Models of Variability - Produktlinienentwicklung mit KOALA

Sergej Fries
251296

Betreut von Dipl.-Inform. Christian Fuß

Zusammenfassung

Die rasante Entwicklung der Softwaresysteme in der Unterhaltungsindustrie macht den Einsatz neuer Entwicklungstechniken unabdingbar, um auf dem Markt konkurrenzfähig zu bleiben. Zu diesen Techniken gehört auch die sogenannte Produktlinienentwicklung. Dabei wird eine Menge von ähnlichen Produkten gleichzeitig entwickelt und produziert. Der Einsatz dieser Technik führt zur Reduzierung der Entwicklungskosten und Steigerung der Entwicklungsgeschwindigkeit.
In dieser Ausarbeitung werden Wege aufgezeigt, wie verschiedene Formen der Variabilität, die die Unterschiede zwischen den einzelnen Produkten in der Produktlinie beschreiben, mit Hilfe des Komponentenmodells KOALA von Philips realisiert werden können.

Inhaltsverzeichnis

1 Einleitung **195**

2 Grundlagen **196**

2.1 Komponentenmodelle 196

2.2 Produktlinienentwicklung 197

3 Formen der Variabilität und ihr Einsatz **199**

3.1 Architekturdesignphase 200

3.2 Detaillierte Design-Phase 203

3.3 Implementierungsphase 205

3.4 Linkphase 206

4 KOALA **207**

4.1 Features von KOALA 208

4.2 Implementierung der Variabilität in KOALA 213

4.3 Fazit 219

5 Zusammenfassung **219**

1 Einleitung

„Alle 18 Monate verdoppelt sich die Leistung der Prozessoren“ besagt das Mooresche Gesetz. Diese Entwicklung gilt jedoch nicht nur in der Hardware-Branche. Die gleiche Beobachtung wird auch in der Software-Industrie gemacht. Die Komplexität der Software und ihre Größe wächst genauso wie die Prozessorleistung exponentiell. Dabei ist nicht nur der Markt für Anwender-Software, sondern auch die Branche der eingebetteten Systeme von dieser Entwicklung betroffen. So haben viele moderne Geräte in der Unterhaltungsindustrie bereits mehrere Megabyte an Software. Im Vergleich dazu reichte in den 70-er Jahren noch 1 kB Speicher und ein 8-Bit Prozessor aus, um die komplette Steuerung zu übernehmen. Bezeichnend für den heutigen Markt ist außerdem eine große Palette der gleichzeitig angebotenen Produktvarianten, wie zum Beispiel bei Mobiltelefonen. Jedes dieser Produkte braucht aufgrund der unterschiedlichen Hardware unterschiedliche Software, so dass schnelles und effektives Entwickeln der Software eine immer größere Bedeutung bekommt.
Um diesen steigenden Anforderungen gerecht zu werden stehen zwei Lösungen zur Verfügung. Zum Einen kann die Anzahl der Entwickler jährlich vergrößert werden, wodurch jedoch mit dem Wachsen der Kosten für die Entwicklung in mindestens der gleichen Größenordnung gerechnet werden muss, oder es muss der Einsatz anderer Techniken, wie z. B. *Produktlinienentwicklung*, die neuen Anforderungen erfüllen. Produktlinienentwicklung ist ein Verfahren bei dem eine Vielzahl verschiedener aber ähnlicher Produkte parallel entwickelt und produziert wird. Die Voraussetzung, dass Produkte einer Produktlinie ähnlich sein müssen, setzt sicherlich gewisse Grenzen für den Einsatz dieser Technik. Im Falle der Unterhaltungsindustrie ist diese Voraussetzung jedoch oft erfüllt.
Das Ziel dieser Ausarbeitung ist eine kritische Auseinandersetzung mit dem Komponentenmodell KOALA der Firma Philips. Der Hauptaugenmerk wird dabei auf die Möglichkeiten gelegt, die dieses Modell für Softwareentwicklung in Produktlinien zur Verfügung stellt.
Die Ausarbeitung gliedert sich folgendermaßen. Im Kapitel 2 werden die Begriffe Produktlinienentwicklung und Komponentenmodelle erklärt. Kapitel 3 stellt verschiedene Formen der Variabilität vor, die in Produktlinien zu finden sind . Im Kapitel 4 wird das Komponentenmodell KOALA vorgestellt und Wege der Implementierung für die Variabilitätsarten aus Kapitel 3 erarbeitet und erklärt. Schließlich werden die Resultate im Kapitel 5 zusammengefasst und ein Ausblick gegeben.

2 Grundlagen

2.1 Komponentenmodelle

Im folgenden Abschnitt wird erklärt was ein Komponentenmodell ist, was es leistet und welche Unterschiede zu objektorientierten Frameworks existieren.
Die Basiseinheit eines solchen Modells ist die *Komponente*, sie ist eine Einheit der Wiederverwendung, die durch folgende Merkmale beschrieben werden kann [5]:

- *Schnittstellendefiniertheit* - Eine Komponente besitzt eine wohldefinierte Schnittstelle zum Rest des Systems. Diese Schnittstelle beschreibt ihre Funktionalität.
- *Ersetzbarkeit* - Eine Komponente kann durch eine andere mit dem gleichen Interface ersetzt werden, ohne dass es Auswirkungen auf das Restsystem hat.
- *Identifizierbarkeit* - Der Funktionsumfang einer Komponente muss erkennbar sein. In der Praxis wird das oft durch eindeutige Benennung erreicht. Ist die eindeutige Benennung nicht möglich, so ist die Funktionalität der Komponente vermutlich nicht klar abgegrenzt, was zu Schwierigkeiten beim Einsatz führen kann.
- *Abgeschlossenheit* - Die Funktionalität der Komponente darf nicht von externen Komponenten abhängig sein. Es muss eine in sich abgeschlossene Entität sein. Ist die Unabhängigkeit nicht möglich, so kann eine zusammengestellte Komponente konstruiert werden, die abgeschlossen ist. Dadurch verringert sich jedoch ihr Wiederverwendungswert, da größere Komponenten im Allgemeinen spezifischer, als kleine Komponenten sind.
- *Unabhängigkeit* - Eine Komponente soll nicht von dem Kontext abhängen, in dem sie eingesetzt wird, da dies dem Prinzip der Wiederverwendung widerspricht.

Unter einem *Komponentenmodell* wird eine konkrete Ausprägung des Paradigmas der komponentenbasierten Entwicklung verstanden.[1] Es handelt sich also um eine Sammlung verschiedener Komponenten, ihrer Dokumentation und der Beschreibung wie die Komponenten zu verwenden sind. Die Idee der komponentenbasierten Programmierung basiert auf der Wiederverwendung der vorhandenen

[1] `http://de.wikipedia.org/wiki/Komponentenmodell`

Komponenten als Bausteine zur Implementierung eines Produktes. Die Komponenten bilden das Grundgerüst, das an wohldefinierten Stellen erweitert werden muss, um die komplette Funktionalität eines Programms zu erhalten.
Ein Komponentenmodell unterscheidet sich von einem OO-Framework in der Hinsicht, dass ein Framework in sich unflexibel ist. Es kann erweitert bzw. verfeinert werden, aber es stellt eine bestimmte Domäne dar, die nicht verändert werden kann. In einem Komponentenmodell kann im Gegensatz dazu durch Neuordnung der Komponenten eine neue Architektur entstehen.

Zur Implementierung der Komponenten können aber durchaus OO-Frameworks benutzt werden. Dadurch wird eine hohe Flexibilität des Gesamtsystems erreicht, da sowohl einzelne Komponente als auch die Architektur Möglichkeiten zur Veränderung bieten.

2.2 Produktlinienentwicklung

Das Ziel einiger Komponentenmodelle ist die Tauglichkeit für die Produktlinienentwicklung. Wie der Name „Produktlinie“ vermuten lässt, wird dabei eine ganze Reihe verschiedener aber ähnlicher Produkte - im Falle der Softwareindustrie sind das Softwaresysteme - gleichzeitig entwickelt bzw. hergestellt. Eine genauere Definition liefert SEI[2]:

> Eine Softwareproduktlinie ist eine Menge von Softwaresystemen, die gemeinsame Features besitzen, und spezifische Bedürfnisse eines Marktsegments oder einer Aufgabe erfüllen und aus einem gemeinsamen Basisbestand in vorgeschriebener Weise entwickelt werden.

Produkte einer Produktlinie zeichnen sich durch Vorhandensein einer gemeinsamen Basis aus, d. h. sie unterscheiden sich nur an wohldefinierten Stellen.
Das Vorhandensein einer großen gemeinsamen Basis bringt den Vorteil, dass die Entwicklung neuer Produkte sich auf die Spezifizierung und Implementierung der Unterschiede reduziert, wodurch zum Einen die Zeit bis zur Herausgabe des fertigen Produktes sich verkürzt und zum Anderen im Allgemeinen die Anzahl der Fehler im gesamten System sinkt, da alle Produkte die gleiche Basis haben und diese immer wieder weiterentwickelt und korrigiert wird, sobald Fehler festgestellt werden. Unter anderem genau wegen dieser höheren Qualität, die die Produktlinienentwicklung bietet, sind Produktlinien im Bereich der eingebetteten Systeme interessant. Während es im Falle eines Fehler in einer „gewöhnlichen“ An-

[2]Carnegie Mellon Software Engineering Institute, `http://www.sei.cmu.edu/productlines/index.html`

wendung möglich ist den Fehler durch eine neue Version des Programms zu korrigieren, ist es nur schwer vorstellbar, wie z. B. Millionen verkaufter Fernseher umprogrammiert werden müssen, falls ein schwerwiegender Fehler gefunden wird.
Im Folgenden wird auf einen wichtigen Begriff aus dem Bereich der Produktlinien eingegangen - die *Variabilität*.

Unter *Variabilität* wird Veränderungs- bzw. Anpassungsfähigkeit eines Systems verstanden. Abbildung 1 zeigt ein Beispiel für einen Feature-Graphen eines Automobil-Multimediasystems. Ein Feature-Graph ist eine graphische Darstellung der vorkommenden Komponenten in einem System.Das CarMultimedia-System besteht aus drei Komponenten: einem Abspielgerät, das Audio-CDs, MP3-CDs oder DVDs abspielen kann, einem TV-Tuner zum Empfang von Fernsehsignalen und optional einem GPS-Navigationssystem.

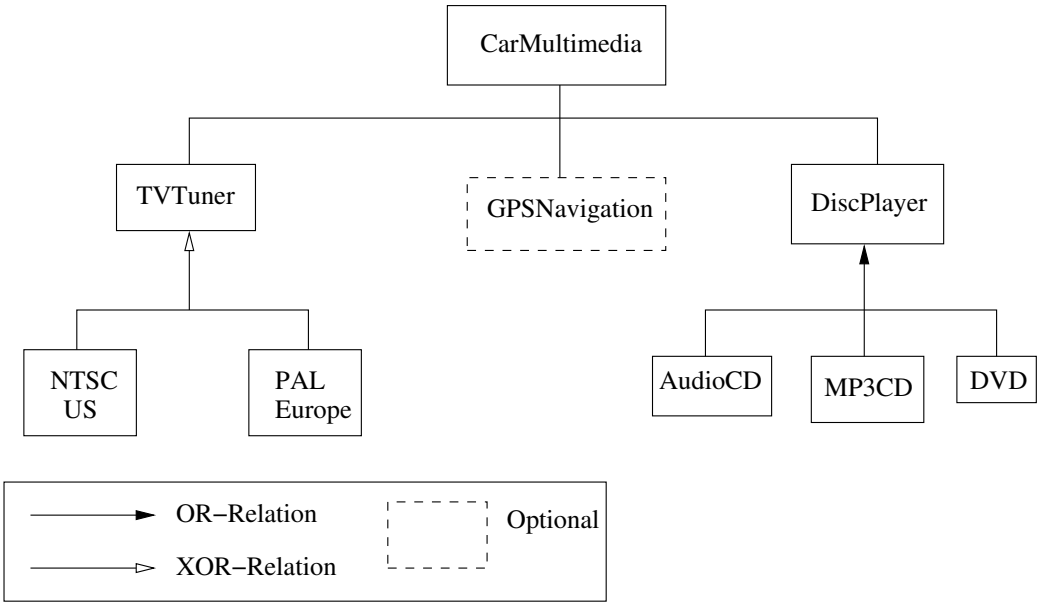

Abbildung 1: Featuregraph eines CarMultimedia-Systems

Die Implementierung der einzelnen Varianten des Multimediasystems für verschiedene Autos wird also ähnlich sein, aber nicht gleich. Es sind nur bestimmte Punkte, die das Endprodukt unterschiedlich machen, in diesem Fall sind das der DicsPlayer, der TVTuner und die GPSNavigation. Aber trotzdem können sogar in diesem kleinen Beispiel über zwanzig verschiedene Varianten abgeleitet werden. Diese Fähigkeit des Systems, verschiedene Produkte aus der gleichen Architektur hervorzubringen, wird Variabilität genannt. Das Beispiel zeigt, dass Variabilität bereits auf der höchsten Ebene der Softwareentwicklung - der Anforderungsspezifikation - zu finden ist. Natürlich gibt es die Variabilität auch auf den anderen

Stufen des Entwicklungsprozesses. Im nächsten Kapitel werden unterschiedliche Formen der Variabilität vorgestellt und erläutert.

3 Formen der Variabilität und ihr Einsatz

Genauso wie in anderen Bereichen der Softwarekonstruktion stellen sich bestimmte Vorgehensweisen bei der Entwicklung als produktivitätssteigernd dar, so dass sie zu Mustern erweitert werden. Ein *Variabilitätsmuster* ist hiermit ein domänenunabhängiges allgemeines Verfahren um Variabilität bestimmter Art implementieren zu können.
In [2] werden drei grundlegende Entitäten der Variabilitätsformen unterschieden:

- **Variant Entity.** Dies entspricht einer XOR-Relation bestimmter Eigenschaften eines Systems. Es kann also nur eine Variante aus einer nichtleeren Menge von Varianten gleichzeitig aktiv sein. In Abbildung 1 kann der TV-Tuner entweder das NTSC oder das PAL System dekodieren, aber nicht beide Systeme gleichzeitig. Die Variant Entity wurde in der Abbildung durch einen *Pfeil mit nicht gefüllter Pfeilspitze* gekennzeichnet.

- **Optional Entity.** Die optionale Entität entspricht der Variant Entity mit dem Unterschied, dass die Menge der Varianten auch leer sein kann. Ein Beispiel für optionale Entität wäre das GPS-Navigationssystem in Abbildung 1, da ein Fahrzeug nur optional über ein Navigationssystem verfügen sollte. Optional Entities werden durch ein *gestricheltes Rechteck* dargestellt.

- **Multiple Coexisting Entities.** In dem Fall der mehrfachen koexistierenden Entitäten gibt es im Unterschied zu den beiden oberen Punkten keine Beschränkung auf nur eine der möglichen Varianten. Der DiscPlayer in Abbildung 1 ist ein Beispiel für die Multiple Coexisting Entity, da ein Player jede Kombination aus den drei möglichen Varianten sein kann. Multiple Coexisting Entities werden durch einen *Pfeil mit gefüllter Pfeilspitze* graphisch dargestellt.

In den folgenden Abschnitten wird aufgezeigt, wie diese drei generellen Arten in den verschiedenen Phasen der Produktlinienentwicklung eingesetzt werden können.

3.1 Architekturdesignphase

Auf der Ebene des Architekturdesigns können *Architecture Reorganization, Variant Architecture Component, Infrastructure-Centered Architecture* und *Optional Architecture Component* verwendet werden. Die ersten drei Variabilitätsarten gehören zur Variant Entity, während die Optional Architecture Component ein Optional Entity-Muster ist.

3.1.1 Architecture Reorganization

Trotz der Ähnlichkeit der Produkte und der daraus resultierenden Ähnlichkeit der Architekturen in einer Produktlinie, sind oft Veränderungen der Architektur eines bestimmten Produktes notwendig, um seine auszeichnenden Features zu modellieren. Diese Veränderungen spiegeln sich oft in den anders gesetzten Kontroll- und Datenflussen zwischen den Komponenten wider.
Abbildung 2 zeigt einen möglichen Architekturausschnitt eines GPS-Navigationssystem-Empfängers. Der Empfänger besteht aus vier Modulen: einem Datenempfänger (RadioReceiver), einem Decoder (Decoder), einem Prüfsummenberechner (ChecksumCalculator) und einem Controller, der die Abarbeitungsreihenfolge festlegt. Die Controller-Komponente ist mit dem Datenempfänger und dem Decoder verbunden. Der Prüfsummenbaustein kann entweder an den Decoder oder an den Datenempfänger angeschlossen werden. Dadurch verändert sich jedoch die Semantik des gesamten Bausteins. Falls die Prüfsummen-Komponente an den Datenempfänger angeschlossen wird, wird als erstes der ankommende Datenfluss auf Korrektheit überprüft und erst dann dekodiert. In dem Fall, dass der Decoder eine Verbindung zum ChecksumCalculator hat, wird der Datenfluss zunächst dekodiert und erst dann auf Korrektheit überprüft. Diese beiden Vorgehensweisen sind denkbar und könnten unterschiedliche Protokolle beschreiben, die in verschiedenen Systemen verwendet werden.

In [2] wird die Verwendung von sogenannten *Architecture Description Languages* (ADL), die dieses Muster unterstützen, als Lösung vorgeschlagen. Eine Architecture Description Language ist eine Sprache zur Beschreibung gesamter (Software-) Architekturen und somit auch von Kontroll- und Datenflüssen zwischen den einzelnen Komponenten. Die Änderung der Architektur wird durch Verschiebung der Verbindungen zwischen den Schnittstellen der betreffenden Komponenten erreicht. Im Abschnitt 4.2.1 (Seite 213) wird das obige Beispiel in einer von Philips intern entwickelten ADL beschrieben.

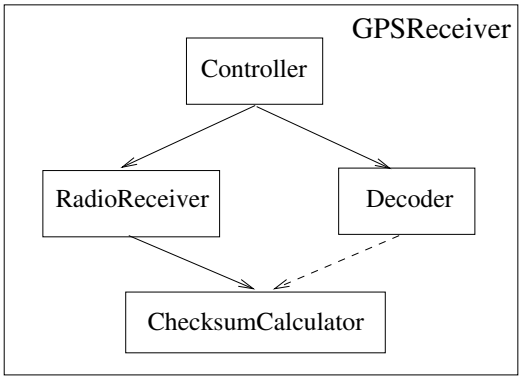

Abbildung 2: Beispiel für das Architecture Reorganization Muster. Der ChecksumCalculator kann entweder mit der Decoder- oder der RadioReceiver-Komponente verbunden werden.

3.1.2 Variant Architecture Component

Das Variant Architecture Component-Muster wird eingesetzt, falls an einer bestimmten Stelle in der Architektur mehrere verschiedene Komponenten zur Verfügung stehen, aber nur eine davon verwendet werden darf. Dabei ist nicht festgelegt, dass die Komponenten gleiche Interfaces haben müssen, wodurch die Aufgabe zusätzlich erschwert wird, da die Verbindung zum Restsystem nicht mehr einheitlich sein muss.
Ein Beispiel in dem das Variant Architecture Component-Muster eingesetzt werden kann, wäre ein Multimediasystem, das in Abhängigkeit davon, ob es mit einem CD-ROM-Laufwerk oder einer Festplatte ausgestattet ist, nur Lese- oder Lese- und Schreib- Operationen durchführen kann. Dadurch haben die Komponenten, die das CD-Rom-Laufwerk bzw. die Festplatte realisieren, unterschiedliche Interfaces.
Das Problem, dass es mehrere Komponenten gibt, wird durch parallele Unterstützung aller möglichen Subsysteme gelöst, d. h. alle Komponenten sind zunächst einmal in der Architektur vorhanden. Erst wenn ein bestimmtes Produkt abgeleitet wird, wird eine bestimmte Komponenten ausgewählt und alle anderen gelöscht. Die Entscheidung welche Komponente in einer bestimmten Konfiguration verwendet wird, wird in einem darauffolgenden Konfigurationsschritt gemacht.
Die Tatsache, dass die Komponenten unterschiedliche Interfaces haben dürfen und die Implementierung des Restsystems sich nicht verändern darf, hat zur Folge, dass dieses Muster nicht direkt eingesetzt werden kann. Es muss auf den unterliegenden Architekturschichten durch das Variant Component Specializations-Muster (Kapitel 3.2.1) unterstützt werden. Das führt zur Verschiebung der Ent-

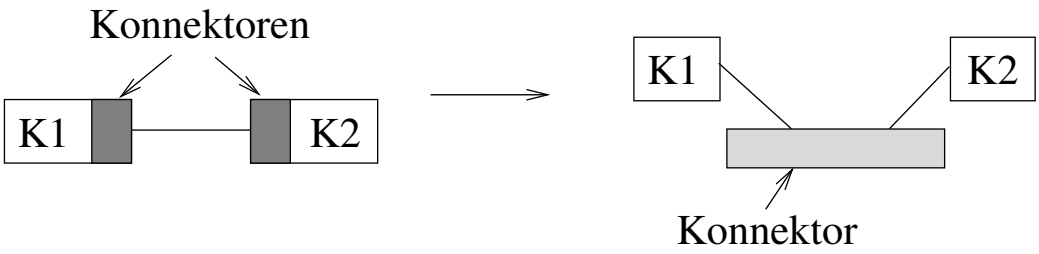

Abbildung 3: Extrahierung der Konnektoren aus den Komponenten nach dem Infrastructure-Centered Architecture Muster.

scheidung welches Interface benutzt wird auf einen späteren Zeitpunkt. Dadurch findet das Handling der verschiedenen Interfaces in den aufrufenden Komponenten und nicht in dem Variationspunkt statt.
Eine Implementierung von Variant Architecture Component in KOALA wird in 4.2.3 vorgestellt.

3.1.3 Infrastructure-Centered Architecture

Die Kommunikation zwischen den Komponenten in einer Architektur setzt die Kenntnis der Verbindungsart zwischen den Bausteinen voraus. Im Falle der direkten Verbindung ist diese Information zusätzlich explizit in den Komponenten als Quellcode der Konnektoren integriert. Das führt zum Einen zur Spezialisierung der Komponente, da sie nur bestimmte Verbindungsarten unterstützt und zum Anderen zur Steigerung der Komplexität aufgrund des zusätzlichen Verbindungsquellcodes.

Um diese Probleme zu umgehen, wird das Infrastructure-Centered Architecture Muster, das in Abbildung 3 zu sehen ist, verwendet. Dabei werden die Konnektoren aus den Komponenten extrahiert und als explizite Entitäten in die Architektur aufgenommen. Die Konnektoren bilden eine Infrastruktur mit der die kommunizierenden Komponenten verbunden werden. Diese Infrastruktur kann sowohl ein existierender Standard wie CORBA oder COM, eine Scripting Sprache oder aber auch eine Eigenentwicklung, wie es im Falle von KOALA der Fall ist, sein.

3.1.4 Optional Architecture Component

Das Optional Architecture Component-Muster ist eine Abwandlung des Variant Architecture Component-Musters, bei der zusätzlich die Möglichkeit besteht, dass eine Komponente in einem Produkt vorkommt in einem anderen aber nicht. We-

gen dieser Ähnlichkeit kann bei dem Optional Architecture Component-Muster ein analoger Lösungsansatz verwendet werden.
Genauso wie in Variant Architecture Component-Muster werden erst einmal alle möglichen Komponenten im System parallel unterstützt, bis in einem späteren Konfigurationsschritt ein Produkt aus der Produktlinie abgeleitet wird. Zusätzlich muss bei Optional Architecture Component sichergestellt werden, dass das System funktioniert, falls keine Komponente eingebunden wird. Dafür existieren zwei Lösungen, eine auf der Seite der aufrufenden und die zweite auf der Seite der aufgerufenen Komponente.
Die erste Lösung delegiert das Problem auf eine weiter unten liegende Entwicklungsstufe, z. B. auf die Implementierungsphase, wo mit Hilfe des Condition on Variable-Musters (s. 3.3.2) überprüft werden kann, ob die Komponente vorhanden ist und dementsprechende Aktion ausführen.
Die Lösung auf der Seite der aufgerufenen Komponente besteht in der Implementierung einer „null“-Komponente, d. h. einer Komponente, die zwar ein korrektes Interface hat, aber beim Aufruf der Funktionen einen Dummy-Wert zurück liefert. Natürlich erfordert auch dieser Lösungsweg von der aufrufenden Komponenten das Wissen, welche Werte Dummy-Werte sind und welche nicht, wodurch nach jedem Aufruf eine Überprüfung des Ergebnisses stattfinden muss.

3.2 Detaillierte Design-Phase

Für die Ebene des detaillierten Designs werden vier Variabilitätsmechanismen vorgestellt - *Variant Component Specialization, Optional Component Specialization, Multiple Coexisting Component Implementation* und *Multiple Coexisting Component Specialization*. Die folgenden Abschnitte beschreiben diese Muster.

3.2.1 Variant Component Specialization

Im Abschnitt über den Variant Architecture Component-Mechanismus wurde eine Lösung für das Problem vorgestellt, wenn mehrere Komponenten an einer bestimmten Stelle in der Architektur parallel unterstützt werden sollen. Dabei wurde darauf hingewiesen, dass aufgrund möglicherweise unterschiedlicher Interfaces die Komponenten nicht direkt miteinander verbunden werden können. Das Variant Component Specialization-Muster löst dieses Problem.
Die Idee, die in Abbildung 4 abgebildet, ist, die vorhandenen Interfaces von der Komponente zu lösen und sie in eigene Module bzw. Klassen zu verschieben. Dadurch wird die ursprüngliche Komponente in zwei Teilkomponenten zerlegt, in eine Interface-unabhängige und eine Interface-beschreibende Komponenten.

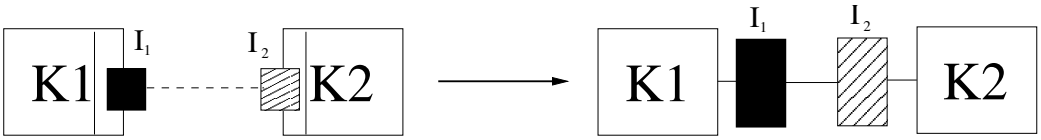

Abbildung 4: Auslagerung der Interfaces I_1 und I_2 aus den Komponenten K1 und K2 zur Realisierung der Variant Component Specialization-Variabilitätsform.

Die Verbindung von zwei Komponenten mit unterschiedlichen Interfaces wird dadurch auf die Implementierung der Interface-beschreibenden Module zurückgeführt.

3.2.2 Optional Component Specializations

Das Optional Component Specializations-Muster wird verwendet, um Teile der Implementierung einer Komponente hinzuzufügen oder zu entfernen. Dadurch kann das Verhalten der Komponente an das jeweilige Produkt angepasst werden.
Als Beispiel für die Verwendung dieses Musters kann folgender Fall betrachtet werden. Ein Multimediasystem, das ein Lied abspielt, soll die Daten über dieses Lied auf einen Bildschirm ausgeben. In Abhängigkeit vom Modell des Wagens steht dafür entweder ein großer hochauflösender oder ein einzeiliger monochromer Bildschirm zur Verfügung.
Ein großer Bildschirm kann aufgrund seiner physikalischen Eigenschaft natürlich mehr Informationen gleichzeitig ausgeben, als der kleine Bildschirm, z.B. neben der Tracknummer auch den Titel des Albums, das Cover, etc.. Deswegen muss die Implementierung der Komponente im Multimediasystem, die für die Ausgabe verantwortlich ist, so angepasst werden, dass an den großen Bildschirm mehr Daten gesendet werden.
Dieses Problem wird durch das Auslagern der Teile der Implementierung in eigene Subkomponenten gelöst. Falls ein Teil des Systems in einem Produkt nicht enthalten sein darf, kann eine „null“-Komponente verwendet werden, die Dummy-Werte als Resultat zurückliefert.
Die Aufspaltung einer Komponente in mehrere Subkomponenten bringt den Vorteil, dass ihre Flexibilität erhöht wird. Auf der anderen Seite hat das den Nachteil, dass ab einem bestimmten Zeitpunkt die verringerte Funktionalität zum Einstellen der Wiederverwendung dieser Komponenten führt, da das Neuprogrammieren der gleichen Funktionalität weniger Aufwand bedeutet. Auch die Identifizierung und die Trennung des Verhaltens aus einer Komponente muss keine einfache Aufgabe sein.

Die Implementierung des Optional Component Specializations-Mustern in KOALA wird im Kapitel 4.2.6 behandelt.

3.2.3 Multiple Coexisting Component Implementation

Der DiscPlayer im Featuregraph des *CarMultimedia*-Systems (Abbildung 1) unterstützt je nach Modell beliebige Kombinationen der MP3-, DVD- und AudioCD-Wiedergabe, so dass der Player z.B. sowohl MP3- als auch AudioCD abspielen können muss. In den bisher vorgestellten Variabilitätsformen war jedoch immer das Ziel, nur eine bestimmte Komponente aus einer Menge auszuwählen, was für dieses Beispiel bedeuten würde, dass der Player entweder nur MP3 oder nur AudioCD abspielen kann. Um Unterstützung mehrerer Formate zu ermöglichen, wird die Multiple Coexisting Component Implementation Form der Variabilität benutzt, die mit Hilfe folgender Konstruktion implementiert wird.
Die einzelnen parallel unterstützten Subsysteme werden als Komponenten mit einem für alle gemeinsamen Interface implementiert. Vor jedem Aufruf einer Methode aus dem Interface wird überprüft, welche Komponente die passende Implementierung zur Ausführung der gestellten Aufgabe enthält. Ist diese Entscheidung getroffen, werden an die entsprechende Komponente die Daten zum Verarbeiten gesendet und falls nötig nach der Verarbeitung ausgelesen.
Dieser Prozess wird von verschiedenen Mustern, wie z. B. Builder- oder Abstract-Factory-Muster (siehe [1, s. 97–106] und [1, s. 87–97]), ermöglicht.

3.3 Implementierungsphase

Auf der Implementierungsebene werden zwei Mechanismen vorgestellt um Variabilität auszudrücken. Zum Einen das *Condition on Constants* und zum Anderen *Condition on Variables*. Die folgenden Unterabschnitte beschreiben diese beiden Formen der Variabilität.

3.3.1 Condition on Constants

Wie der Name Condition on Constants schon sagt, ist die Hauptidee bei dieser Form der Variabilität die Auswertung einer Konstante mit Hilfe einer *if*-Bedingung , um ein bestimmtes Verhalten des Systems sicherzustellen. Die if-Bedingung kann entweder eine Präprozessor-Direktive wie #ifdef in C/C++, oder ein Sprachkonstrukt einer Programmiersprache sein.
If-Bedingungen sind eine gängige Methode um Variabilität in Softwarepro-

dukten zu implementieren. Ihre häufige Verwendung kann jedoch zu einem schlecht überprüfbaren Quellcode führen, da jede if-Bedingung die Anzahl der möglichen Programmläufe verdoppelt. Diese Beobachtung gilt zu unterschiedlichen Zeitpunkten sowohl für die Präprozessor-Anweisung #ifdef, als auch für das if-Sprachkonstrukt. Die Verdopplung der Programmläufe aufgrund der Präprozessor-Direktiven ist vor dem Lauf des Präprozessors auf dem Quellcode zu finden. Nach diesem Schritt führen nur die if-Bedingungen der Programmiersprache zur Erhöhung der Programmläufe, die Präprozessor-Direktiven haben keinen Einfluss auf den restlichen Code. Nach der Auswertung der konstanten Ausdrücke in der Kompilationsphase bleibt nur ein eindeutiger Programmlauf, der in der Laufphase ausgeführt wird. Ein Vorteil bei Verwendung von Condition on Constants ist die Möglichkeit der Optimierung des Codes durch Compiler. Die Codeblöcke, die nicht ausgeführt werden, müssen nicht in das kompilierte Produkt aufgenommen werden, wodurch die Größe der ausführbaren Datei oder der dynamischen Bibliothek sich verkleinert. Die Auswertung der Bedingung erfolgt bereits während der Kompilierzeit und spielt während der Ausführung des Programms keine Rolle. Der Fall, dass auch während der Ausführung der Programmfluß kontrolliert werden muss, wird im nächsten Abschnitt behandelt.

3.3.2 Condition on Variable

Diese Form der Variabilität unterscheidet sich von der oberen durch Verwendung einer Variable statt einer Konstanten in der if-Bedingung. Dadurch entfällt auf der einen Seite die Möglichkeit der Optimierung des kompilierten Codes durch Compiler, auf der anderen Seite geschieht die Auswertung der if-Bedingung während der Laufzeit des Programms, so dass der Ausführungsfluss des Programms gesteuert werden kann.
Auch in diesem Fall ist darauf zu achten, dass eine hohe Anzahl der if-Bedingung zur Erschwerung der Kontrolle der Laufzeiteigenschaften eines Programms führt.

3.4 Linkphase

3.4.1 Binary Replacement - Linkerdirektiven

Unter einem Linker oder Binder (auch: "Bindelader") versteht man ein Programm, das einzelne Programmmodule zu einem ausführbaren Programm zusammenstellt.[3] Das Setzen der Linkerdirektiven legt fest, welche Programmmodule in das

[3] `http://de.wikipedia.org/wiki/Linker`

endgültige ausführbare Programm aufgenommen werden müssen und beeinflusst somit das Verhalten des Programms.

3.4.2 Binary Replacement - Physisch

Bei dieser Form der Variabilität handelt es sich um komplette oder teilweise Ersetzung der ausführbaren Datei durch eine andere. Diese Variabilitätsart wird vor allem bei nachträglichen Korrekturen bereits benutzter Programme verwendet. Es handelt sich um die songenannten *Patchs*.
Das Ersetzen der Dateien bringt jedoch die Gefahr mit sich, dass der Rest des Systems in einen inkonsistenten und funktionsunfähigen Zustand gebracht wird. Zum Einen kann es dadurch passieren, dass andere Systemteile nur wegen der vorhandenen Bugs im ersetzten Teil richtig funktioniert haben. Dieses Problem tritt jedoch bei jeder Ersetzung eines Teilsystems auf. Ein weiterer Grund, der zu Inkonsistenzen führen kann, ist die Tatsache, dass nach der Ersetzung der ausführbaren Datei keine syntaktische Überprüfung zum Restsystem stattfindet. Dadurch ist der Fall vorstellbar, dass die Namen der Methoden im geändertern Teil nicht mehr mit den Funktionsnamen des Restsystems übereinstimmen.

4 KOALA

Das Komponentenmodell KOALA [3] ist eine Entwicklung der Firma Philips aus dem Jahre 1997 für den Bereich der Unterhaltungsindustrie, das ständig weiterentwickelt wird.
Es handelt sich dabei um ein Modell zur Modellierung und Realisierung von Softwareproduktlinien und verfolgt das Ziel die Kosten für die Programmierung zu reduzieren und die Entwicklung neuer Software zu beschleunigen. Dazu wird die Architektur eines Softwareproduktes im ersten Schritt mit Hilfe einer speziellen Architecture Description Language beschrieben. Diese Beschreibung, die sowohl graphisch, als auch textuell sein kann, wird im zweiten Schritt in C-Quellcode übersetzt, der das Grundgerüst für das entworfene Produkt bildet. Nach der Implementierung der nötigen Funktionen kann das fertige Programm kompiliert werden um eine ausführbare Datei zu erhalten.
Der gewählte komponentenbasierte Ansatz erlaubt die Erstellung von *Konfigurationen*, die sowohl strukturelle als auch inhaltliche Variabilität aufweisen. Die Konfigurationen werden durch Instantiierung der Komponenten und Verbindung ihrer Interfaces erstellt und sind somit Instanzen der Produktlinienarchitektur, die ein bestimmtes Produkt beschreiben.

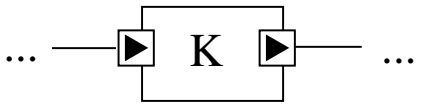

Abbildung 5: Beispiel einer Komponente mit einem *require-* und *provide-*Interface.

Im Folgenden werden einige Features von KOALA detailliert beschrieben und erklärt. Um eine komplette Übersicht über die Möglichkeiten von KOALA zu erhalten, sei auf [4] verwiesen. Die Implementierung der Muster aus Kapitel 3 wird im Abschnitt 4.2 gezeigt.

4.1 Features von KOALA

Dieser Abschnitt behandelt die wichtigsten Features von KOALA wie *Komponenten*, *Module*, *switches*, *diversity interfaces* und *require/provide interfaces*.

4.1.1 Komponenten

Eine Komponenten ist ein Grundbaustein des Designs, der Entwicklung und der Wiederverwendung in KOALA. Sie ist ein in sich abgeschlossenes System, die eine Aufgabe oder eine ganze Domäne repräsentiert. Eine Komponente wird in KOALA durch ein *Rechteck* dargestellt.

Zur Kommunikation mit der Umwelt verfügen Komponenten über sogenannte *provide-* und *require-Interfaces* (s. 4.1.2). Ein Interface wird durch ein schwarzes Dreieck, das von einem Rechteck umgeben wird, und auf dem Rand einer Komponente angebracht ist, dargestellt. Die Richtung des Dreieck beschreibt dabei die Art der Schnittstelle. Zeigt die Spitze des Dreiecks in die Komponente hinnein, dann handelt es sich um ein provide-Interface. Dementsprechend werden require-Interfaces durch ein Dreieck dargestellt, dessen Spitze aus der Komponente hinaus zeigt. Ein Beispiel für eine Komponente mit Interfaces ist in Abbildung 5 zu sehen.

KOALA verfolgt das Ziel, eine strikte Trennung zwischen Komponenten- und Konfigurations-Entwicklung sicherzustellen. Der Entwickler einer Komponente darf keine Annahmen machen, wie die Konfiguration aussehen kann, in der seine Komponente verwendet wird. Auf der anderen Seite dürfen die Entwickler der Konfigurationen nicht die interne Implementierung der Komponenten ändern, um diese an die Konfiguration anzupassen. Diese Trennung hat auf der einen Seite

den Nachteil, dass die Verbindung der einzelnen Komponenten zu Konfigurationen durch unterschiedliche Interfaces erschwert wird und zusätzlichen Programmieraufwand bedeutet. Auf der anderen Seite kann dadurch die Entwicklung der Produkte und der Komponenten von getrennten Programmiererteams durchgeführt werden und Verwendung von Drittanbieter-Komponenten wird erleichtert. Bereits das führt zu einer Erhöhung der Entwicklungsgeschwindigkeit und möglicherweise Reduzierung der entstehenden Kosten.

4.1.2 Interfaces

Interfaces oder *Schnittstellen* sind fest definierte Zugänge zu einer Komponente. Sie legen die Funktionalität fest, die eine Komponente entweder ihrer Umwelt zur Verfügung stellt oder von ihrer Umwelt fordert um die Aufgaben erfüllen zu können. Die Schnittstellen, die angebotene Funktionalität beschreiben, werden *provide*-Interfaces, und ihr Gegenstück, das Dienste fordert *require*-Interfaces genannt. Jede Funktionalität, die von einer Komponente benötigt oder zur Verfügung gestellt wird, wird in mindestens einem Interface spezifiziert. Sogar Zugriffe auf das Betriebssystem werden über die *require*-Interfaces abgewickelt. Dadurch wird die Flexibilität der Komponenten erhöht, da sie nicht von bestimmten Diensten bzw. von einem bestimmten Betriebssystem abhängig sind.
Bei Verbindungen von Interfaces muss die Regel beachtet werden, dass ein require-Interfaces mit genau einem provide-Interface und ein provide-Interface mit beliebig vielen require-Interfaces verbunden werden dürfen. Das entspricht der Tatsache, dass eine Funktion genau eine Implementierung hat, aber dabei mehrere andere aufrufen kann.
Entscheidend für die Verbindung zweier Schnittstellen ist ihr Typ. Der Typ eines Interfaces ist eine Menge von Deklarationen der Funktionen. Für ihre Beschreibung wird eine Java-ähnliche Syntax benutzt, wie folgendes Beispiel zeigt.

```
interface I {
  double f( int x );
}
```

Dieser Quellcode beschreibt eine Schnittstelle mit einer Funktion, die einen Integer-Parameter bekommt und einen double-Wert als Resultat liefert. Zwei Interfaces dürfen nur dann verbunden werden, falls ihre Typen gleich sind, oder die Menge der Funktionen des require-Interfaces eine echte Untermenge des provide-Interfaces sind.

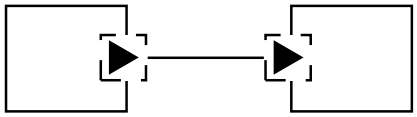

Abbildung 6: Beispiel eines optionalen Interfaces.

4.1.3 Unterstützung der Variabilität in KOALA

Die bisher betrachteten Features sind in jedem Komponentenmodell vorhanden. Zur Unterstützung der Variabilität bietet KOALA zusätzlich *optionale Interfaces*, *Diversity-Interfaces*, *Module* und *Diversity spreadsheets* an.

Optionale Interfaces

Eine besondere Form der Interfaces bilden *optionale Interfaces*. Diese Art der Schnittstellen hat eine zusätzliche Funktion *iPresent*, die angibt, ob dieses Interface in einer bestimmten Konfiguration vorkommt. Dadurch kann auf einfache Weise Funktionalität eines Subsystems hinzugefügt oder entfernt werden. Optionale Interfaces werden durch ein gestricheltes Rechteck mit einem schwarzen Dreieck dargestellt (Abbildung 6).

Diversity Interfaces

Wiederverwendbare Komponenten dürfen nach Ansicht der Autoren in [4] keine konfigurationsspezifischen Informationen enthalten, sondern müssen vollständig parametrisierbar sein, d. h. alle Parameter der Komponente müssen einstellbar sein.

So eine hochparametrisierbare Komponente hat jedoch den Nachteil, dass sie nicht in Systemen mit begrenzten Ressourcen eingesetzt werden könnte, falls keine Möglichkeit bestehen würde, um nicht benötigte Funktionalität bzw. Flexibilität entfernen zu können.

Um dieses Ziel zu erreichen, werden in KOALA *Diversity Interfaces* eingeführt. Diversity Interfaces sind require-Interfaces einer Komponente, die Diversity Funktionen enthält. Da diese Funktionen in einem require-Interface definiert werden, müssen sie an einer anderen Stelle im System implementiert werden. Durch zusätzliche Definition eines jeden Diversity Parameters als ein Makro, führt das dazu, dass Abfragen der Art

if (diversity_parameter ()) { do_something() };

durch den Präprozessor eliminiert werden können, falls der Wert, den diversity_parameter() liefert, eine Konstante ist.

Diese Vorgehensweise führt zur Reduzierung vom erstellten Code, wodurch der Einsatz von hochparametrisierbaren Komponente in KOALA ermöglicht wird.

Module

Oft müssen Subkomponenten einer größeren Komponente vor der Benutzung initialisiert werden, um korrekte Funktionsweise garantieren zu können. Dafür stellt jede von ihnen ein provide-Initialisierungsinterface zur Verfügung, das die entsprechenden Funktionen zum Initialisieren bereitstellt. Dabei entsteht das Problem, dass diese Interfaces nicht an das Initialisierungsinterface der zusammengesetzten Komponente angeschlossen werden können, da dabei die Regel verletzt wird, dass ein provide-Interface mit genau einem require-Interface verbunden werden darf. Um dieses Problem zu lösen, werden in KOALA *Module* benutzt, die durch *Rechtecke ohne Interfaces* dargestellt werden. Der zweite große Einsatzgebiet der Module ist die Verbindung von Komponenten mit unterschiedlichen Interfaces. Das Beispiel in Abbildung 7 zeigt wie das Modul m die Komponenten K1 und K2 initialisiert.

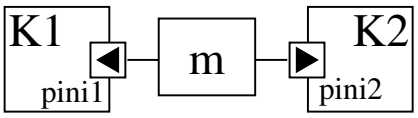

Abbildung 7: Initialisierung der Komponenten K1 und K2 mit Hilfe des Moduls m.

Ein Modul ist eine interfacelose Komponente, die in einer Komponente definiert ist. Es kann mit beliebig vielen Schnittstellen anderer Komponenten oder Subkomponenten verbunden werden. Falls ein Modul mit einem require-Interface verbunden ist, werden alle Funktionen dieser Schnittstelle im Modul implementiert. Falls eine Verbindung mit einem provide-Interface besteht, können alle Funktionen dieser Schnittstelle in dem Modul verwendet werden. Die Funktionen dürfen dabei sowohl in C als auch in der Component Description Language (CDL) implementiert werden. CDL ist eine Untermenge der C, die die durch andere Interfaces bereitgestellten Konstanten, Operatoren und Funktionen verwenden kann.

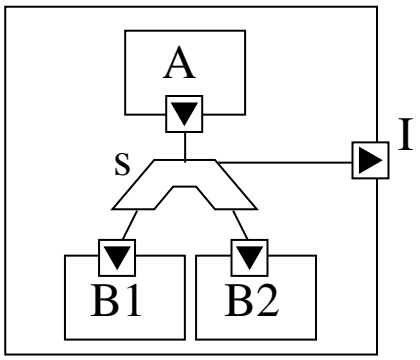

Abbildung 8: Der Switch s verbindet A mit B1 oder B2 in Abhängigkeit vom Wert des Diversity-Interfaces I.

Im Unterschied zu Komponenten haben Module einen geringen Wiederverwendungswert, da ihre Aufgaben als Initialisierungs- oder Verbindungsbausteine stark von dem Produkt bestimmt werden, in dem sie eingesetzt werden.

Switch

Angenommen eine Komponente A benutzt B1 in einem Produkt und B2 in einem anderen. Eine Möglichkeit dieses Szenario zu lösen, besteht im Definieren von zwei Konfigurationen. Da A Teil einer komplexen zusammengesetzten Komponente sein kann und der Rest des Systems nicht dupliziert werden soll, wird der Baustein *switch* zwischen A, B1 und B2 geschaltet.
Ein switch ist ein Element, das die Verbindung der Interfaces steuert. Es enthält einen Steueranschluss und drei Anschlüsse für die Interfaces der zu verbindenden Komponenten. In Abhängigkeit vom Wert des Steueranschlusses wird entweder B1 oder B2 mit A verbunden (Abbildung 8). Im Falle eines konstanten Wertes am Eingang vom Interface I, kann KOALA den Code optimieren und in der Produktkonfiguration die Komponente A mit B1 oder B2 direkt verbinden, ohne dass zusätzlicher Overhead entsteht.

Diversity Spreadsheets

Aufgrund der Möglichkeit die Funktionen der require-Interfaces in einem Modul mit Hilfe der Component Description Language zu beschreiben, können Diversity-Parameter einer innenliegenden Komponente durch Diversity-Parameter der außenliegenden Komponenten ausgerechnet bzw. beschrieben werden. D.h. der Zustand eines Variationspunktes kann in einen Zusammenhang zu den Zu-

ständen anderer Variationspunkte gesetzt werden.
Ein interessantes Beispiel für die Verwendung der Diversity Spreadsheets ist die Feststellung des Speicherbedarfs einer Konfiguration. Dafür muss jede Subkomponente ihren Speicherbedarf der Vater-Komponente in einem provide-Interface bekannt geben. Der gesamte Speicherbedarf kann dann ausgerechnet werden, indem die zusammengesetzte Komponente die zur Verfügung gestellten Werte summiert.

4.2 Implementierung der einzelnen Arten der Variabilität in KOALA

In diesem Abschnitt werden mögliche Implementierungen der Muster aus Kapitel 3 vorgestellt.

4.2.1 Architecture Reorganization

Die Realisierung der Architecture Reorganization Variabilitätsform kann in KOALA durch keinen eindeutigen Mechanismus beschrieben werden. Die in Kapitel 3.1.1 erwähnte Änderung des Kontroll- bzw. des Datenflusses kann in Abhängigkeit davon, welche Veränderung der Architektur das nach sich zieht, auf verschiedene Arten modelliert werden. Somit ist die Architecture Reorganization Variabilitätsform in KOALA eine übergeorndete Kontrollstruktur, die die Variabilitätformen der anderen Designstufen bentutzt, um die Variabilität auf der Architekturstufe zu erreichen.

Das Beispiel aus Kapitel 3.1.1 kann z. B. mit Hilfe des in Kapitel 4.2.3 beschriebenen Variant Architecture Component Musters gelöst werden. Wie in Abbildung 9 zu sehen ist, werden dafür die Komponente RadioReceiver, Decoder und ChecksumCalculator mit einem Switch verbunden. Der Switch verbindet in Abhängigkeit vom Wert des Interfaces I_d die ChecksumCalculator-Komponente mit dem RadioReceiver oder mit dem Decoder. Dadurch ändert sich der Kontrollfluss und dadurch auch das gesamte Produkt.

4.2.2 Infrastructure-Centered Architecture

Die Aufgabe des Infrastructure-Centered Architecture-Musters besteht in dem Lösen der impliziten Bindung der Komponenten an die Architektur, indem die Konnektoren der Komponenten zu expliziten Entitäten der Architektur gemacht wer-

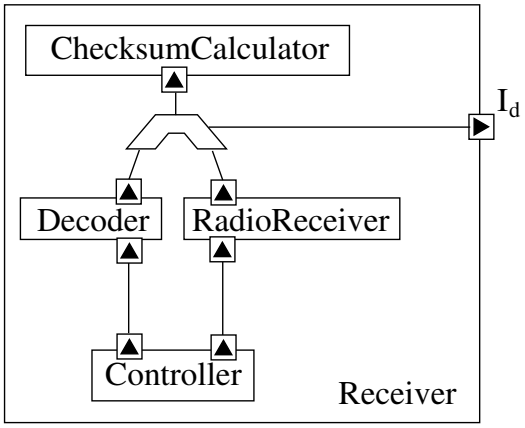

Abbildung 9: Architecture Reorganization: Änderung der Architektur mit Hilfe der Variant Architecture Component Variabilitätsform.

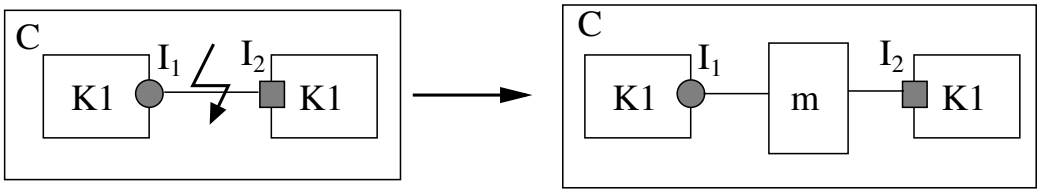

Abbildung 10: Infrastructure-Centered Architecture: Verbindung der Komponenten durch ein Modul.

den.
Im folgenden Beispiel wird gezeigt, wie zwei Komponenten mit unterschiedlichen Interfaces verbunden werden können. Im Linken Teil der Abbildung 10 ist die Ausgangssituation abgebildet. Die Architektur enthält zwei Komponenten K1 und K2. Da die Interfaces der beiden Komponenten, die durch das graue Rechteck und den grauen Kreis dargestellt sind, unterschiedliche Typen haben, ist die Verbindung dieser Komponenten nicht erlaubt. Dieses Problem wird durch das Hinzufügen eines Moduls gelöst. Der rechte Teil der Abbildung 10 bildet die grafische Lösung ab und der folgende Quellcode zeigt eine schematische Implementierung.

```
component C {
    provides I1 p;
    requires I2 r;
    contains module m;
    connects p = m;
             m = r;
}
```

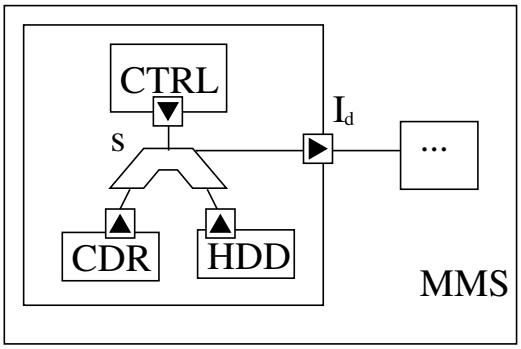

Abbildung 11: Variant Architecture Component. Auswahl eines Bausteins mit Hilfe eines Switches.

Da das Modul *m* alle Funktionen des Interfaces I1 mit Hilfe der Funktionen des Interfaces I2 implementiert, werden die Interfaces I1 und I2 implizit miteinander verbunden.

4.2.3 Variant Architecture Component

Zur Implementierung des Variant Architecture Component-Musters kann in KOALA der Baustein *switch* verwendet werden. Das gleiche Beispiel wie im Kapitel 3.1.1, das in Abbildung 11 zu sehen ist, wird verwendet um die Implementierung des Musters in KOALA zu zeigen.
Ein Teil der Architektur des Multimediasystems (MMS) besteht aus einer Komponente CDR (CD-ROM), einer Komponente HDD (Festplatte) und einem Steuerelement, das durch die Komponente CTRL dargestellt wird. Das Steuerelement CTRL wird in Abhängigkeit davon, um welches Produkt der Produktlinie es sich handelt, entweder an ein CD-ROM, d. h. Komponente CDR, oder an eine Festplatte-Komponente HDD angeschlossen sein. Das wird durch Verwendung des Switches *s*, der zwischen CTRL und CDR mit HDD geschaltet ist, erreicht.

Wie bereits in Kapitel 4.1.3 vorgestellt wurde, verbindet der Baustein Switch die durch den Wert des Diversity Interfaces I_d festgelegte Interfaces der Komponenten. Falls das Interface I_d variable Werte liefert, kann die Komponente CTRL sowohl mit CDR als auch mit HDD verbunden werden. Das entspricht der Multiple Coexisting Component Implementation (s. Abschnitt 4.2.9), da in einem Produkt sowhl HDD als auch CDR parallel vorhanden und benutzt sein können. In dem Fall, dass I_d einen konstanten Wert liefert, wird in einer Konfiguration die CTRL-Komponente direkt mit der betroffenen Komponenten verbunden werden, ohne das ein zusätzlicher Switch dazwischen steht.

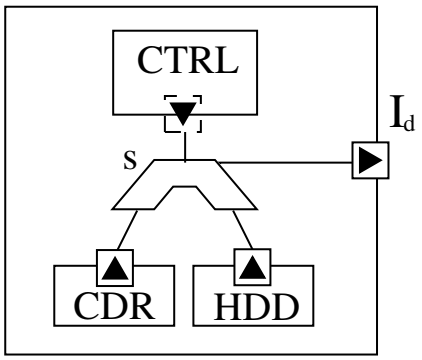

Abbildung 12: Optional Architecture Component. Verwendung eines optionalen Interface zur Verbindung mit anderen Komponenten.

Falls die Wahl einer der Komponenten CDR oder HDD Änderungen in dem require-Interface der CTRL-Komponente zur Folge hätte, z. B. weil die provide-Interfaces der CDR- und HDD-Komponenten unterschiedlich sind, wird das Variant Component Specialization-Muster(s. 4.2.5) verwendet.

4.2.4 Optional Architecture Component

Durch optionale Interfaces unterstützt KOALA direkt den Optional Architecture Component-Mechanismus. Falls im Beispiel in Abbildung 11 erlaubt werden soll, dass weder ein CD-ROM noch eine Festplatte im System vorhanden ist, muss nur das Interface an der CTRL-Komponente verändert werden, so dass die Architektur wie in Abbildung 12 entsteht. Durch das Setzen des Wertes in der iPresent-Funktion des optionalen Interfaces wird entschieden, ob eine der Komponenten in das Produkt aufgenommen werden soll, oder der komplette switch-Teil unberücksichtigt bleibt.

4.2.5 Variant Component Specialization

Bereits in Kapitel 4.2.2 wurde ein Lösungsweg gezeigt, wie Komponenten mit unterschiedlichen Interfaces verbunden werden können. Dabei wurde ein Modul zwischen die Komponenten geschaltet, das die Aufgabe der Delegation der Funktionsaufrufe übernommen hat.

Diese Lösung funktioniert sehr gut, solange alle Funktionen des require-Interfaces der aufrufenden Komponente implementiert werden können. Sobald aber eine Funktion nicht realisiert werden kann, weil z. B. wie im Multimediasystem-

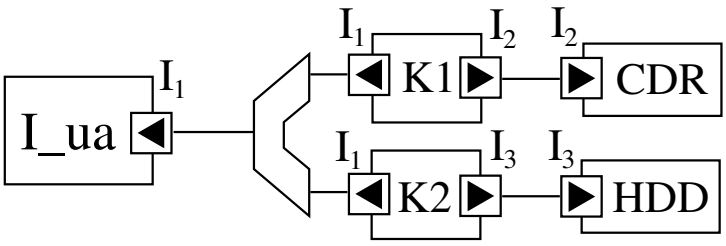

Abbildung 13: Durch die Auslagerung des Interface-Parts der Komponenten, können Komponenten mit unterschiedlichen Interfaces einfacher verbunden werden.

Beispiel in Abbildung 11 das CD-ROM-Laufwerk keine Schreiboperationen durchführen kann, ist die Lösung nicht mehr ausreichend.
In diesem Fall wird das Variant Component Specialization-Muster eingesetzt. Die Idee dieses Musters wurde bereits im Kapitel 3.2.1 erklärt. Eine Implementierung dieser Idee in KOALA ist in Abbildung 13 dargestellt. Die ursprüngliche Komponente CTRL ist in drei Teile aufgespalten - die verbindungsunabhängige Teilkomponente I_ua und zwei Komponenten K1 und K2, die als Wrapper zwischen I_ua und CDR bzw. HDD funktionieren. In Abhängigkeit von der Stellung des Switches wird I_ua mit CDR oder mit HDD verbunden. Der Implementierungsweg ist somit dem Vorgehen in dem Infrastructure-Centered Architecture Muster sehr ähnlich. Die Verwendung von Komponenten statt Module begründet sich damit, dass Module nicht an Switches angeschlossen werden dürfen, da sie keine expliziten Interfaces besitzen.

4.2.6 Optional Component Specialization

Das Optional Component Specializations-Muster wird verwendet, um Teile der Implementierung einer Komponente hinzuzufügen oder zu entfernen, indem die veränderbaren Teile in eigenständige Subkomponenten ausgelagert werden.
Diese Aufspaltung der Funktionalität wird in KOALA durch Verwendung einer zusätzlichen Komponente gelöst, die über ein optionales Interfaces an den Rest der Komponente angeschlossen ist. Falls eine Funktionalität, die in Abbildung 14 durch die Subkomponente SK dargestellt ist, entfernt werden muss, wird der entsprechende Wert der iPresent-Funktion der HK-Komponente auf *false* gesetzt.

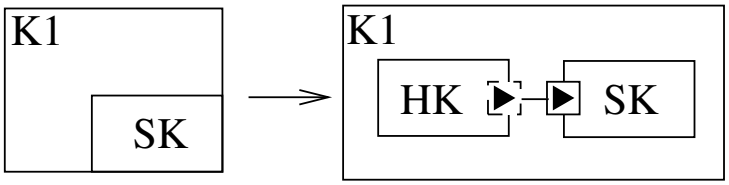

Abbildung 14: Optional Component Specialization. Aufspaltung der Funktionalität durch Verwendung einer zusätzlichen Subkomponente.

4.2.7 Condition on Variable, Condition on Constant

Für diese beiden Formen der Variabilität sind in KOALA keine besonderen Methoden nötig. Da die Grundprogrammiersprache von KOALA C ist, können die Beschreibungen dieser Methoden, die in 3.3.1 und 3.3.2 vorgestellt wurden, direkt in KOALA übernommen werden.

4.2.8 Binary Replacement - Physisch, Binary Replacement - Linkerdirektiven

Da KOALA die Beschreibung der Architektur durch die ADL in einen C-Quellcode übersetzt und diesen kompiliert, sind die beiden *Binary Replacement*-Variationsformen auch in KOALA vorhanden. Zu ihrer Umsetzung sind keine KOALA-eigene Konstrukte nötig.

4.2.9 Multiple Coexisting Component Implementation

Wie bereits im Abschnitt 4.2.3 erwähnt, entspricht die Konstruktion der Multiple Coexisting Component Implementation Variabilitätsform der Konstruktion aus dem Abschnitt 4.2.3, die in Abbildung 11 zu sehen ist. Der Unterschied zwischen beiden Implementierungen besteht darin, dass die Schnittstelle bei bei der Multiple Coexisting Component Implementation Variabilitätsform mit einem Interface verbunden werden kann, das variable Werte liefert, während bei Variant Architecture Component die Schnittstelle stets einen konstanten Ausdruck liefern muss.

4.3 Fazit

Wie in den vorhergehenden Abschnitten gezeigt wurde, können alle Formen der Variabilität aus Kapitel 3 mit KOALA realisiert werden. Die vorhandenen Features bieten flexible Werkzeuge zum Handhaben der Variabilität auf jeder Ebene der Entwicklung.
In bestimmten Fällen kann die Bindung des Komponentenmodells an die Sprache C ein Nachteil sein, da dadurch andere Programmierparadigmen, wie z. B. Objektorientierung nicht mehr zur Verfügung stehen. Es muss auch mit Schwierigkeiten und Problemen gerechnet werden, die der Programmiersprache C zugerechnet werden, wie falscher Umgang mit Zeigern oder Speicherfreigaben. Bei großen Projekten kann das möglicherweise zu schwer wartbaren Programmen führen. Auf der anderen Seite ist die Programmiersprache C seit Jahrzehnten im Einsatz, so dass unter Umständen auf Millionen Zeilen von Code zugegriffen werden kann, was den Entwicklungsprozess beschleunigen kann. Auch die Effizienz dieser Sprache ist vor allem bei eingebetteten Systemen ein Vorteil.

5 Zusammenfassung

Diese Ausarbeitung beschäftigte sich mit der Frage wie verschiedene Formen der Variabilität mit Hilfe von KOALA-Komponentenmodell realisiert werden können. Dafür wurde als Erstes definiert, dass ein Komponentenmodell eine Sammlung von Komponenten, ihrer Dokumentation und der Gebrauchsanweisung ist, das die Idee der Wiederverwendung im Vordergrund hat. Es wurde außerdem behauptet, dass das Ziel einiger Komponentemodelle die Tauglichkeit für die Produktlinienentwicklung ist.
Als Nächstes wurden verschiedene Formen der Variabilität allgemein untersucht und Wege aufgezeigt, wie die jeweilige Variabilitätsform in einem System verwendet werden kann. Es stellte sich heraus, dass Variabilität auf jeder Stufe der Software-Entwicklung zu finden ist.
Bevor schließlich im letzten Teil dieser Arbeit gezeigt wird, dass alle vorgestellten Variabilitätsarten ohne Schwierigkeiten in KOALA implementiert werden können, wurde das KOALA-Komponentenmodell vorgestellt. Neben den allgemeinen Features eines Komponentenmodells wie require- und provide- Interfaces, wurden KOALA-spezifischen Features wie Diversity Interfaces, Diversity Spreadsheets, Optionale Interfaces und Module vorgestellt und erläutert. Diese Features dienen der Beschreibung der Variabilität in einer Software-Architektur.
Wie bereits erwähnt, wurde im letzten Teil der Ausarbeitung gezeigt, dass die vorgestellten Variabilitätsarten in KOALA implementiert werden können. Das lässt

den Schluß zu, dass KOALA ein flexibles Werkzeug im Bereich der Produktlinien in der Unterhaltungsindustrie sein kann. Die Zukunftspläne der Entwickler von KOALA bestehen in der Integration dieses Komponentenmodells mit anderen Modellen wie COM oder JavaBeans. Diese Entwicklung könnte den Wert von KOALA erhöhen, da dadurch Integration in bereits vorhandene Architekturen möglich wird.

Literatur

[1] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[2] GURP, JILLES VAN, JAN BOSCH und MIKAEL SVAHNBERG: *On the Notion of Variability in Software Product Lines.* In: *WICSA*, Seiten 45–54. IEEE Computer Society, 2001.

[3] OMMERING, ROB C. VAN, FRANK VAN DER LINDEN, JEFF KRAMER und JEFF MAGEE: *KOALA*. `http://www.extra.research.philips.com/SAE/koala/`.

[4] OMMERING, ROB C. VAN, FRANK VAN DER LINDEN, JEFF KRAMER und JEFF MAGEE: *The Koala Component Model for Consumer Electronics Software.* IEEE Computer, 33(3):78–85, 2000.

[5] WACHSMANN, CHRISTIAN: *White Paper - Component based Development*. OpenKnowledge, August 2002. `http:/www.openknowledge.de/pdf/cbd.de.pdf`.