschaften der Einheiten und deren Beziehung untereinander. ER-Modelle werden bei der Implementierung zur Ableitung von Objekten und ihrer Attribute genutzt.

Funktionale-Analyse Die Funktionale-Analyse baut auf den Ergebnissen der vorangegangenen Aktivitäten auf. In dieser Tätigkeit werden funktionale Gemeinsamkeiten und Unterschiede zwischen Domänenmitgliedern identifiziert. Die dafür aufgestellten Funktions-Modelle schaffen detaillierte Zusammenhänge zwischen den im Feature- und ER-Modell modellierten Einheiten. Darunter fallen bspw. die Spezifizierung von Funktionen mit ihren Eingaben, Aktivitäten und Datenflüssen. Darüber hinaus fließen die Verhaltensweisen einer Anwendung mit ihren Ereignissen, Zuständen und Bedingungen mit ein.

Feature-Diagramme

Im Zusammenhang mit der Domänenmodellierung stehen Feature-Diagramme. Sie werden innerhalb der Feature-Modellierung benutzt um den Charakter von Programmeigenschaften abzubilden. Die Abbildung 2.4 zeigt ein Feature-Diagramm, in dem auszugsweise Features zur Beschreibung von Autos dargestellt sind. Der Aufbau von Feature-Diagrammen wird an diesem Beispiel erläutert.

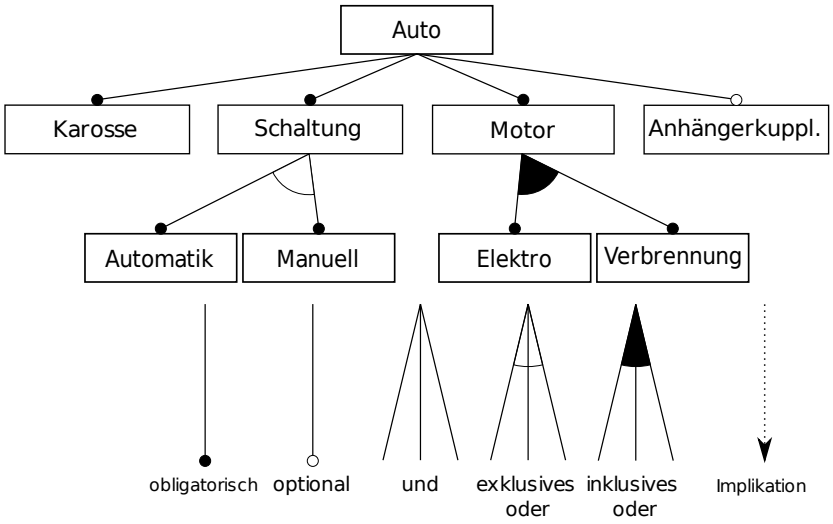


Abbildung 2.4: Diagramm zu Features eines Autos [CE99]

Alle innerhalb der Domänenanalyse bestimmten Programmeigenschaften werden gemäß einer Baumstruktur logisch strukturiert. So bildet Schaltung mit Manuell und Automatik eine logische Einheit. Features sind nicht gleichermaßen Bestandteil aller zu modellierenden Domänenmitglieder. Aus diesem Grund sind einzelne Merkmale entweder obligatorisch oder optional und können durch und/oder Operationen weiter in Beziehung zueinander gesetzt werden. Die strikte Baumstruktur lässt die Abbildung von Abhängigkeiten zwischen verschiedenen Merkmalen nicht zu. Beispielsweise ist es denkbar, dass der Elektromotor eine Automatikschaltung voraussetzt. Derlei Abhängigkeiten werden in dieser Arbeit durch den Implikationspfeil ausgedrückt.

Binding Time

Bei der Umsetzung der Features tritt die Frage auf, zu welchem Zeitpunkt die Wahl für optionale und/oder alternative Merkmale getroffen werden soll. Diese Frage ist eng mit dem Zeitpunkt der Einbindung (engl. Binding Time) verknüpft. Grundsätzlich wird nach folgenden Zeitpunkten unterschieden:

Kompilierungszeit Die Auswahl von Merkmalen bezieht sich auf unterschiedliche Zusammenstellungen von Anwendungen und wird während der Kompilierung getroffen. Entscheidungsgrundlagen können bereits vor der Ausführung getroffen werden und sind unabänderlich. Dieser Zeitpunkt wird vor allem aus Effizienzgründen (Programmlaufzeit und -größe) gewählt.

Startphase In der Startphase erfolgt die Auswahl von Merkmalen, die in Abhängigkeit zur Umgebung stehen. Für die Laufzeit der Anwendung stehen diese dann fest.

Laufzeit Die Bestimmung von Merkmalen zur Laufzeit beruht auf Interaktionen des Programms mit der Umgebung während des Programmbetriebes. Die dafür notwendige Logik ist direkter Bestandteil übersetzter Programme. Infolgedessen steigt die Programmgröße und die Ausführungszeit erhöht sich.

2.4.3 Architekturmodellierung

Phase drei der Domänenanalyse (Architekturmodellierung) umfasst die Bildung eines Referenz-Design-Modells, von dem Software-Lösungen abgeleitet werden können. Dabei wird auf die Ergebnismodelle der vorangegangenen Phasen zurückgegriffen. Da das erklärte Ziel der Domänenentwicklung die höhere Wiederverwendbarkeit ist, wird in der Architekturmodellierung ein Schichtenmodell benutzt. Einzelne Schichten werden gemäß eines Anwendungsfalles angepasst und zusammengefügt. Darüber hinaus werden in dieser Phase wiederverwendbare Software-Artefakte bestimmt und für erneute Einsätze vorbereitet (Bildung von Software-Modulen).

Das Erstellen eines Architekturmodells folgt einem Prozess, der genau in Kang et al. [KCH+90] beschrieben ist. Ein Teilaspekt dieses Prozesses umfasst die Kategorisierung von Prozessen innerhalb der Anwendung. Dazu werden folgende Kriterien verwandt:

- Nebenläufigkeiten,
- Input/Output Abhängigkeiten,
- funktionale und zeitliche Zusammenhänge,
- zeitkritische Abschnitte und
- Periodizitäten.

Erkenntnisse aus der Architekturmodellierung fließen in die Kontextanalyse und Domänenmodellierung ein. Dadurch entsteht, neben den iterativen Prozessen der einzelnen Phasen, ein Zyklus über alle Abschnitte der Domänenanalyse hinweg.

2.5 Software-Produktlinien

Nachdem in Abschnitt 2.2 (S. 7) bereits HPL eingeführt wurden, soll nun der Begriff *Software-Produktlinien (SPL)* erläutert werden. Northrop [Nor02] definiert sie wie folgt:

> *A software product line is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*

Entsprechend dieser Definition besitzen SPL einen ähnlichen Funktionsumfang und sind auf ein bestimmtes Marktsegment ausgerichtet. Um die Vorteile ähnlicher Funktionalität in der Software-Entwicklung nutzen zu können, schlägt Czarnecki et al. [CE00] die Entwicklung von SPL in Form von Programmfamilien vor. Der Begriff Programmfamilien geht auf Parnas [Par76] zurück und beschreibt eine Menge von Programmen, die viele technische Eigenschaften teilen. Dabei geht der technische Aspekt in erster Linie aus dem Entwicklungsprozess und den damit verbundenen Designentscheidungen während der Software-Entwicklung zurück. Als Prozess zur Entwicklung von Programmfamilien empfehlen Parnas [Par76] und Wirth [Wir71] die Entwicklung ausgehend von einer minimalen Basis. Alle notwendigen Funktionalitäten sollen durch Erweiterungen hinzugefügt werden. Dies führt zum Entwicklungsprozess der schrittweisen Verfeinerung (engl. Stepwise Refinement).

Die Nutzung von Programmfamilien zur Entwicklung von SPL führt zu einer höheren Wiederverwendbarkeit bestehender Software-Komponenten. Im Ergebnis lässt sich neben höherer Software-Qualität vor allem ein großer ökonomischer Nutzen erzielen.

SPL und Programmfamilien

Für den Rest dieser Arbeit wird die Umsetzung von SPL durch Programmfamilien angenommen. Die durch den Prozess der FODA bestimmten Programmmerkmale stehen stellvertretend für die jeweiligen Funktionalitäten, die im Sinne der schrittweisen Verfeinerungen umgesetzt werden sollen.

Klassifikation Programmmerkmale

Alle Programmmerkmale, die nach der im vorherigen Abschnitt definierten Methode bestimmt wurden, weisen unterschiedliche Charakteristika bzgl. ihres Wirkungsverhaltens auf. Eine Beobachtung in diesem Zusammenhang ist, dass sich Merkmale querschneidend zu anderen Software-Komponenten verhalten können [KLM+97]. Dieses Verhalten wird auch als querschneidende Belange (engl. Crosscutting Concerns) bezeichnet. Das Wirkungsverhalten unterscheidet sich dahingehend, dass einige Merkmale jeweils nur eine Stelle im Programm erweitern. Andere Merkmale werden an verschiedenen Stellen im Programm wirksam. Darüber hinaus existieren Abhängigkeiten, wie Merkmale als Erweiterung greifen. Eine Übersicht dazu gibt die Tabelle 2.3.

		Wirken im Quelltext	
		homogen	**heterogen**
Abhängigkeiten	**statisch**	Einführung einer Methode in mehreren Klassen	Einführung einer Methode in eine Klasse
	einfach dynamisch	Erweiterung mehrerer Methoden gleichzeitig	Erweiterung einer Methode
	erweitert dynamisch	Erweiterung mehrerer Methoden in Abhängigkeit vom Programmfluss	Erweiterung einer Methode im Abhängigkeit vom Programmfluss

Tabelle 2.3: Klassifizierung querschneidender Belange [Ape07]

Homogene Crosscuts erweitern unterschiedliche Stellen im Programm mit der gleichen Erweiterung. Im Gegensatz dazu stehen heterogene Crosscuts. Diese fassen ver-

schiedene Erweiterungen zu einer Einheit zusammen. Jede einzelne Erweiterung ergänzt jeweils nur eine Stelle im Programm. Laut Apel [Ape07] bilden die beiden betrachteten Programmierparadigmen AOP und FOP die in der Tabelle 2.3 klassifizierten querschneidenden Belange unterschiedlich gut ab. Beide stellen entsprechende Erweiterungen bereit, um querschneidende Belange umsetzen zu können. Im kommenden Abschnitt erfolgt eine Einführung in die beiden Paradigmen. Dabei wird auch ihre Eignung zur Abbildung querschneidender Belange erläutert.

2.6 Erweiterte Programmierparadigmen

Die beiden in dieser Arbeit verwendeten erweiterten Programmierparadigmen stehen in Bezug auf die Trennung querschneidender Belange (engl. Separations of Concerns) in Konkurrenz zueinander [ALS08]. Während in verschiedenen Arbeiten [KAB07, FC08] u. a. ihre Eignung aus konzeptioneller Sicht zur Umsetzung von SPL untersucht wurde, beschäftigt sich ein Teil dieser Arbeit mit der technischen Sicht auf beide Programmierparadigmen. Dieser Sicht kommt in eingebetteten Systemen aufgrund der vorherrschenden Ressourcenbeschränkungen eine große Bedeutung zu.

2.6.1 Feature-Orientierte Programmierung

Feature-Orientierte Programmierung (FOP) ist ein Programmierparadigma zur Entwicklung von SPL. Es stellt u. a. eine Erweiterung der OOP dar.[8] Wesentliches Merkmal der FOP ist das Feature.[9] Features werden nach [CE00, KCH+90] wie folgt definiert (aus [Ape07]):

> *... features reflect directly the requirements of the stakeholders and are used to specify and distinguish different software products, ...*

Features sind abstrakt und werden durch Feature-Module innerhalb der Programmierung repräsentiert. Ein Feature-Modul besteht aus Software-Einheiten (Rollen) die eine gegebene Basis an verschiedenen Stellen erweitern (Feature-Komposition). Es wird zur Umsetzung von Kollaborationen, d. h. die Erweiterung verschiedener Klassen für die Ergänzung eines Features, eingesetzt [SB02]. Ein Feature bildet somit eine Schicht, die alle Erweiterungen in sich vereint, die zur Umsetzung des angestrebten Features notwendig sind. Die grafische Darstellung dieses Sachverhalts erfolgt häufig in Kollaborationendiagrammen [RWL95]. Ein Beispiel gibt die Abbildung 2.5. Die Darstellung zeigt drei Kollaboration k1, k2 und k3, die jeweils ein Feature repräsentieren. Rollen werden durch weiße Boxen, Verfeinerungen durch Pfeile dargestellt.

[8] FOP ist ein allgemeines Konzept und nicht auf OOP beschränkt.

[9] Für die weitere Arbeit wird Features stellvertretend für Merkmale eines Programms benutzt. Die Nutzung dieser Bezeichnung erfolgt dabei unabhängig vom verwendeten Programmierparadigma.

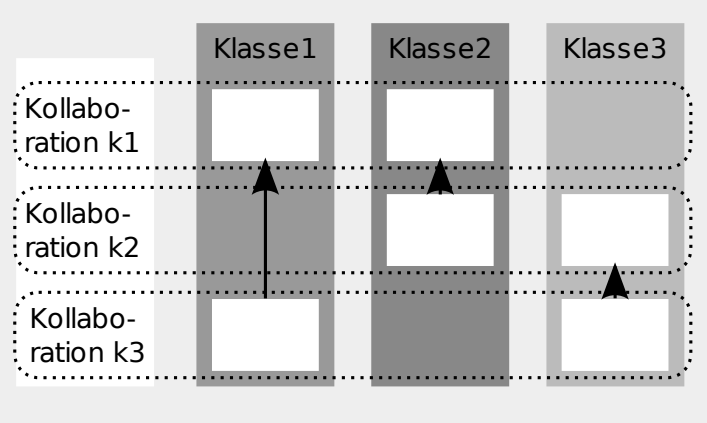


Abbildung 2.5: Beispiel Kollaborationendiagramm

heterogener Crosscut

Die Rollen eines Feature-Moduls stehen stellvertretend für Erweiterungen von existierenden Software-Einheiten. Dabei bezieht sich eine Rolle jeweils nur auf eine Einheit. Sind unterschiedliche Stellen im Programm jeweils mit der gleichen Erweiterung zu ergänzen, muss für jede Erweiterung eine separate Rolle definiert werden. Laut Apel [Ape07] eignet sich FOP für die Umsetzung von statischen und einfach dynamischen heterogenen Crosscuts. Dabei konnte in einer Studie gezeigt werden, dass es möglich war 94 % der Funktionalität einer SPL durch OOP und FOP abzubilden.

Die Nutzung von FOP dient der Vermeidung von Klassenexplosion, die das Ergebnis minimaler Erweiterungen unter Nutzung der Vererbung in OOP ist. Für jede Kombination von Features müsste eine eigene Klasse entwickelt werden, was im Ergebnis mit der Anzahl an Features zu einem exponentiellen Wachstum führt. Auch wenn die tatsächliche Anzahl an Variationen durch Abhängigkeiten zwischen verschiedenen Features darunter liegen kann, ist die Entwicklung mit viel Aufwand verbunden. Um dem entgegenzuwirken werden in FOP unterschiedliche Funktionalitäten bereits in die Basisklasse integriert. Verschiedene Features können dennoch einander aufsetzen.

Jampack-Komposition

Zur internen Realisierung von FOP wird Jampack-Komposition vorgeschlagen. Dieser Ansatz stellt eine Alternative zu den Klassenhierarchien dar, die ebenfalls in Batory et al. [BSR04] zur Diskussion gestellt werden.[10] Jampack-Kompositionen erlauben die direkte Integration von Erweiterungen in bestehenden Quellcode. In Rosenmüller [Ros05] wird auf die Vorteile von Jampack-Kompositionen gegenüber Klassenhierarchien hingewiesen, die in Kuhlemann et al. [KAL07] experimentell bestätigt wurden. Dazu gehören im Einzelnen:

1. Verzicht virtueller Funktionen und
2. Zugriffsmöglichkeit auf Membervariablen von Verfeinerungen in Erweiterungen.

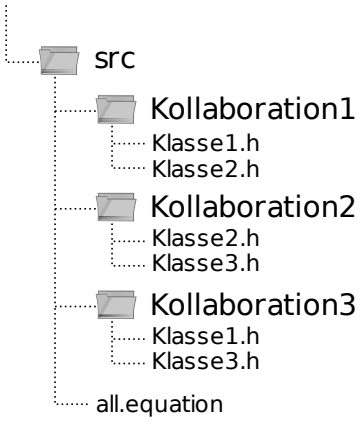
Vor allem der erste Punkt wirkt sich positiv in der Software-Entwicklung für eingebettete Systeme aus. Die Nutzung virtueller Funktionen zieht Nachteile in Bezug

[10] Weitere Vorschläge sind virtuelle Klassen, verschachtelte Vererbung u. a.

auf Performance und Programmgröße nach sich. Darüber hinaus werden die einzelnen Stufen der Verfeinerungshierarchie in der Feature-Komposition aus der Sicht der verfeinerten Klassen nicht verwendet, da die polymorphen Eigenschaften der Funktionen von Klassenhierarchien zu keinem Zeitpunkt genutzt werden. Der in der Arbeit eingesetzt FeatureC++-Compiler nutzt Jampack-Komposition zur Umsetzung von Feature-Kompositionen.

FeatureC++

Mit FeatureC++ steht eine Erweiterung für C++ zur Verfügung, die den Einsatz von FOP erlaubt [ALRS05].[11] Verfeinerungen, die zu einem Feature gehören, werden in einer Ordnerstruktur zusammengefasst (Abbildung 2.6 (a)). Die Reihenfolge der Anwendung einzelner Feature wird in einer Datei spezifiziert (Abbildung 2.6 (b)). Damit lassen sich Kollaborationen abbilden (Abbildung 2.5, S. 18). Der FeatureC++-Compiler agiert als Precompiler zur Erzeugung von C++-Quellcode aus FeatureC++-Quellen.



(a) Organisation von Features

```
1 // all.equation - Datei
2 Kollaboration1
3 Kollaboration2
4 Kollaboration3
```

(b)

Abbildung 2.6: Verarbeitung von Features in FeatureC++

2.6.2 Aspekt-Orientierte Programmierung

Aspekt-Orientierte Programmierung (AOP) hat die Kapselung von Programmfunktionalität insbesondere in Bezug auf die Behandlung querschneidender Belange zum Ziel [KLM+97, Spi02]. AOP dient damit der Modularisierung von System-Eigenschaften, die über die Kapselung von Funktionen und Eigenschaften, wie sie mit Klassen in der OOP für Objekte abgebildet werden können, hinausgehen [KLA06]. Folglich liegt das Einsatzgebiet bei der Behandlung von Crosscutting Concerns, wie sie einführend in Abschnitt 2.5 beschrieben wurden.

Aufbau und Nutzung

Der grundlegende Abstraktionsmechanismus besteht aus Aspekten, die der Programmerweiterung dienen. Ein Aspekt besteht aus Bedingungen (Pointcuts) und dazugehörenden Codeblöcken (Advices). Ein Pointcut ist ein deklarativer Ausdruck, der

[11] In dieser Arbeit wird die light Version des Compilers verwendet. Diese unterstützt ausschließlich FOP. Daneben existiert eine erweiterte Version, die außer FOP auch AOP unterstützt.

bspw. Signaturen von Methoden beschreibt (Joinpoint), die erweitert werden sollen. Mit Hilfe von Musterausdrücken lassen sich gleichzeitig mehrere Joinpoints erfassen. Innerhalb von Adviceblöcken wird Programmcode spezifiziert, der am Joinpoint ausgeführt werden soll. Mit zusätzlichen Angaben zur Wirkung eines Adviceblocks (before, around, after) lässt sich die Aufrufhierarchie beeinflussen. Aspekte werden vor der Kompilierung des eigentlichen Programms durch einen Aspektweber in das Basisprogramm eingefügt. Der Prozess als solches wird auch als „Weben“ bezeichnet. Das Eintreten eines Joinpoints wird im Verständnis der AOP als Ereignis verstanden. Diese Abstraktion lässt grundsätzlich jedwede Form von Erweiterungen im Basiscode zu, da alle Operationen (Zuweisungen, bedingte Sprünge u. a.) als Ereignis aufgefasst werden können. Dennoch liegt das Hauptaugenmerk Aspekt-Orientierter Sprachen bei der Erweiterung von Methoden. Typische Einsatzgebiete AOP sind Logging, Tracing und Caching.

Einsatz und Kritik

Dennoch ist der Einsatz von AOP zur Kapselung von Programmfunktionalität nicht unumstritten. Wie in Apel et al. [ALS06] beschrieben, eignet sich AOP vor allem zur Behandlung homogener Crosscuts in der Software-Entwicklung. In erster Linie kann durch den Einsatz von AOP auf Redundanz bei der Implementierung von Features verzichtet werden. Während in Spinczyk [Spi02] generell nur Aspekt-Orientierung eingesetzt wird, zweifeln die Autoren anderer Arbeiten [Ste06, EA06] die Eignung an. Dabei wird die Möglichkeit kritisiert, Quelltext an nicht vorhandenen Schnittstellen erweitern zu können. Die bedienten Joinpoints ergeben sich teilweise anhand der Schnittstellen eingesetzter Objekte, die innerhalb der Methoden einer Klasse eingesetzt werden und beziehen sich nicht auf die Schnittstellen der Klasse selbst. Dadurch bricht AOP mit der Forderung nach „Information Hiding“ innerhalb der strukturierten Programmierung [KLM+97].

Erweiterungen mit Hilfe von AOP sind implizit.[12] Sie ergeben sich durch die Definition von Pointcuts, die zu erweiternde Stellen innerhalb des Quellcodes beschreiben. Dazu werden deklarative Pointcut-Ausdrücke verwendet. Zur Behandlung homogener Crosscuts eingesetzte Musterausdrücke haben den Nachteil evtl. falsche Joinpoints zu erweitern bzw. Joinpoints auszulassen. Das Resultat von Pointcuts auf existierende Joinpoints im Basisprogramm ist nicht direkt ersichtlich. Abhilfe schaffen Visualisierungen innerhalb der Programmierumgebung, die das Wirken von Pointcuts an entsprechenden Joinpoints anzeigen.[13]

Ein weiterer Nachteil ist die Erhöhung der Komplexität zur Definition eines Pointcuts unter Berücksichtigung des Kontrollflusses. Einige Aspekte-Orientierte Erweiterungen bieten die Möglichkeit erweitert dynamische Abhängigkeiten abzubilden. Dadurch lassen sich Aspekte in Abhängigkeit von Vorbedingungen und der Aufrufhierarchie in den Basiscode weben. In Kästner et al. [KAB07] konnte bei der Refaktorisierung des DBMS Berkeley DB gezeigt werden, dass diese dynamischen Konstrukte nur selten notwendig sind. Darüber hinaus ergeben sich aufgrund der Vorbedingungen häufig komplexe Pointcut-Definitionen.

[12] Die Aussage bezieht sich auf AspectJ-ähnliche Sprachen.
[13] http://www.aspectc.org/ oder http://eclipse.org/aspectj/

Einschränkungen AOP

Existierende Aspekt-Orientierte Sprachen arbeiten häufig auf der Basis von Methoden und Konstruktoren. Einzelne Anweisungen oder Anweisungsblöcke können, wie in Kästner [Käs07] beschrieben, nicht erweitert werden. Erweiterungen auf Anweisungsebene sind bislang nicht im Sprachumfang enthalten.[14] In Harbulot et al. [HG06] wird die Erweiterung von AspectJ für Schleifen vorgestellt. Die Autoren nutzen den Pointcut-Advice Mechanismus zur Umsetzung von Schleifenparallelisierungen in der Programmiersprache Java. Sie betonen dabei besonders die Schwierigkeit der Identifikation von Schleifen, da diese keine eindeutigen Signaturen wie Methoden bereitstellen. Eindeutige Signaturen sind jedoch die Voraussetzung für Erweiterungen durch den Pointcut-Advice Mechanismus. Darüber hinaus wird die Identifizierung von Schleifen erschwert durch: verschiedene Schleifenkonstrukte (*while*, *do-while*, *for*), verschiedene Bedingungen und die Anwesenheit mehrerer Schleifen. Die Autoren stellen in ihrer Ausarbeitung auch die Erweiterung auf andere Anweisungen, wie *if-then-else* Konstrukte, in Aussicht.

AspectC++

In dieser Arbeit wird AspectC++ zur Umsetzung von AOP für C++ verwendet. Eine Übersicht zur Verarbeitung zeigt die Abbildung 2.7. Der AspectC++-Compiler (Aspekt-Weber) erzeugt aus Pointcut- und Advice-Definitionen (Aspekt A und B), die in separaten Dateien vorliegen, C++-Quellcode und integriert diesen in bestehende Basisklassen (Komponenten A - C) [SGSP02]. Der generierte Programmcode kann im Anschluss durch einen C++-Compiler zu lauffähigen Programmen übersetzt werden. Damit ist AspectC++ genau wie FeatureC++ ebenfalls ein Precompiler. Entsprechend der Klassifizierungsmatrix (Tabelle 2.3, S. 16) unterstützt AspectC++ alle Formen statischer und dynamischer Erweiterungen. Dabei ist eine Erweiterung nicht auf einzelne Komponenten beschränkt, so dass querschneidende Belange damit umgesetzt werden können (Abbildung 2.7). AspectC++ definiert zur Umsetzung eine eigene Sprache, die sich an der C++-Syntax orientiert. Die Erweiterungsfähigkeiten in AspectC++ beziehen sich in erster Linie auf Funktionen. Für eine vollständige Übersicht zur AspectC++-Syntax sei auf [Pur05] verwiesen.

[14] Die Aussage bezieht sich auf AspectJ und AspectC++.

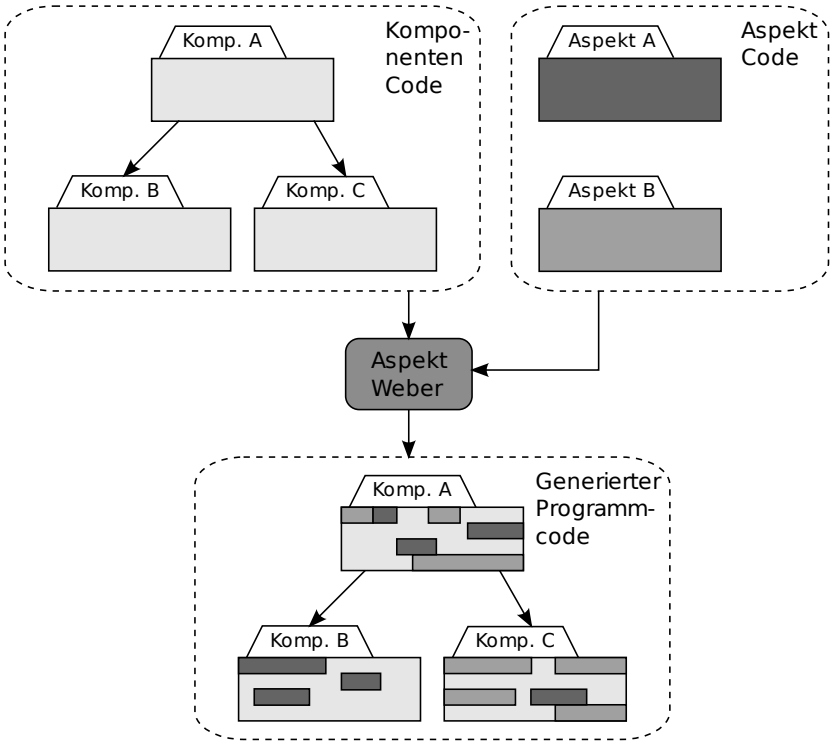


Abbildung 2.7: Weben von Aspekt Code in Komponenten Code [SGSP02]

Kapitel 3

Einsatz des Präprozessors cpp zur Implementierung von Features in eingebetteten Systemen

Nachdem im Grundlagenkapitel bereits ein Überblick zu Sprachen und Methoden zur Programmierung eingebetteter Systeme gegeben wurde, wird im folgenden Kapitel die Umsetzung von Software-Produktlinien mit Hilfe des Präprozessors cpp auf eingebetteten Systemen untersucht. Die Ergebnisse des Kapitels werden in den Abschnitten 4 und 6 (S. 35 bzw. 73) wieder aufgegriffen. In diesen Abschnitten erfolgt eine Gegenüberstellung der Resultate aus diesem Kapitel mit den Ergebnissen des Einsatzes moderner Programmierparadigmen in Software-Projekten. Aus diesem Grund gibt dieser Abschnitt eine Übersicht über den Einsatz des Präprozessors zur Implementierung von Features in Software-Projekten eingebetteter Systeme.

verwandte Arbeiten

Der Einsatz des Präprozessors in der Anwendungsprogrammierung wurde bereits in einer Studie untersucht [EBN02]. Die Autoren weisen in dieser Arbeit darauf hin, dass die Analyse auf weitere Domänen ausgeweitet werden sollte. Diese Aussage stützt sich auf die Annahme, dass sich der Einsatz des Präprozessors bspw. in der Programmierung von Betriebssystemen oder Bibliotheken vom Einsatz in der Anwendungsprogrammierung unterscheidet. Aus diesem Grund erfolgt im Rahmen dieser Arbeit eine Analyse über den Einsatz des Präprozessors bei der Programmierung eingebetteter Systeme. Dazu wird auch die Arbeit von Kästner et al. [KAK08] heranzogen, die sich mit der Granularität von Erweiterungen auseinandersetzt. Darin geben die Autoren eine Klassifikation für eine Teilmenge von Erweiterungen gegeben, die auch in dieser Arbeit verwendet wird.

3.1 Klassifikationen

Zunächst werden zwei Klassifikationen für Features vorgestellt, die später zur Analyse verschiedener Software Projekte genutzt werden. Wie bereits in Abschnitt 2.3.2 (S. 9)

erwähnt, bildet der Präprozessor cpp u. A. eine Grundlage zur Umsetzung von Konfigurationen in der C/C++-Programmierung. Einzelne Konfigurationen bestehen jeweils aus Erweiterungen einer bestehenden Basis um Quelltext, der ein Programm-Feature umsetzt. Präprozessordirektiven rahmen diesen Quelltext ein und gemäß der Feature-Konfiguration wird der Quelltext vom Präprozessor entfernt oder nicht. Dabei fallen Feature unterschiedlich aus, weil die jeweilig umgesetzte Erweiterung verschieden in den Quelltext der bestehenden Basis eingreift. Eine Untersuchung von Programm-Features in Software-Projekten führte laut Kästner et al. [KAK08] zu dem Ergebnis, dass Feature unterschiedliche Granularitäten bei der Einführung einer Erweiterung aufweisen. Neben dem Hinzufügen von neuen Attributen und Methoden in OOP (grobgranulare Erweiterung), waren auch Erweiterungen in den Methodenrümpfen notwendig. Diese auch als feingranulare Erweiterungen bezeichneten Ergänzungen lassen sich nach [KAK08] unterscheiden in:

feingranulare Erweiterungen

- Änderung von Methodensignaturen,
- Erweiterung von Ausdrücken und
- Erweiterung von Anweisungen.

Die Abbildung 3.1 (a) zeigt Beispiele für die genannten, feingranularen Erweiterungen. Änderungen von Methodensignaturen (Z. 2, *Transaction txn*) beziehen sich auf das Hinzufügen eines oder mehrerer Parameter zu einer bestehenden Funktion, die dann im Funktionsrumpf verwendet werden. Erweiterungen von Ausdrücken (Z. 3, || *txn==null* betreffen Ausdrücke als Bestandteil von Schleifen oder bedingten Anweisungen. Hierbei erfolgt die Ergänzung des Ausdrucks um weitere Operatoren mit dazugehörigen Operanden. Die Erweiterung von Anweisungen betrifft das Einfügen von Anweisungen an beliebiger Stelle im Funktionsrumpf (Z. 4 und 6, *Lock l=txn.lock(o);* bzw. *l.unlock();*). Eine mögliche Umsetzung des Beispiels mit Hilfe von Präprozessordirektiven zeigt die Abbildung 3.1 (b).

heterogene und homogene Erweiterungen

Neben der Granularität stellt sich auch die Frage, was für Erweiterungen im Programmcode gemacht werden. Im Grundlagenkapitel wurde dazu bereits eine Klassifizierungsmatrix (Tabelle 2.3, S. 16) vorgestellt. Entsprechend dieser Tabelle erlauben Präprozessordirektiven statische und einfach dynamische Erweiterungen. Das Hinzufügen von Attributen und Methoden kann durch Makros im Quelltext konfiguriert werden (explizite Erweiterung). Erweiterungen, die an Bedingungen geknüpft sind und erst zur Laufzeit bestimmt werden können (erweitert, dynamische Erweiterungen), lassen sich durch Präprozessordirektiven nicht umsetzen. Der Präprozessor stellt dafür keine Sprachelemente zur Verfügung, um auf dynamisch eintretende Ereignisse während der Programmausführung zu reagieren.[1] Insgesamt lassen sich durch Direktiven sowohl heterogene als auch homogene Erweiterungen umsetzen. Die entsprechenden Erweiterungen für beide Formen werden durch Makros im Quelltext des Basisprogramms explizit gemacht.

[1] Die Aussage lässt außer Betracht, dass durch zusätzlichen Quellcode in einer Erweiterung ein entsprechendes Verhalten umgesetzt werden könnte. Dessen ungeachtet werden dynamische Erweiterungen, wie sie bspw. AspectC++ anbietet, konzeptionell vom Präprozessor cpp nicht unterstützt.

```
1 class Stack {
2   void push(Object o
        , Transaction txn) {
3     if (o==null || txn==null)
          return;
4     Lock l=txn.lock(o);
5     elementData[size++] = o;
6     l.unlock();
7     fireStackChanged();
8   }
9 }
```

(a) Beispiele für feingranulare Erweiterungen

```
1  class Stack {
2    void push(Object o
3  #ifdef TXN
4    , Transaction txn
5  #endif
6  ) {
7      if (o==null
8  #ifdef TXN
9    || txn== n u l l
10 #endif
11     ) return ;
12 #ifdef TXN
13     Lock l=txn.lock(o);
14 #endif
15     elementData[size++] = o;
16 #ifdef TXN
17     l.unlock();
18 #endif
19     fireStackChanged();
20   }
21 }
```

(b) Umsetzung feingranularer Erweiterungen mit Hilfe des Präprozessors cpp

Abbildung 3.1: Feingranulare Erweiterungen: Beispiele und Umsetzung [KAK08]

Die beiden vorgestellten Klassifikationen (Granularität und heterogen/homogene Erweiterung) werden nachfolgend in der Analyse verwendet. Beide sind für spätere Diskussionen, die im Zusammenhang mit den in diesem Kapitel erarbeiteten Ergebnissen stehen, hinreichend genug. Eine genauerer Aufschlüsselung zum Aufbau von Makrodefinitionen, wie sie in der Studie gegeben wird, ist nicht notwendig, da letztlich nur ihr Wirken im Quelltext von Interesse ist. Dafür reichen die Betrachtungen fein- und grobgranularer Erweiterungen zusammen mit deren Hetero- und Homogenität aus.

3.2 Analyse Einsatz cpp

Die im vorherigen Abschnitt genannten Klassifizierungen werden nachfolgend für die Untersuchung verschiedener Software-Projekte für eingebettete Systeme verwendet. Alle dafür ausgewählten Software-Projekte werden zunächst vorgestellt und anschließend auf die in der Klassifizierung gegebenen Eigenschaften hin untersucht.

3.2.1 Vorstellung zu evaluierender Projekte

Im Gegensatz zur eingangs genannten Studie, die sich ausschließlich mit dem Einsatz des Präprozessors cpp in der Anwendungsprogrammierung auseinandersetzt, wird im folgenden Abschnitt der Präprozessoreinsatz in drei umfangreichen Software-Projekten für eingebettete Systeme untersucht. Die ausgewählten Projekte werden mit *Lines of*

Code (LOC) vorgestellt.[2] Im Einzelnen gehören dazu:

AVR LibC
AVR LibC[3] ist eine freie C-Bibliothek zur Programmierung von Mikroprozessoren auf AVR Basis. In der Bibliothek werden alle Hardware-Parameter unterstützter Prozessoren definiert und allgemeingültige Funktionen bereitgestellt. Die Bibliothek ist Bestandteil einer Werkzeug-Sammlung zur Entwicklung von Programmen für die AVR-Modellreihe. Das Projekt AVR LibC umfasst 50785 LOC. Für die Analyse wird die Version 1.6.1 der Bibliothek verwendet.

Ethernut
Ethernut[4] ist ein Projekt zur Realisierung einer Software-Plattform für eingebettete Systeme mit einer Ethernet-Schnittstelle. Dazu verbindet es das Echtzeitbetriebssystem Nut/OS mit einem TCP/IP Stack. Ethernut unterstützt verschiedene eingebettete Systeme, darunter auch Systeme auf der Basis von AVR-Mikroprozessoren. Für die Untersuchung wird die Version 4.2.1 verwendet. Das Projekt umfasst insgesamt 73055 LOC.

Femto OS
Femto OS[5] ist ein kleines, ressourcenschonendes Betriebssystem für eingebettete Systeme auf Basis von AVR-Mikroprozessoren. Die Quellcode-Basis von Femto OS besteht aus 13341 LOC. Obwohl der Umfang des Projekts vergleichsweise gering ist, stehen in der verwendeten Version (0.84) 623 Konfigurationsparameter zur Parametrisierung zur Verfügung.

Gründe für die Auswahl

Die Auswahl umfasst Projekte, die sich u. a. auch an eingebettete Systeme auf der Basis der AVR-Modellreihe richten. Die in Abschnitt 2.1 (S. 5) genannten Systeme werden teilweise unterstützt. Alle Projekte zeichnen sich durch einen starken Bezug zur Hardware aus. Das äußert sich darin, dass neben der allgemeinen Unterstützung verschiedener Hardware-Plattformen, viele Funktionalitäten der jeweils verwendeten Hardware angesprochen werden. Infolgedessen kann die Auswahl als repräsentativ für Software-Projekte eingebetteter Systeme angenommen werden. Da C als Programmiersprache im Bereich eingebetteter Systeme immer noch die weiteste Verbreitung findet (Abschnitt 2.3.2, S. 9), besteht die Auswahl zu analysierender Projekte ausschließlich aus Systemen, die in dieser Sprache geschrieben wurden. Für die Untersuchung wird die komplette Quellcode-Basis einschließlich evtl. vorhandener Beispielprogramme herangezogen.[6]

[2] Um die verschiedenen Software-Projekte vergleichen zu können, wurde der Quellcode vor der Analyse in ein einheitliches Format gebracht (keine Kommentare, keine Leerzeilen und gleiche Formatierung).
[3] http://www.nongnu.org/avr-libc/
[4] http://www.ethernut.de/
[5] http://www.femtoos.org/
[6] Einzelne Teile der Projekte sind auch in Assembler und wenige Beispielprogramme sind auch in C++ geschrieben. In ihnen kommen Präprozessoranweisungen genauso zum Einsatz.

Projekt	LOC	cpp	#include	#define	#ifdef u. a.	#others
AVR LibC	50785	89,12 %	1,41 %	93,14 %	5,44 %	0,01 %
Ethernut	73055	23,83 %	15,81 %	48,09 %	35,49 %	0,61 %
Femto OS	13264	69,28 %	0,44 %	54,46 %	41,57 %	3,53 %
Durchschnitt	45701	60,74 %	5,89 %	65,23 %	27,50 %	1,38 %
Studie [EBN02]	37457	8,4 %	15,48 %	32,14 %	47,62 %	4,76 %

Tabelle 3.1: Übersicht Vorkommen Präprozessoranweisungen

Umfang der Präprozessor-anweisungen in den Projekten

Die Tabelle 3.1 zeigt eine Übersicht zum Einsatz des Präprozessors in den drei Projekten.[7] Neben den bereits genannten Kennzahlen zum Umfang der jeweiligen Projekte (LOC), steht der Prozentsatz der Präprozessoranweisungen im Projekt (cpp). Im Weiteren ist der prozentuale Anteil der Direktiven nach Tabelle 2.2 (S. 10) separat aufgeschlüsselt. Die Unterscheidung ist wesentlich, da die einzelnen Klassen von Direktiven jeweils für einen anderen Einsatzzweck verwendet werden. Darüber hinaus zeigt die Tabelle, dass selbst in den eingebetteten Systemen der Umfang einzelner Direktiven stark schwankt.

In der eingangs erwähnten Studie [EBN02] liegt der Anteil der Präprozessoranweisungen in den untersuchten Anwendungsprogrammen im Durchschnitt bei 8,4 %. Die Bandbreite reicht im Einzelnen von < 4 % bis 22 %. Gerade Anwendungen, die eine große Anzahl von Systemen unterstützen, gehören hierbei zu den Projekten, in denen am häufigsten Präprozessoranweisungen zum Einsatz kommen. Viele Direktiven werden dabei für die Abfrage von Systemmerkmalen verwendet, um die von einander abweichenden Basisoperationen verschiedener Systeme anzusprechen. Die Tabelle 3.1 zeigt, dass der Anteil der Präprozessoranweisungen in allen Projekten über den Werten der Studie liegt. Der Durchschnitt liegt mit 60,74 % sogar um das 7,2-fache höher. Dabei treten vor allem bedingte Übersetzungen hervor. Dies geht zum einen auf die größere Anzahl unterstützter Systeme zurück. Zum anderen fallen die Unterschiede zwischen den unterstützten Plattformen höher aus. Ein weiterer Grund ist die Tatsache, dass angesichts der Ressourcenbeschränkungen eingebetteter Systeme eine allgemein höhere Konfiguration der Software ermöglicht wird, um Anforderungen eines Szenarios besser umsetzen zu können. Darum müssen vielfach Anpassungen gemacht werden, die mit Hilfe der strukturierten Programmierung in C nicht umgesetzt werden können. Im Weiteren werden die Klassifizierungen aus dem Abschnitt im Einzelnen für die Analyse der Software-Projekte verwendet. Diese sollen als Ausgangspunkt für Erklärungen zum vergleichsweise hohen Einsatz von Makros in der Programmierung eingebetteter Systeme dienen.

[7] Die Werte wurden u. a. durch ein eigens dafür entwickeltes Programm bestimmt.

3.2.2 Feingranulare Erweiterungen

AVR LibC

Der hohe Anteil an Präprozessoranweisungen hat in den einzelnen Projekten unterschiedliche Ursachen. Im AVR LibC Projekt geht dies v. a. auf die große Anzahl an Konstantendefinitionen zurück (Abbildung 3.2). Damit werden überwiegend Hardware- und Zugriffsparameter der 101 unterstützten AVR-Mikroprozessoren festgelegt. Mit Hilfe von feingranularen Erweiterungen werden im AVR LibC Projekt einzelne Anweisungen oder Anweisungsblöcke umgesetzt. Teilweise erfassen Direktiven auch ganze Funktionen. Doch diese bilden insgesamt die Ausnahme.

```
1 #define PINA    _SFR_IO8(0x00)
2 #define DDRA    _SFR_IO8(0x01)
3 #define PORTA   _SFR_IO8(0x02)
```

Abbildung 3.2: Konstantendefinitionen in AVR LibC

Ethernut

Der Anteil bedingter Übersetzungen fällt in den übrigen Projekten vergleichsweise höher aus. Jedoch ist dies auf die unterschiedliche Nutzung im jeweiligen Projekt zurückzuführen. In Ethernut gehen bedingte Übersetzungen zum Großteil auf Konfigurationen zurück, die nicht mit dem Quellcode des Systems vermischt sind. Dadurch werden in erster Linie Konstanten definiert, die an unterschiedlichen Stellen im Programmtext direkt verwendet werden. Präprozessordirektiven in Methodenrümpfen treten dann auf, wenn sich die unterstützten Hardware-Plattformen unterscheiden oder Quellcode in Abhängigkeit vom verwendeten Compiler steht. Jene Anpassungen beziehen sich ausschließlich auf Erweiterungen durch Anweisungen. Im Einzelnen bedeutet das, dass Hinzufügen einer Anweisung, einer Gruppe von Anweisungen oder ganzer Blöcke (z. B. *for*-Schleife oder eine weitere *case*-Verzweigung). Darüber hinaus gibt es vereinzelt Fälle, in denen Datenstrukturen mit Hilfe von Makros konfiguriert werden. Dies bezieht sich auf die Hinzunahme eines Datentyps bei der Zusammensetzung neuer Strukturen. Ein Beispiel zeigt die Abbildung 3.3. Der Parameter *ir_count* ist nur Bestandteil der Struktur *IRQ_HANDLER*, wenn *NUT_PERFMON* definiert ist. Neben dieser Strukturerweiterung werden auch alle Initialisierungen wie in dem Beispiel über Makros angepasst.

Femto OS

Wie die Tabelle 3.1 zeigt, machen Makros im Femto OS Projekt den überwiegenden Teil der Quellcode-Basis aus. Mit ihrer Hilfe werden sowohl ganze Methoden als auch einzelne Bestandteile von ihnen konfiguriert (Abbildung 3.4). Dabei sind die dafür eingesetzten bedingten Übersetzungen nicht selten von verschiedenen Konfigurationsparametern abhängig (Z. 1). In einzelnen Fällen treten Abhängigkeiten von bis zu fünf verschiedenen Parametern auf. Deren Zusammenhänge lassen sich häufig schlecht erfassen, weil die Schnittmenge zwischen Features an den vorkommenden Stellen nicht klar ersichtlich ist. Bei den vielen unterschiedlichen Verbindungen zwischen Features und deren Abhängigkeiten können Erweiterungen nur mit genauer Kenntnis aller Zusammenhänge erfolgen. Im Gegensatz zu den beiden anderen Projekten treten in Femto OS auch Änderungen von Methodensignaturen auf (Abbildung 3.5). Der Parameter *uiTick-*

```
typedef struct {
#ifdef NUT_PERFMON
  u_long ir_count;
#endif
  void* ir_arg;
  void (*ir_handler)(void*);
  int (*ir_ctl)(int cmd, void* param);
} IRQ_HANDLER;

...

IRQ_HANDLER sig_RSCTS = {
#ifdef NUT_PERFMON
  0,
#endif
  NULL,
  NULL,
  NplRsCtsCtl
};
```

Abbildung 3.3: Konfiguration einer Datenstruktur in Ethernut

sToWait wird bspw. in verschiedenen Methoden hinzugefügt. Innerhalb des Methodenrumpfes wird für diesen Parameter jeweils der gleiche Quellcode ausgeführt.

```
#if (cfgCheckReset == cfgTrue) || (defCheckReportingError == cfgTrue)
static void portShortDelay(Tuint08 uiShift) {
  TIMSK = 0x00;
  TCCR0B = 0x1D;
  TCCR0A = 0x80;
  OCR0B = 1 << uiShift;
  OCR0A = 0xFF;
  TCNT0H = 0x00;
  TCNT0L = 0x00;
  TIFR = 0x10;
  TCCR0B &= ~0x10;
  #if (cfgSysDebug == cfgFalse)
  while ( (TIFR & 0x10) == 0 );
  #endif
}
#endif
```

Abbildung 3.4: Beispiel Makroeinsatz in Femto OS

An vielen Stellen in den Projekten Ethernut und Femto OS treten Makros verschachtelt auf. Das heißt eine Präprozessoranweisung ist in den Kontext einer anderen eingebettet (Abbildung 3.4). Vereinzelt treten Verschachtelungstiefen bis zu einer Größe von 11 ein. Dies führt in erster Linie zu einer schlechten Lesbarkeit des Quelltextes. Darüber hinaus geht der Zusammenhang zwischen einzelnen Features, die an dieser Stelle umgesetzt werden, verloren. Dabei treten nicht selten unterschiedliche Kombinationen verschiedener Features auf. Beides trägt dazu bei, dass sich das Projekt nur schwer um weitere Features erweitern lässt.

Nachdem nun einzelne Bestandteile verschiedener Erweiterungen und Nutzung besprochen wurden, wird nun deren Zusammenhang bei der Umsetzung von Programm-

```
1 #if (cfgUseTimeoutOnSync == cfgTrue)
2 void privSyncWaitBody(Tuint08 uiSlot, Tuint16 uiTicksToWait)
      __attribute__ ( ( naked ) );
3 #else
4 void privSyncWaitBody(Tuint08 uiSlot) __attribute__ ( ( naked ) );
5 #endif
```

Abbildung 3.5: Änderung Methodensignatur in Femto OS

Features betrachtet. Diese Analyse trägt dazu bei, wie Features alternativ umgesetzt werden können.

3.2.3 Homogene und Heterogene Erweiterungen

Für die Untersuchung homogener und heterogener Erweiterungen wird der Quelltext auf bedingte Übersetzungen untersucht. Dafür werden die entsprechenden Direktiven (Tabelle 2.2, S. 10) analysiert und ausgewertet. Diese greifen auf zuvor per *#define* festgelegte Konfigurationsparameter zu, mit deren Hilfe bedingte Übersetzungen gesteuert werden. Für die Analyse werden zudem logischen Operatoren (*&&* und *||*) bedingter Übersetzungen ausgewertet. Die Kombination verschiedener Konfigurationsparameter kann jeweils die Umsetzung unterschiedlicher Features bedeuten. Grundlage dieser Annahme ist die Tatsache, dass einzelne Features voneinander abhängig sind und zusammen Programmeigenschaften definieren. Ein Beispiel zeigt die Abbildung 3.6. Die Zeilen 2 - 3 oder Zeile 5 hängen von den Konfigurationsparametern *F_CPU* und *U2X* ab mit denen die Übertragungsrate einer Schnittstelle konfiguriert wird (*UBRRL*).

```
1 #if F_CPU<2000000UL && defined(U2X)
2   UCSRA = _BV(U2X);
3   UBRRL = (F_CPU / (8UL * UART_BAUD)) - 1;
4 #else
5   UBRRL = (F_CPU / (16UL * UART_BAUD)) - 1;
6 #endif
```

Abbildung 3.6: Beispiel für sich überschneidende Programmmerkmale

Kombination heterogener und homogener Erweiterungen

Die strikte Trennung in heterogene und homogene Features kann bei der Umsetzung jedoch nicht immer aufrecht erhalten werden. Die Kombination beider Ansätze wird bereits in Apel et al. [ALS08] vorgeschlagen. Aus diesem Grund erfolgt in der Analyse die Aufteilung in heterogene und homogene Erweiterungen, sowie deren Kombination (heterogen und homogen). Die Ergebnisse der Analyse zeigt die Abbildung 3.7. Die Tabelle in der Abbildung gibt den prozentualen Umfang des Quellcodes in LOC an, der über bedingte Übersetzungen konfiguriert wird. Dabei zeigt sich, dass der Anteil an bedingten Übersetzungen im ganzen Quellcode (Tabelle 3.1, S. 27) nicht deren Umfang im Quellcode widerspiegelt. Mit bedingten Übersetzungen wird nicht ausschließlich nur Quellcode konfiguriert. Teilweise legen *#ifdef*-Konstruktionen , die über *#define* gesteuert werden, wiederum neue Parameter fest, sodass sie sich nicht direkt im Quellcode reflektie-

ren. Die Abbildung 3.7 zeigt die Ergebnisse der Analyse der drei untersuchten Software-Projekte. Dabei werden Erweiterungen entsprechend in heterogen, homogen sowie deren Kombination einzeln aufgeschlüsselt. Es zeigt sich, dass der überwiegende Anteil aus heterogenen Erweiterungen besteht. Dieser liegt im Einzelnen zwischen 85,23 % und 93,35 %. Homogene Erweiterungen machen in den Projekten jeweils zwischen 2,73 % und 6,47 % aus. Die Kombination beider Ansätze beläuft sich auf bis zu 4,19 % (von 1,08 %) der Quellcode-Basis. Gleichwohl ähnliche Werte in [AB08] et al. das Resultat einer Analyse sind, so sind die Ergebnisse der eigenen Analyse nicht einfach darauf übertragbar. Die in [Ape07] et al. untersuchten Programme nutzten Aspekte für die Umsetzung. Diese sind sowohl syntaktisch als auch semantisch im Sinne einer Erweiterung vollständig. Dies trifft auf Erweiterungen mit Hilfe von Präprozessordirektiven nicht zu (Abschnitt 3.3, S. 32).

Projekt	**Anteil**
AVR LibC	15,00 %
Ethernut	16,99 %
Femto OS	46,10 %

Projekt	***heterogen***	***heterogen & homogen***	***homogen***
AVR LibC	92,45 %	1,08 %	6,47 %
Ethernut	85,23 %	4,19 %	10,57 %
Femto OS	93,36 %	3,92 %	2,73 %

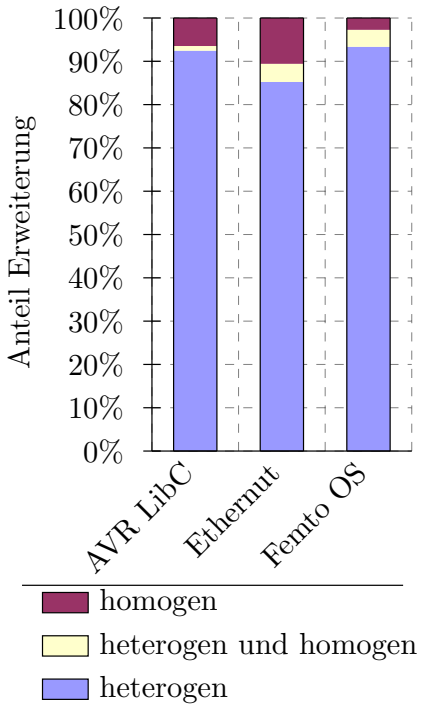


Abbildung 3.7: Gesamtumfang und Aufteilung der Erweiterungen in den Projekten

Umfang der Erweiterungen

Tiefergehende Analysen zeigen, dass einzelne Erweiterungen im Durchschnitt zwischen 5 und 11 LOC ausmachen. Diese fassen die in Abschnitt 3.2.2 (S. 28) besprochenen feingranularen Erweiterungen zu größeren Einheiten zusammen. Homogene Erweiterungen werden jeweils durch Quellcode-Replikation umgesetzt. Änderungen dieser Erweiterungen sind mit einem höheren Aufwand verbunden, da die einzelnen Erweiterungspunkte separat gepflegt werden müssen. Durch das Auslassen einzelner Erweiterungspunkte steigt die Fehleranfälligkeit. Doch dies ist nicht der einzige Nachteil der sich durch den Einsatz von Präprozessordirektiven ergibt. Im folgenden Abschnitt wird auf weitere Schwächen eingegangen.

3.3 Kritik am Einsatz cpp

Der Einsatz von Präprozessordirektiven gilt als schwierig für den Programmierer, weil Direktiven als unstrukturiert und invasiv gelten [Str94, SC92, EBN02]. Der Transformationsprozess zur Anwendung von Präprozessordirektiven erfolgt ohne Rücksicht auf die Programmiersprache und Semantik des geschriebenen Quellcodes. Dadurch werden bestehende Geltungs- und Namensbereiche aufgelöst. Infolgedessen sind Fehler im Zusammenhang mit dem Einsatz von Präprozessoranweisungen schwierig zu finden, da sie keine geeignete Darstellung/Kennzeichnung für Erweiterungen sind. In Spencer et al. [SC92] bezeichnen die Autoren den Einsatz des Präprozessors als schädlich und weisen auf häufigen Missbrauch bei der Umsetzung von Software-Projekten hin. In Ernst et al. [EBN02] wird zudem noch auf Seiteneffekte hingewiesen. Diese sind das Ergebnis unbeabsichtigter Textersetzungen. Die Folge sind sowohl syntaktische als auch semantische Fehler. Letztere sind schwierig zu finden, da sie nicht direkt in der Quellcode-Basis ersichtlich sind. Erweiterungen mit Hilfe von Präprozessoranweisungen müssen selbst nicht syntaktisch richtig sein. Es ist ausreichend, wenn der Quelltext nach dem Transformationsprozess der Grammatik einer Programmiersprache genügt. Diese Eigenschaft wird teilweise in den Projekten genutzt, führt jedoch zu einer schlechten Lesbarkeit (Abbildung 3.8) und ist problematisch, weil sie eine nahezu willkürliche Umsetzung von Features zulässt [KAK08].

```
1 #if SCANF_LEVEL > SCANF_MIN
2   if (width == SCHAR_MAX)
3     width = 1;
4   while (width-- > 0) {
5 #endif /* SCANF_LEVEL > SCANF_MIN */
```

Abbildung 3.8: Teildefinition eines Features

Modularität

Obwohl Makros für die Umsetzung von Features und Konfigurationen eingesetzt werden, sind sie nicht greifbar. Einzelne Features verteilen sich in Abhängigkeit von der Modellierung des Programms auf verschiedene Module und sind nicht zusammenhängend extrahierbar. Aus diesem Grund sind Erweiterungen immer Bestandteil des Programms.

C++ und der Präprozessor

Laut Stroustrup [Str94] wurde C++ entworfen einen Teil der Direktiven zu ersetzen. So findet sich für die Definition von Konstanten mit *const* ein entsprechendes Äquivalent. Zudem lassen sich Makros die Anweisungen oder Ausdrücke enthalten, durch Inline- oder Template-Methoden ersetzen. Hingegen gibt es laut [Str94] keine sprachliche Entsprechung für den Einsatz von bedingten Übersetzungen mit Hilfe von *#ifdef*-Konstruktionen. Diese bilden jedoch häufig die Grundlage für Erweiterungen.

3.4 Zusammenfassung

Die Analyse des Präprozessors cpp für Umsetzung von Software-Produktlinien in eingebetteten Systeme zeigte, dass bis zu 46 % der Eigenschaften eines Programms direkt über den Präprozessor konfiguriert werden. Dabei wurde festgestellt, dass mindestens 85 % der Programmerweiterungen heterogen sind. Die einzelnen Erweiterungen werden sowohl zur Unterstützung verschiedener Hardware-Plattformen in der Software als auch zur Umsetzung von Programmkonfigurationen eingesetzt. Dabei sind jeweilige Ergänzungen oftmals minimal.

Das immer noch an Makros festgehalten wird, geht laut Ernst et al. [EBN02] auf die Tatsache zurück, dass Präprozessoranweisungen die Konfiguration von Quellcode so erlauben, dass er nicht in den Übersetzungsprozess bei der Kompilierung durch einen Compiler mit einbezogen werden muss. Jedoch wiegt diese Eigenschaft nicht die Nachteile auf, die sich im Zusammenhang mit dem Einsatz des Präprozessors ergeben. Darüber hinaus ist die Fähigkeit des Präprozessors zur Umsetzung von Features fragwürdig. Infolgedessen sind Alternativen für die Feature-Umsetzung wünschenswert. Der Kompositionsansatz auf der Grundlage von Feature-Modulen und/oder Aspekten bietet eine Möglichkeit Features ohne Präprozessoranweisungen umzusetzen. Dieser wird nachfolgend für die Programmierung tief eingebetteter Systeme untersucht.

Kapitel 4

Abbildung von Hardware-Produktlinien

Bei der Vorstellung der für diese Arbeit genutzten eingebetteten Systeme wurde bereits der Begriff der HPL eingeführt (Abschnitt 2.2, S. 7). Im folgenden Kapitel wird deren Abbildung in der Software-Entwicklung tief eingebetteter Systeme analysiert und diskutiert. Grundlage für diese Untersuchung sind u. a. die Ergebnisse aus dem vorherigen Kapitel 3 (S. 23). Darin wurde festgestellt, dass SPL zur Umsetzung verschiedener Hardware-Varianten einer HPL eingesetzt werden. Jedoch weist der zur Realisierung dieser SPL eingesetzte Präprozessor cpp Schwächen auf, sodass dessen Eignung zur Umsetzung von SPL angezweifelt werden kann. Ein weiterer Grund für die Untersuchung ist die mangelnde Produktivität bei der Umsetzung von Software-Projekten für eingebettete Systeme. Laut Johansson [Joh03] liegt die Produktivität in einem Teilaspekt der Software-Entwicklung (Gerätetreiber-Programmierung) bei nur einem Viertel im Vergleich zu gewöhnlicher Software-Entwicklung. Aufgrund dessen werden im folgenden Kapitel verschiedene Lösungsansätze zur Entwicklung von Gerätetreibern vorgestellt und im Einzelnen untersucht. Gegenstand der Untersuchung sind die Auswirkungen der einzelnen Lösungsansätze auf die Programmgröße und die erzielte Wiederverwendbarkeit der entwickelten Software-Module.

Zusammenhang Gerätetreiber und HPL/SPL

Die Untersuchung der Abbildung von HPL anhand von Gerätetreibern ist hinreichend, da Gerätetreiber die Schnittstelle zu verschiedenen Hardware-Konfigurationen einer HPL bilden und diese im jeweiligen Software-Projekt repräsentieren. Eine Hardware-Konfiguration beschreibt dabei jeweils eine definierte Zuordnung von Geräten zum Mikroprozessor des eingebetteten Systems. Die Vielzahl verschiedener Hardware-Konfigurationen, die auf das breite Anwendungsfeld eingebetteter Systeme zurückgeht, legt den Schluss nahe, Gerätetreiber als SPL zu entwickeln. Damit können Veränderungen an der Konfiguration in der Software aufgefangen werden. Die an dieser Stelle auftretenden Interaktionspunkte zwischen HPL und den dazugehörigen SPL stellen ein weiteres Ergebnis dieses Kapitels dar.

4.1 Komponenten für die Evaluierung

Für die Betrachtung verschiedener Ansätze zur Umsetzung von Gerätetreibern müssen passende Beispiele für Komponenten ausgewählt werden. Die Umsetzung verschiedener Gerätetreiber erfolgt in diesem Kapitel exemplarisch am *Liquid Crystal Display (LCD)* 1640[1] von ANAG VISION [DST01] und den jeweils auf der Platine der eingebetteten Systeme vorhandenen *Light Emitting Diodes (LED)*. Beide Komponenten kommen vielfach in unterschiedlichen Hardware-Konfigurationen eingebetteter Systeme zum Einsatz und sind auch Bestandteil der Systeme, die für diese Arbeit eingesetzt werden. Dabei treten sie in den verwendeten Systemen einzeln (LCD) oder mehrfach (LED) auf. Gleichwohl die Gerätetreiber beider Komponenten im Umfang gering ausfallen, sind sie doch ausreichend, um Ansätze zur Entwicklung wiederverwendbarer Gerätetreiber und deren Effizienz zu zeigen. Darüber hinaus können an diesen Beispielen Interaktionspunkte zwischen HPL und SPL gezeigt werden. Alle Untersuchungen finden auf der Basis der AVR-Modellreihe des Herstellers Atmel statt. Betrachtungen unterschiedlicher Modellreihen oder Systeme verschiedener Hersteller benötigen umfangreichere Untersuchungen, die nicht im Rahmen dieser Arbeit erfolgen können.

Hardware-Konfiguration

Die eingangs dieses Kapitels genannten Hardware-Konfigurationen sollen zunächst näher erläutert werden. Dafür wird die Abbildung 4.1 herangezogen.[2] Alle Geräte die innerhalb einer Konfiguration einer HPL genutzt werden, sind über eine definierte Anzahl an Daten- und Steuerleitungen mit den Anschlusskontakten des Mikroprozessors verbunden. Die Zuordnung ist für ein eingebettetes System fest definiert (statische Konfiguration). Jedoch sind die Anschlusskontakte auf Seiten des Prozessors nicht fest vorgegeben. Infolgedessen können die Daten- und Steuerleitungen aus der Abbildung 4.1 mit anderen Anschlusskontakten des Prozessors verbunden sein.

Ursachen für Variationen in Hardware-Konfigurationen

Unterschiedliche Varianten gehen auf Änderungen an der Hardware-Konfiguration eines Systems zurück. Eine Hauptursache ist die Verwendung eines anderen Prozessors. Dieser verfügt i. d. R. über andere Anschlusskontakte, sodass eine bestehende Hardware-Konfiguration nicht übernommen werden kann. Darüber hinaus können zusätzliche oder andere Komponenten eine andere Hardware-Konfiguration notwendig machen. Dies geht auf die Tatsache zurück, dass die Anschlusskontakte des Prozessors, die für die Verbindung zu den Komponenten benutzt werden, häufig mit einer zweiten Belegung versehen sind. Diese Belegung setzt eine alternative Funktion um, die nicht selten gegensätzlich zur ersten Funktionalität steht.

Das LCD weist selbst eine zusätzliche Variation auf, die Einfluss auf die Hardware-Konfiguration hat. Das Display lässt sich sowohl mit einem 4- als auch 8-Bit breiten Datenbus betreiben.[3] Dafür werden neben den Daten- und Steuerleitungen des LCD, die bereits für den 4-Bit Betrieb eingesetzt werden vier weitere Datenleitungen verwendet (Abbildung 4.1, Markierung 8-Bit). Demzufolge ändert sich die Hardware-Konfiguration

[1] Das LCD wird für die Realisierung einfacher Anzeigen eingesetzt.

[2] Ein ähnliches Bild ergibt sich für die ebenfalls untersuchten LED. Der Umfang der Daten- und Steuerleitungen fällt aufgrund der einfacheren Struktur geringer aus.

[3] Der Datenbus wird bspw. zur Übertragung von Steuerzeichen.

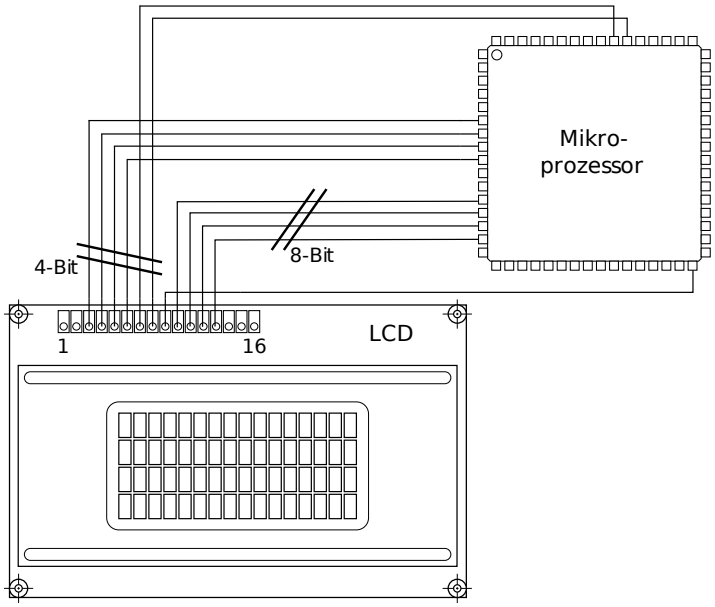


Abbildung 4.1: Anschlusskonfiguration LCD-Display an Mikroprozessor

des Systems, da zusätzliche Anschlusskontakte des Prozessors verwendet werden. Diese Variation hat zudem Einfluss auf den Gerätetreiber, da die zu übertragenden Datenwörter an das LCD nicht getrennt in zwei Halb-Bytes übertragen werden müssen. Die verschiedenen Variationen von Hardware-Konfigurationen, die zuvor genannt wurden, zeigen die Notwendigkeit für einfache adaptierbare Gerätetreiber. Der Gerätetreiber selbst umfasst dabei neben der Initialisierung einer Komponente eine Reihe von Funktionen für dessen Betrieb.

Bevor die Lösungsansätze im Einzelnen vorgestellt werden, erfolgt eine Übersicht zur Programmierung eingebetteter Systeme. Die Ausführungen sind für die Erläuterungen der einzelnen Lösungsansätze und für die abschließende Diskussion notwendig. Dieser Abschnitt kann jedoch nur einen Einblick in diese Thematik geben. Für detailliertere Ausführungen sei auf [Bar99, Bar06, BHK02, Gan91, Sch05] verwiesen.

4.2 Einführung Cross-Plattform Entwicklung

Programmübersetzung

Eingebettete Systeme auf Basis der AVR-Modellreihe besitzen nicht die notwendigen Ressourcen zur Entwicklung von Programmen auf dem System selbst. Aus diesem Grund werden Programme für diese Systeme auf einem Computer entwickelt und für eine gegebene AVR-Zielplattform übersetzt. Dafür wird in dieser Arbeit der *avr-gcc* Cross-Compiler[4] in der Version 4.0.2 eingesetzt. Der Cross-Compiler ist Bestandteil einer Verarbeitungskette (engl. Toolchain), die aus der Programmübersetzung (Kompilierung), dem Linken und Überspielen auf den Mikroprozessor besteht. Mit Hilfe der

[4] http://gcc.gnu.org/

Programmiersprachen zur Evaluierung

Toolchain können Quellprogramme der Programmiersprachen C und C++ für die AVR-Modellreihe übersetzt werden. Beide Sprachen finden laut [Bar99, BHK02, Zur00] in der Programmierung eingebetteter Systeme die weiteste Verbreitung und werden vom Hersteller der AVR-Modellreihe direkt unterstützt.

Die Toolchain zur Programmübersetzung wird um die beiden Precompiler AspectC++[5] und FeatureC++[6] ergänzt (Abbildung 4.2). Infolgedessen können AspectC++- und FeatureC++-Quellen einzeln und in Kombination verarbeitet werden. Die beiden Precompiler erzeugen aus den jeweiligen Quellen den C++-Sourcecode, der mit Hilfe des C++-Compilers *avr-gcc* für eine entsprechende AVR-Zielplattform zu Objektdateien übersetzt werden kann (Abbildung 4.2 (a)). Im Weiteren wird mit Hilfe des Linkers aus der Toolchain ein ausführbares Programm *elf* erzeugt. Anschließend erfolgt das Überspielen des erstellten Programms in den Programmspeicher des Mikroprozessors mittels *avrdude* (Abbildung 4.2 (b)). Während der Programmübersetzung durch den *avr-gcc*-Compiler treten eine Reihe von Optimierungen auf, die nachfolgend besprochen werden. Diese Optimierungen sind für die Programmierung eingebetteter Systeme unerlässlich, da sie die Grundlage zur Generierung effizienter Programme bilden. Ohne ihren Einsatz werden selbst bei einfachen Programmen die zur Verfügung stehenden Ressourcen schnell überschritten.

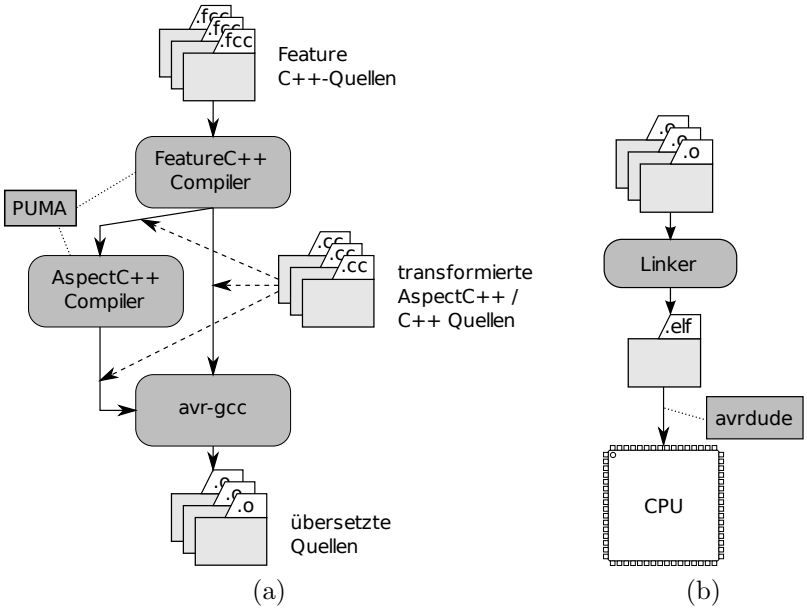


Abbildung 4.2: Übersetzung der Programmquellen (adaptiert aus [Ape07]); Linken und Überspielen von Programmen

Optimierungen Compiler

Der Cross-Compiler *avr-gcc* besitzt über 100 Optimierungsparameter[7], mit denen Einfluss auf den Übersetzungsprozess eines Programms genommen werden kann. Zur

[5] AspectC++ Version 1.0pre3
[6] FeatureC++ Version vom 10.04.2008
[7] Die Anzahl geht auf das Handbuch des Compilers *avr-gcc* zurück.

besseren Handhabung dieser Parameter werden die meisten Optionen in vier Optimierungsstufen zusammengefasst. Sie reichen von `-O0` (keine Optimierung) bis `-O3` (höchste Optimierungsstufe). Die zusätzliche Stufe `-Os` weist den Compiler an, die Programmgröße nach Möglichkeit zu reduzieren. Mit `-O2` wird in erster Linie nur die Programmlaufzeit optimiert. Sie umfasst die meisten Optimierungsparameter und stellt einen guten Kompromiss zwischen Programmlaufzeit und Programmgröße dar. Durch die höchste Optimierungsstufe (`-O3`) werden alle möglichen Optimierungen bzgl. der Programmlaufzeit angewendet. Dies führt aber häufig auch zu größeren Programmen.

Debug-Tools

Neben den Programmen zur Übersetzung von Quellcode, sind auch Werkzeuge zur Auswertung und zum Debuggen von Programmen Bestandteil der Toolchain. Die in dieser Arbeit gezeigten Abbildungen mit Assembler-Befehlen stammen aus der Auswertung übersetzter Programme mit Hilfe des Programms *avr-objdump*. Mit *avr-size* steht ein Programm zur Verfügung, mit dem der Speicherverbrauch eines Programms bestimmt werden kann. Die Abbildung 4.3 zeigt die Ausgabe des Programms. Bei der Auflistung des Speicherverbrauches wird in Programmspeicher *text* und statisch zugewiesenem Arbeitsspeicher für Variablen *data* und *bss* unterschieden. Mit *data* und *bss* werden jeweils initialisierte bzw. nicht initialisierte Variablen separat aufgelistet. Die Kennzahl *dec* gibt den vollständigen Speicherverbrauch an. Der Speicherverbrauch eines Programms wird bei der abschließenden Diskussion der vorgestellten Lösungsansätze verwendet. Er dient als Indikator für die Effizienz der jeweiligen Lösung.

```
1 text    data     bss     dec     hex filename
2 3654      38      18    3710     e7e main.elf
```

Abbildung 4.3: Beispielausgabe *avr-size*

Hinweise zu den Beispielprogrammen

Alle in diesem Kapitel folgenden Beispielprogramme werden mit Optimierungen übersetzt. Dabei sind die einzelnen Programme in erster Linie auf eine geringe Programmgröße optimiert. Obwohl unterschiedliche Mikroprozessoren der AVR-Modellreihe zur Verfügung stehen, werden alle Beispielprogramme für ein Referenzsystem übersetzt. Dies ist notwendig, da geringe Unterschiede zwischen den einzelnen Systemen Auswirkungen auf die übersetzten Programme haben (Abschnitt 2.2 und 2.3.1, S. 7 bzw. 8).

4.3 Fallstudie Gerätetreiber

In den folgenden Abschnitten werden verschiedene Treiberimplementierungen für das LCD und die LED in einer Fallstudie vorgestellt. Dafür kommen unterschiedliche Programmiersprachen zum Einsatz. Diese stehen stellvertretend für verschiedene Programmierparadigmen, die laut Kuhlemann et al. [KLA06] Vor- und Nachteile bei der Umsetzung von Software-Projekten haben. Aus diesem Grund werden die im Weiteren vorgestellten Ansätze auf ihre Eignung zur Umsetzung der Gerätetreiber untersucht. Ziel dieser Untersuchung sind Empfehlungen an Programmierer eingebetteter Systeme

zur Erstellung von Gerätetreibern. Die Eignung der verschiedenen Ansätze wird anhand verschiedener Bewertungskriterien ermittelt. In Kuhlemann [Kuh06] wurden dazu bereits unterschiedliche Kriterien definiert, die auch in dieser Arbeit verwendet werden. Dazu gehören im Einzelnen:

- Performance (Leistungsfähigkeit),
- Ressourcenverbrauch,
- Modularisierbarkeit,
- Wiederverwendung,
- Erweiterbarkeit und
- Zeitpunkt der Adaptierung.

Zusammenfassung und Einschränkung der Kriterien

Eine vollständige Untersuchung der genannten Kriterien kann im Rahmen dieser Arbeit nicht erfolgen. Zur Vereinfachung werden einige Eigenschaften zusammengefasst und zusammenhängend betrachtet. Die Performance und der Ressourcenverbrauch stehen in Korrelation, da bspw. kleinere Programme eine schnellere Ausführung nach sich ziehen.[8] Im weiteren Verlauf werden Modularisierbarkeit, Wiederverwendbarkeit und Erweiterbarkeit zusammenhängend für einzelne Programmieransätze diskutiert. Modularisier- und Erweiterbarkeit stehen laut Kuhlemann [Kuh06] *„in engem Zusammenhang“*. Die Aufteilung von Software in Modulen und deren Erweiterbarkeit spielt auch bei der Wiederverwendung eine Rolle. In der abschließenden Bewertung werden Modularisierbarkeit, Wiederverwendung und Erweiterbarkeit jedoch einzeln aufgeführt, da die genutzten Programmiersprachen laut Kuhlemann [Kuh06] unterschiedliche Stärken und Schwächen in diesen Teilbereichen aufweisen. Der Zeitpunkt der Adaptierung ist bei der Betrachtung der Gerätetreiber nicht wesentlich. In allen Beispielen sind Konfigurationen zur Übersetzungszeit ausreichend, weshalb keine Betrachtungen von Adaptionen zur Laufzeit erfolgen. Dies steht im Widerspruch zu Evaluierungen, die in Kuhlemann [Kuh06] und Lohmann et al. [LSSP06] gemacht wurden. Darin stellen die Autoren verschiedene Lösungsansätze zur Entwicklung von SPL gegenüber, die teilweise auf Laufzeitadaptionen beruhen. Die Ergebnisse der Evaluierungen zeigen, dass Ansätze, die auf Laufzeitadaptionen basieren, in Bezug auf die Programmgröße und -laufzeit schlechter abschneiden. Die Ursache für das schlechtere Abschneiden ist die zusätzliche Programmlogik, die zur Umsetzung der Laufzeitadaptionen benötigt wird und für den Mehrverbrauch an Programmspeicher verantwortlich ist (Abschnitt 2.3.4 und 2.4.2, S. 10 und 13).

In der Programmierung eingebetteter Systeme wird jedoch versucht auf Adaptionen zur Laufzeit zu verzichten, da Laufzeitadaptionen infolge zusätzlicher Programmlogik Overhead nach sich ziehen. Sofern möglich wird die Entscheidung einer Konfiguration bereits zur Übersetzungszeit getroffen. Dafür wird neben den Mitteln, die die

[8] Eine differenziertere Betrachtung erfolgt anhand gegebener Beispiele.

eingesetzte Programmiersprache zur Verfügung stellt, auch auf den Präprozessor cpp zurückgegriffen (Abschnitt 3, S. 23).

4.3.1 Gerätetreiber in C

Für den ersten Lösungsansatz zur Entwicklung der Gerätetreiber wird die Programmiersprache C herangezogen. Zur Umsetzung von Software-Projekten für die AVR-Modellreihe wird häufig die Bibliothek AVR LibC verwendet. Diese wurde bereits in Abschnitt 3.2.1 (S. 25) eingeführt. Die in der Bibliothek definierten Hardware-Parameter[9] unterstützter Prozessoren werden häufig direkt in der Programmierung eingesetzt. Die Abbildung 4.4 zeigt den Quelltext zur Initialisierung und Nutzung der LED zusammen mit den dazugehörigen Assembler-Anweisungen (Abbildung (a) bzw. (b)). Es zeigt sich, dass sowohl die Initialisierung (Abbildung 4.4 (a), Z. 6 - 9 und Z. 18) als auch das Einschalten der LED (Abbildung 4.4 (a), Z. 12 - 15 und Z. 19 - 20) zu einfachen Load-, Store- (Abbildung 4.4 (b), Z. 7 - 10) und Bit-Operationen aufgelöst werden (Abbildung 4.4 (b), Z. 11 - 12). Die Abbildung belegt weiter, dass keine Funktionsaufrufe vorhanden sind. Dies geht auf Optimierungen des Compilers zurück, der die Funktionalität der aufgerufenen Methode in den Kontext der aufrufenden Methode integriert. Daher ist das erzeugte Programm gleichsam performant und speicherschonend.

Variationen und Wiederverwendung

Änderungen der Anzahl eingesetzter LED oder Variationen in der Anschlusskonfiguration haben zur Folge, dass der komplette Quellcode angepasst werden muss. Damit wirken sich Änderungen in der HPL auf den Gerätetreiber aus (Interaktionspunkt zwischen HPL und SPL). Neue Varianten können bspw. mit Hilfe von Präprozessordirektiven umgesetzt werden (Abbildung 4.5). Die Abbildung zeigt den Quellcode für zwei Hardware-Konfigurationen. Der zu einer Konfiguration gehörende Quellcode wird mit Hilfe des Makros *BOBBYBOARD* konfiguriert. Jedoch birgt dieser Ansatz zwei Nachteile. Zum einen führt dieses Vorgehen zu einer geringen Wiederverwendbarkeit, da nur Quellcode unter den verschiedenen Varianten geteilt werden kann, der unabhängig von direkt angesprochenen Hardware-Parametern (*PORTB* und *PORTC*) ist. Die hinzugefügte Funktionalität wird direkt im Quellcode konfiguriert. Bestehende Gemeinsamkeiten zwischen Ausprägungen verschiedener Implementierungen werden nicht ausgenutzt. Zum anderen zeigt das Beispiel bereits im Ansatz, dass durch das Hinzufügen weiterer Hardware-Konfigurationen die Übersichtlichkeit des Quellcodes sinkt und der Aufwand für Wartung und Pflege ansteigt.

Erhöhung der Wiederverwendbarkeit

Zur Verbesserung der Wiederverwendbarkeit des Quellcodes ist die Trennung der Treiberfunktionalität von den Speicheradressen notwendig. Präprozessoranweisungen bieten eine Möglichkeit diese Trennung umzusetzen. Die Abbildung 4.6 zeigt ausschnittsweise die Definition von Parametern für das LCD. *LCD_PORT* u. a. legen jeweils symbolische Adressen fest. Diese werden bei der Übersetzung des Programms im Quelltext durch die Hardware-Parameter des Prozessors *PORTA* u. a. ersetzt.

[9] Hardware-Parameter stehen für Registernamen hinter denen Funktionalitäten des Prozessors stehen. Ein Beispiel sind die Register *DDRA* und *PORTA* (Abbildung 4.4, S. 42 Z. 7 und 8).

```
#include <avr/io.h>
#define sbi(ADDRESS,BIT) (ADDRESS |= (1<<BIT))
#define cbi(ADDRESS,BIT) (ADDRESS &= ~(1<<BIT))

// Initialisierung LEDs
static inline void initLed() {
  DDRA  = 240;
  PORTA = 255;
}

// setze Bit -> LED an.
static inline void ledOn(uint8_t i) {
   if (i>3) return;
   sbi(PORTA, i);
}

int main() {
   initLed();
   ledOn(1);
   ledOn(3);
   while(1)
     ;
}
```

(a) Treiberimplementierung und Beispiel

```
...
000000d6 <main>:
   d6: cf ef   ldi   r28, 0xFF  ; 255
   d8: d0 e1   ldi   r29, 0x10  ; 16
   da: de bf   out   0x3e, r29  ; 62
   dc: cd bf   out   0x3d, r28  ; 61
   de: 80 ef   ldi   r24, 0xF0  ; 240 // Initialisierung Anfang
   e0: 81 b9   out   0x01, r24  ; 1
   e2: 8f ef   ldi   r24, 0xFF  ; 255
   e4: 82 b9   out   0x02, r24  ; 2   // Initialisierung Ende
   e6: 11 9a   sbi   0x02, 1    ; 2   // schalte LED 1 ein
   e8: 13 9a   sbi   0x02, 3    ; 2   // schalte LED 3 ein
...
```

(b) Assembler-Programm zum Treiber

Abbildung 4.4: Gerätetreiber LED in C

Während dieser Ansatz praktikabel für Komponenten ist, die nur einmal im System auftauchen, wird die Treiberimplementierung für eine variable Anzahl an LED ungleich aufwendiger. Für jede Komponente wäre eine separate Treiberimplementierung erforderlich, die jeweils durch die Definition symbolischer Adressen mittels Präprozessoranweisungen konfiguriert wird. Dabei kommt es insbesondere nicht mehr zur gewünschten Trennung von Treiberfunktionalitäten und Speicheradressen. Jede Treiberimplementierung steht somit in Abhängigkeit zur Anzahl der auftretenden Komponenten. Damit ergeben sich die gleichen Nachteile wie bei der Umsetzung verschiedener Hardware-Konfigurationen einer HPL (Abbildung 4.5). Ein Ansatz, der die Probleme umgeht, die sich aus der Umsetzung von Treibern für gleichartige Komponenten ergeben, ist das Klassenkonzept aus der OOP. Dadurch lässt sich die Kapselung der

```
1  #define BOBBYBOARD
2
3  void ledOn(uint8_t i) {
4  #ifdef BOBBYBOARD
5          if (i>3) return;
6  #else
7          if (i>1) return;
8  #endif
9
10 #ifdef BOBBYBOARD
11         cbi(PORTB, i+4);
12 #else
13         cbi(PORTC, i+6);
14 #endif
15 }
```

Abbildung 4.5: Umsetzung von Software-Varianten für eine HPL

```
1  #ifndef LCD_H_CONFIG
2  #define LCD_H_CONFIG
3
4  #define LCD_PORT   PORTA   //LCD PORT
5  #define LCD_DDR    DDRA    //LCD DDR register
6  #define LCD_DATA0_PORT    LCD_PORT
7  #define LCD_DATA1_PORT    LCD_PORT
8  ...
```

Abbildung 4.6: Konfiguration der Speicheradressen mittels Präprozessoranweisungen

Treiberfunktionalität für gleichartige Komponenten erzielen.

4.3.2 Kapselung der Komponenten in Klassen - C++

Mit Hilfe von Klassen lässt sich die Funktionalität gleicher Objekte (Instanzen) kapseln. Ein Klasse stellt eine Schablone für daraus hervorgehende Objekte dar und fasst gemeinsame Datentypen und Funktionen zusammen. Im Fall der Treiberimplementierung durch Klassen hält diese für jede zum Gerät gehörende Daten- oder Steuerleitung eine Variable vor, mit der der Zugriff auf die Speicheradresse der Leitung erfolgt.

Zur Veranschaulichung dient die Abbildung 4.7. Für die Initialisierung eines LED-Objekts (Abbildung 4.7 (a)) werden die Speicheradressen (Hardware-Parameter) an den Konstruktor der LED-Klasse übergeben. Diese werden innerhalb verschiedener Funktionen (bspw. *ledOn*) verwendet. Obwohl eine Reihe von Attributen für die Klassendefinition von LED verwendet wird, kommen diese im erzeugten Programm (Abbildung 4.7 (b)) nicht vor. Der Grund dafür liegt in der Möglichkeit des Compilers, die in den Variablen (*_ddr*, *_port* und *_bit*) zwischengespeicherten Werte, zu realen Adressen (Hardware-Parameter) des Prozessors aufzulösen. Infolgedessen reduziert sich die in Abbildung 4.7 (a) aufgelistete Initialisierung (Z. 8 - 13 und Z. 22) zu einfachen Assembler-Befehlen (Abbildung 4.7 (b), Z. 7 - 8). Das gleiche gilt für das Einschalten der LED. Damit erfolgt die Reduktion der Gerätetreiberklasse wie zuvor bei der C-Implementierung

```
1  class Led {
2  private:
3    volatile uint8_t & _ddr;
4    volatile uint8_t & _port;
5    uint8_t _bit;
6  public:
7    // Initialisierung
8    Led(volatile uint8_t & ddr, volatile uint8_t & port,
9        uint8_t bit) :
10     _ddr(ddr), _port(port), _bit(bit) {
11     _ddr  |= (1<<_bit);
12     _port &= ~(1<<_bit);
13   }
14
15   // schalte LED ein
16   void ledOn() {
17     _port |= (1<<_bit);
18   }
19 ...
20
21 int main(int argc, char **argv) {
22   Led l0(DDRA, PORTA, 0);
23   l0.ledOn();
24 ...
```

(a) Treiberimplementierung und Beispiel

```
1  ...
2  000000d6 <main>:
3    d6: cf ef   ldi  r28, 0xFF  ; 255
4    d8: d0 e1   ldi  r29, 0x10  ; 16
5    da: de bf   out  0x3e, r29  ; 62
6    dc: cd bf   out  0x3d, r28  ; 61
7    de: 08 9a   sbi  0x01, 0    ; 1  // Initialisierung Anfang
8    e0: 10 98   cbi  0x02, 0    ; 2  // Initialisierung Ende
9    e2: 10 9a   sbi  0x02, 0    ; 2  // schalte LED 0 ein
10 ...
```

(b) Assembler-Programm zum Treiber

Abbildung 4.7: Gerätetreiber LED in C++

(Abschnitt 4.3.1, S. 41) zu einfachen Bit-Operationen.

Geräteschnittstellen

Durch den Einsatz von Klassen zur Umsetzung von Gerätetreibern konnte eine höhere Wiederverwendung erzielt werden. Bei der Implementierung des LCD-Gerätetreibers zeigt sich jedoch, dass die 4-Bit/8-Bit Variante nur unzureichend umgesetzt werden konnte. Dies äußert sich darin, dass entweder eine Neuimplementierung des Treibers notwendig war oder auf den Einsatz von Präprozessoranweisungen zurückgegriffen werden musste.[10] Jedoch treffen alle Einschränkungen bzgl. des Präprozessoreinsatzes, die bereits in Abschnitt 4.3.1 (S. 41) diskutiert wurden, glei-

[10] Die Neuimplementierung des Treibers lässt sich ebenfalls durch die Nutzung von Vererbung in OOP vermeiden. Die abweichende Funktionalität der 8-Bit-Variation lässt sich in einer separaten Klasse, die von einem Basisfunktionstreiber für das LCD erbt, umsetzen. Nachteil dieses Ansatzes ist der exponentielle Aufwand bei der Implementierung, wenn weitere abweichende Funktionalitäten und Varianten umgesetzt werden sollen. Aus diesem Grund wird dieser Ansatz nicht weiter erörtert.

chermaßen an dieser Stelle zu.

In [CRKH05] et al. wird ein Ansatz beschrieben, der eine unabhängige Implementierung von Gerätetreibern erlaubt. Die Idee dazu wird mit Hilfe der Abbildung 4.8 genauer erläutert. Alle Komponenten, die an den Mikroprozessor angeschlossen sind, können im Speicher durch Hardware-Parameter angesprochen werden. Werden alle Hardware-Parameter, die von einem Gerät verwendet werden, in einer Schnittstelle (Interface) zusammengefasst, kann darauf aufbauend eine Treiberimplementierung erfolgen, die unabhängig vom Gerät selbst erfolgt und damit in verschiedenen Hardware-Konfigurationen einsetzbar ist. Zur Umsetzung dieser Schnittstellen[11] eignen sich Klassen. Die Nutzung wird anhand von Klassen in C++ erläutert, da die nachfolgenden Lösungsansätze auf C++ basieren. Gleichwohl sei erwähnt, dass die Nutzung von Geräteschnittstellen gleichermaßen in C möglich ist.

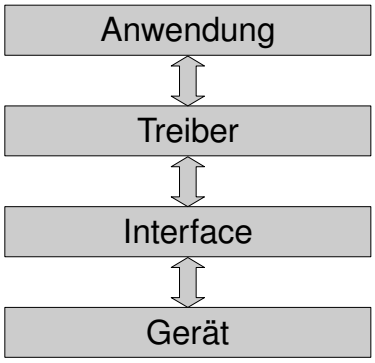


Abbildung 4.8: Schichtenmodell für die Implementierung von Treibern und Anwendungen

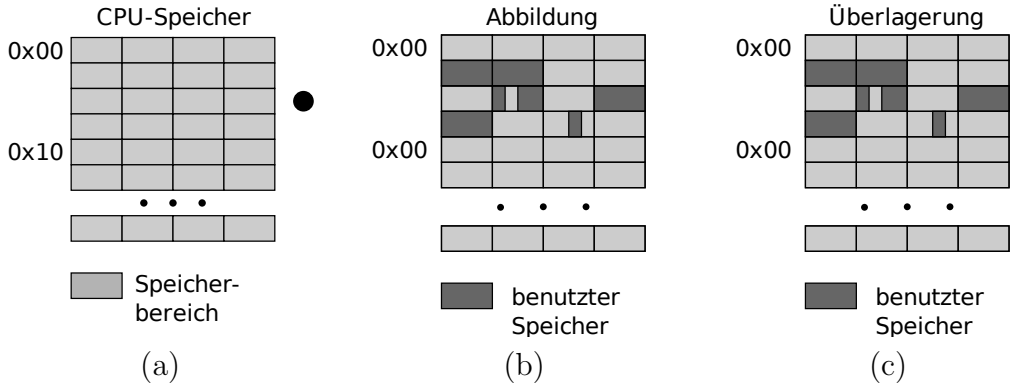


Abbildung 4.9: Abbildung einer Gerätestruktur auf den Speicher

[11] Im Weiteren werden diese Schnittstellen als Geräteschnittstellen bezeichnet.

Klassen als Gerätestrukturen

Die Umsetzung von Geräteschnittstellen in C++ wird mit Hilfe der Abbildung 4.9 erläutert.[12] Die Geräteschnittstelle einer Komponente wird durch eine Datenstruktur repräsentiert. Diese fasst alle zum Gerät gehörenden Hardware-Parameter in einer Klasse zusammen und bildet dadurch eine Schnittstelle, die zur Implementierung eines allgemeingültigen Gerätetreibers verwendet werden kann. Zur Veranschaulichung dient die Abbildung 4.9. Im linken Bild (Abbildung (a)) ist der Speicherbereich des Prozessors dargestellt. Die Bildmitte (Abbildung (b)) zeigt die Geräteschnittstelle zusammen mit den zu ihr gehörenden Blöcken (dunkle Markierung). Durch die Überlagerung (•) der Geräteschnittstelle auf den Speicherbereich des Prozessors ergibt sich die Abbildung 4.9 (c). Die Blöcke der Geräteschnittstelle definieren symbolische Adressen, die für die Treiberimplementierung genutzt werden können. Um existierende Gerätetreiber für eine Hardware-Konfiguration verfügbar zu machen, ist es ausreichend alle notwendigen Geräteschnittstellen für die Variante zu definieren. Die Abbildung 4.10 zeigt ausschnittsweise die Definition einer Schnittstelle. Ausgehend von der Startadresse *0x00* werden Variablen für verwendete Blöcke innerhalb der Geräteschnittstelle deklariert. Dabei verläuft die Definition fortlaufend von der Startadresse, wobei nicht verwendete Bereiche durch Platzhalter (*_gap01[0x29]*) kenntlich gemacht werden. Der häufige Zugriff auf einzelne Bits lässt sich anhand von Bitfeldern realisieren [Man04]. Dadurch können einzelne Bits in der Geräteschnittstelle adressiert werden (benanntes Bitfeld).

```
1  class AT90CAN128LCD4Bit {
2  private:
3    volatile uint8_t _gap01[0x29]; // Platzhalter
4  public:
5  ...
6    volatile uint8_t
7                   : 4, // anonymes Bit-Feld
8      LCDDDR_D4    : 1, // benanntes Bit-Feld
9      LCDDDR_D5    : 1,
10     LCDDDR_D6    : 1,
11     LCDDDR_D7    : 1;
12 ...
```

Abbildung 4.10: Definition Gerätestruktur zur Abbildung einer Geräteschnittstelle

Treiberimplementierungen

Nachdem zuvor verschiedene Geräteschnittstellen für gleichartige, externe Komponenten definiert wurden, folgt nun die Treiberimplementierung. Aufgrund der Kapselung der Komponenten und der dadurch vorhandenen Abstraktion lassen sich nun Techniken der OOP, AOP und FOP einsetzen. Dabei fällt auf, dass die Geräteschnittstellen eines Typs (z. B. LED) mit der gleichen Treiberimplementierung erweitert werden müssen. Die Erweiterung von mehrfach im System vorhandenen Komponenten mit der gleichen Funktionalität führt zu einem homogenen Crosscut. Aus diesem Grunde scheidet Vererbung zur Umsetzung der Treiberimplementierung für die einzelnen Klassen aus. Der Einsatz der Vererbung führt hierbei zu Quellcode-Replikation zusammen mit

[12] Die Nutzung dieses Ansatzes ist auch in C möglich. Auf die abweichende Anwendung wird in dieser Arbeit nicht eingegangen.

einem höherem Implementierungs- und Wartungsaufwand. Die Erweiterung der einzelnen Geräteschnittstellen mit dem notwendigen Funktionsumfang stellt eine statische Erweiterung dar. Zwei Ansätze zur Umsetzung dieser Erweiterung werden nachfolgend vorgestellt.

4.3.3 Statische Template-Metaprogrammierung

Für die Implementierung des Gerätetreibers eignet sich Statische Template-Metaprogrammierung. Statische Template-Metaprogrammierung gehört in den Bereich der Generischen Programmierung. Mit Hilfe von Templates kann Funktionalität für eine Menge gleicher Basisklassen definiert werden. Templates sind Klassenschablonen, die anhand von Datentypen parametrisiert werden können. Während der Kompilierung werden alle benötigten Template-Varianten durch Quellcode-Replikation erzeugt und übersetzt [Str00]. Die aus Templates hervorgehenden Klassen teilen die gleiche Funktionalität, wodurch der Aufwand bei der Entwicklung verringert wird. Typische Beispiele für Templates sind Implementierungen von Container-Klassen, wie z. B. list und vector aus der *Standard Template Library (STL)*. Für eine detaillierte Einführung in Templates sei auf Stroustrup [Str00] und Czarnecki et al. [CE00] verwiesen. Zur Umsetzung des Gerätetreibers wird auf den Template-Mechanismus von C++ zurückgegriffen.[13]

Template als Gerätetreiber

Im Beispiel der Umsetzung eines Gerätetreibers für das LCD-Modul enthält das Template die Treiberimplementierung. Die Instantiierung des Templates erfolgt anhand der zuvor definierten Geräteschnittstellen aus dem vorherigen Abschnitt. Während des Übersetzungsprozesses wird für jede benutzte Schnittstelle eine Klasse (Template-Ausprägung) erzeugt. Die Abbildung 4.11 zeigt die Initialisierung und Nutzung eines Gerätetreibers auf Basis eines Templates. Die Gerätestruktur wird bei der Initialisierung der Template-Klasse als Parameter (*LCDComponent*, Z. 1) übergeben. Für die Initialisierung des LCD-Objekts erfolgt die Überlagerung der übergebenen *LCDComponent* Schnittstelle auf die Startadresse des Prozessors im Speicher (Z. 11). Damit wird der Ansatz, der bereits in der Abbildung 4.9 dargestellt wurde, umgesetzt. Die Treiberimplementierung geht nun auf die definierten Variablen und Bitfelder aus der Geräteschnittstelle zurück (Z. 7 - 8).

Effizienz Template Einsatz

Der Einsatz von Templates wird häufig in Zusammenhang mit Quellcode-Explosion (engl. Code-Bloat) gebracht. Die Bezeichnung geht auf die durch den Generierungsprozess erzeugten Klassen zurück, die eine Vergrößerung des Programmcodes zur Folge haben können. Gemeinsame Funktionen und Eigenschaften, die zuvor noch in Templates gekapselt waren, werden nicht unter den generierten Klassen geteilt. Obwohl dies gegen den Einsatz von Templates für die Programmierung spricht, ergibt sich das gleiche Ergebnis, das sich bereits in den Abbildungen 4.4 und 4.7 (S. 42 bzw. 44) zeigte. Die in den Geräteschnittstellen definierten Variablen stellen wiederum nur den Zugriff zu symbolischen Adressen des Prozessors bereit. Diese Variablen können wie in den Beispielen zuvor aufgelöst werden, sodass der Einsatz von Templates nicht zu dem

[13] Im Folgenden auch mit C++ (T) bezeichnet.

```
1  template <typename LCDComponent>
2  class LCD {
3  private:
4    LCDComponent & lcd;
5
6    void waitLCD() {
7      lcd.LCDDDR_D7  = 0;
8      lcd.LCDPORT_D7 = 1;
9  ...
10 public:
11   // Initialisierung
12   LCD() : lcd((*(LCDComponent *) 0x00)) {};
13 ...
```

Abbildung 4.11: Gerätetreiber LCD als C++ (T)

erwarteten Overhead durch Quellcode-Explosion bei der Programmgröße führt (Abbildung 4.12). Es zeigt sich abermals, dass alle Operationen der Treiberimplementierung zu einfachen Speicheroperationen aufgelöst werden (Abbildung 4.12 (b), Z. 7 ff). Damit steht die Template-Lösung den zuvor vorgestellten Lösungen in Bezug auf die Effizienz in nichts nach.

Die aus der Template-Metaprogrammierung generierten Klassen beinhalten den vollen Funktionsumfang der zugrunde liegenden Templates. Auf der Basis von Quellcode-Analysen bestimmt der Compiler welche Bestandteile der generierten Klassen benutzt werden und übersetzt diese. Alle übrigen Bestandteile werden nicht verwendet. Die in Templates definierten Funktionen sind als Inline-Funktion deklariert.[14] Inline-Deklarationen weisen den Compiler an, Funktionsaufrufe durch den im Funktionsrumpf enthaltenen Quellcode zu ersetzen. Dadurch reduziert sich der Overhead eines Funktionsaufrufs, der aus der Sicherung des Prozessorstatus, Funktionsausführung und Wiederherstellung des Prozessorstatus besteht. Inline-Funktionen erhöhen die Ausführungsgeschwindigkeit, weil durch deren Einbettung Anweisungen, die in Verbindung mit dem Funktionsaufruf stehen, eingespart werden können. Der Nachteil, der sich durch das Duplizieren der Funktionsrümpfe ergibt und zu einer Vergrößerung der übersetzten Programme führt, ist in diesem Fall tolerierbar. Die meisten Funktionen enthalten entweder nur wenige Anweisungen oder treten nur einmal im Programm auf. Im Gegenzug würde die alternative Nutzung in Form von Funktionsaufrufen zu mehr Anweisungen und damit zu einem größeren Speicherverbrauch führen.

Erweiterbarkeit und Wiederverwendung

Die Treiberimplementierung auf Basis von Templates führt dazu, dass alle Funktionalitäten in einem Modul gekapselt sind. Durch Typenparametrisierung wird eine hohe Wiederverwendung erreicht. Ein Mangel, der bei der Treiberentwicklung mittels Templates auftrat, sind die Erweiterungsfähigkeiten von Templates. Dies geht darauf zurück, dass für Erweiterungen nur Vererbung oder Template-Spezialisierungen möglich sind.[15]

[14] Dies ist eine allgemeine Einschränkung von Templates. Der Template-Parameter muss bereits zur Übersetzungszeit bekannt sein, da keine spätere Bindung des Template-Parameters an das Template, bspw. während des Linkens von Objektdateien oder zur Laufzeit, möglich ist.

[15] Die Nutzung des Präprozessors cpp wird an dieser Stelle ausgeschlossen. Erweiterungen sollen ausschließlich mit Hilfe von OOP umgesetzt werden.

```
#include "AT90CAN128LCD4Bit.h"
#include "LCD.h"

int main() {
  LCD<AT90CAN128LCD4Bit> lcd; // Initialisierung
  lcd << "initialized";

  while(1)
    ;
}
```

(a) Beispielprogramm Initialisierung und Nutzung LCD

```
...
000000d6 <main>:
  d6:  cf ef  ldi   r28, 0xFF  ; 255
  d8:  d0 e1  ldi   r29, 0x10  ; 16
  da:  de bf  out   0x3e, r29  ; 62
  dc:  cd bf  out   0x3d, r28  ; 61
  de:  80 e4  ldi   r24, 0x40  ; 64   // Initialisierung Anfang
  e0:  9c e9  ldi   r25, 0x9C  ; 156
  e2:  01 97  sbiw  r24, 0x01  ; 1
  e4:  f1 f7  brne  .-4        ; 0xe2 <main+0xc>
  e6:  57 9a  sbi   0x0a, 7    ; 10
  e8:  6c 9a  sbi   0x0d, 4    ; 13
  ea:  6d 9a  sbi   0x0d, 5    ; 13
  ec:  6e 9a  sbi   0x0d, 6    ; 13
  ee:  6f 9a  sbi   0x0d, 7    ; 13
  f0:  86 9a  sbi   0x10, 6    ; 16
  f2:  87 9a  sbi   0x10, 7    ; 16
  f4:  8e 98  cbi   0x11, 6    ; 17
  f6:  8f 98  cbi   0x11, 7    ; 17
  f8:  5f 98  cbi   0x0b, 7    ; 11
  fa:  77 98  cbi   0x0e, 7    ; 14
  fc:  76 98  cbi   0x0e, 6    ; 14
...
```

(b) Assembler-Programm zum Beispiel

Abbildung 4.12: Beispielprogramm Gerätetreiber LCD mit C++ (T)

Beide Erweiterungsmöglichkeiten führen zu einer schlechten Wiederverwendung, da die Umsetzung unabhängiger Features nur durch Quellcode-Replikation erfolgen kann. Zu dem gleichen Ergebnis kommen auch Kuhlemann et al. [KLA06], die aufgrund dieser Tatsache die Kombination der Generischen Programmierung mit anderen Ansätzen vorschlagen, um den Problemen des Template-Einsatzes zu begegnen. Für die Entwicklung eines Gerätetreibers wurde dazu auf die Kombination von Template-Programmierung und FOP zurückgegriffen. Die mangelnde Erweiterbarkeit von Templates ist jedoch nicht der einzige Nachteil dieses Lösungsansatzes.

Nachteile Template-Metaprogrammierung

Die Entwicklung und Nutzung von Templates ist laut Josuttis et al. [JV02] problembehaftet. Die Autoren beschreiben, dass bei der Entwicklung eines Templates sichergestellt werden muss, dass der Template-Parameter (Typ) alle Operationen unterstützt, die innerhalb des Templates in Anspruch genommen werden. Infolgedessen müssen semantische Randbedingungen für den übergebenen Typ definiert werden, um

die falsche Verwendung eines Templates auszuschließen. Darüber hinaus wird in [JV02] angemerkt, dass die fehlerhafte Nutzung von Templates zu unübersichtlichen Fehlermeldungen führt, die die Fehlerbestimmung erschweren. Der Template-Mechanismus in C++ erlaubt die Erstellung eigenständiger Programme, die zur Übersetzungszeit vom Compiler ausgewertet werden (Template-Metaprogramme). Mit Hilfe von Template-Metaprogrammen lassen sich zur Übersetzungszeit Konfigurationen bestimmen. Die Nutzung von Template-Metaprogrammen bedeutet jedoch eine Vermischung von Quellcode, der zur Generierung einer Konfiguration verwendet wird (Konfigurationsebene), mit Quellcode der zur Umsetzung einer Problemlösungsstrategie (Implementierungsebene) benutzt wird. Ferner wird in Czarnecki et al. [CE00] auf die schlechte Lesbarkeit von Template-Metaprogrammen, die mangelnde Unterstützung des Template-Mechanismus in C++-Compilern und den Mehraufwand bei der Programmübersetzung hingewiesen. Demzufolge sind der Template-Metaprogrammierung bezogen auf deren Komplexität und damit Eignung zur Umsetzung von Problemlösungsstrategien Grenzen gesetzt [CE00]. Aufgrund dessen ist von der alleinigen Nutzung statischer Template-Metaprogrammierung abzuraten.

In AspectC++ steht mit Slice-Klassen eine weitere Möglichkeit zur Verfügung den homogenen statischen Crosscut umzusetzen.

4.3.4 Slice-Klassen in AOP

Mit Slice-Klassen wird innerhalb AspectC++ ein Wirkungsbereich definiert, mit dessen Hilfe die statische Struktur von Programmen erweitert werden kann. Slice-Klassen sind Bestandteil von Advice-Blöcken und ihr Wirken wird über Pointcuts gesteuert [Pur05]. Dadurch lassen sich neue Member (Variablen und Funktionen) bestehenden Klassen hinzufügen. Mit Hilfe von Platzhaltern in Pointcut-Ausdrücken können gleichzeitig mehrere Klassen erfasst werden. Slice-Klassen dienen ausschließlich statischen Erweiterungen (Tabelle 2.3, S. 16).

Slice-Klasse vs. Template

Der Slice-Klassen Ansatz ist mit der Template-Metaprogrammierung vergleichbar. Dennoch gibt es einige Unterschiede. Alle durch den Pointcut-Ausdruck erfassten Klassen werden direkt erweitert. Damit stehen alle erweiterten Klassen bereits vor der Kompilierung mit dem C++-Compiler zur Verfügung und werden nicht erst zur Übersetzungszeit erstellt. Die in Abschnitt 4.3.2 (S. 43) definierten Gerätestrukturen lassen sich direkt für Erweiterungen einsetzen. Die Erweiterungen durch Aspekte sind im Gegensatz zur expliziten Erweiterung durch Templates implizit. Die Abbildung 4.13 zeigt die Definition eines Aspekts zur Erweiterung verschiedener LED Gerätestrukturen.

Slice-Klassen setzen auf dem Pointcut-Advice Mechanismus von AOP auf. Alle zu erweiternden Klassen werden im Pointcut-Ausdruck definiert (Abbildung 4.13 (a), Z. 2 und 3). Mit Hilfe von Slice-Klassen lassen sich zwar Konstruktoren hinzufügen, dennoch ist eine separate *init*-Methode erforderlich. Dies trägt dem Umstand Rechnung, dass die Erzeugung des LED-Objekts (Abbildung 4.13 (b), Z. 1) keiner Überlagerung der erweiterten Geräteschnittstelle nach Abbildung 4.9 (S. 45) entspricht. Diese Abbildung ist jedoch notwendig, damit die in der Geräteschnittstelle vorhandenen symbolischen

```
1  aspect LedFunc {
2    pointcut p() = "LED%";
3    // pointcut p() = "LED1" || "LED3";
4
5    advice p() : slice class S {
6    public:
7      // Initialisierung
8      void init() {
9        LEDPORT = 0;
10       LEDDDR  = 1;
11     }
12
13     // schalte LED ein
14     void ledOn() {
15       LEDPORT = 1;
16     }
17 ...
```

(a) Gerätetreiber als Aspekt

```
1 LED0 &led = (*(LED0 *) 0x00);
2 led.init();
3 led.ledOn();
```

(b) Assembler-Programm zum Treiber

Abbildung 4.13: Gerätetreiber LED in AspectC++

Variablen zu den entsprechenden Speicheradressen im Adressraum des Mikroprozessors aufgelöst werden.

Effizienz und Wiederverwendung Slice-Klassen

Dieser Ansatz steht den Lösungen aus den vorherigen Abschnitten mit C, C++ oder C++-Templates in nichts nach. Die Ergebnisse hängen wie bereits zuvor von Optimierungen des Compilers ab. Die in Aspekten umgesetzten Treiberimplementierungen sind für verschiedene Varianten der HPL einsetzbar. Damit ist die in dem Aspekt gekapselte Treiberimplementierung auch in anderen Hardware-Konfigurationen verwendbar.

4.4 Evaluierung Entwicklung Gerätetreiber

Die Kernfrage dieses Kapitels ist die Eignung verschiedener Programmieransätze zur Umsetzung von Gerätetreibern. Bei der Untersuchung wurden Ansätze auf der Basis erweiterter Programmierparadigmen nur verfolgt, wenn dadurch die Fähigkeiten des Paradigmas zur Umsetzung der Funktionalitäten wirklich in Anspruch genommen werden mussten. Die Untersuchung erfolgte anhand verschiedener Gerätetreiber für die LED und das LCD-Modul eingebetteter Systeme. Im Folgenden werden die in Abschnitt 4.3 (S. 39) genannten Kriterien (Ressourcenverbrauch, Modularisierbarkeit, Wiederverwendung und Erweiterbarkeit) vergleichend für die vorgestellten Lösungsansätze untersucht.

LED

Für den Vergleich verschiedener Gerätetreiber für die LED werden Lösungen auf der Basis von C, C++, C++ (T) und AspectC++ herangezogen. Die Ansteuerung

der LED erfordert jeweils nur das Setzen oder Löschen eines Bits. Die Ergebnisse (Tabelle 4.1) zeigen, dass mit eingeschalteten Optimierungen (`-O1`, `-O2` und `-O3`) kein Unterschied in der Programmgröße zwischen den einzelnen Lösungsansätzen festgestellt werden kann. Eine Ausnahme bildet die Optimierungsstufe `-Os`. Hier wird mit Hilfe von Aspekten ein Vorteil von ca. 13 % und 17 % gegenüber C/C++ bzw. C++ (T) erzielt. Der Grund liegt beim Übersetzungsprozess durch den Compiler bei dem anstelle von Inline-Methoden Funktionsaufrufe benutzt werden. Der Mehrverbrauch wird mit Hilfe der Abbildung 4.14 verdeutlicht. Im Quelltext aller Lösungsansätze wird das Einschalten der LED durch eine Funktion umgesetzt. Diese Funktion wird im Fall von C/C++ und C++ (T) bei eingeschalteten Optimierungen (`-Os`) zu einer eigenständigen Funktion mit einem Funktionsaufruf übersetzt (Abbildung 4.14, Z. 1 - 6 und Z. 11 - 12). Dies bedeutet zum einen einen Mehrverbrauch an Programmspeicher, da sich die gleiche Funktionalität durch eine Inline-Funktion (Abbildung 4.14 (b)) umsetzen lässt. Zum anderen fallen die vom Compiler vollzogenen Optimierungen in der Funktion *ledOn* geringer aus, so dass in diesem Fall keine Ersparnis erzielt wird.

```
1  000000d6 <ledOn>:
2    d6:  38 2f   mov   r19, r24
3    d8:  84 30   cpi   r24, 0x04   ; 4
4  ... // + 10 Zeilen Assembler
5    ee:  22 b9   out   0x02, r18   ; 2
6    f0:  08 95   ret
7  ...
8  000000f2 <main>:
9    f2:  cf ef   ldi   r28, 0xFF   ; 255
10 ...
11  102:  81 e0   ldi   r24, 0x01   ; 1
12  104:  0e 94 6b 00   call 0xd6   ; 0xd6 <ledOn>
```

(a) Assembler-Programm mit Funktionsaufruf

```
1    e6:  11 9a   sbi   0x02, 1   ; 2
```

(b) Assembler-Programm mit Inlining

Abbildung 4.14: Funktionsaufruf vs. Inlining

In Bezug auf die Wiederverwendbarkeit lässt sich bei diesem Beispiel sagen, dass alle Ansätze auf der Basis von OOP Programmierung der strukturierten Programmierung mit C überlegen sind. Die Lösungen auf der Basis von C++, C++ (T) und AspectC++ erlauben die Kapselung der Treiberfunktionalität, die somit unabhängig von der vorliegenden Hardware-Konfiguration eingesetzt werden kann. Dadurch ist die Umsetzung verschiedener Varianten in Bezug auf die Anzahl der eingesetzten Komponenten und die Hardware-Konfiguration möglich. Erweiterungen des Gerätetreibers auf der Grundlage von Aspekten oder Feature-Modulen waren aufgrund des eingeschränkten Funktionsumfangs der LED nicht notwendig.

LCD Da der LED-Gerätetreiber keine Features aufweist, erfolgt für den LCD-Gerätetreiber die gleiche Untersuchung. Für die Analyse werden Lösungsansätze auf

Opt.	LEDs	text	data+bss	dec
`-O0`	C	354	0	354
	C++	564	0	564
	C++ (T)	638	0	638
	AspectC++	502	0	502
`-O1` bis `-O3`	C	238	0	238
	C++	238	0	238
	C++ (T)	238	0	238
	AspectC++	238	0	238
`-Os`	C	274	0	274
	C++	274	0	274
	C++ (T)	286	0	286
	AspectC++	238	0	238

Tabelle 4.1: Übersicht Programmgrößen LED (in Bytes)

der Basis von C, C++, C++ (T) und FeatureC++ verwendet. Die Konfiguration des Gerätetreibers in C und C++ erfolgt mit Hilfe von Präprozessoranweisungen (Abschnitt 3, S. 23). Die C++ (T)-Variante basiert auf Geräteschnittstellen und Templates, die mit Hilfe von Features konfiguriert werden. Aspekte waren zur Umsetzung des Treibers nicht notwendig und werden nicht untersucht. Die Ergebnisse der übersetzten Beispielprogramme zeigt die Tabelle 4.2. Es zeigt sich, dass die C- und die FeatureC++-Variante den anderen Lösungen im Hinblick auf die Programmgröße bei eingeschalteten Optimierungen überlegen ist. Der Grund dafür liegt in der Nutzung von Funktionsaufrufen im Vergleich zu Inline-Funktionen die bspw. in der C++ (T)-Variante eingesetzt werden. Dabei wird kein Unterschied zwischen Methoden gemacht, die nur einmal oder mehrere Male verwendet werden. Die Ersetzung aller Funktionsaufrufe durch Inline-Funktionen führt zu ähnlichen Programmgrößen wie bei den anderen Lösungsansätzen.

In Bezug auf die Wiederverwendbarkeit zeigt sich, dass alle Lösungen zwar eine hohe Wiederverwendung aufweisen, jedoch einzelne Ansätze (C, C++ und C++ (T)) Schwächen in Bezug auf Modularisierbarkeit und Erweiterbarkeit aufweisen. Der Einsatz des Präprozessors macht dabei in den Lösungen C und C++ bereits 38.28 % bzw. 42.91 % aus und liegt damit im Bereich der Projekte, die in Abschnitt 3 (S. 23) untersucht wurden. Alle Features des LCD werden mit Hilfe von Makros umgesetzt. Aufgrund der geringen Anzahl an Features ist der Einsatz von Makros in diesem Fall tolerierbar. Gleichwohl ist zu erwarten, dass der Aufwand zur Implementierung und Wartung komplexerer Gerätetreiber für andere Komponenten höher ausfällt. Dies geht mit der geringen Modularisierbarkeit der Präprozessoranweisungen einher. Der C++ (T) Ansatz besitzt den Nachteil, dass Templates eine geringe Erweiterbarkeit aufweisen. Dies äußert sich darin, dass bestehende Methoden einer Template-Klasse nicht erweitert werden können.

Bei der Untersuchung der verschiedenen Lösungsansätze zeigte sich, dass der Res-

Opt.	LCD	text	data+bss	dec
-O0	C[1]	5208	16	5224
	C++[2]	1532	42	1574
	C++ (T)	5330	16	5646
	FeatureC++	5104	16	5120
-O1	C	974	8	982
	C++	1512	42	1554
	C++ (T)	1764	8	1772
	FeatureC++	1894	8	1902
-O2	C	992	8	1000
	C++	1512	42	1554
	C++ (T)	1736	8	1774
	FeatureC++	1902	8	1910
-O3	C	1632	8	1640
	C++	1512	42	1554
	C++ (T)	1736	8	1774
	FeatureC++	1902	8	1910
-Os	C	936	8	944
	C++	1512	42	1554
	C++ (T)	1342	8	1350
	FeatureC++	1070	8	1078

[1] http://kk.elektronik-4u.de (LCD Library)
[2] http://www.avrfreaks.net (LCD HD44780 C++ library)

Tabelle 4.2: Übersicht Programmgrößen LCD (in Bytes)

sourcenverbrauch im wesentlichen auf den Einsatz von Inline-Funktionen (Inlining) zurückgeht. Die Methode des Inlinings wird nachfolgend noch genauer diskutiert.

Inlining vs. Funktionsaufrufe

Die Unterschiede zwischen den verschiedenen Lösungsansätzen gehen zumeist auf die Nutzung von Inline-Funktionen (Inlining) bzw. Funktionen mit entsprechenden Funktionsaufrufen zurück. Geschwindigkeitsvorteile, die durch den Einsatz von Inline-Funktionen entstehen, werden durch die Replikation von Programmanweisungen und dem dadurch steigenden Programmspeicherverbrauch erkauft. Die Kennzeichnung einer Funktion als Inline-Funktion erfolgt entweder durch das Schlüsselwort *inline* in der Methodendeklaration (explizites Inlining) oder durch direkte Definition der Methode an der Stelle ihrer Deklaration (implizites Inlining). Inline-Definitionen werden in allen Ansätzen (C, C++, AspectC++ und FeatureC++) unterstützt und der Programmierer kann dadurch Einfluss auf den Übersetzungsprozess nehmen. Jedoch ist die Definition von Inline-Methoden nach dem obigen Schema keine Garantie für Inlining. Sowohl explizites als auch implizites Inlining stellen jeweils nur eine Empfehlung an den Compiler dar. Während der Übersetzung eines Programms durch den Compiler bestimmt die-

ser anhand des Umfangs und der Häufigkeit des Aufrufs einer Methode, ob Inlining zur Anwendung kommt. Jedoch besteht die Möglichkeit durch die Angabe von Methodenattributen Inlining für eine Methode zu Erzwingen oder Auszuschließen. Für den eingesetzten Compiler lauten die entsprechenden Angaben *__attribute__((always_inline))* bzw. *__attribute__((noinline))*. Da diese Inline-Kennzeichnungen durch den Programmierer selbst festgelegt werden, können falsche Kennzeichnungen laut Serrano [Ser97] zu einem unerwünschten Anwachsen des übersetzten Programms führen. Darüber hinaus setzt die Kennzeichnung ein genaues Verständnis über die Auswirkungen des Inlinings seitens des Programmierers voraus.

AspectC++/ FeatureC++ und Inlining

Kenntnisse zur Umsetzung von Inline-Optimierungen auf der Seite des Programmierers sind auch bei der Anwendung von AspectC++ und FeatureC++ notwendig. Beide unterstützen sowohl explizites als auch implizites Inlining. Ihre Anwendung wird jeweils durch die Definition in Feature-Modulen oder Aspekten umgesetzt. Diese Tatsache erschwert den Umgang mit Aspekten und Feature-Modulen. Im Fall homogener Erweiterungen mit AspectC++ legt der Aspekt Inlining für alle Erweiterungspunkte fest. Deren Anzahl kann sich im Laufe der Projektentwicklung ändern, sodass sich die Zweckmäßigkeit des Inlinings von passend zu ungeeignet bzw. umgekehrt ändern kann. Die mittels FeatureC++ übersetzten Quellen enthalten je nach Definition in den FeatureC++-Quellen implizites und explizites Inlining. In Anbetracht der Tatsache, dass die einzelnen Verfeinerungsstufen der Erweiterungshierarchie nicht benutzt werden, sollte Inlining durch das *__attribute__((always_inline))* erzwungen werden, da jede Verfeinerungsstufe jeweils nur einmal verwendet wird. Dadurch kann von den Optimierungen des Compilers für Inline-Methoden Gebrauch gemacht werden.

Übersicht und Empfehlungen

Nachdem die einzelnen Gerätetreiber-Implementierungen ausgewertet wurden, folgt nun eine Übersicht zu den Ergebnissen der untersuchten Programmieransätze. Darauf basierend werden Empfehlungen für den Einsatz verschiedener Ansätze gegeben. Die Tabelle 4.3 zeigt die Übersicht der Kriterien zusammen mit deren Erfüllbarkeit für die untersuchten Lösungsansätze. Wie der Tabelle zu entnehmen ist, weisen C und C++ große Schwächen in den Bereichen Erweiterbarkeit, Modularisierbarkeit und Wiederverwendung auf. Dies geht in erster Linie auf den Einsatz des Präprozessors zur Feature-Konfiguration zurück. Einige der Probleme, die im Zusammenhang mit dem Einsatz von C und C++ stehen, konnten mit Hilfe von C++ (T) vermieden werden. Dennoch ist der Einsatz von C++ (T) unnötig, da die generischen Eigenschaften dieses Ansatzes in der Gerätetreiber-Programmierung nicht genutzt werden. Die zur Umsetzung der Gerätetreiber eingesetzten Geräteschnittstellen lassen sich direkt durch Aspekte oder Feature-Module erweitern, sodass von Templates oder anderen Lösungen kein Gebrauch gemacht werden muss. Die gewählten Ansätze und Vorgehensweisen sind jeweils als Empfehlungen zur Umsetzung von Gerätetreibern zu sehen.

Die bisherigen Betrachtungen befassten sich ausschließlich mit der Anbindung von Geräten an den Mikroprozessor und dem Auftreten möglicher Varianten. Weitere Änderungen für die HPL ergeben sich durch Variationen angeschlossener Komponenten. Eine Untersuchung dafür erfolgt im nächsten Abschnitt.

Kriterium	**C**	**C++**	**C++ (T)**	***AspectC++***	***FeatureC++***
Erweiterbarkeit	-	-	-	*	+
Ressourcenverbrauch	+	o	o	+	+
Modularisierbarkeit	-	-	o	+	+
Wiederverwendbarkeit	-	o	o	+	+

+ gut erfüllbar
o teilweise erfüllbar
- schlecht erfüllbar
* keine Aussage, da nicht notwendig

Tabelle 4.3: Übersicht Lösungsansätze für Gerätetreiber und deren Eignung

4.5 Variationen von Komponenten

Wie bereits in Abschnitt 4.1 (S. 36) beschrieben, lässt sich das LCD-Modul mit einem 4- oder 8-Bit breiten Datenbus betreiben. Während die Variationen in den Beispielen zuvor sich ausschließlich auf die Verbindung von Komponenten zum Mikroprozessor bezogen, zeigt sich, dass Varianten auch in den Geräten selbst auftreten. Die beiden unterschiedlichen Ansteuerungsmöglichkeiten schlagen sich im Fall des LCD-Moduls in der Treiberimplementierung und, sofern benutzt, auch in der Geräteschnittstelle nieder. Für das LCD-Modul beziehen sich notwendige Erweiterungen auf die Ergänzung einer Methode. Eine mögliche Umsetzung auf der Grundlage einer Geräteschnittstelle zusammen mit der Treiberimplementierung in Form eines Templates zeigt die Abbildung 4.15. Das Vorgehen wurde bereits im Abschnitt 4.3.3 (S. 57) gezeigt. Die Ergänzung wird durch eine Methodenverfeinerung mittels FeatureC++ umgesetzt. Die Methode *write* (Z. 15) der Verfeinerung (Z. 13) kapselt die zusätzliche Funktionalität, die für die 4-Bit Variante des LCD notwendig ist. Ein ähnliches Bild ergibt sich für die 8-Bit Variante.

Komponenten und Features

Varianten treten aber auch an anderen Stellen in der HPL auf. Bereits bei der Vorstellung der für die Arbeit eingesetzten HPL (Abschnitt 2.2, S. 7) wurde auf die unterschiedlichen Ausstattungsmerkmale (Komponenten) hingewiesen. Jede Komponente weist für sich wieder unterschiedliche, teils auch optionale Features auf. Das LCD-Modul und die LED besitzen wenige bis keine Features, weshalb ein weiteres Beispiel angeführt wird. Ein optionales Ausstattungsmerkmal der AVR-Mikroprozessoren ist der CAN-Controller. Dieser kann für Datenübertragungen des Controllers zu anderen Controllern eingesetzt werden. Entsprechend des Datenblattes besitzt der im Mikroprozessor integrierte CAN-Controller bspw. folgende Features [Atm07]:

- maximale Übertragungsrate 1 MBit/s,

```
1  template <typename T>
2  class LCD {
3  private:
4    T& lcd;
5  ...
6  public:
7    void write(uint8_t data, bool command) {
8      lcd.LCDDDR_EA  = 1;
9  ...
10
11
12 template <typename T>
13 refines class LCD<T=AT90CANLCD4Bit> {
14   public:
15     void write(uint8_t data, bool command) {
16       super::write(data, command);
17       lcd.LCDPORT_D7 = (data & 0x80) ? 1 : 0;
18       lcd.LCDPORT_D6 = (data & 0x40) ? 1 : 0;
19 ...
```

Abbildung 4.15: Methodenverfeinerung LCD-Modul 4-Bit Variante

- Unterstützung der CAN-Standards 2.0 A und B,
- 15 Nachrichtenobjekte (engl. Message Objects) und
- Abhörmodus (engl. Listening Mode).

Die genannten Features haben Auswirkungen auf die Gerätetreiber-Implementierung. So hat bspw. die Nutzung der Nachrichtenobjekte Einfluss auf die Initialisierung, das Senden und Empfangen in der Treiberimplementierung. Darüber hinaus hat die Unterstützung der beiden CAN-Standards größeren Einfluss auf den Gerätetreiber, da strukturelle Unterschiede zwischen beiden Standards Konsequenzen in der Treiberimplementierung nach sich ziehen. Damit ergibt sich ein weiterer Interaktionspunkt zwischen der HPL und der SPL.

4.6 Zusammenfassung

Das Kapitel zeigte verschiedene Ansätze zur Umsetzung von HPL am Beispiel der Entwicklung von Gerätetreibern für eingebettete Systeme. Dabei zeigte sich, dass mit Hilfe erweiterter Programmierparadigmen auf den Präprozessor cpp zur Umsetzung von Konfigurationen und Varianten verzichtet werden konnte. Die jeweiligen Lösungen auf der Grundlage von AspectC++ und FeatureC++ standen effizienten Referenz-Beispielen, die in C geschrieben und jeweils durch Präprozessoranweisungen konfiguriert wurden, in Bezug auf die Programmgröße in nichts nach.

Bei der Umsetzung der Ergebnisse zeigte sich jedoch, dass die erzielten Programmgrößen von der Implementierung abhängen und das sich der Programmierer der Wirkung einer Implementierung bewusst sein muss. Dies betrifft jedoch alle in dem Kapitel untersuchten Lösungsansätze gleichermaßen. Da AspectC++ und FeatureC++

als Precompiler agieren, könnte durch zusätzliche Angaben zu Aspekten bzw. Feature-Modulen Einfluss auf ihre Integration in den bestehenden Quellcode genommen werden. Eine Möglichkeit Einfluss auf die Programmgröße und die Ausführungsgeschwindigkeit zu nehmen, bieten die im Kapitel genannten Methodenattribute.

Die Ergebnisse zu den untersuchten Komponenten lassen sich, obgleich sie für die Untersuchungen in diesem Kapitel ausreichend waren, nicht verallgemeinern. Weitere Untersuchungen anhand umfangreicherer Funktionstreiber sind notwendig, um die in diesem Kapitel erzielten Ergebnisse zu belegen. Bei der Umsetzung verschiedener Gerätetreiber wurden zudem folgende Interaktionspunkte zwischen HPL und SPL festgestellt:

Hardware-Konfiguration Verschiedene Anschlusskonfigurationen von Mikroprozessor und anzuschließenden Komponenten haben Auswirkungen auf die Treiberimplementierung.

Verbindung Mikroprozessor Komponente Die Anzahl der Anschlusskontakte zwischen Mikroprozessor und Komponenten ist z. T. variabel. Dies geht auf die in den Komponenten vorhandenen Features zurück und hat Auswirkungen auf die Gerätetreiber-Implementierungen.

Features von Komponenten Komponenten, die an einen Mikroprozessor angeschlossen werden sollen, bilden selbst eine HPL und haben Einfluss auf den Gerätetreiber.

Es ist davon auszugehen, dass weitere Interaktionspunkte auftreten, wenn die Untersuchung auf weitere Modellreihen oder andere Hardware-Plattformen ausgedehnt wird.

Kapitel 5

DBMS für tief eingebettete Systeme

Nachdem im vorherigen Kapitel der Einsatz erweiterter Programmierparadigmen in der Gerätetreiber-Entwicklung untersucht wurde, werden in diesem Kapitel die Grundlagen zur Entwicklung eines DBMS für tief eingebettete Systeme gelegt. Dazu werden im ersten Schritt in einer Domänenanalyse Anforderungen für DBMS in tief eingebetteten Systemen herausgearbeitet. Die Erläuterung des dafür notwendigen Vorgehens erfolgte bereits in Abschnitt 2.4 (S. 11). Die Ergebnisse der Analyse werden mit einer Auswahl existierender DBMS-Lösungen verglichen. In diesem Zusammenhang wird deren Einsatzfähigkeit auf den in dieser Arbeit eingesetzten Systemen diskutiert. Ziel des Kapitels ist ein Feature-Diagramm für ein DBMS. Das Diagramm bildet die Grundlage zur Entwicklung der DBMS-Lösung RobbyDBMS. Anhand der Implementierung RobbyDBMS erfolgt die konzeptionelle und technische Untersuchung erweiterter Programmierparadigmen (Kapitel 6, S. 73).

Warum Untersuchung anhand eines DBMS?

Die Auswahl des DBMS als komplexe Beispielanwendung hat mehrere Gründe. Die Entwicklung maßgeschneiderter Datenmanagement-Lösungen ist immer noch Gegenstand aktueller Forschung [RSS+08]. Der Grund dafür sind vielfältige Anforderungen an DBMS, die in unterschiedlichen Features umgesetzt werden müssen. Dabei stellen insbesondere die Kapselung optionaler Eigenschaften sowie deren Integration in ein bestehendes Basissystem die Entwickler vor Herausforderungen. Darüber hinaus sind einzelne Features voneinander abhängig, sodass deren Kombination die Entwickler vor weitere Aufgaben stellt. Demzufolge eignet sich die Entwicklung eines DBMS zur konzeptionellen und technischen Untersuchung erweiterter Programmierparadigmen.

Notwendigkeit DBMS

In den 1960er Jahren haben auftretende Probleme bei der Verwaltung von Datenmengen zur Entwicklung von DBMS geführt. Zuvor wurden alle Informationen in den Computersystemen manuell verwaltet, was gleichsam aufwendig, fehleranfällig und kostenintensiv war. In den folgenden Jahrzehnten wurden viele DBMS-Lösungen entwickelt, die mit ihrem großen Funktionsumfang die Verwaltung von Datenmengen vereinfachten. Der große Funktionsumfang ist zudem das Ergebnis der Ausrichtung auf möglichst viele Anwendungsszenarien (Mehrzwecksystem). Gleichzeitig zeigt er sich für die vergleichsweise hohen Ansprüche an die zugrunde liegende Hardware-Plattform verantwortlich. Die Hardware-seitigen Ressourcenbeschränkungen tief eingebetteter Syste-

me führen jedoch dazu, dass diese Mehrzwecksysteme nicht auf diesen Systemen eingesetzt werden können. Allerdings fallen auch in tief eingebetteten Systemen Daten zur Speicherung und Verwaltung an. Dazu gehören bspw.:

- Sensordaten,
- Ereignisprotokolle,
- Wartungsdaten,
- Daten zur Kalibrierung von Sensoren,
- Konfigurationsdaten und
- Störungsprotokolle.

Ungeachtet der Tatsache, dass der Datenumfang im Einzelnen anscheinend gering ausfallen kann, können zusätzliche Anforderungen an die Datenhaltung und -speicherung Mechanismen zur konsistenten Speicherung der genannten Daten notwendig machen. Dies geht mit der Forderung einher, DBMS-Lösungen auch für die Domäne tief eingebetteter Systeme umzusetzen. Die genauen Anforderungen an DBMS für tief eingebettete Systeme werden im ersten Abschnitt erarbeitet.

5.1 FODA DBMS tief eingebettetes System

Aufgrund des Umfangs einer vollständigen Domänenanalyse nach Kang et al. [KCH+90] werden nur Teile der FODA verwendet und deren Ergebnisse präsentiert. Im ersten Schritt werden Anforderungen und Randbedingungen der Datenspeicherung in tief eingebetteten Systemen herausgearbeitet. Diese werden im Weiteren für den Vergleich zu existierenden DBMS-Lösungen verwendet. Deren Eignung wird anhand von Feature-Diagrammen diskutiert, die den Funktionsumfang der jeweiligen Lösung widerspiegeln. Im Anschluss an die Diskussion wird ein Feature-Diagramm präsentiert, das die Grundlage für die Entwicklung von RobbyDBMS bildet.

Gemeinsamkeiten und Unterschiede

Bei der Vorstellung der Methodik FODA zur Entwicklung von SPL wurde bereits darauf hingewiesen, dass existierende Lösungen die wichtigste Grundlage zur Entwicklung neuer Systeme darstellen. Infolgedessen werden zunächst einige Unterschiede zu existierenden DBMS-Lösungen aufgelistet und erläutert. Diese wurden in einem Vergleich zu existierenden Systemen bestimmt, die in diesem Kapitel noch genannt oder vorgestellt werden.

5.1.1 Struktur anfallender Daten

Wie bereits eingangs erwähnt, beziehen sich die zu speichernden Datenmengen in tief eingebetteten Systemen häufig auf die Speicherung von Sensordaten oder Konfigurationen. Diese Informationen lassen sich mit Hilfe weniger Bits oder Bytes darstellen und ihre Anzahl kann für die Anwendung jeweilig als fest angenommen werden. Grund zu

dieser Annahme ist die Tatsache, dass alle Datenquellen und ihr Umfang im voraus bekannt sind. Die Datenspeicherung kann in Tupeln erfolgen. Ein Tupel besteht dabei aus einem Schlüssel und einer definierten Menge von Attributen, die zusammen sequentiell abgespeichert werden. Im Weiteren lassen sich die zu speichernden Attribute mit einfachen Datentypen abbilden. Erweiterte Datentypen, wie z. B. Zeitstempel und binäre Objekte, die in Mehrzweck-DBMS zum Einsatz kommen, sind nicht notwendig. Damit reduziert sich insgesamt der Aufwand zur Speicherung und Verwaltung aller anfallenden Daten.

5.1.2 Permanente Datenspeicher eingebetteter Systeme

In Abschnitt 2.1 (S. 5) wurden bereits ein paar Beispiele eingebetteter Systeme vorgestellt. Für die permanente Datenspeicherung bringen diese Systeme selbst nur den *Electrically Erasable Programmable Read Only Memory (EEPROM)*-Speicher mit.[1] Bei der Vorstellung der eingebetteten Systeme wurde nur auf Restriktionen bzgl. der Leistungsfähigkeit des Prozessors und Ausstattung an Programm-, Arbeits- und Datenspeicher hingewiesen. Für den permanenten Datenspeicher eingebetteter Systeme ergibt sich neben der geringen Größe von wenigen Kilobyte eine weitere Einschränkung. Aufgrund der eingesetzten Speichertechnologie ist die Anzahl möglicher Schreibzyklen begrenzt [CGOZ99]. Typischerweise werden von den Herstellern 10.000 bis 1.000.000 Schreibvorgänge pro Speicherzelle garantiert. Auch wenn die tatsächliche Anzahl häufig höher liegt, müssen während des Einsatzes besondere Vorkehrungen getroffen werden, um Datenverlust durch Ausfall der Speicherfähigkeit eingesetzter Komponenten vorzubeugen [Lin07, CGOZ99, Atm07].

EEPROM

Neben EEPROM-Speichern sind auch Flash-Speicher-basierte Datenspeicher in eingebetteten Systemen weit verbreitet. Diese auf NAND- oder NOR-Strukturen basierenden Speicher unterscheiden sich technologisch von EEPROM-Speichern. Die Unterschiede liegen im Bereich der Adressierbarkeit einzelner Speicherzellen. EEPROM-Speicher erlauben Byte-weise Adressierungen. Im Gegensatz dazu sind Flash-Speicher intern in Blöcke unterteilt, die wiederum aus mehreren Seiten bestehen. Lese-, Schreib- und Löschoperationen sind jeweils auf diese Unterstrukturen begrenzt, sodass ein wahlfreier Zugriff nicht möglich ist. Während Lese- und Schreiboperationen auf Seiten ausgeführt werden können, ist das Löschen nur auf Blockebene möglich. Flash-Speicher unterliegen bezogen auf die Anzahl möglicher Schreibzyklen den gleichen Einschränkungen wie EEPROM-Speicher.

Flash-Datenspeicher

Die genannten Speichertechnologien unterscheiden sich neben der Adressierbarkeit auch im Zeitverhalten bei verschiedenen Operationen. Eine Übersicht zu den Zeitspannen einzelner Operationen auf EEPROM- und Flash-Speichern zeigt die Tabelle 5.1.

Zeitverhalten der Speicher

Während der Lesezugriff bei beiden Speichertechnologien sehr schnell ist, dauert ein Schreibzugriff beim EEPROM-Speicher vergleichsweise lange. Infolgedessen macht dies, je nach Anwendung den Einsatz einer Pufferverwaltung notwendig. Damit kann

[1] Es existieren jedoch Projekte, in denen Mikroprozessoren der AVR-Modellreihe mit Flash-basierten Speichern erweitert werden.

Speichertechnologie	EEPROM	Flash
Lesen (/Wort)	60 bis 150 ns	70 bis 200 ns
Schreiben (/Wort)	10 ms	5 bis 10 μs
Löschen (/Block)	nicht definiert	500 bis 800 ms

Tabelle 5.1: Flash und EEPROM Zeitverhalten bei verschiedenen Operationen [PBVB01]

die Leistungsschwäche beim Schreiben vermindert werden. Löschoperationen sind im Byte-weise adressierbaren EEPROM-Speicher nicht definiert. Vor Schreibzugriffen wird eine Speicherzelle automatisch gelöscht. Das Löschen von Speicherzellen ist in Flash-Datenspeichern ungleich aufwendiger, da Löschoperationen nur für Blöcke definiert sind. Demzufolge müssen Informationen eines Blockes, die noch benötigt werden, zwischengespeichert werden, um sie nach dem Löschvorgang weiter verwenden zu können.

5.1.3 Nebenläufigkeit in eingebetteten Systemen

Obwohl eingebettete Systeme keine Mehrbenutzersysteme oder Mehrprozesssysteme sind, können in ihnen auch eine eingeschränkte Form von Nebenläufigkeit auftreten. Quelle dieser Nebenläufigkeiten sind Interrupts, die Hardware-seitige Unterbrechungsanforderungen darstellen. Diese werden für die ereignisbasierte Verarbeitung in eingebetteten Systemen eingesetzt. Folgende Interruptquellen treten bspw. in den eingesetzten eingebetteten Systemen auf:

- Timer,
- externe Interruptquelle (bspw. Sensor zur Kollisionserkennung) und
- Sende- und Empfangsbereitschaft von Schnittstellen.

Interrupt-Verarbeitung

Mit Eintreten eines Interrupts wird der aktuelle Prozessorzustand gesichert und die zum Interrupt gehörende Vektorfunktion (*Interrupt-Service-Routine (ISR)*) abgearbeitet. Nach Beenden der ISR wird der Prozessorstatus wiederhergestellt und die Bearbeitung der Aufgabe, die vor Eintreten des Interrupts bearbeitet wurde, fortgesetzt. Während der Bearbeitung einer ISR können weitere Interrupts auftreten, sodass, sofern erlaubt, eine verschachtelte Verarbeitung von Interrupts auftritt (Abbildung 5.1). Die Interruptverarbeitung erfolgt nur, wenn das entsprechende Bit im Interrupt-Mask-Register maskiert ist und die Verarbeitung von Interrupts generell erlaubt ist.

Folgen Interrupt-Verarbeitung

Während der Abarbeitung von ISR kann es vorkommen, dass der Zugriff auf gemeinsame Datenstrukturen und Daten notwendig ist. Da Interrupts zu jedem Zeitpunkt eintreten können, ist es möglich, dass die Daten und Datenstrukturen noch in der Bearbeitung innerhalb eines anderen Programmabschnitts sind. Das betrifft bspw. das Hinzufügen oder Löschen von Elementen zur einer Liste. Externe Veränderungen können hierbei zur Zerstörung der Datenstruktur führen. Darüber hinaus sind Speicherlecks, Systemabstürze oder falsche Ergebnisse zu erwarten. Neben den Datenstrukturen

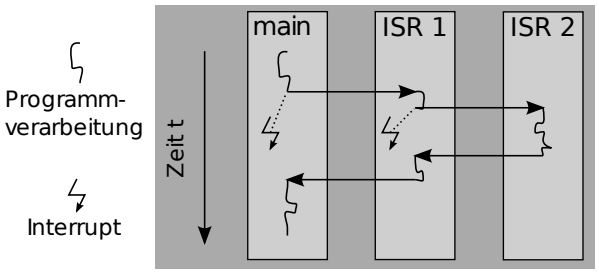


Abbildung 5.1: Interrupt-Verarbeitung in eingebetteten Systemen

sind auch die zu speichernden Datenelemente selbst zu schützen. Die Verarbeitung eines Interrupts kann dazu führen, dass Datenelemente gelesen oder geschrieben werden sollen, die noch in der Verarbeitung sind. Eine Folge ist bspw. das Eintreten von Inkonsistentem Lesen bzw. Lost Update [SHS05]. Beide Phänomene machen den Einsatz einer Transaktionsverwaltung notwendig.

5.1.4 Autonome eingebettete Systeme

Die Stromversorgung in autonomen eingebetteten Systemen erfolgt häufig durch Batterien und Akkumulatoren. Da jederzeit diese Versorgung ausfallen kann, müssen je nach Anwendung Vorkehrungen zur Bestimmung der Integrität gespeicherter und zu speichernder Daten getroffen werden. Dies trifft insbesondere dann zu, wenn ein System nach dem Zusammenbrechen der Stromversorgung wieder mit Energie versorgt wird, um auf den Datenbestand zuzugreifen. Die Integrität gespeicherter Daten lässt sich durch den Einsatz von Prüfsummen sicherstellen.

5.2 Existierende DBMS für eingebettete Systeme

Nachdem im Abschnitt zuvor der Aufgabenbereich und die Randbedingungen für DBMS in tief eingebetteten Systemen erläutert wurden, werden im folgenden Abschnitt existierende DBMS-Lösungen vorgestellt und ihre Eignung für den Einsatz auf den in Abschnitt 2.1 (S. 5) diskutiert. Dazu werden zunächst eine Reihe von DBMS (Mehrzwecksystem) genannt, deren Einsatz von vornherein ausscheidet, weil sie grundlegende Anforderungen der eingesetzten Systeme nicht erfüllen. Die Auswahl bezieht sich auf Systeme, deren Ressourcenverbrauch bekannt ist.

Die betreffenden Systeme werden zusammen mit ihrem Verbrauch an Programmspeicher aufgeführt, der neben der vorausgesetzten Plattform als maßgebendes Kriterium gilt (Tabelle 5.2, S. 71). Der minimale Verbrauch an Programmspeicher liegt bei den genannten Systemen zwischen 50 und 2048 KB. Damit übersteigt der benötigte Speicher in den meisten Fällen bereits den zur Verfügung stehenden Programmspeicher. Darüber hinaus setzen viele Lösungen die Unterstützung durch ein Betriebssystem voraus, dessen Einsatz für einen weiteren Verbrauch an Programmspeicher verantwortlich ist. Dieser

zusätzliche Speicherverbrauch wird jedoch in keinem Fall angegeben, übersteigt aber die zur Verfügung stehenden Ressourcen der eingesetzten Systeme um ein Vielfaches.

Refaktorisierung Berkeley DB

Der hohe Verbrauch an Programmspeicher geht bei den genannten Mehrzweck-DBMS auf den hohen Funktionsumfang zurück. Durch Refaktorisierung lässt sich der Speicherverbrauch des DBMS reduzieren. Dabei werden obligatorische Features in optionale Features überführt, die dann nur noch in Abhängigkeit der gewählten Konfiguration Bestandteil des Zielsystems sind. In Rosenmüller et al. [RLA07] wird Refaktorisierung am Beispiel von DBMS Berkeley DB angewendet. Durch das Herauslösen von Features ließ sich eine DBMS-Variante in der minimalen Größe von 256 KB erzeugen. Die Autoren von [RLA07] gehen davon aus, dass sich durch Refaktorisierung weiterer Features noch kleinere Systeme erzielen lassen. Jedoch wird in Rosenmüller et al. [RSS+08] darauf hingewiesen, dass der Aufwand zur Extraktion weiterer Features aufgrund der Vermischung von Funktionalitäten unannehmbar ansteigt. Ferner motivieren die Autoren die Entwicklung eines DBMS von Grund auf neu. Darüber hinaus liegt der Speicherverbrauch selbst bei einer Halbierung des bereits refaktorisierten Beispiels Berkeley DB immer noch oberhalb des zur Verfügung stehenden Programmspeichers der eingesetzten Systeme.

Nach der Vorstellung verschiedener Mehrzweck-DBMS wird nun eine Reihe von DBMS-Lösungen präsentiert. Die Auswahl beschränkt sich auf Systeme, die speziell für die Domäne tief eingebetteter Systeme entwickelt wurden. Ihr Einsatz wird vor dem Hintergrund des Speicherverbrauchs und der Variabilität bezogen auf mögliche Anwendungsfälle diskutiert. Beide Kriterien werden in dieser Arbeit als maßgebend angesehen.

5.2.1 TinyDB

TinyDB stellt ein verteiltes Anfragesystem für Sensornetzwerke zur Verfügung. Die zugrunde liegende Architektur besteht aus einem Rechner für die Formulierung und Verteilung einer Anfrage (Root-Knoten) und vielen autonomen Sensorknoten. Anfragen erfolgen in einer SQL-ähnlichen Syntax. Sie erlauben die zyklische Abfrage, die Forderung der Speicherung von Datenreihen auf den Sensorknoten, sowie die Aggregation von Daten über vorhandene Datenreihen und Sensorknoten. Für die Bearbeitung von Anfragen werden zusätzlich Metainformationen (Schemata) auf den Sensorknoten gehalten. Der Datenaustausch zwischen den einzelnen Einheiten erfolgt drahtlos. TinyDB baut auf dem Betriebssystem TinyOS auf, das grundlegende Funktionalitäten für die Verwaltung und den Zugriff auf die Hardware des eingesetzten eingebetteten Systems bereitstellt [MFHH05]. Die Anwendungsgebiete liegen bspw. in der verteilten Umweltüberwachung [SOP+04].

eingeschränkter Funktionsumfang

TinyDB stellt kein vollständiges DBMS für autonome eingebettete Systeme zur Verfügung. Die vorgegebene Struktur aus Root- und Sensorknoten schränkt den Einsatz auf verteilte Anwendungen mit Sensorknoten als Datenquelle ein. Mit dem SQL-Interpreter und der Metadatenhaltung sind nur eine Reihe der Bestandteile bestehender DBMS umgesetzt. Der Fokus von TinyDB liegt in der ressourcenschonenden und ener-

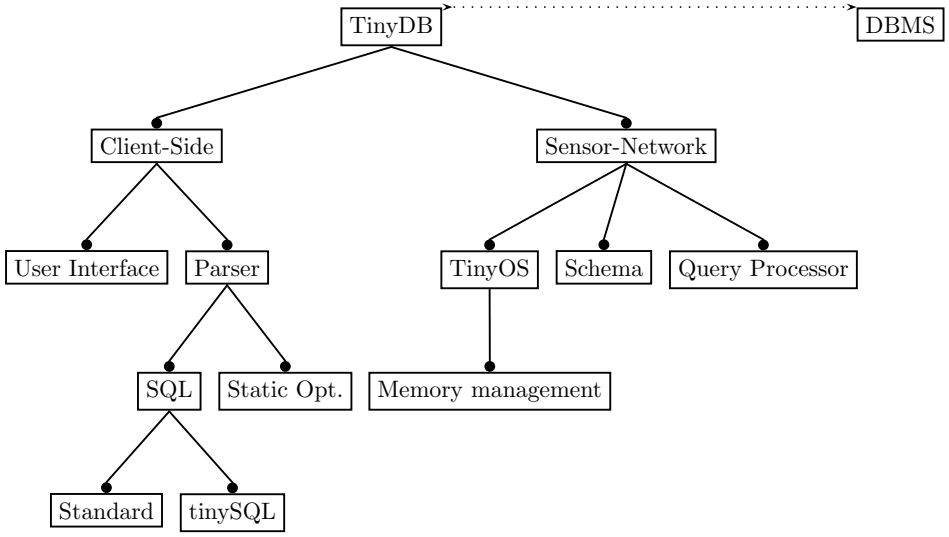


Abbildung 5.2: Feature-Diagramm TinyDB

giesparenden Umsetzung des Anfragesystems.

TinyDB und TinyOS sind in nesC[2] geschrieben. nesC ist eine Erweiterung für die Programmiersprache C, um die Handhabung von Nebenläufigkeiten und Reaktionsfähigkeiten bei der Programmierung von eingebetteten Systemen zu vereinfachen. Ein voll ausgestattetes TinyDB führt zusammen mit dem gekoppelten TinyOS zu einem Speicherverbrauch von $\sim 64\,\mathrm{KB}$ [LPCS04]. Damit lässt es sich nicht auf allen in Abschnitt 2.2 (S. 7) vorgestellten eingebetteten Systemen einsetzen. Der Programmiersprachendialekt nesC setzt zudem vor dem Einsatz die Portierung auf das entsprechende Zielsystem voraus.

5.2.2 PicoDBMS

PicoDBMS ist ein DBMS für Smartcards [PBVB01]. Die Notwendigkeit der Datenmanagement-Funktionalität motivieren die Autoren am Beispiel einer Gesundheitskarte für Patienten. Das tief eingebettete System (Smartcard) enthält beispielsweise neben Notfallinformationen (Blutgruppe, Allergien, Impfschutz u. a.) auch Teile oder die komplette Krankengeschichte eines Patienten. Verschiedene Nutzergruppen und sensitive Daten erfordern z. B. unterschiedliche Sichten, Transaktionen oder Anfragesprachen.

Einschränkung Zielsysteme

Mit der Spezialisierung auf Smartcards werden Datenmanagement-Funktionalitäten nur für eine Teilmenge existierender eingebetteter Systeme umgesetzt. Die Struktur des DBMS ist vollständig auf die Hardware und die Anwendung ausgerichtet. Alternative Speichertechnologien, wie z. B. NAND- oder NOR-FLASH, wurden innerhalb der Umsetzung nicht berücksichtigt. Ein Großteil des zur Verfügung stehenden Arbeitsspeichers wird vom Betriebssystem der Smartcard belegt. Aus diesem Grund ist PicoDBMS

[2] http://nescc.sourceforge.net/

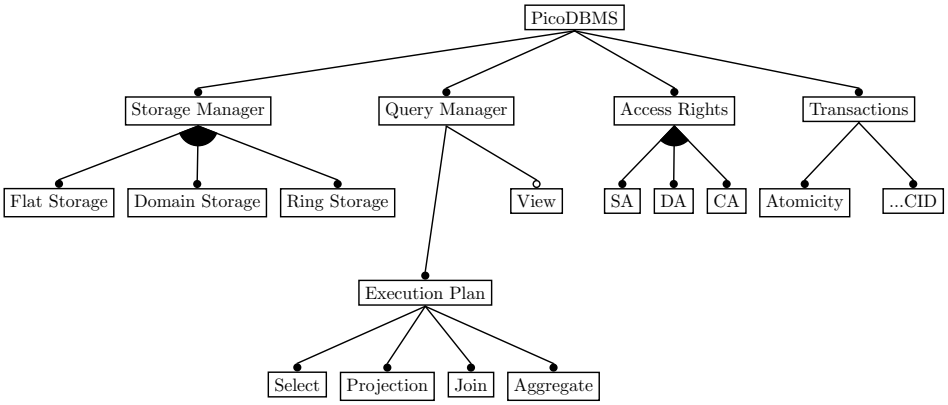


Abbildung 5.3: Feature-Diagramm PicoDBMS

gerade im Bereich der Nutzung des Arbeitsspeichers optimiert. Trotz des modularen Aufbaus sind viele Komponenten des PicoDBMS verbindlich (Abbildung 5.3, S. 66). Aufgrund dessen ergeben sich nur minimale Konfigurationsmöglichkeiten. Einsparungen an Arbeits- und Programmspeicher lassen sich nur durch die Auswahl der zugrunde liegenden Speichermodelle erzielen [ABP06].

Die Anwendung Gesundheitskarte ist für datenintensive DBMS nicht repräsentativ. Neue Datenbankeinträge oder Updates sind bezogen auf die Zeitspanne zwischen Operationen unregelmäßig und selten. Dadurch entfällt bspw. die Notwendigkeit des Cachings von Einträgen, die der Abnutzung des Speichers entgegenwirken. Die kompakten Speicherstrukturen des PicoDBMS sind Ergebnis der Domäne Gesundheitskarte und nicht allgemeingültig auf andere Anwendungen übertragbar, wie bereits in Anciaux et al. [ABP06] erwähnt.

5.2.3 COMET DBMS

COMET DBMS umfasst eine Sammlung von Werkzeugen zur Generierung von Echtzeit DBMS in heterogenen Umgebungen mit eingebetteten Systemen. Zu den Werkzeugen gehören Datenmanagement-, Analyse- und Konfigurationsprogramme mit deren Hilfe aus vorgefertigten Software-Bausteinen maßgeschneiderte Zielsysteme erzeugt werden [NTNH03]. Die Zielanwendung sind eingebettete Systeme, die in Automobilen zusammen ein Netzwerk bilden. Darin vorkommende Echtzeitanforderungen sind direkter Bestandteil des COMET Designs.

Einschränkung Zielanwendung

Für die Umsetzung maßgeschneiderter Software werden bestehende Komponenten mit Aspekten entsprechend einer Konfiguration zusammengefügt. Existierende Konfigurationsparameter beziehen sich auf die Behandlung von Echtzeitanforderungen und Nutzung erweiterter Indexstrukturen. Eine Basisimplementierung (COMET BaseLine), bestehend aus einer relationalen Anfragesprache, einem Transaktionssystem und einer Speicherverwaltung führt zu einem Footprint der kompilierten Bibliothek von $\sim 20\,\mathrm{KB}$ [Nys03, Nys05].

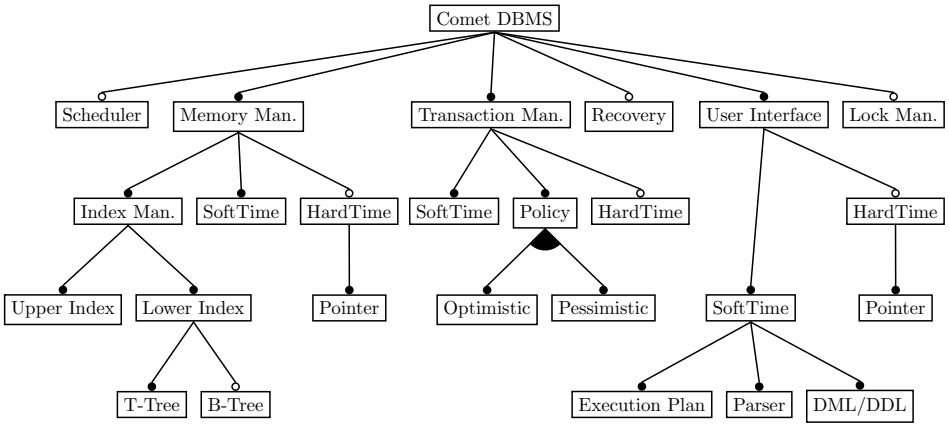


Abbildung 5.4: Feature-Diagramm COMET DBMS

5.2.4 FAME-DBMS

FAME-DBMS ist ein DBMS, das für verschiedene Zielanwendungen und -systeme konzipiert ist. Im Gegensatz zu den bereits vorgestellten Lösungen TinyDB, PicoDBMS und COMET DBMS zeichnet es sich durch eine Vielzahl von Konfigurationsmöglichkeiten aus. Dabei ist FAME-DBMS auf kein spezielles Anwendungsszenario ausgerichtet. Ausgehend von einer Basisimplementierung sollen Anforderungen von Anwendungen oder Zielsystemen durch Erweiterungen umgesetzt werden. Dafür wird auf optionale Komponenten und die Auswahl von Alternativen zurückgegriffen [RSS+08]. Das zum System gehörige Feature-Diagramm zeigt die Abbildung 5.5. Erweiterungen der obligatorischen Features werden mit Hilfe von FOP und AOP umgesetzt. Zu den unterstützten Zielsystemen gehören Windows, Linux und NutOS.[3] Durch die Ausrichtung auf verschiedene Zielsysteme werden vor allem Anforderungen tief eingebetteter Systeme nicht vollständig bedient. Dies zeigt sich bspw. daran, dass der Speicherverbrauch einer Referenzimplementierung mit 40 KB angegeben wird. Demzufolge lässt sich das System nicht auf allen in Abschnitt 2.2 (S. 7) vorgestellten Systemen einsetzen.

5.3 Feature-Diagramm RobbyDBMS

Von den vorgestellten DBMS für tief eingebettete Systeme erlaubt FAME-DBMS die größte Flexibilität. Alle weiteren Lösungen (TinyDB, PicoDBMS und COMET DBMS) sind für ein bestimmtes Anwendungsszenario entworfen worden und erlauben nur minimale Konfigurationsmöglichkeiten. Die Reduzierung auf eine bestimmte Anwendung zieht zudem im Fall PicoDBMS und TinyDB eine Eingrenzung verfügbarer eingebetteter Systeme nach sich. Aus diesem Grund wird der Ansatz des FAME-DBMS Projekts

[3] NutOS bildet die Grundlage für den Einsatz auf dem eingebetteteten System BTNode (http://www.btnode.ethz.ch/). BTNode ist ein tief eingebettetes System, das mit den in dieser Arbeit eingesetzten Systemen (Abschnitt 2.2, S. 7) vergleichbar ist, jedoch über deutlich mehr Arbeitsspeicher verfügt.

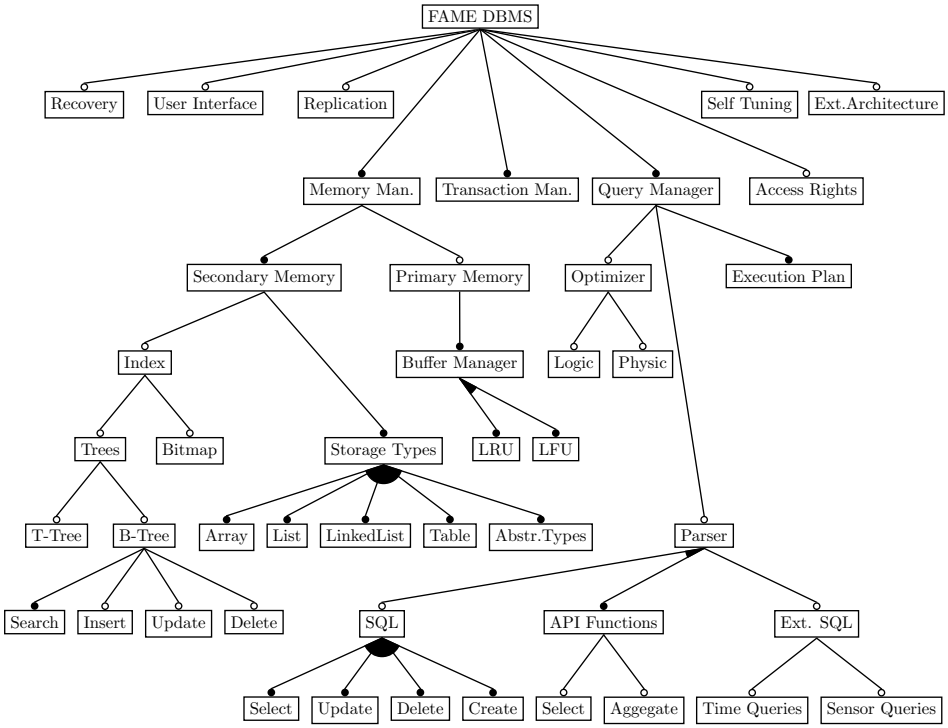


Abbildung 5.5: Feature-Diagramm FAME-DBMS

zur Entwicklung von RobbyDBMS verwendet. Dieser sieht vor möglichst viele Funktionalitäten in optionale Komponenten auszulagern. Dadurch lässt sich der Verbrauch an Programmspeicher soweit wie möglich reduzieren. Demzufolge wird der Spielraum für die Anwendungsentwicklung auf dem eingesetzten System vergrößert. Aus diesem Grund werden einige Funktionalitäten und Bestandteile, der zuvor eingeführten Systeme, als optionale Komponenten modelliert. Nachfolgend werden die einzelnen Bestandteile der DBMS-Lösung und deren Zusammenhänge erläutert. Eine vollständige Übersicht aller Features zeigt die Abbildung 5.6.

Pufferverwaltung (BufferManager) In Abschnitt 5.1.2 (S. 61) wurde beschrieben, dass der zur Verfügung stehende permanente Datenspeicher aufgrund der eingesetzten Technologie in der Anzahl möglicher Schreibzyklen begrenzt ist. Um deren Anzahl zu reduzieren, wird eine Pufferverwaltung eingesetzt. Datensätze werden solange in der Pufferverwaltung zwischengespeichert bis sie durch andere Einträge ersetzt werden. Darüber hinaus wird dadurch die Leistung gesteigert, da die Anzahl vergleichsweise langsamer Schreibzugriffe reduziert wird. Der für die Pufferverwaltung benötigte Speicher wird entweder zur Übersetzungszeit reserviert (statisch) oder zur Laufzeit dynamisch allokiert. Der Vorteil der statischen Reservierung liegt in der einfacheren Handhabung, da die fehleranfällige Verwaltung des benötigten Speichers (Allokierung und Freigabe) entfällt. Im Gegensatz dazu erlaubt die dynamische Reservierung die

Anforderung für Speicher beliebiger Größe.[4]

Transaktionsverwaltung (Transaction) Das Feature Transaktionsverwaltung setzt auf dem Feature Pufferverwaltung auf. Die in der Pufferverwaltung zwischengespeicherten Tupel werden durch Transaktionsobjekte vor externen Veränderungen geschützt. Transaktionen werden sowohl zur Sicherung von Lese- als auch Schreibzugriffen eingesetzt. Ein konkurrierender Zugriff während der Abarbeitung einer Transaktion wird bis zu deren Ende aufgeschoben und dann verarbeitet. Die Umsetzung der Transaktionsverwaltung erfordert das zeitweise Abschalten von Interrupts. Dies ist notwendig, um benötigte Datenstrukturen zu schützen.

Back-up (Backup) Durch den Einsatz der Pufferverwaltung stehen neue oder veränderte Datensätze nur im Arbeitsspeicher. Das Zusammenbrechen der Stromversorgung führt in diesem Fall zum Verlust der Daten. Um dies zu vermeiden, werden die Einträge in der Pufferverwaltung in regelmäßigen Abständen in den permanenten Datenspeicher zurückgeschrieben.

Prüfsummen (Checksum) Mit Hilfe des Prüfsummen-Features kann die Verarbeitung fehlerhafter Datensätze vermieden werden. Ausgangspunkt fehlerhafter Datensätze sind zum einen abgebrochene Speichervorgänge infolge eines Stromausfalls. Zum anderen kann der permanente Datenspeicher nach längerem Gebrauch seine Speicherfähigkeit verlieren. In beiden Fällen lassen sich fehlerhafte Datensätze durch die Gegenprüfung mit einer zum Datenelement gespeicherten Prüfsumme identifizieren.

Indexstruktur (Index) Indexstrukturen dienen dem schnelleren Auffinden von Datensätzen im DBMS und ersetzen damit die sequentielle Suche im Datenbestand. Am weitesten verbreitet sind dabei Baumstrukturen oder Hash-Verfahren. Für weitere Informationen zu beiden Ansätzen sei auf Saake et al. HSS05 verwiesen. Für RobbyDBMS wird eine Indexstruktur auf der Basis von statischem Hashing eingesetzt [ZYLK+05].

Anmerkungen zum Feature-Diagramm

Der vollständige Funktionsumfang für RobbyDBMS beträgt 16 Features, wobei eine Reihe von Features weiter unterteilt ist. So besteht das Feature zur Umsetzung von Prüfsummen (*Checksum*) aus zwei getrennten Features jeweils für Lese- und Schreibunterstützung. Dies geht auf die Modellierung der Schreibunterstützung als optionales Feature zurück.[5] Dadurch wächst die Anzahl an Features auf 19 an. Von den 19 umgesetzten Features sind 7 obligatorisch und 12 optional. Alle obligatorischen Features repräsentieren die Grundstruktur des DBMS und sind infolgedessen unverzichtbar.

Das Feature-Diagramm zeigt darüber hinaus, dass alle optionalen Features von anderen, teils obligatorischen und teils optionalen Features abhängen. Die im Feature-Diagramm verwendeten Implikationspfeile stehen stellvertretend für Referenzen oder

[4] Die zur Datenspeicherung eingesetzten Tupel sind nicht auf eine bestimmte Größe festgelegt.

[5] Dieses Phänomen wird auch als Feature-Optionality Problem bezeichnet und wird im Rahmen dieser Arbeit nicht weiter untersucht.

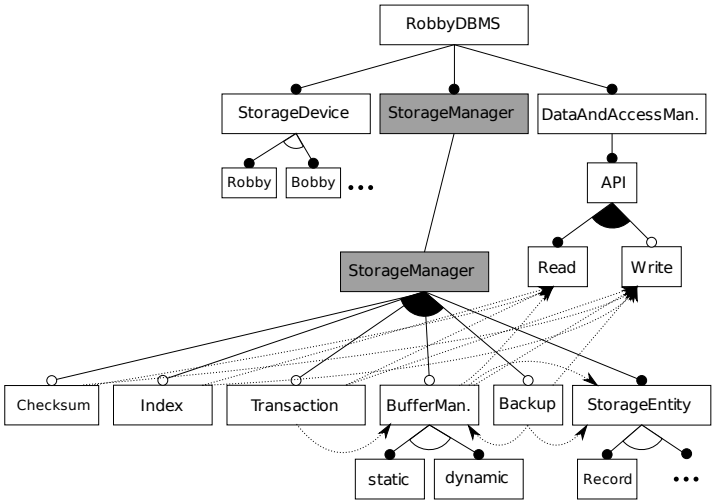


Abbildung 5.6: Feature-Diagramm RobbyDBMS

Erweiterungen anderer Features. Während Referenzen nur das Vorhandensein eines Features ausdrücken, beziehen sich Erweiterungen direkt auf Ergänzungen vorhandener Methoden und Attribute in bestehenden Klassen.

5.4 Zusammenfassung

In diesem Kapitel erfolgte eine Domänenanalyse mit Hilfe der Methode FODA für das Datenmanagement tief eingebetteter Systeme. Dazu wurden zunächst Anforderungen an das zu entwickelnde Datenmanagement erläutert. Im Einzelnen gehörten dazu die zu speichernden Daten, die zur Verfügung stehenden Datenspeicher, die Nebenläufigkeiten in tief eingebetteten Systemen und die Besonderheiten autonomer eingebetteter Systeme. In einer anschließenden Analyse wurde der Einsatz verschiedener DBMS-Lösungen diskutiert. Im ersten Schritt erfolgte ein Diskussion zum Einsatz verschiedener Mehrzweck-DBMS-Lösungen auf den vorgestellten eingebetteten Systemen. Dabei stellte sich heraus, dass die meisten Systeme nicht eingesetzt werden können, da der von ihnen benötigte Speicherbedarf, die zur Verfügung stehenden Ressourcen der Systeme z. T. bei weitem überschreitet. Nach der Analyse der Mehrzweck-DBMS erfolgte eine Betrachtung einer Auswahl verschiedener DBMS-Lösungen, die insbesondere für den Einsatz auf tief eingebetteten Systemen entworfen wurden. Dabei stellte sich heraus, dass die meisten Systeme jeweils für ein bestimmtes Anwendungsszenario entworfen wurden oder auf eine bestimmte Zielplattform ausgerichtet sind. Demzufolge ist die Wiederverwendbarkeit dieser Lösungen gering.

Die Ergebnisse dieser Analyse bildeten u. a. den Ausgangspunkt zur Entwicklung eines Feature-Diagramms für die DBMS-Lösung RobbyDBMS. Alle darin enthaltenen Features wurden mit Bezug auf ihren Einsatzzweck und auf die in der Domänenanalyse ermittelten Anforderungen erläutert.

DBMS	nicht erfüllte Voraussetzungen	Speicherverbrauch
Apache Derby[1]	setzt Betriebssystem mit Java Unterstützung voraus; Speicherverbrauch zu groß	2048 KB
Berkeley DB[2]	setzt Unterstützung durch Betriebssystem voraus (Linux, Windows u. a.); Speicherverbrauch zu groß	484 KB
DB2 Everyplace [KLLP01]	setzt Unterstützung durch Betriebssystem voraus (PalmOS, WinCE u. a.); Speicherverbrauch zu groß	150 KB
eXtremeDB[3]	Datenspeicherung im Arbeitsspeicher oder auf Festplatte	50 KB
HSQLDB[4]	setzt Betriebssystem mit Java Unterstützung voraus; Speicherverbrauch zu groß	100 - 200 KB
LGeDBMS [KBL+06]	setzt Unterstützung durch Betriebssystem voraus (REX, pSOS, QNX, Windows u. a.); Flash-Speicher; Speicherverbrauch zu groß	600 KB
Metakit[5]	setzt Unterstützung durch Betriebssystem voraus; Speicherverbrauch zu groß; für 16- bis 64-Bit Architekturen	125 KB
Oracle Lite[6]	setzt Unterstützung durch Betriebssystem voraus; Speicherverbrauch zu groß	ca. 1024 KB
RDM Embedded[7]	setzt Unterstützung durch Betriebssystem voraus (AIX, Linux u. a.); Speicherverbrauch zu groß	ca. 270 KB
SQLite[8]	setzt Unterstützung durch Betriebssystem voraus (Linux, Windows u. a.); Speicherverbrauch zu groß	225 KB

[1] http://db.apache.org/derby/
[2] http://www.oracle.com/technology/products/berkeley-db/
[3] http://www.mcobject.com/extremedbfamily.shtml
[4] http://hsqldb.org/
[5] http://www.equi4.com/metakit/
[6] http://www.oracle.com/technology/products/lite/
[7] http://www.raima.com/products/rdm-embedded/
[8] http://www.sqlite.org/

Tabelle 5.2: Auswahl existierender Mehrzweck-DBMS-Lösungen

Kapitel 6

Fallstudie RobbyDBMS

Nachdem im vorherigen Kapitel 5 (S. 59) Anforderungen und Features des zu implementierenden DBMS erörtert wurden, wird im folgenden Kapitel zum einen der Aufbau des RobbyDBMS vorgestellt. Zum anderen werden einige Aspekte der Implementierung besprochen sowie Ergebnisse der erzielten DBMS-Lösung präsentiert. Diese beziehen sich in erster Linie auf die verschiedenen generierbaren Varianten und der dazugehörigen Programmgrößen. Zur Umsetzung des DBMS wird FeatureC++ und AspectC++ herangezogen. Die Einschränkung auf die beiden Ansätze trägt dem Umstand Rechnung, dass eine Gegenüberstellung verschiedener Lösungsansätze auf der Grundlage verschiedener Programmierparadigmen, wie sie in Kapitel 4 (S. 35) am Beispiel von Gerätetreibern erfolgte, für die Entwicklung einer DBMS-Lösung im Rahmen dieser Arbeit nicht erfolgen kann.

Auswahl FeatureC++ und AspectC++

Basierend auf den Ergebnissen und Analysen dieses Kapitels werden Empfehlungen für den Einsatz moderner Programmierparadigmen für die Programmierung eingebetteter Systeme gegeben. Bevor zu den Ergebnissen der Implementierung übergegangen wird, folgen noch Erläuterungen zum Aufbau der DBMS-Lösung RobbyDBMS. Diese Erläuterungen dienen dem besseren Verständnis und fließen in die Diskussion zur Umsetzung und in die abschließende Evaluierung mit ein.

6.1 Aufbau RobbyDBMS

Für den Aufbau von RobbyDBMS wird der Architekturvorschlag der Fünf-Schichten-Architektur nach Härder [Här87] herangezogen (Abbildung 6.1). Dieser Vorschlag beschreibt ein Schichtenmodell für relationale DBMS. In den einzelnen Schichten finden unterschiedliche Transformationsprozesse statt. Durch diese Transformationen werden nutzerseitige Anfragen und Operationen an das DBMS schrittweise in Manipulationen des Datenbestandes auf dem externen Speicher umgesetzt. Zwischen benachbarten Schichten bestehen Schnittstellen für den Datenaustausch. Aufgrund der Annahmen an die zu speichernden Daten und den dazugehörigen Anfragen und Operationen (Abschnitt 5.1.1, S. 60) lässt sich das Schichtenmodell für RobbyDBMS im Umfang vereinfachen. Dazu trägt ebenso der geringe Funktionsumfang (Abschnitt 5.6, S. 70)

bei. Ein Beispiel dafür ist die Zugriffskontrolle als Bestandteil des Datensystems aus der Datensicherungshierarchie. In Saake et al. [SH00] ist zudem beschrieben, dass je nach Einsatzgebiet und Ausprägung nicht alle Bestandteile des Schichtenmodells in existierenden DBMS-Lösungen umgesetzt werden oder zwei Schichten zusammengelegt werden können. Darüber hinaus weisen die Autoren darauf hin, dass die Zuordnung von Funktionalitäten zur Herstellung der Datensicherheit (Datensicherungshierarchie) zu einzelnen Schichten nicht zwingend ist. Diverse Funktionalitäten, wie z. B. die Sperrverwaltung, lassen sich in unterschiedlichen Schichten des Modells realisieren [SH00].

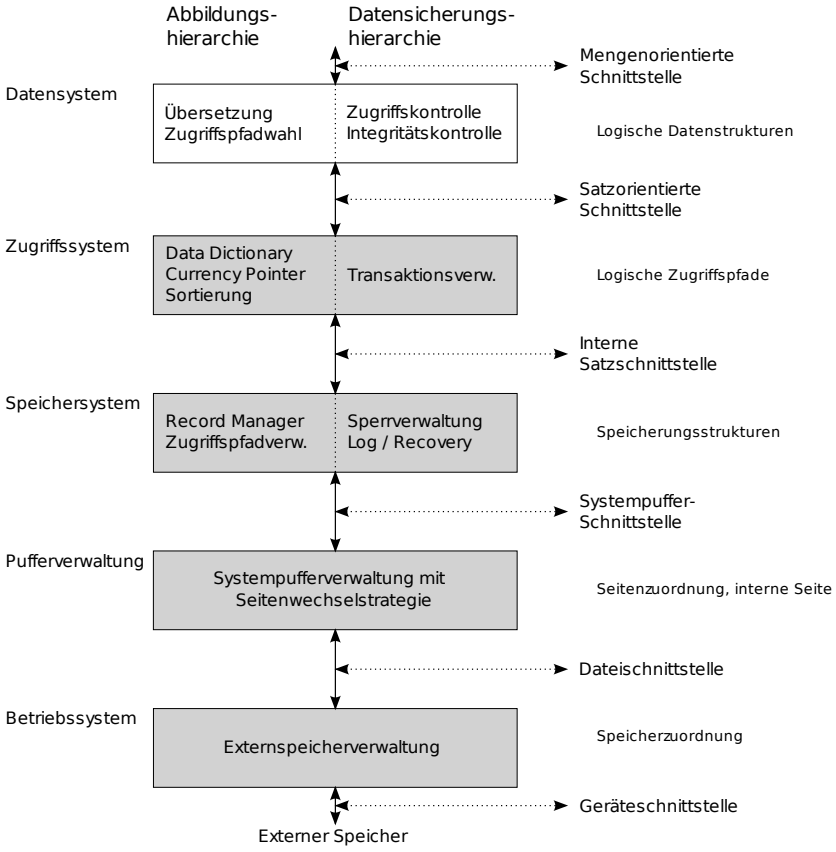


Abbildung 6.1: Funktions-Orientierte Sicht auf die Fünf-Schichten-Architektur (adaptiert aus [SH00])

Schichtenmodell RobbyDBMS

Für die Architektur RobbyDBMS werden alle grau hinterlegten Schichten der Fünf-Schichten-Architektur herangezogen. Diese werden auf ein eigenes Schichtenmodell (Abbildung 6.2) abgebildet. Da kein Betriebssystem vorausgesetzt wird, entspricht die Externspeicherverwaltung der Nutzung von Gerätetreibern. Alle für den Betrieb notwendigen Treiber werden in einer Hardware-Bibliothek zusammengefasst. Die oberhalb der Externspeicherverwaltung liegenden Schichten in der Fünf-Schichten-Architektur (Speichersystem und Pufferverwaltung) können zu einer Schicht zusammengefasst werden. Der Grund für diese Zusammenlegung ist die Modellierung der Pufferverwaltung als optionales Feature. Dadurch entfällt die Notwendigkeit eines eigenständigen Transfor-

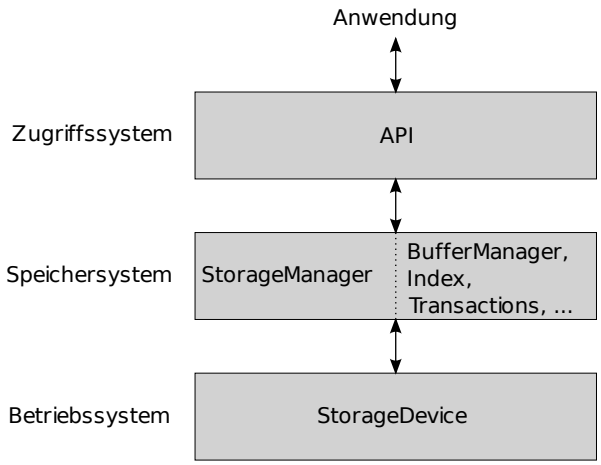


Abbildung 6.2: Schichtenmodell RobbyDBMS

mationsprozesses, der in einer separaten Schicht abgebildet werden muss.

Alle Einschränkungen und Anmerkungen aus der Domänenanalyse (Abschnitt 5, S. 59) führen im Ergebnis zu einem DBMS mit zugriffsmethodenorientierter Programmierschnittstelle [Här87]. Diese Architektur (Abbildung 6.2) stellt eine einfache *Application Programming Interface (API)* für die Datenspeicherung bereit. Mit Hilfe dieser Schnittstelle werden die internen Datenstrukturen, die im Speichersystem vorgehalten werden, direkt manipuliert. Anfragesprachen wie *Structured Query Language (SQL)* zur Manipulation von Datensätzen sind nicht vorgesehen.[1] Im folgenden Abschnitt wird die Implementierung RobbyDBMS erläutert. Dabei wird insbesondere auf die Art und die Integration der Features eingegangen.

6.2 Implementierung

Das im vorherigen Abschnitt gezeigte Schichtenmodell (Abbildung 6.2) für RobbyDBMS definiert das Grundgerüst für alle Features, die in RobbyDBMS integriert werden können. Diese Features wurden im Einzelnen bereits in Abschnitt 5.3 (S. 67) erläutert. Die Funktionalitäten der umgesetzten Features verteilen sich auf unterschiedliche Klassen. Eine Übersicht zur Verteilung der Funktionalitäten im DBMS zeigt die Abbildung 6.3.

6.2.1 Heterogene und Homogene Erweiterungen

Während der Implementierung zeigte sich, dass alle umgesetzten Features in RobbyDBMS durchweg heterogen sind. Kein Feature benötigte den Einsatz von Aspekten, um homogene Erweiterungen an verschiedenen Stellen im Basiscode umsetzen zu

[1] SQL wird in der Fünf-Schichten-Architektur nach Härder [Här87] auf einer höheren Ebene realisiert.

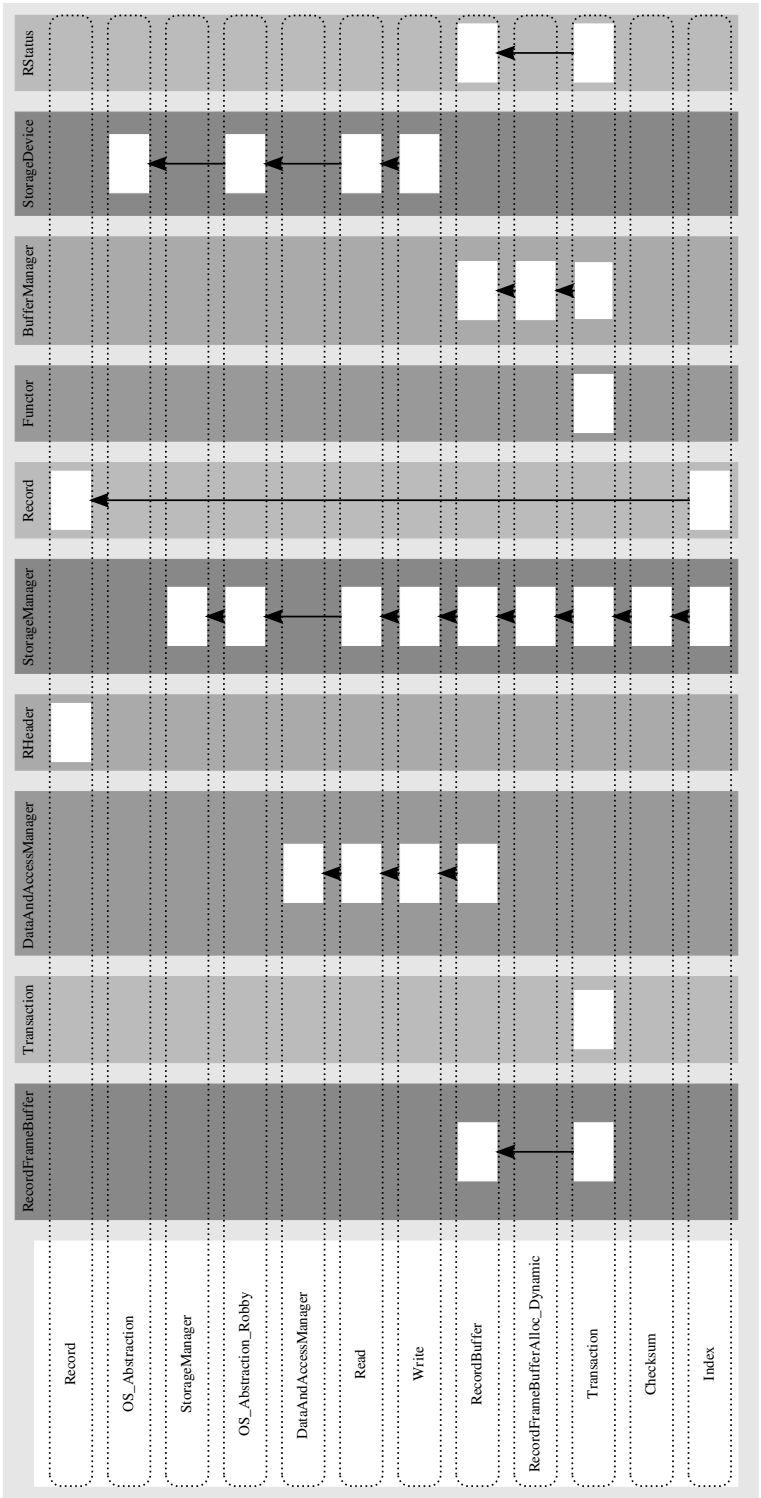


Abbildung 6.3: Kollaborationendiagramm RobbyDBMS

können. Dieses Ergebnis ist bemerkenswert, da in anderen Arbeiten [Puk06, Käs07] zu diesem Thema festgestellt wurde, dass der Anteil homogener Erweiterungen insgesamt zwar gering ausfällt, aber nie Null ist. Funktionalitäten, die in diesen Arbeiten mit Hilfe von Aspekten umgesetzt wurden, sind jedoch entweder nicht vorhanden oder werden alternativ realisiert. In Pukall [Puk06] wird zur Umsetzung von Logging-Funktionalitäten auf AOP zurückgegriffen. Durch Aspekte wird hierbei die Protokollierung von Transaktionsausführungen verwirklicht. Ein Logging-Feature ist in RobbyDBMS nicht vorgesehen, da der geringe zur Verfügung stehende Datenspeicher ausschließlich für die zu speichernden Datenelemente eingesetzt wird. In Kästner [Käs07] werden homogene Erweiterungen bspw. zur Synchronisierung konkurrierender Zugriffe eingesetzt. Die zur Synchronisierung eingesetzten Sperren erweitern ganze Methoden. Dieser Ansatz ist jedoch in RobbyDBMS nicht möglich, da Sperren in RobbyDBMS feingranularer umgesetzt werden müssen.[2]

Erklärungen warum Aspekte nicht notwendig

Der Vergleich zu einzelnen Features aus anderen Arbeiten [Puk06, Käs07] ist jedoch nicht ausreichend, um den Verzicht von Aspekten bei der Realisierung RobbyDBMS zu erklären. Der Verzicht ist auch das Ergebnis der Struktur von RobbyDBMS. Im Vergleich zu größeren Systemen wie z. B. Berkeley DBMS, weist RobbyDBMS einen geringen Funktionsumfang auf. Die Implementierung umfasst insgesamt 20 Klassen, die die entsprechenden Erweiterungen einzelner Features aufnehmen. Dabei zeigt sich, dass sich der Großteil aller Funktionalitäten auf das Speichersystem bezieht und hier insbesondere auf die Klasse *StorageManager* (Abbildung 6.3). Somit mussten nur wenige Klassen und Methoden erweitert werden, wodurch mögliche Ansatzpunkte für homogene Erweiterungen bereits aufgrund der schlanken Struktur wegfallen. Bei der Vorstellung von AspectC++ in Abschnitt 2.6.2 (S. 19) wurde bereits darauf hingewiesen, dass mit AspectC++ in erster Linie Funktionen erweitert werden können. Beliebige, feingranulare Erweiterungen auf Anweisungsebene sind nicht möglich. Somit konnten homogene Erweiterungen in dem Feature *Transaction* nicht umgesetzt werden. Diese waren bspw. notwendig, um feingranulare Sperren zum Schutz von Datenstrukturen und Datenelementen umzusetzen. FeatureC++ bietet ebenso keine Möglichkeiten zur Realisierung feingranulare Erweiterungen. Detaillierte Erläuterungen zur Umsetzung des Features *Transaction* werden in Abschnitt 6.3.2 (S. 79) gegeben.

Aufgrund der Tatsache, dass keine Aspekte zur Realisierung von Features in RobbyDBMS notwendig waren, bezieht sich die folgende Evaluierung ausschließlich auf FeatureC++.

6.3 Evaluierung RobbyDBMS

Für die Evaluierung werden die in Abschnitt 4.3 (S. 39) verwendeten Kriterien (Ressourcenverbrauch, und Erweiterbarkeit) herangezogen. Diese bildeten u. a. bereits die

[2] Sperren und Freigaben werden in RobbyDBMS durch das Abschalten bzw. Einschalten von Interrupts realisiert. Um die Reaktionsfähigkeit des eingebetteten Systems zu erhalten, sollten Interrupts nach Möglichkeit nur für wenige Programmanweisungen abgeschalten werden.

Grundlage für die Evaluierung verschiedener Gerätetreiber in Abschnitt 4.4 (S. 51).

6.3.1 Ressourcenverbrauch RobbyDBMS

Ein wesentliches Kriterium für die DBMS-Lösung ist der Verbrauch an Programm- und Arbeitsspeicher. Beide Kennzahlen spiegeln die Effizienz der Implementierung wieder und werden gemäß der Angaben aus dem Abschnitt 4.2 (S. 37) bestimmt. Für die folgende Übersicht (Tabelle 6.1) wurden eine Reihe von Beispielprogrammen geschrieben, die die jeweiligen Features einer RobbyDBMS-Variante ansprechen. Alle Varianten werden zusammen mit ihrem Verbrauch an Programmspeicher (*text*) und Arbeitsspeicher (*data+bss*) angegeben.

Programmspeicher text

Die Übersicht zeigt, dass ausgehend von einem minimalen System, das nur eine Lese-Unterstützung bietet, verschiedene DBMS-Varianten in unterschiedlichen Größen erzeugt werden können. Die Beispielprogramme wurden mit der Optimierungsstufe `-Os` übersetzt. Sie umfassen einen Lesevorgang für alle Varianten mit dem *Read*-Feature und jeweils einen Lese-, Schreib- und Löschvorgang für Systeme mit den Features *Read* und *Write*. Ein voll ausgestattetes RobbyDBMS liegt bei einem Gesamtspeicherverbrauch von 6045 Bytes. Infolgedessen kann entsprechend der Anforderungen eines gegebenen Anwendungsszenarios eine maßgeschneiderte DBMS-Lösung generiert werden. Bei der Kombination unterschiedlicher Features zeigt sich, dass der Speicherverbrauch nahezu linear mit den Features ansteigt.[3] Demzufolge entsteht durch die Feature-Komposition kaum Overhead. Insgesamt zeigen die erzielten Programmgrößen eindrucksvoll die Skalierbarkeit der DBMS-Lösung. Der geringe Speicherverbrauch von RobbyDBMS lässt dadurch ausreichend Spielraum für die Anwendungsentwicklung.

Optimierung Feature-Komposition

Bei der Untersuchung der verschiedenen Beispielprogramme (Tabelle 6.1) stellte sich heraus, dass die erzielten Programmgrößen weiter reduziert werden können. Die Grundlage dafür bildet der Einsatz von Methodenattributen. Diese wurden bereits in Abschnitt 4.4 (S. 51) besprochen. Durch ihren Einsatz konnte der Verbrauch an Programmspeicher um bis zu 6,3 % gesenkt werden.

Arbeitsspeicher data+bss

Die Übersicht in der Tabelle 6.1 spiegelt nicht den Verbrauch an Arbeitsspeicher während der Programmausführung wider. In die Kennzahl geht nicht der Speicherplatz von Variablen ein, der im Zuge der Programmausführung belegt wird. Die Bestimmung des tatsächlich verbrauchten Arbeitsspeichers setzt eine genaue Analyse der Beispielprogramme voraus. Eine solche Untersuchung geht jedoch über den Rahmen dieser Arbeit hinaus.

Einfluss der Reihenfolge von Features

Bei der Erläuterung des Feature-Diagramms in Abschnitt 5.3 (S. 67) wurden die einzelnen Features und ihre Abhängigkeiten untereinander aufgeführt. Dabei zeigte sich, dass einige Features voneinander unabhängig sind. Das bedeutet, dass die Reihenfolge ihrer Anwendung in der Feature-Komposition keinen Einfluss auf das Programmergebnis hat. In RobbyDBMS sind bspw. die beiden Features *Checksum* und *Index* von-

[3] Die Abweichung des Speicherverbrauchs zwischen der Summe einzelner Feature und der Feature in Kombination beträgt bei verschiedenen Messungen weniger als 0,3 %.

Variante	*Read*	*Write*	*BufferMan.*	*Transaction*	*Checksum*	*Index*	**text**	**data + bss**	**dec**
1	✓	o	o	o	o	o	658	0	658
2	✓	✓	o	o	o	o	996	0	996
3	✓	✓	o	o	✓	o	1278	0	1278
4	✓	✓	o	o	o	✓	1686	0	1686
5	✓	✓	o	o	✓	✓	2002	0	2002
6	✓	✓	✓	o	o	o	2294	10	2304
7	✓	✓	✓	o	✓	✓	3330	10	3340
8	✓	✓	✓	✓	✓	✓	3734	12	3734

✓ Feature Bestandteil der Variante
o Feature nicht Bestandteil der Variante

Tabelle 6.1: Übersicht Programmgrößen RobbyDBMS Varianten (in Bytes)

einander unabhängig.[4] Die Unabhängigkeit wirft die Frage auf, ob die Änderung der Reihenfolge in der Komposition unabhängiger Features Einfluss auf die Programmgröße hat. Grund zu dieser Annahme ist die Vermutung, dass sich infolge der geänderten Reihenfolge andere Synergieeffekte ergeben, die sich während der Übersetzung positiv auf die Programmgröße auswirken. Die beiden Features *Checksum* und *Index* eignen sich für diese Untersuchung, da sie abgesehen von der Unabhängigkeit untereinander eine weitere Eigenschaft teilen. Beide erweitern z. T. die gleichen Klassen (Abbildung 6.3, S. 76) und mit ihnen die gleichen Methoden.

Es zeigte sich, dass die Änderung der Reihenfolge in der Feature-Komposition von *Checksum* und *Index* einen geringen, aber messbaren Einfluss auf das übersetzte Programm hat. So führt der Wechsel in der Feature-Komposition von *Index/Checksum* in *Checksum/Index* (Tabelle 6.1, Variante 5) zu einem um 10 Bytes (ca. 0,53 %) geringeren Verbrauch an Programmspeicher. Die tiefergehende Analyse zeigte, dass die erzielten Einsparungen über das Programm verteilt sind und keiner bestimmten Stelle zugeordnet werden können. Aufgrund der geringen Einsparungen dürfte die Reihenfolge der Feature-Komposition in der Praxis jedoch kaum relevant sein.

6.3.2 Feingranulare Erweiterungen

Bei der Umsetzung der Features trat es bisweilen auf, dass die Nutzung von Methodenverfeinerungen nicht ausreichte, um ein Feature umsetzen zu können. Dies äußert sich darin, dass zum einen für die Umsetzung eines Features auf lokale Variablen zugegriffen werden musste (Abbildung 6.4 (a), Z. 3 und 4). Zum anderen mussten Anweisungen

[4] In die Prüfsummenberechnung fließen die Metadaten aus der Indexspeicherung nicht mit ein.

```
1 ...
2 bool getDataDirect(Record &r) {
3   RSIZE_u8t cs;
4   ADDRESS_u16t ca = search(r.
        header.id);
5 ...
```

(a) Zugriff auf lokale Variable

```
1  ...
2  bool removeElem(
       RecordFrameBuffer* del) {
3    RecordFrameBuffer* cur  = this
       ->rbfhead;
4    RecordFrameBuffer* prev = NULL
       ;
5
6    while (cur) {
7      if (cur == del) {
8        if (prev) {
9          prev->next = cur->next;
10       } else {
11         this->rbfhead = this->
             rbfhead->next;
12       }
13       this->numofelemsinbuf--;
14       this->bufsize -= del->r.
           header.size;
15       this->bufsize -= sizeof(
           RecordFrameBuffer);
16       free(cur);
17       return true;
18     }
19 ...
```

(b) Erweiterung um Anweisungen

Abbildung 6.4

im Rumpf einer zu verfeinernden Funktion hinzugefügt werden. Im Beispiel (Abbildung 6.4 (b)) mussten die Zeilen 7 - 16 durch eine Sperre geschützt werden, um Fehler im Programmablauf zu vermeiden.

eingeschränkter Zugriff

Sowohl FeatureC++ als auch AspectC++ bieten keine Möglichkeiten für den Zugriff auf lokale Variable oder die Umsetzung feingranularer Erweiterungen auf Anweisungsebene. Beide Probleme werden nachfolgend zusammen mit Ansätzen zur Lösung diskutiert.

lokale Variable Zugriff

Jede Methodenverfeinerung in FeatureC++ wird mit Hilfe von Funktionen umgesetzt. Dabei ruft die verfeinernde Methode jeweils die verfeinerte Methode in ihrem Kontext auf. Zur Verdeutlichung dient die Abbildung 6.5. Sie zeigt die Reihenfolge der Features (a) und das Ergebnis nach der Feature-Anwendung (b). Infolgedessen entsteht eine Funktionshierarchie. Da jede Methode ihren eigenen Gültigkeitsbereich (engl. Scope) definiert, kann in der Funktion *putDataDirect* nicht auf lokale Variable der Funktionen *Checksum_Write_putDataDirect* und *Write_putDataDirect* zugegriffen werden. Diese Einschränkung gilt gleichermaßen für Inline-Funktionen. Für die Umsetzung von Features in RobbyDBMS war jedoch der Zugriff auf lokale Variable anderer Features notwendig. Im Einzelnen mussten bspw. Zeiger manipuliert oder Zählvariablen verändert werden. Die Einschränkung des Zugriffs auf lokale Variable wurde bereits in Kästner [Käs07] thematisiert.[5] Mögliche Lösungsansätze zur Umgehung des Pro-

[5] Gleichwohl in dieser Arbeit AspectJ untersucht wurde, so treffen die gleichen Einschränkungen auch hier zu.

```
1 // all.equation
2 ...
3 Write
4 Checksum_Write
5 Index_Write
6 ...
```

(a) Reihenfolge der Feature-Komposition

```
1  inline
2  bool Write_putDataDirect(ADDRESS_u16t saddress, Record& r) {
3    ...
4  }
5
6  inline
7  bool Checksum_Write_putDataDirect(ADDRESS_u16t saddress, Record& r) {
8    bool ret = Write_putDataDirect(saddress, r);
9    ...
10 }
11
12 bool putDataDirect(ADDRESS_u16t saddress, Record& r) {
13   ...
14   if (Checksum_Write_putDataDirect(saddress, r)) {
15   ...
16 }
```

(b) Ergebnis der Feature-Komposition

Abbildung 6.5: Feature Umsetzung

blems sind:

- Änderung lokaler Variable in eine Instanzvariable und
- Einsatz von Hook-Methoden mit den zu manipulierenden lokalen Variablen als Parameter.

Instanzvariable

Gegen den Einsatz von Instanzvariablen spricht, dass sie im Vergleich zu lokalen Variablen zu einem Mehrverbrauch an Arbeits- und Programmspeicher führen. Lokale Variablen werden bevorzugt in den Registern des Prozessors gehalten, wodurch häufige Lade- und Speicherbefehle vermieden werden können. Im Beispiel Abbildung 6.6 führt die Änderung der lokalen Variablen *RSIZE_u8t cs* und *ADDRESS_u16t ca* in die Instanzvariablen *h_cs* bzw. *h_ca* zu einer Vergrößerung des übersetzten Programms von 8 Byte. *RSIZE_u8t* und *ADDRESS_u16t* sind einfache Datentypen. Aus diesem Grund fällt der Mehrverbrauch an Speicher in diesem Fall gering aus. Wenngleich die Auswirkungen in diesem Fall gering sind, so zeigte sich an anderen Stellen, dass der Einsatz von Instanzvariablen zu einem Anstieg des Programmspeicherverbrauchs um bis zu 3 % führte. Der jeweilige Anstieg hängt dabei in erster Linie vom Variablentyp und den notwendigen Operationen ab.

Hook-Methoden

Die zweite Möglichkeit zur Manipulation lokaler Variabler ist der Einsatz von Hook-Methoden. Hook-Methoden stellen explizite Erweiterungspunkte in Methodenrümpfen zu erweiternder Klassen zur Verfügung. Dazu wird eine leere Methode definiert, die

```
1  refines class StorageManager {
2    private:
3      bool getDataDirect(Record &r) {
4        RSIZE_u8t cs = 0;
5        ADDRESS_u16t ca = search(r.header.id);
6        ...
7  };
```

(a) Lokale Variable

```
1  refines class StorageManager {
2    private:
3      RSIZE_u8t h_cs;
4      ADDRESS_u16t h_ca;
5      ...
6      bool getDataDirect(Record &r) {
7        h_cs = 0;
8        h_ca = search(r.header.id);
9  ...
10 };
```

(b) Instanzvariable

Abbildung 6.6: Lokale Variable und Instanzvariable

später durch Aspekte, Feature-Module oder innerhalb einer Vererbungshierarchie erweitert wird. Dabei stellen Hook-Methoden keine Funktionalität für die Klasse selbst bereit und stehen nur für eine zu einem späteren Zeitpunkt zu definierende spezifische Erweiterung zur Verfügung. Hook-Methoden werden an entsprechender Stelle im Methodenrumpf der Basisklasse eingefügt und bekommen alle zu verändernden lokalen Variablen als Parameter übergeben. In Erweiterungen können beliebige Manipulationen an der Variablen vollzogen werden.

Den Einsatz von Hook-Methoden zur Manipulation lokaler Variable verdeutlicht die Abbildung 6.7. Durch die Hook-Methode *hook_modifyCurAddress* kann der Wert der Zählvariablen *cadd* in einem Feature verändert werden. Die Hook-Methode wird wie alle anderen zu verfeinernden Methoden in eine Inline-Hierarchie überführt. Durch Compiler-Optimierungen wird die aufgerufene Methode *hook_modifyCurAddress* vollständig in den Kontext der aufrufenden Methode *search* integriert. Das führt dazu, dass neben der Ersparnis eines Funktionsaufrufs (Sichern und Wiederherstellen des Methodenkontexts) auch Optimierungen für den Zugriff auf die zu manipulierenden Variablen erfolgen können. So wird die zusätzliche Addition zur Variablen *cadd* (Abbildung 6.7 (b)) mit anderen Operationen in der aufrufenden Methode zusammengefügt. Weitere Untersuchungen haben ergeben, dass Hook-Methoden vom Compiler wegoptimiert werden, sofern sie nicht durch ein Feature erweitert werden.

Erweiterung Anweisungsebene

Neben dem Zugriff auf lokale Variable waren auch feingranulare Erweiterungen in Methoden notwendig. Ein Beispiel dafür ist die Realisierung des Features *Transaction*. Hierbei mussten in einer existierenden Methode Erweiterungen vorgenommen werden, die eine Menge von Anweisungen umfassen. Wie eingangs des Abschnitts erwähnt stellen sowohl FeatureC++ als auch AspectC++ keine Möglichkeiten zur Verfügung, um

```
1  refines class StorageManager {
2    private:
3      void hook_modifyCurAddress(ADDRESS_u16t& cadd) {
4        return;
5      };
6
7      ADDRESS_u16t search(RID_u16t key) {
8        ADDRESS_u16t cur = 0;
9        while (cur < (sd.MAXSTORAGE-1)) {
10         ...
11         hook_modifyCurAddress(cur);
12       }
13     }
14   ...
15 };
```

(a) Definition Hook-Methode

```
1  refines class StorageManager {
2    private:
3      void hook_modifyCurAddress(ADDRESS_u16t& cadd) {
4        super::hook_modifyCurAddress(cadd);
5        cadd++;
6      }
7  };
```

(b) Verfeinerung der Hook-Methode in einem Feature

Abbildung 6.7: Einsatz Hook-Methode als expliziter Erweiterungspunkt

derartige Erweiterungen vornehmen zu können. Eine Lösung bieten, wie bereits in dem Beispiel mit den lokalen Variablen zuvor, Hook-Methoden. Sie stellen explizite Erweiterungspunkte zur Verfügung und markieren all jene Stellen in der Quellcode-Basis, an denen später Erweiterungen umgesetzt werden sollen (Abbildung 6.8). Durch die Einführung expliziter Erweiterungspunkte in Form von Hook-Methoden ändert sich die Art der Erweiterung von einem homogenen in einen heterogenen Crosscut. Die Hook-Methoden stellen nur noch einen Punkt im Programm dar, der durch Erweiterungen ergänzt werden muss.

6.4 Zusammenfassung

In diesem Kapitel wurde die Implementierung von RobbyDBMS auf der Grundlage des Feature-Diagramms (Abbildung 5.6, S. 70) erläutert. Insgesamt wurden 19 Features (davon 7 obligatorisch und 12 optional) umgesetzt. Die Realisierung der Features erfolgte dabei ausschließlich durch Feature-Module mit Hilfe von FeatureC++. Bei der Evaluierung zeigte sich, dass FOP der Erstellung einer maßgeschneiderten DBMS-Lösung entgegenkommt. Dies äußerte sich darin, dass sich alle Features in Feature-Modulen kapseln ließen und auf den Einsatz des Präprozessors cpp verzichtet werden konnte. Darüber hinaus konnte ein Mittel zur Verbesserung der Feature-Komposition aufgezeigt werden, um den Bedarf an Programmspeicher zu senken. Im Einzelnen konnten

```
1  ...
2  void hook_disableInterrupts() {};
3  void hook_enableInterrupts() {};
4
5  bool removeElem(RecordFrameBuffer* del) {
6    RecordFrameBuffer* cur  = this->rbfhead;
7    RecordFrameBuffer* prev = NULL;
8
9    while (cur) {
10     if (cur == del) {
11       hook_disableInterrupts();
12         ...
13       hook_enableInterrupts();
14       return true;
15     }
16     hook_disableInterrupts();
17       ...
18     hook_enableInterrupts();
19   }
20   return false;
21 }
22 ...
```

(a) Definition und Nutzung Hook-Methode

```
1  refines class BufferManager {
2    private:
3      void hook_disableInterrupts() {
4        super::hook_disableInterrupts();
5        cli();
6      }
7      void hook_enableInterrupts() {
8        super::hook_enableInterrupts();
9        sei();
10     }
11 };
```

(b) Verfeinerung der Hook-Methode in einem Feature

Abbildung 6.8: Einsatz Hook-Methode für Erweiterungen auf Anweisungsebene

dadurch Einsparungen von bis zu 6,3 % erzielt werden.

Ein Mangel, der im Zusammenhang mit der Umsetzung von Features für RobbyDBMS auftrat, ist die Umsetzung feingranularer Erweiterungen. Diese beziehen sich konkret auf den Zugriff lokaler Variable von Methoden eines Features und auf feingranulare Erweiterungen auf der Anweisungsebene in zu verfeinernden Methoden. Sowohl AspectC++ als auch FeatureC++ stellen keine Möglichkeiten zur Umsetzung dieser feingranularen Erweiterungen zur Verfügung. Jedoch konnte gezeigt werden, dass mit Hilfe von Hilfskonstruktionen auf der Basis von Hook-Methoden eine Realisierung feingranulare Erweiterungen möglich ist. In diesem Zusammenhang wurde festgestellt, dass sich keine Nachteile durch den Einsatz von Hook-Methoden ergeben und sie eine annehmbare Lösung zur Realisierung feingranularer Erweiterungen darstellen.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit erfolgte eine Untersuchung der Anwendung erweiterter Programmierparadigmen für die Programmierung eingebetteter Systeme. Im Einzelnen wurden dazu die beiden Paradigmen AOP und FOP anhand AspectC++ bzw. FeatureC++ untersucht. Ausgangspunkt für diese Untersuchung ist die bis dato vorherrschende Nutzung der Programmiersprache C und des Präprozessors cpp zur Umsetzung von Software-Projekten für eingebettete Systeme.

Nach der Einleitung und dem Grundlagenkapitel wurde im ersten Schritt der Einsatz des Präprozessors cpp in der Programmierung eingebetteter Systeme untersucht. Dabei stellte sich heraus, dass der Präprozessor vergleichsweise häufig zum Einsatz kommt und zur Umsetzung von SPL genutzt wird. Die Eignung des Präprozessors zur Umsetzung von SPL ist jedoch fragwürdig, da sein Einsatz mit vielen Nachteilen verbunden ist. Infolgedessen wurde die Verwendung der erweiterten Programmierparadigmen zur Umsetzung von Software-Projekten für eingebettete Systeme vorgeschlagen. Ihr Einsatz wurde in zwei Fallstudien genauer untersucht. *Kapitel 3*

In der ersten Fallstudie erfolgte die Entwicklung von zwei Gerätetreibern. Der Fokus lag dabei auf der Umsetzung von Variationen in der Hardware (HPL) in einer SPL. In diesem Zusammenhang wurden verschiedene Interaktionspunkte zwischen HPL und SPL herausgearbeitet. Bei der Entwicklung der Gerätetreiber konnte gezeigt werden, dass die Lösungsansätze auf der Basis von AspectC++ und FeatureC++ in verschiedenen Kriterien (darunter Erweiterbarkeit, Modularisierbarkeit und Wiederverwendbarkeit) anderen Lösungen, die u. a. den Präprozessor cpp zur Konfiguration einsetzen, überlegen sind. Darüber hinaus konnten keine Nachteile beim Einsatz von AspectC++ und FeatureC++ in Bezug auf den Ressourcenverbrauch festgestellt werden. *Kapitel 4*

In der zweiten Fallstudie wurde die Entwicklung eines DBMS (RobbyDBMS) mit Hilfe von AspectC++ und FeatureC++ für eingebettete Systeme untersucht. Als Grundlage dienten die Ergebnisse einer Domänenanalyse zum Datenmanagement eingebetteter Systeme. Um die Anforderungen an das Datenmanagement anwendungsgerecht umsetzen zu können, wurden die meisten Bestandteile des umgesetzten Datenmanagements als optionale Komponenten modelliert. Bei der anschließenden Implementierung zeigte sich, dass FeatureC++ zur Umsetzung der gewählten Features aus- *Kapitel 5 & 6*

reichend ist und auf den Einsatz von AspectC++ verzichtet werden konnte. In diesem Zusammenhang wurde festgestellt, dass sowohl FeatureC++ als auch AspectC++ keine Möglichkeiten bieten, um feingranulare Erweiterungen umsetzen zu können. Zur Lösung dieses Problems wurden verschiedene Ansätze diskutiert und zweckmäßige Lösungen präsentiert.

Fazit Die Arbeit hat gezeigt, dass der Kompositionsansatz der den untersuchten erweiterten Programmierparadigmen zu Grunde liegt, der Programmierung eingebetteter Systeme entgegenkommt. Dies äußert sich darin, dass durch den Einsatz von AspectC++ und FeatureC++ eine effiziente Entwicklung von SPL für tief eingebettete Systeme möglich ist. In zwei Fallstudien konnte diese Einschätzung belegt werden. Somit hat diese Arbeit gezeigt, dass erweiterter Programmierparadigmen eine gute Alternative zur Programmierspache C und der damit verbundenen Entwicklungsmethodik bieten.

Ausblick Die Ergebnisse dieser Arbeit werfen eine Reihe von Fragen auf, die in weiteren Arbeiten untersucht werden sollten. Dazu gehören im Einzelnen:

feingranulare Erweiterungen
: Bei der Umsetzung von Features zeigte sich, dass feingranulare Erweiterungen nur mit Hilfe von Hilfskonstruktionen umgesetzt werden konnten. Obwohl keine Nachteile durch den Einsatz dieser Hilfskonstruktionen festgestellt werden konnten, sollte in weiteren Arbeiten untersucht werden, wie eine direkte Unterstützung feingranularer Erweiterungen in den untersuchten Programmierparadigmen realisiert werden kann.

weitere Fallstudien
: Die in dieser Arbeit untersuchten Fallstudien zeigen Vorgehensweisen und Lösungsansätze zur Umsetzung von Software-Projekten für tief eingebettete Systeme. Aufgrund des geringen Umfangs beider Studien sollten die in den Kapiteln erarbeiteten Ergebnisse mit weiteren Fallstudien belegt werden. Dazu gehört bspw. die Umsetzung komplexerer Gerätetreiber und die Ausweitung der Untersuchung auf weitere Systeme. Darüber hinaus existieren weitere Funktionalitäten für DBMS, deren Integration in die bestehende DBMS-Lösung RobbyDBMS untersucht werden sollte. Die Untersuchung von AspectC++ kam in dieser Arbeit aufgrund mangelnder Notwendigkeit zu kurz. Aus diesem Grund sollte AspectC++ in weiteren Fallstudien untersucht werden.

FeatureC Der immer noch weit verbreitete Einsatz von C zur Programmierung eingebetteter Systeme legt die Umsetzung des Feature-Orientierten Ansatzes auch für diese Programmiersprache nahe.[1] Darüber hinaus sprechen weitere Gründe für die Umsetzung von FOP für C. Mit den Mikrocontrollern-Serien PIC von Microchip Technology Inc.[2] und F^{2}MC (8- und 16-Bit) von Fujitsu Limited[3] exis-

[1] Mit AspectC existiert bereits eine Aspekt-Orientierte Spracherweiterung für die Programmiersprache C (http://www.cs.ubc.ca/labs/spl/projects/aspectc.html).

[2] http://www.microchip.com/

[3] http://mcu.emea.fujitsu.com

tieren mindestens zwei Modellreihen für die zum gegenwärtigen Zeitpunkt keine C++-Compiler zur Verfügung stehen.[4] Darüber hinaus existieren für viele eingebettete Systeme Spezialcompiler, die mitunter nur die Programmiersprache C unterstützen. Aufgrund der Vorteile, die sich durch den Einsatz von FOP in der Programmierung eingebetteter Systeme ergeben, ist die Umsetzung des Feature-Orientierten Ansatzes für die Programmiersprache C wünschenswert.[5]

[4] Die Aussage lässt außer Betracht das Compiler existieren, die C++-Quellcode in C-Quellcode transformieren können.

[5] In dem Projekt FeatureHouse wird bereits an der Programmiersprachen-unabhängigen Feature-Komposition gearbeitet (http://www.infosun.fim.uni-passau.de/cl/staff/apel/FH/).

Literaturverzeichnis

[AB08] Apel, Sven ; Batory, Don S.: How AspectJ is Used: An Analysis of Eleven AspectJ Programs / Universität Passau, Fakultät für Informatik und Mathematik. Passau, April 2008 (MIP-0801). – Forschungsbericht

[ABP06] Anciaux, Nicolas ; Bouganim, Luc ; Pucheral, Philippe: Smart Card DBMS: where are we now? / Institut National de Recherche en Informatique et Automatique (INRIA). Le Chesnay Cedex, France, Juni 2006 (80840). – Forschungsbericht

[ALRS05] Apel, Sven ; Leich, Thomas ; Rosenmüller, Marko ; Saake, Gunter: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)* Bd. 3676, Springer-Verlag, 2005 (Lecture Notes in Computer Science), S. 125–140

[ALS06] Apel, Sven ; Leich, Thomas ; Saake, Gunter: Aspectual Mixin Layers: Aspects and Features in Concert. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM Press, 2006, S. 122–131

[ALS08] Apel, Sven ; Leich, Thomas ; Saake, Gunter: Aspectual Feature Modules. In: *IEEE Transactions on Software Engineering* 34 (2008), Nr. 2, S. 162–180

[Ape07] Apel, Sven: *The Role of Features and Aspects in Software Development.* Magdeburg, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Diss., 2007

[Atm07] Atmel Corporation (Hrsg.): *8-bit AVR Microcontroller with 32K/64K/128K Bytes of ISP Flash and CAN Controller.* San Jose, CA, USA: Atmel Corporation, Juli 2007. – Version 7679E

[Bar99] Barr, Michael: *Programming Embedded Systems in C and C++.* Sebastopol, CA, USA : O'Reilly & Associates, Inc., 1999

[Bar06] Barr, Michael: *Programming Embedded Systems.* Sebastopol, CA, USA : O'Reilly Media, Inc., 2006

[BHK02] Bollow, Friedrich ; Homann, Matthias ; Köhn, Klaus-Peter: *C und C++ für Embedded Systems*. Bonn : mitp Verlag, 2002

[BKR+05] Biagosch, Andreas ; Knupfer, Stefan ; Radtke, Philipp ; Näher, Ulrich ; Zielke, Andreas E.: Automotive Electronics - Managing Innovations on the Road / McKinsey & Company, Inc. Düsseldorf, Juli 2005. – Forschungsbericht

[BSR04] Batory, Don S. ; Sarvela, Jacob N. ; Rauschmayer, Axel: Scaling Step-Wise Refinement. In: *IEEE Transactions on Software Engineering* 30 (2004), Nr. 6, S. 355–371

[CE99] Czarnecki, Krzysztof ; Eisenecker, Ulrich W.: Components and Generative Programming. In: *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)* Bd. 1687, Springer-Verlag, 1999 (Lecture Notes in Computer Science), S. 2–19

[CE00] Czarnecki, Krzysztof ; Eisenecker, Ulrich: *Generative Programming: Methods, Tools, and Applications*. Indianapolis, IN, USA : Addison-Wesley, 2000

[CGOZ99] Cappelletti, Paulo (Hrsg.) ; Golla, Carla (Hrsg.) ; Olivo, Piero (Hrsg.) ; Zanoni, Enrico (Hrsg.): *Flash Memories*. Norwell, MA, USA : Kluwer Academic Publishers, 1999

[CRKH05] Corbet, Jonathan ; Rubini, Alessandro ; Kroah-Hartmann, Greg: *Linux Device Drivers*. Sebastopol, CA, USA : O'Reilly Media, Inc., 2005

[DST01] DST GmbH - Elektronik-Bauteile Vertrieb (Hrsg.): *AV1640*. Germering: DST GmbH - Elektronik-Bauteile Vertrieb, Januar 2001. – Version SMT99007

[EA06] Eaddy, Marc ; Aho, Alfred V.: Statement Annotations for Fine-Grained Advising. In: *Proceedings of the ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, 2006, S. 89–99

[EAK+01] Elrad, Tzilla ; Aksit, Mehmet ; Kiczales, Gregor ; Lieberherr, Karl J. ; Ossher, Harold: Discussing Aspects Of AOP. In: *Communications of the ACM* 44 (2001), Nr. 10, S. 33–38

[EBN02] Ernst, Michael D. ; Badros, Greg J. ; Notkin, David: An Empirical Analysis of C Preprocessor Use. In: *IEEE Transactions on Software Engineering* 28 (2002), Nr. 12, S. 1146–1170

[FC08] Figueiredo, Eduardo ; Cacho, Nélio: Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM Press, 2008, S. 261–270

[Gan91] Ganssle, Jack G.: *The Art of Programming Embedded Systems*. Orlando, FL, USA : Academic Press, Inc., 1991

[Gol04] Goldthwaite, Lois: Technical Report on C++ Performance / International Organization for Standardization, International Electrotechnical Commission. Geneva, Switzerland, Juli 2004 (TR 18015:2004(E)). – Forschungsbericht

[Här87] Härder, Theo: Realisierung von operationalen Schnittstellen. In: Lockemann, Peter C. (Hrsg.) ; Schmidt, Joachim W. (Hrsg.): *Datenbankhandbuch*. Berlin : Springer-Verlag, 1987, S. 163–335

[HG06] Harbulot, Bruno ; Gurd, John R.: A join point for loops in AspectJ. In: *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, ACM Press, 2006, S. 63–74

[HKM+05] Helmerich, Alfred ; Koch, Nora ; Mandel, Luis ; Braun, Peter ; Dornbusch, Peter ; Gruler, Alexander ; Keil, Patrick ; Leisibach, Roland ; Romberg, Jan ; Schätz, Bernhard ; Wold, Thomas ; Wimmel, Guido: Study of Worldwide Trends and R&D Programmes in Embedded Systems in View of Maximising the Impactof a Technology Platform in the Area / F.A.S.T. Gesellschaft für angewandte Softwaretechnologie mbH, Technische Universität München. München, November 2005 (FAST 2005). – Forschungsbericht

[Joh03] Johansson, Evert: Device Drivers and the Special Function Register Hell. In: *Atmel Applications Journal* 1 (2003), Nr. 1, S. 18,40

[JV02] Josuttis, Nicolai M. ; Vandevoorde, David: *C++ Templates: The Complete Guide*. Amsterdam, Holland : Addison-Wesley, 2002

[KAB07] Kästner, Christian ; Apel, Sven ; Batory, Don S.: A Case Study Implementing Features Using AspectJ. In: *Proceedings of the International Software Product Line Conference (SPLC)*, IEEE Computer Society, 2007, S. 223–232

[KAK08] Kästner, Christian ; Apel, Sven ; Kuhlemann, Martin: Granularity in Software Product Lines. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, ACM Press, 2008, S. 311–320

[KAL07] Kuhlemann, Martin ; Apel, Sven ; Leich, Thomas: Streamlining Feature-Oriented Designs. In: *Proceedings of the ETAPS International Symposium on Software Composition (SC)* Bd. 4829, Springer-Verlag, 2007 (Lecture Notes in Computer Science), S. 177–184

[Käs07] Kästner, Christian: *Aspect-Oriented Refactoring of Berkeley DB*. Magdeburg, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Diplomarbeit, 2007

[KBL+06] Kim, Gye-Jeong ; Baek, Seung-Cheon ; Lee, Hyun-Sook ; Lee, Han-Deok ; Joe, Moon J.: LGeDBMS: a Small DBMS for Embedded System with Flash Memory. In: *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, ACM Press, 2006, S. 1255–1258

[KCH+90] Kang, Kyo C. ; Cohen, Sholom G. ; Hess, James A. ; Novak, William E. ; Peterson, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study / Carnegie-Mellon University, Software Engineering Institute. Pittsburgh, PA, USA, November 1990 (CMU/SEI-90-TR-21). – Forschungsbericht

[KLA06] Kuhlemann, Martin ; Leich, Thomas ; Apel, Sven: Einfluss erweiterter Programmier-Paradigmen auf die Entwicklung eingebetteter DBMS. In: *Proceedings of the GI-Workshop on the Foundations of Databases*, Institute of Computer Science, Martin-Luther-University, 2006, S. 100–104

[KLLP01] Karlsson, Jonas S. ; Lal, Amrish ; Leung, Cliff ; Pham, Thanh: IBM DB2 Everyplace: A Small Footprint Relational Database System. In: *Proceedings of the International Conference on Data Engineering (ICDE)*, IEEE Computer Society, 2001, S. 230–232

[KLM+97] Kiczales, Gregor ; Lamping, John ; Mendhekar, Anurag ; Maeda, Chris ; Lopes, Cristina V. ; Loingtier, Jean-Marc ; Irwin, John: Aspect-Oriented Programming. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* Bd. 1241, Springer-Verlag, 1997 (Lecture Notes in Computer Science), S. 220–242

[Kri05] Krishnan, Ravi: Future of Embedded Systems Technology / BCC Research. Wellesley, MA, USA, Juni 2005 (GB-IFT016B). – Forschungsbericht

[Kuh06] Kuhlemann, Martin: *Moderne Modularisierungstechniken und ihre Bedeutung für qualitativ hochwertige Software*. Magdeburg, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Diplomarbeit, 2006

[Lin07] Lindhorst, Timo: *Wear Out Behavior of NAND Flash Memory.* Magdeburg, 2007. – Studienarbeit

[LPCS04] Levis, Philip ; Patel, Neil ; Culler, David E. ; Shenker, Scott: Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In: *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI)*, USENIX Association Berkeley, 2004, S. 15–28

[LSSP06] Lohmann, Daniel ; Spinczyk, Olaf ; Schröder-Preikschat, Wolfgang: Lean and Efficient System Software Product Lines: Where Aspects Beat Objects. In: *Transactions on Aspect-Oriented Software Development II (TAOSD)* Bd. 4242, Springer-Verlag, 2006 (Lecture Notes in Computer Science), S. 227–255

[Man04] Mann, Richard: How to Program a 8-bit Mikrocontroller Using C Language. In: *Atmel Applications Journal* 1 (2004), Nr. 3, S. 14–16

[MFHH05] Madden, Samuel ; Franklin, Michael J. ; Hellerstein, Joseph M. ; Hong, Wei: TinyDB: An Acquisitional Query Processing System for Sensor Networks. In: *ACM Transactions on Database Systems (TODS)* 30 (2005), Nr. 1, S. 122–173

[Nor02] Northrop, Linda M.: SEI's Software Product Line Tenets. In: *IEEE Software* 19 (2002), Nr. 4, S. 32–40

[NTNH03] Nyström, Dag ; Tešanović, Aleksandra ; Norström, Christer ; Hansson, Jörgen: The COMET Database Management System / Mälardalen University, Real-Time Research Centre. Västerås, Sweden, April 2003 (MDH-MRTC-98/2003-1-SE). – Forschungsbericht

[Nys03] Nyström, Dag: *COMET: A Component-Based Real-Time Database for Vehicle Control-Systems.* Västerås, Sweden : Licentiate Thesis, Mai 2003

[Nys05] Nyström, Dag: *Data Management in Vehicle Control-Systems.* Västerås, Sweden, Mälardalen University, Department of Computer Science and Electronics, Diss., 2005

[Par76] Parnas, David L.: On the Design and Development of Program Families. In: *IEEE Transactions on Software Engineering* 2 (1976), Nr. 1, S. 1–9

[PBVB01] Pucheral, Philippe ; Bouganim, Luc ; Valduriez, Patrick ; Bobineau, Christophe: PicoDBMS: Scaling Down Database Techniques for the Smartcard. In: *The International Journal on Very Large Data Bases (VLDB)* 10 (2001), Nr. 2-3, S. 120–132

[Pla97] Plauger, Philip J.: Embedded C++: An Overview. In: *Embedded Systems Programming* 10 (1997), Nr. 12, S. 40–52

[Pre97] Prehofer, Christian: Feature-Oriented Programming: A Fresh Look at Objects. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* Bd. 1241, Springer-Verlag, 1997 (Lecture Notes in Computer Science), S. 419–443

[Puk06] Pukall, Mario: *FAME-DBMS: Entwurf ausgewählter Aspekte der Transaktionsverwaltung*. Magdeburg, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Diplomarbeit, 2006

[Pur05] pure-systems GmbH (Hrsg.): *AspectC++ Language Reference*. Magdeburg: pure-systems GmbH, März 2005. – Version 1.6

[RLA07] Rosenmüller, Marko ; Leich, Thomas ; Apel, Sven: Konfigurierbarkeit für ressourceneffiziente Datenhaltung in eingebetteten Systemen am Beispiel von Berkeley DB. In: *Proceedings of the Workshop on Business, Technologie und Web (BTW)*, Verlagshaus Mainz, Aachen, 2007, S. 329–341

[Ros05] Rosenüller, Marko: *Merkmalsorientierte Programmierung in C++*. Magdeburg, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Diplomarbeit, 2005

[RSS+08] Rosenmüller, Marko ; Siegmund, Norbert ; Schirmeier, Horst ; Sincero, Julio ; Apel, Sven ; Leich, Thomas ; Spinczyk, Olaf ; Saake, Gunter: FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In: *Proceedings of the EDBT Workshop on Software Engineering for Tailor-made Data Management (SETMDM)*, ACM Press, 2008, S. 1–6

[RWL95] Reenskaug, Trygve ; Wold, Per ; Lehne, Odd A.: *Working With Objects: The OOram Software Engineering Method*. Greenwich, CT, USA : Prentice Hall, 1995

[SB02] Smaragdakis, Yannis ; Batory, Don S.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11 (2002), Nr. 2, S. 215–255

[SC92] Spencer, Henry ; Collyer, Geoff: #ifdef Considered Harmful, or Portability Experience With C News. In: *Proceedings of the USENIX Technical Conference*, USENIX Association Berkeley, 1992, S. 185–197

[Sch79] Schaefer, Eugen: *Zuverlässigkeit, Verfügbarkeit und Sicherheit in der Elektronik : eine Brücke von der Zuverlässigkeitstheorie zu den Aufgaben der Zuverlässigkeitspraxis*. Würzburg : Vogel Verlag, 1979

[Sch02] Schulze, Michael: *Maßgeschneidertes Planungswesen in Betriebssystemfamilien*. Magdeburg, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Diplomarbeit, 2002

[Sch05] Scholz, Peter: *Softwareentwicklung eingebetteter Systeme*. Berlin : Springer-Verlag, 2005

[Ser97] Serrano, Manuel: Inline Expansion: When and How? In: *Proceedings of the International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP)* Bd. 1292, Springer-Verlag, 1997 (Lecture Notes In Computer Science), S. 143–157

[SGSP02] Spinczyk, Olaf ; Gal, Andreas ; Schröder-Preikschat, Wolfgang: AspectC++: An Aspect-Oriented Extension to C++. In: *Proceedings of International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)* Bd. 21, Australian Computer Society, 2002, S. 53–60

[SH00] Saake, Gunter ; Heuer, Andreas: *Datenbanken: Konzepte und Sprachen*. Bonn : mitp Verlag, 2000

[SHS05] Saake, Gunter ; Heuer, Andreas ; Sattler, Kai-Uwe: *Datenbanken: Implementierungstechniken*. Bonn : mitp Verlag, 2005

[SOP+04] Szewczyk, Robert ; Osterweil, Eric ; Polastre, Joseph ; Hamilton, Michael ; Mainwaring, Alan ; Estrin, Deborah: Habitat Monitoring with Sensor Networks. In: *Communications of the ACM* 47 (2004), Nr. 6, S. 34–40

[Spi02] Spinczyk, Olaf: *Aspektorientierung und Programmfamilien im Betriebssystembau*. Magdeburg, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Diss., 2002

[SPLSS07] Schröder-Preikschat, Wolfgang ; Lohmann, Daniel ; Scheler, Fabian ; Spinczyk, Olaf: Dimensions of Variability in Embedded Operating Systems. In: *Informatik - Forschung und Entwicklung* 22 (2007), Nr. 1, S. 5–22

[SSPSS98] Schön, Friedrich ; Schröder-Preikschat, Wolfgang ; Spinczyk, Olaf ; Spinczyk, Ute: Design Rationale of the PURE Object-Oriented Embedded Operation System. In: *Proceedings of the IFIP International Workshop on Distributed and Parallel Embedded Systems (DIPES)*, Kluwer Academic Publishers, 1998, S. 231–240

[Ste06] Steimann, Friedrich: The Paradoxical Success of Aspect-Oriented Programming. In: *Proceedings of the International Conference on Object Oriented-Programming Systems Languages and Applications (OOPSLA)*, ACM Press, 2006, S. 481–497

[Str94] Stroustrup, Bjarne: *Design und Entwicklung von C++.* Bonn : Addison-Wesley, 1994

[Str00] Stroustrup, Bjarne: *The C++ Programming Language.* Amsterdam, Holland : Addison-Wesley, 2000

[Str02] Stroustrup, Bjarne: C and C++: Siblings. In: *C/C++ Users Journal* 20 (2002), Nr. 7, S. 28, 30, 32–36

[Str04] Stroustrup, Bjarne: Abstraction and the C++ Machine Model. In: *Proceedings of the International Conference on Embedded Software and Systems (ICESS)* Bd. 3605, Springer-Verlag, 2004 (Lecture Notes in Computer Science), S. 1–13

[Str07] Stroustrup, Bjarne: Evolving a language in and for the real world: C++ 1991-2006. In: *Proceedings of the Conference on History of Programming Languages (HOPL)*, ACM Press, 2007, S. 1–59

[Ten00] Tennenhouse, David: Proactive Computing. In: *Communications of the ACM* 43 (2000), Nr. 5, S. 43–50

[TOHSMS99] Tarr, Peri ; Ossher, Harold ; Harrison, William ; Stanley M. Sutton, Jr.: N degrees of separation: multi-dimensional separation of concerns. In: *Proceedings of the International Conference on Software Engineering (ICSE)*, IEEE Computer Society, 1999, S. 107–119

[Tur02] Turley, Jim: *The Essential Guide to Semiconductor Technology.* Upper Saddle River, NJ, USA : Prentice Hall Press, 2002

[Tur08] Turley, Jim: *Intel Has Just 2% Market Share!* 2008. – `http://www.jimturley.com/articles.htm#2percent`; [Online; Stand 29.08.2008]

[VDI2422] *Entwicklungsmethodik für Geräte mit Steuerung durch Mikroelektronik.* Februar 1994 VDI/VDE Richtlinie 2422

[Wir71] Wirth, Niklaus: Program Development by Stepwise Refinement. In: *Communications of the ACM* 14 (1971), Nr. 4, S. 221–227

[Zur00] Zurell, Kirk: *C Programming for Embedded Systems.* Lawrence, KS, USA : R&D Books, 2000

[ZYLK+05] Zeinalipour-Yazti, Demetrios ; Lin, Song ; Kalogeraki, Vana ; Gunopulos, Dimitrios ; Najjar, Walid A.: MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. In: *Proceedings of the Conference on File and Storage Technologies (FAST)*, USENIX Association Berkeley, 2005, S. 31–44

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 28.11.2008

Jörg Liebig