# Evolution und Komposition von Softwaresystemen

*Software-Produktlinien als Beitrag zu Flexibilität und Langlebigkeit*

vorgelegt als Habilitationsschrift
zur Erlangung des akademischen Grads Dr.-Ing. habil.
der Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von Dr.-Ing. Matthias Riebisch

Gutachter: 1. Prof. Dr.-Ing.habil. I. Philippow, TU Ilmenau
2. Prof. Dr.-Ing.habil. H. Züllighoven, Universität Hamburg
3. O.Univ.-Prof. Dipl.-Ing. Mag. Dr. G. Kappel, TU Wien

Datum der Einreichung: 3. Juni 2003
Datum der Probevorlesung: 2. Juni 2004
Datum des wissenschaftlichen Vortrags und Kolloquiums: 20.August 2004
Datum des Vollzugs der Habilitation: 27. September 2004

## Vorwort

Wer beruflich Software entwickelt, weiß um den hohen manuellen Aufwand, um den großen Anteil immer in ähnlicher Form wiederkehrenden Lösungen und um die Schwierigkeiten, Lösungen zu verstehen, die ein anderer oder man selbst erst kürzlich erdacht hat. Objektorientierung, CASE-Werkzeuge, Software-Wiederverwendung und viele weitere Ideen wurden nacheinander als ultimativer Durchbruch des technischen Fortschritts proklamiert. Was nach Abklingen der verschiedenen Hypes blieb, war die Erkenntnis, dass es ein *Silver Bullet* zur Lösung aller Probleme wohl nicht geben wird, dass es eine Anzahl von Beiträgen sein werden, die in ihrer Summe zu wirksamen technologischen Verbesserungen führen.

Diese Arbeit soll einen Beitrag zum technischen Fortschritt leisten. Sie stellt Methoden vor, die auf Software-Wiederverwendung aufbauen und die Anwendungsbreite vorgefertigter Software erweitern. Die Methoden helfen dem Entwickler, die Komplexität der innerhalb von Software bestehenden Abhängigkeiten zu beherrschen. Sie helfen dem Produzenten, die Softwareentwicklung schneller und flexibler zu gestalten. Sie tragen im Interesse des Kunden dazu bei, dass Software gleichzeitig Änderungen ermöglicht und eine lange Lebenserwartung hat.

**Danksagung.** Die vorliegende Arbeit entstand während meiner wissenschaftlichen Tätigkeit im Fachgebiet Prozessinformatik der Technischen Universität Ilmenau unter Leitung von Frau Prof. Dr.-Ing. habil. Ilka Philippow. Ihr gilt mein Dank für die ausgezeichneten Arbeitsbedingungen. Ich erhielt die Möglichkeit, die Entwicklung der ALEXANDRIA-Methodik im Rahmen des gleichnamigen Forschungsprojekts selbständig durchzuführen und mich an Industrie-Kooperationen sowie an der Betreuung von Promotionen zu beteiligen. Ich möchte allen Mitstreitern am Projekt ALEXANDRIA und am Digitalen Video-Projekt für ihre engagierte Arbeit danken: Kai Böllert, Detlef Streitferdt, Ilian Pashov, Periklis Sochos und Michael Hübner. Eine Reihe von Studenten trugen mit ihren Studien- und Diplomarbeiten zum Gelingen des Projekts bei, auch ihnen gilt mein Dank. Außerdem möchte ich allen Fachkollegen danken, die Ideen und Vorschläge beigesteuert haben, wie Prof. John Leaney, Prof. Vaclav Rajlich und Prof. Dr. Johannes Sametinger.

Mein Dank gilt den Projektpartnern Dr. Günter Böckle (Siemens AG), Michael Zettler (Siemens Dematic GmbH) und Norbert Aschenbach (WSA Electronic GmbH) für die gute Zusammenarbeit sowie dem Förderverein der Fakultät Informatik und Automatisierung der TU Ilmenau für die Förderung des Digitalen Video-Projekts, das ohne diese Unterstützung nicht als studentisches Forschungsprojekt durchführbar wäre.

Ganz besonders möchte ich jedoch meiner Familie für die Unterstützung in den vergangenen Jahren danken, sowie meinem Vater für das Durchsehen des Manuskripts.

**Zusammenfassung**

Softwaresysteme sind heute umfangreicher, komplexer und von entscheidenderer Bedeutung für Produkte und Dienstleistungen als eine Dekade zuvor. Gleichzeitig sind Änderungen viel häufiger und in größerem Umfang erforderlich. Sie müssen auch schneller realisierbar sein. Zudem muss die Software eine höhere Lebensdauer erreichen, vor allem wegen des Aufwandes zu ihrer Entwicklung. Objektorientierte Programmierung und Wiederverwendungstechniken haben dabei nicht den erwarteten Erfolg gebracht. Die Einführung von Software-Produktlinien beziehungsweise Systemfamilien ermöglichen es, einen der Serienfertigung ähnlichen Vorfertigungsgrad zu erreichen und erlauben es gleichzeitig, kurzfristig Produktvarianten zu erstellen.

In dieser Arbeit werden Methoden der Domänenanalyse mit Wiederverwendungsansätzen und Generativen Programmiertechniken verknüpft und eine Methodik zur Produktlinien-Entwicklung vorgestellt. Featuremodelle werden als Ausdrucksmittel für Variabilität und Produktkonfigurationen eingesetzt, damit die Vorfertigung geplant und die Erstellung von kundenspezifischen Produkten gesteuert werden kann. Durch Präzisierung ihrer Syntax und Erweiterung ihrer Semantik werden Featuremodelle einer Nutzung in Werkzeugen zugänglich gemacht. Objektorientierte Entwurfsmodelle und Architektur werden so in feingranulare Komponenten zerlegt, dass Varianten als neue Produkte mit geringem Aufwand erstellbar sind. Die Erstellung der Implementierung solcher Produkte wird durch die Komposition von Quelltext-Komponenten automatisiert. Die Komposition von ebenfalls zerlegten Objektmodellen ermöglicht eine durchgehende Automatisierung der Produkterstellung, die durch einen Kunden mittels der Feature-Auswahl gesteuert wird. Dafür wird mit Hyper/UML eine Umsetzung des Hyperspace-Ansatzes auf die Modellierungssprache UML entwickelt, die eine Feature-gesteuerte Zerlegung und Komposition von Objektmodellen ermöglicht. Damit lassen sich schlanke Produkte entwickeln, die nur die tatsächlich benötigte Funktionalität enthalten. Zur Evolution von Produktlinien und zur Einbindung existierender Lösungen und Komponenten in die Evolution werden Reverse-Engineering- und Refactoring-Techniken integriert. Anforderungen, Modelle und Implementierung werden durch Traceability-Links verbunden, damit Änderungen konsistent durchgeführt werden können. Diese Mittel tragen dazu bei, dass während einer iterativen Entwicklung der Verlust an Architektur-Qualität, das sogenannte Architectural Decay, vermieden werden kann. Maßnahmen zur Verbesserung des Projekt- und Qualitätsmanagements werden kurz betrachtet, soweit sie wichtige Randbedingungen für die Wirksamkeit der Methoden schaffen müssen. Die Anwendbarkeit und Eignung der Ergebnisse der Arbeiten wurde in mehreren industriellen Projekten überprüft.

**Abstract**

Software systems are today bigger, more complex and of higher importance for products and services than a decade before. At the same time changes are required many more frequently and of a larger size. Furthermore, they have to be implemented faster. Additionally, the software must achieve a higher life span, particularly because of the cost of its development. In the past, Object-Oriented Programming and Reuse techniques did not provide the expected success. The introduction of software product lines respectively system families makes possible it to reach a degree of prefabrication similar to the one of serial production. At the same time they facilitate the delivery of product variants with a short time to market.

In this work methods of the methods of domain analysis are integrated with Reuse approaches and techniques of Generative Programming, and a methodology for product line development is presented. Feature models are used as means expressing variability and product configurations, so that the prefabrication be planned and the production of customer-specific products can be controlled. By enforcing the formalization in terms of syntax and semantics, feature models are made accessible to tools and automation. Object-oriented design models and architecture are separated into fine-granular components in such a way that new products can easily be developed as combinations of those components. The implementation of such products is automated by the composition of source code components. The composition of object models separated similarly enables a uninterrupted automation for the product development, which is controlled by a customer by means of a feature selection. To facilitate such a composition, the Hyperspace approach is applied to UML to Hyper/UML, which makes possible a feature-driven separation and composition of object models. In this way slim products can be developed, containing only the actually needed functionality. For the evolution of product lines and for the integration of existing solutions and components into the evolution, Reverse Engineering and Refactoring techniques are integrated. Requirements, models and implementation are connected by Traceability links to perform changes consistently. As a consequence, the loss of architectural quality - so-called Architectural Decay - can be avoided during the iterative development process. Measures for the improvement of the project and quality management are regarded briefly, as far as they are of importance for the effectiveness of the developed methods. The applicability and suitability of the results of the work were examined in several industrial projects.

**Keywords**: Software Product Line; Feature Model; Evolution, Traceability, Architecture, Domain Engineering, Generative Programming, Variability, UML, Hyperspace

## Gliederung

**Abbildungsverzeichnis**

**Tabellenverzeichnis**

**Abkürzungsverzeichnis**

| | |
|---|---|
| API | Application Program Interface |
| COTS | Components of the Shelf |
| ELOC | Effective Lines of Code |
| ERM | Entity-Relationship-Modell |
| ggf. | gegebenenfalls |
| IT | Informationstechnologie |
| LOC | Lines of Code |
| OCL | Object Constraint Language |
| OR | logisches ODER |
| PC | Personal Computer |
| PDA | Personal Digital Assistent |
| TQM | Total Quality Management |
| UML | Unified Modeling Language |
| XML | Extensible Markup Language |
| XOR | logisches Exklusiv-ODER |
| XP | Extreme Programming |

# 1 Einleitung

Eine Vielzahl von Produkten enthalten heute Computer. Deren Software bestimmt wesentliche Teile der Funktion von Produkten wie zum Beispiel Telefonen, Haushaltgeräten, Fahrzeugsteuerungen, Werkzeugmaschinen und Energieversorgungsanlagen. Software wird bei der Automatisierung von Prozessen in allen Lebensbereichen eingesetzt, damit Effektivitäts- und Produktivitätssteigerungen erreicht werden. Dabei verlagert sich ihr Einsatzschwerpunkt zunehmend von Einzelsystemen zu Systemübergreifenden Integrationsmitteln. Die Software wird zum unverzichtbaren Bestandteil, sogar zum Kern von Prozessen und Produkten, aber auch zu einem entscheidenden Kostenfaktor.

Anbieter von Produkten und Dienstleistungen stehen in einem ständigen Verdrängungswettbewerb untereinander. Für ihren Erfolg ist entscheidend, dass neue Produkte und die ihnen zugrunde liegende Software so schnell und so kostengünstig wie möglich entwickelt werden. Die Einführungszeit der Produkte, die sogenannte Time to Market, wird wesentlich von der für die Entwicklung der Software benötigten Zeit bestimmt.

Wenn Funktionalität oder Produktivität von der Software abhängen, würde ihr Nutzungsausfall ein wirtschaftliches Scheitern zur Folge haben. Das mit der Software verbundene Risiko wird dann als *geschäftskritisch* bezeichnet. Die Software soll genauso lange nutzbar sein wie die Geschäftsprozesse gelten. Die Ablösung durch eine Neuentwicklung wäre nicht nur mit Risiken verbunden; sie würde zugleich hohe Kosten verursachen, deshalb ist die Langlebigkeit der Software wirtschaftlich notwendig.

Änderungen an Software werden zunehmend schwieriger, weil ihre Komplexität gegenüber früheren Systemen stark gewachsen ist. Dies resultiert zum einen aus der Integration in die Geschäftsprozesse oder Produkte und zum anderen aus der Verknüpfung einer Vielzahl von Systemen mit verschiedenen Strukturen, mit verschiedener Umgebung und über Plattform- und Generationsgrenzen hinweg. Die hohe Komplexität der Software führt zu hohem Fehlerrisiko bei Änderungen. Gleichzeitig erfordert jede Änderung an Geschäftsprozessen eine Veränderung der verwendeten Software. Die Änderungszyklen von Softwaresystemen werden dabei immer kürzer, wie eine Studie zeigte (Abb. 1).

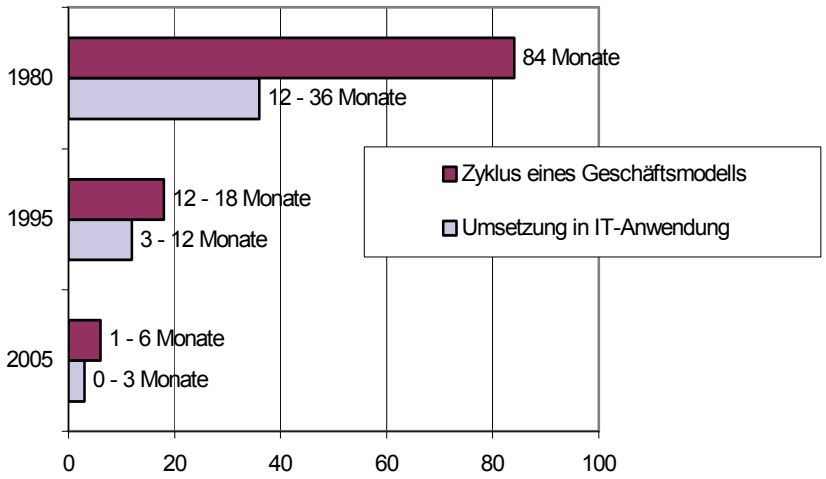



*Abb. 1: Änderungszyklen von Geschäftsmodellen und entsprechenden IT-Anwendungen [Meta 2002]*

Änderungen an Software führen häufig zu einer fortschreitenden Zerstörung ihrer Struktur. Dieser im Englischen als Architectural Decay bezeichnete Effekt (siehe 1.2) führt nach einiger Zeit dazu, dass Änderungen überhaupt nicht mehr durchgeführt werden können, weil ein Softwaresystem danach nicht mehr stabilisiert werden könnte. In verschiedenen Fällen war die Entwicklung neuer Produkte nicht möglich, weil die vorhandene Software die dazu erforderlichen Änderungen nicht mehr erlaubte [Rajlich Bennet 2000]. An solchen Beispielen wird deutlich, welche Bedeutung Flexibilität und Effektivität des Software Engineering haben.

Software-Änderungen zur Anpassung an geänderte Anforderungen gehören neben der Beseitigung von Mängeln zur Wartung von Software. Die meisten Prinzipien und Methoden des Software Engineering sollen Änderungen unterstützen und vereinfachen. Die Bedeutung der Wartung wird daran deutlich, dass sie den größten Aufwand während des Lebenszyklus von Software erfordert, sowohl von den Kosten als auch von der Zeitdauer her. Angaben in der Literatur stellen einen prozentualen Wartungsanteil von 50 % am Gesamtaufwand fest [Bosch 2000], nach eigenen Erfahrungen des Autors ist dieser Anteil häufig deutlich größer. Dieser hohe Kostenanteil führt zu abnehmender Konkurrenzfähigkeit des Softwareherstellers, weil einerseits ein hoher Teil der Kompetenzen und Ressourcen für die Wartung aufgebraucht werden, und andererseits die Softwaresysteme zu unflexibel für eine einfache Anpassung an veränderte Anforderungen sind. Das Qualitätsmerkmal Wartbarkeit [DIN 66272] beschreibt die Eigenschaften von Software, die den Aufwand für Änderungen bestimmen. Langlebigkeit und Flexibilität von Software hängt von solchen Eigenschaften ab.

Die vorliegende Arbeit soll Möglichkeiten für die Verbesserung von Langlebigkeit und Flexibilität von Software schaffen. Es sollen Methoden entwickelt werden, die eine Erhaltung und Verbesserung der Wartbarkeit unterstützen. In den folgenden Abschnitten werden damit zusammenhängende Problembereiche und Lösungsmöglichkeiten genauer untersucht. Aus den identifizierten Teilproblemen wird die Zielstellung der Arbeit abgeleitet.

## 1.1 Senkung des Änderungsaufwands

Der Zeitaufwand für Änderungen an Softwaresystemen wird durch die Größe und Komplexität des Systems, durch die Qualifikation der Softwareentwickler, die Methoden- und Werkzeugunterstützung und den Umfang der einzelnen Änderungen bestimmt.

Der Aufwand für eine einzelne Änderung ist vor allem dann hoch, wenn sie wegen des Umfangs des Systems und der Menge der internen Abhängigkeiten große Anteile des zu ändernden Systems betrifft. Die Verringerung solcher Abhängigkeiten kann dazu beitragen, den Aufwand je Änderung drastisch zu reduzieren. Es gilt, alle Möglichkeiten der Softwaretechnik so zu nutzen, dass die Auswirkungen von Änderungen gering bleiben. Spezielle Software-Architekturen und Maßnahmen wie Separation of Concerns und Information Hiding können dazu Beiträge leisten.

Zur Durchführung von Änderungen gehören folgende Aktivitäten:

1. Struktur des Systems und Änderungsbedarf verstehen
2. das von einer Änderung betroffene Element identifizieren
3. betroffenes Element und Struktur des Systems ändern
4. Änderung aus Aktivität 3 verifizieren
5. Aktivitäten 1 bis 4 für alle von der Änderung betroffenen Elemente durchführen
6. geändertes System in Betrieb nehmen

Die Aktivitäten 1 und 2 gehören zu Reverse Engineering (siehe 1.3) und Anforderungsanalyse. Der hierfür nötige Aufwand kann vor allem durch die Gewährleistung einer guten Verständlichkeit des Systems und seiner Architektur sowie eine klare Zuordnung von Systembestandteilen zu Anforderungen verringert werden. Dokumentationen leisten hierzu einen wichtigen Beitrag.

Die Aktivität 3 erfordert Aktivitäten der Anforderungsanalyse, des Entwurfs und der Implementierung. Dabei ist eine Analyse der Eigenschaften des vorhandenen Systems nötig. Das System muss um neue Eigenschaften erweitert werden, damit die neuen Anforderungen erfüllt werden. Der Aufwand dafür kann durch die Nutzung unterstützender Werkzeuge verringert werden. Für Änderungen der Struktur des Systems haben besonders Werkzeuge zur Modellierung Bedeutung.

In Aktivität 4 erfolgt ein Vergleich des erzielten Systemverhaltens mit dem gewünschten Verhalten. Da der Aufwand für Verifikation und Validation meist höher ist, als der für die Entwicklung von Software, kommt einer Reduktion dieses Aufwands große Bedeutung zu. Die Automatisierung von Tests, die Nutzung von Modell-Informationen zur Generierung von Testfällen sowie genaue Spezifikationen stellen wesentliche Möglichkeiten zur Reduktion des Aufwands dar.

Die Aktivität 5 umfasst die iterative Anpassung weiterer Elemente eines Systems. Die Anzahl der Abhängigkeiten innerhalb des Systems bestimmt darüber, wie viele Elemente betroffen sind.

Die Aktivität 6 umfasst neben der Integration vor allem die Koordination mit organisatorischen Maßnahmen, die Konvertierung von Daten und die Umstellung der Umgebung einschließlich der Information der Benutzer. Sie hat weniger Bedeutung wegen des dazu erforderlichen Aufwands, sondern vielmehr wegen der damit möglichen Vermeidung von Fehlern.

Eine Verringerung des Aufwands ist möglich durch Steigerung der Effektivität einzelner Aktivitäten, durch eine Verringerung ihres Umfangs – insbesondere durch die Verringerung von Abhängigkeiten – sowie durch das Vermeiden von Fehlern. Die Aktivitäten 1, 2 und 3 werden durch vorhandene Methoden und Werkzeuge nur unzureichend unterstützt, sind aber für die erfolgreiche Ausführung einer Änderung von grundlegender Bedeutung. Die Entwicklung von Methoden und Werkzeugen für diese Aktivitäten stellt deshalb einen wichtigen Ansatzpunkt zur Erhöhung der Effektivität dar.

## 1.2 Evolution, Vereinfachung und Vermeidung von Architectural Decay

Bei der Durchführung von Änderungen geht die Eigenschaft Wartbarkeit einer Architektur „verloren“ – ein Effekt, der in der Literatur als *Architectural Decay* [Mayrhauser Wang 1999] bezeichnet wird. Der Effekt verhindert die Weiterentwicklung einer Software, die im Folgenden als Evolution bezeichnet wird. Dieser Effekt entsteht dadurch, dass mangelndes Verständnis interner Zusammenhänge der Systeme (o.g. Aktivitäten 1 und 2) eine fehlerhafte Ausführung von Veränderungen (Aktivitäten 3, 4) zur Folge hat. Dadurch kommt es, noch verstärkt durch die daraufhin notwendige Behebung von Fehlern, zu einem Verlust der Struktur und zur Störung der Architektur. Die Architektur eines Systems wird dann nicht an geänderte Anforderungen angepasst, die Struktur der Software ähnelt mit zunehmender Zahl der Änderungen einem Flickenteppich. Als Konsequenz verringert sich die Verständlichkeit des Systems weiter, und die Anzahl von internen Abhängigkeiten steigt. Dies erschwert nachfolgende Änderungen und Fehlerkorrekturen noch stärker. Dadurch ergibt sich eine lawinenartige Verschlechterung von Struktur, Verständlichkeit und Änderbarkeit eines Systems führt. Dieser Prozess führt zu einem Zustand des Systems, in dem Änderungen nicht mehr durchführbar sind, weil dann keine stabile Arbeitsweise des Systems mehr erreicht werden kann. Evolution ist dann nicht mehr möglich. Dieser Effekt tritt besonders dann ein, wenn Änderungen aufgrund von Zeitdruck, mangelnder Kompetenz, geringer Verständlichkeit der Software oder fehlerhaftem Management nur unvollständig durchgeführt werden.

Damit dieser Effekt vermieden werden kann, sind

- Änderungen weitgehend zu erleichtern,
- Entwicklungs- und Änderungsprozesse so zu gestalten, dass Architekturverluste minimiert werden,
- Methoden zur Wiederherstellung der Architekturqualität zu entwickeln, insbesondere durch Neustrukturieren bei Beibehaltung des Verhaltens durch Refactoring (siehe 1.3)
- Änderungen vorzubereiten und vorzufertigen: Komposition (als Zusammenfügen vorgefertigter Komponenten) und Konfiguration (als deren anforderungsgerechtes Zusammenstellen)

Gelingt es, Systeme *einfacher zu gestalten* und ist damit ihre Komplexität zu verringern, hat dies einen geringeren Änderungsaufwand zur Folge, der ebenfalls eine Erhöhung der Flexibilität und eine Verringerung der Gefahr des Architectural Decay bewirkt, wodurch sich wiederum die Langlebigkeit verbessert. Zum Verringern der Komplexität stehen Grundprinzipien der Softwaretechnik wie Strukturieren, Aufteilen, Kapseln und hierarchisches Gliedern zur Verfügung. Sie sollen dazu genutzt werden, interne Abhängigkeiten zwischen Elementen zu verringern. Dadurch soll erreicht werden, dass sich eine geänderte Anforderung auf weniger Elemente auswirkt (Aktivität 3 in Abschnitt 1.1), wodurch der Änderungsaufwand sinkt.

Durch die explizite Darstellung von Bedingungen und Abhängigkeiten sollen diese für Entwickler sichtbar gemacht werden, damit die Klarheit von Strukturen verbessert und dadurch der Aufwand für das Verstehen gesenkt wird (Aktivitäten 1 und 2 in Abschnitt 1.1).

Die Automatisierung von Prüfungen verringert den dafür notwendigen Aufwand (Aktivität 4) und trägt zusätzlich zur Verminderung der Fehlerrisiken nach Veränderungen bei, weil Prüfungen häufiger und umfassender durchgeführt werden können.

Mittel zur Automatisierung der Pflege von Dokumentationen tragen dazu bei, den Aufwand solcher Folgetätigkeiten von Änderungen zu verringern (Aktivität 3 und 5). Zusätzlich helfen sie, Dokumentationen aktuell und konsistent zu halten, wodurch die Klarheit eines Systems erhalten, Architectural Decay verringert und alle daraus resultierenden Folgefehler und deren Korrektur vermieden werden.

## 1.3 Erhaltung der Wartbarkeit durch Refactoring

Ein System kann so lange genutzt werden, wie es flexibel bleibt und an sich ändernde Anforderungen angepasst werden kann. Die Flexibilität eines Softwaresystems wird wesentlich durch den Änderungsaufwand bestimmt, der durch das Qualitätsmerkmal Wartbarkeit erfasst wird. Die Architektur eines Systems hat großen Einfluss auf seine Flexibilität. Änderungen von Anforderungen erfordern häufig Änderungen der Architektur, insbesondere wenn sie nichtfunktionale Merkmale wie Robustheit, Zeitverhalten oder Verfügbarkeit betreffen. Trotz der Änderungen soll die resultierende Architektur wieder so gestaltet werden, dass sie eine hohe Flexibilität aufweist. Klarheit ist hierbei wieder ein wichtiges Ziel.

Architekturänderungen sollen zur Erhaltung der Klarheit durchgängig und konsistent durchgeführt werden und zu Vereinfachungen führen. Dies erfordert Änderungen an Elementen eines Softwaresystems, wobei vorrangig Strukturen, weniger aber Verhalten und Funktionalität betroffen sind. Strukturelle Änderungen bei weitgehender Beibehaltung der Funktion werden als *Refactoring* bezeichnet, weil sie auf das Neu-Schneiden dieser Elemente abzielen [Fowler 1999]. Beispiele für Refactoring-Maßnahmen sind die Zerlegung einer Software in unabhängige Module und die Kapselung variabler Softwareteile in separate Komponenten. Da in vielen Fällen Informationen über Anforderungen, Verhalten und Struktur der Software fehlen und nur der Quellcode als zuverlässige Informationsquelle zur Verfügung steht, ist vor dem Refactoring die Wiedergewinnung dieser Informationen erforderlich, was als *Reverse Engineering* bezeichnet wird. Änderungen müssen von Refactoring begleitet werden, damit die Architektur angepasst und eine Langlebigkeit von Systemen erreicht werden kann.

## 1.4 Zielstellung und Lösungsansatz

Die vorliegende Arbeit baut auf den Methoden der Objektorientierung und Wiederverwendung auf, zu deren Zielen ebenfalls Wartbarkeit und Flexibilität gehören. In der Arbeit wird der Ansatz verfolgt, durch Vorfertigung von Varianten von Software die Flexibilität zu sichern. Flexibilität von Software kann durch ihre Variabilität erreicht werden, eine Eigenschaft von Software, mit möglichst geringem Zusatzaufwand statt genau einer Aufgabe weitere, verwandte Aufgaben erfüllen zu können (siehe 2.2).

Der Begriff Serienfertigung bezeichnet in der Produktion materieller Güter einen solchen Ansatz der Vorfertigung; er hebt dabei den Unterschied zur Einzelfertigung - mit hoher Flexibilität bei hohem Aufwand - und zur Massenfertigung - mit geringem Aufwand bei geringer Flexibilität - hervor. Die Durchführung von Änderungen soll bezüglich Aufwand, Umfang und Auswirkungen dadurch vereinfacht werden, dass Varianten vorgefertigt werden. Software-Produktlinien realisieren einen derartigen Ansatz der Vorfertigung von Teilen, aus denen Softwaresysteme geschaffen werden können, die Ähnlichkeiten in Funktionalität und technischer Lösung aufweisen (siehe 2.1.5).

Die Forderung nach Flexibilität wird erfüllt, wenn vorgefertigte Elemente schnell und mit geringem Aufwand zu neuen Produkten zusammengefügt werden können. Änderungen sollen durch Komposition solcher Elemente umgesetzt werden. Für eine werkzeuggestützte Komposition und Konfiguration von Systemen müssen Eigenschaften,

Abhängigkeiten und Leistungen der vorgefertigten Elemente in einer Form beschrieben werden, die von den Werkzeugen ausgewertet werden kann. Der manuelle Aufwand bei der Bearbeitung, Komposition sowie Konfiguration von Systemen ist zu minimieren.

Damit die Produkte die jeweiligen Kundenanforderungen erfüllen, ist die Vorfertigung der Elemente bezüglich funktionaler und nichtfunktionaler Eigenschaften zu planen. Damit die Elemente passgenau verwendet werden können, müssen sie in einer Weise portioniert sein, dass sie ohne großen Aufwand entsprechend den Anforderungen genutzt werden können. Dabei muss die Komplexität der Elemente und der entstehenden Produkte gering bleiben, damit zukünftige Änderungen nicht erschwert werden.

Von besonderer Bedeutung für die Erhaltung der Wartbarkeit ist die Benutzung von Modellen und Dokumentationen, weil sie hilft, die Verständlichkeit und die Architektur zu erhalten. Die Akzeptanz und Nutzung dieser Hilfsmittel in der Praxis muss bei der Methoden- und Werkzeugentwicklung berücksichtigt werden.

Damit die Methoden in der industriellen Praxis erfolgreich einsetzbar sind, müssen sie bei großen Systemen eingesetzt werden können. Skalierbarkeit drückt die Eigenschaft einer Methode aus, von einer Anwendung an kleinen Gegenständen auf große übertragbar zu sein. Bei der Methodenentwicklung für Software-Produktlinien ist diese Eigenschaft ein wichtiges Ziel, weil Produktlinien typischerweise einen großen Umfang aufweisen.

Die zu entwickelnden Methoden sollen neben der Neuentwicklung auch für die Weiterentwicklung existierender Software verwendbar sein, damit eine Langlebigkeit der Systeme erreicht wird. Der Effekt des Architectural Decay soll dabei durch Reengineering verringert werden.

Dazu wird die Entwicklungsmethodik ALEXANDRIA erarbeitet, die die klassischen Entwicklungsmethoden des Forward Engineering und die Analysetechniken des Reverse Engineering mit Software-Komposition verbindet (Abb. 2) und die auf Software-Produktlinien basiert. Die entwickelten Methoden sind durch folgende Elemente und Prinzipien gekennzeichnet:

- Planung der wiederverwendbaren Plattform einer Produktlinie mittels Domain Engineering und Featuremodellen (siehe Kap. 4)
- Gliederung von Systemen gemäß Features nach dem *Separation-of-Concerns*-Prinzip in Komponenten nach dem Hyperspace-Ansatz (siehe Kap. 6)
- Werkzeugunterstützte Komposition und Konfiguration variabel kombinierbarer Modelle zu Produkten auf Basis der UML (siehe Kap. 5)
- Unterstützung der modellbasierten Weiterentwicklung durch Traceability-Links zwischen Modellen (siehe 4.2)
- Automatisierung der Erstellung und Aktualisierung von Dokumentationen (siehe 6.1.2) und Testmitteln (siehe 6.1.3)
- Einbindung von Migrations- und Refactoring-Maßnahmen in den Entwicklungsprozess, damit Legacy-Komponenten mittels Reengineering weiterentwickelt werden können (siehe Kap. 1).

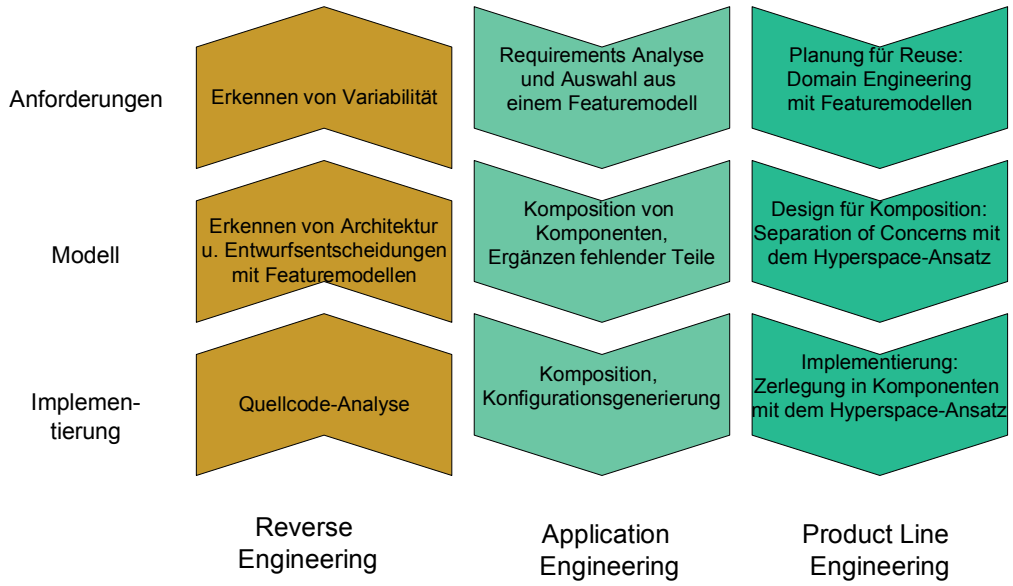



*Abb. 2: Prozesse der Softwareentwicklung mit Produktlinien nach der ALEXANDRIA-Methodik*

Die Erfahrungen bei der industriellen Einführung der Software-Wiederverwendung zeigen, dass neben den softwaretechnischen und methodischen noch weitere Aspekte wie

- technische Unterstützung durch Werkzeuge und Verfahren,
- organisatorisches Umfeld,
- Beachtung strategischer Unternehmensziele,
- langfristiger Ausrichtung aller Entscheidungen und Maßnahmen und
- Führung und langfristige Motivation der Mitarbeiter

zu berücksichtigen sind, damit die Ergebnisse erfolgreich umgesetzt werden können. Solche Aspekte werden deshalb ergänzend zur softwaretechnischen Methodenentwicklung betrachtet (Kap. 9).

## 1.5 Aufbau der Arbeit

Die Einleitung beschrieb die Problemstellung, der die Arbeit gewidmet ist. Die in dieser Arbeit vorgestellten Lösungen basieren auf Ergebnissen von Vorarbeiten in verschiedenen Disziplinen; diese werden im zweiten Kapitel zusammengefasst, im Überblick vorgestellt und bewertet. Im dritten Kapitel wird die ALEXANDRIA-Methodik vorgestellt und das Fallbeispiel eingeführt, das zur Illustration der Ergebnisse verwendet wird. Die Kapitel 4, 5 und 6 enthalten die Hauptbeiträge der Featuremodellierung, der modellbasierten Komposition mit Hyper/UML und der Produktlinien-Methode HyperFeatuRSEB. In Kapitel 7 werden diese Lösungen durch Maßnahmen zur Laufzeit-Variabilität von Produktlinien ergänzt. In Kapitel 1 wird die Produktlinien-Methode um die Einbindung existierender Software erweitert, damit wird eine evolutionäre Weiterentwicklung erreichbar. Kapitel 9 enthält über softwaretechnische Lösungen hinaus Prinzipien, Empfehlungen und Hinweise zur Einbettung in ein Umfeld, die aus Erfahrungen mit industriellen Projekten gewonnen wurden und die zur erfolgreichen praktischen Umsetzung der Lösung benötigt werden. In Kapitel 10 erfolgt eine Bewertung der Eigenschaften der erarbeiteten Lösungen.

# 2 Existierende Beiträge zu Komposition und Evolution

*In diesem Kapitel werden die Ergebnisse von existierenden Arbeiten zusammengefasst und im Überblick vorgestellt, die eine Grundlage für die später in dieser Arbeit vorgestellten Lösungen darstellen. Dabei wird bewertet, wie diese Methoden, Techniken und Vorgehensweisen zum Erreichen der mit dieser Arbeit verfolgten Ziele beitragen.*

## 2.1 Vorfertigung und Wiederverwendung zwecks Evolution und Komposition

Die Vision der Wiederverwendung von vorgefertigten Ergebnissen hat die Entwicklung der Softwaretechnik schon sehr lange beeinflusst. Durch wiederholte Nutzung einmal geschaffener Leistungen soll eine Einsparung von Kosten und Zeit erreicht werden. Damit würden Beiträge zur schnellen Umsetzung von Änderungen geleistet. Daneben wird auch eine evolutionäre Weiterentwicklung von Elementen für einen erweiterten Einsatzbereich angestrebt.

Erste intuitive Ideen zur Wiederverwendung ließen sich nicht erfolgreich umsetzen [Bosch 2000]:

- Wiederverwendung von gerade vorhandenem Code (engl. Code Scavenging oder Code Salvation) ist nicht in ausreichendem Umfang möglich; Wiederverwendung in einer Organisation muss geplant und vorbereitet werden.
- das Zusammenfügen beliebiger Code-Komponenten zu Systemen als reine Bottom-up-Wiederverwendung funktioniert in der Praxis nicht; Wiederverwendungsvorhaben benötigen für eine erfolgreiche Umsetzung einen Top-Down-Ansatz, bei dem existierende Komponenten dadurch eingebunden werden, dass von Anforderungen und Architektur ausgegangen wird.

In der Entwicklung der Wiederverwendungsmethoden seit etwa 1992 wurden die Struktur und die Architektur größerer Software-Einheiten sowie organisatorische Aspekte stärker beachtet, was zuerst zur Etablierung der Komponententechnologie und danach zur Wiederverwendung von teilweise abstrakten Einheiten wie von Frameworks führte. Beide Ansätze stellen eine wichtige Basis der vorliegenden Arbeit dar, sie werden im Folgenden vorgestellt.

### 2.1.1 Komponententechnologie

Bereits in den ersten höheren Programmiersprachen wurden Unterprogramme und Funktionsbibliotheken als Mittel eingesetzt, damit entwickelte Codebausteine mehrfach verwendet werden können. Techniken der Kapselung wie in Modula-2 und Ada wurden dazu eingeführt, dass durch Entkopplung eine Verbreiterung der Einsatzmöglichkeiten und ein geringerer Aufwand bei Änderungen erreicht wird. Die Komponententechnologie wurde mit der Absicht etabliert, Vorräte an Bausteinen mit genormter Schnittstelle anbieten zu können, aus denen sich schnell und billig neue Softwaresysteme zusammensetzen lassen. Im Idealfall braucht der Entwickler eines Softwaresystems nur die Schnittstelle und nicht die Implementierung einer Komponente zu kennen; ihre Schnittstellenbeschreibung ist leicht verständlich. Die Unterstützung der Beschreibung von Komponenten [Tracz 1987][Riebisch 1993] sowie ihre Einbindung in eine Architektur [Sametinger 2002] wurden zur Verringerung des Aufwands für die Suche und Anpassung von Komponenten entwickelt. Für den Effektivitätsgewinn bei der Entwicklung eines neuen Softwaresystems erweist sich der Aufwand für die Anpassung und Integra-

tion von Komponenten als entscheidende Größe. Daraus resultieren die Hauptprobleme der Komponententechnologie - das Treffen der richtigen Entscheidungen für die Entwicklung neuer Komponenten, wie:

- Passgenauigkeit der Anforderungen – eine Komponente ist meist an der Erfüllung von mehreren Anforderungen beteiligt, Änderungen vermindern die Effektivität der Wiederverwendung,
- Wahl geeigneter Schnittstellen mit Passgenauigkeit zur Architektur – insbesondere nichtfunktionale Anforderungen wie Effizienz und Robustheit werden davon beeinflusst,
- Entscheidung über den Umfang der Komponenten – ist er zu groß, entstehen spezialisierte, schwer verständliche Komponenten mit geringem Anwendungsbereich, ist er zu klein, sind mehr Integrationen je Softwaresystem notwendig; eine solche Komponente ist weniger leistungsfähig und wird seltener eingesetzt, wodurch der Effektivitätsgewinn gering bleibt.

Ein weiteres Problem entsteht durch Veränderungen der Aufgabenstellungen, die auch Veränderungen der bereitgestellten Komponenten erfordern. Für Bereiche mit klar abgegrenzten, wenig veränderlichen Anforderungen ist die Wiederverwendung zur Selbstverständlichkeit geworden, wie bei Datenbanken und Betriebssystemen. Diese Bereiche sind jedoch klein im Vergleich mit der Vielfalt von Aufgabenstellungen.

Für durch Standards und Quasistandards etablierte Komponentenmodelle und Architekturen wird die Komponententechnologie ebenfalls erfolgreich eingesetzt. Die Bedeutung eines Standards wird von der Menge der dafür verfügbaren Komponenten bestimmt, weshalb vor allem industrielle Standards Bedeutung erlangt haben. Die wichtigsten Standards sind die für Microsoft Windows (COM, COM+, DCOM [COM]), für die Programmiersprache Java (JavaBeans [JavaBeans]) und für verteilte Systeme nach dem CORBA-Standard [CORBA]. Diese Komponenten sowie Bibliotheken daraus realisieren eine Wiederverwendung auf der Ebene von Binär- oder Quellcode in Form von Black Boxes. Es konnte ein funktionierender Markt etabliert werden, es sind Komponenten verfügbar, aus denen Softwaresysteme schnell herstellbar sind. Die Lebensdauer solcher Softwaresysteme hängt jedoch direkt von der des Standards ab, der im Fall von COM von der Fa. Microsoft bestimmt wird. Hier finden durchschnittlich alle zwei Jahre sog. Major-Release-Wechsel statt, deren Architekturänderungen oft umfangreiche Änderungen der darauf basierenden Softwaresysteme erfordern.

Der Kontakt zwischen Entwicklern und Konsumenten von Komponenten wird in den meisten Fällen über den Markt hergestellt. Je geringer dabei die Kooperation ist, desto schwieriger wird die Abstimmung von Architektur und Anforderungen. Erfahrungen in der Industrie zeigten, dass eine Verwendung „wie vorhanden“ nur selten möglich ist, weil meist nicht alle Anforderungen erfüllt sind [Bosch 2000] und dann Veränderungen an Komponenten notwendig werden. Das o.g. Problem der Passgenauigkeit der Architektur kann so nur selten gelöst werden.

Die Komponententechnologie als Form der Wiederverwendung hat eine Reihe von Vorteilen [Sametinger 97]:

- Durch den Black-Box-Charakter ist mit Ausnahme der Integration keine weitere Entwicklungsarbeit mehr nötig.
- Auf der Seite des Komponenten-Anwenders kann eine hohe Produktivität der Quellcode-Entwicklung erreicht werden.

- Durch den Black-Box-Charakter ergibt sich ein effektives Information Hiding, die Auswirkungen von Änderungen innerhalb von Komponenten bleiben deshalb begrenzt.
- Das ökonomische Modell ist einfach und funktionstüchtig, da außer Angebot und Kauf keine weiteren Transaktionen notwendig sind.
- Es existieren etablierte Komponentenstandards als Basis eines Marktes.

Den Vorteilen stehen einige schwerwiegende Nachteile gegenüber:

- Anpassungen selbst geringen Umfangs sind nicht möglich.
- Der potentielle Anwender von Komponenten muss die Tätigkeiten der Anforderungsanalyse und des Designs auf jeden Fall ausführen. Der Produktivitätsgewinn bleibt insgesamt gering, da für diese Tätigkeiten ein höherer Aufwand notwendig ist als für die Codierung.
- Quellcode zur Integration und Kopplung von Komponenten (sog. Glue Code) kann nicht wiederverwendet werden.
- Die Erfüllung nichtfunktionaler Anforderungen bleibt aus oder ist zumindest unklar, wodurch das Risiko der Entwicklung steigt.
- Bei der Änderung von Anforderungen ist ein Komponentenaustausch notwendig; ein Plug-And-Play-Ansatz funktioniert hier nicht; meist ist ein Refactoring der Anwendung notwendig. Außerdem entsteht wiederum ein hoher Kodier- und Designaufwand, der fast so hoch wie bei einer Neuentwicklung ist.
- Das Entwurfs- und Anforderungswissen steht der Wiederverwendung nicht zur Verfügung, weil Komponenten nur aus Binär- oder Quellcode bestehen. Eine Dokumentation dient nur der Ergänzung und hat für Anbieter und Anwender eine geringere Priorität als der Code. Die Qualität der Dokumentation ist deshalb geringer.
- Aufgrund des Black-Box-Charakters ist der Anwendungsbereich eingeschränkt. Meist kann nur für einen geringen Teil einer Anwendung die Wiederverwendung von Komponenten genutzt werden.

Die Komponententechnologie allein ist nicht geeignet, die Flexibilität zu gewährleisten und den Änderungsaufwand zu senken sowie Langlebigkeit zu erreichen. Einige dieser Nachteile lassen sich durch Anpassungstechniken wie Variabilität (siehe 2.2.3) und Adaption [Bosch 2000] beheben. Außerdem können Komponenten als Bestandteile flexibler Architekturen (siehe 2.2.3) diese Ziele erreichen; Frameworks bieten ebenfalls vorgefertigte Anpassungsmöglichkeiten (siehe 2.1.2).

Für die Produktlinien-Methodik ALEXANDRIA stellt die Komponententechnologie eine wichtige Basis dar, sie wird in Verbindung mit dem Hyperspace-Ansatz (siehe 2.3.4) und mit Architekturen angewendet.

### 2.1.2 Objektorientierte Application Frameworks

Zur Steigerung der Effektivität der Wiederverwendung wurden auf Basis der Objektorientierung die Application Frameworks entwickelt. Dabei handelt es sich um wiederverwendbare, halbfertige Anwendungen, die zur Entwicklung von Kundensoftware spezialisiert werden können [Johnson Foote 1998]. Ein objektorientiertes Application Framework stellt den wiederverwendbaren Entwurf eines Systems dar, der beschreibt, wie das System in eine Menge interagierender Objekte zerlegt ist. Es beschreibt sowohl die beteiligten Komponenten und Objekte als auch ihre Interaktion. Dazu gehört die Beschreibung der Schnittstellen der Objekte und des Steuerflusses zwischen ihnen

sowie die Abbildung der Systemanforderungen auf die Objekte [Johnson Foote 1998][Wirfs-Brook Johnson 1990].

Neben dem Entwurf wird der Quellcode zwar ebenfalls wiederverwendet, die Aspekte der Komponentenstruktur und damit der Wiederverwendung von Architektur und Entwurf haben jedoch eine größere Bedeutung. Im Gegensatz zu Komponenten wird im Fall von Frameworks ein Softwaresystem wiederverwendet. Der Aufwand für Analyse und Entwurf der Struktur des Softwaresystems ist damit in die Wiederverwendung einbezogen. Komponentenbibliotheken sind als Menge konkreter Verfeinerungen für spezielle Anwendungsfälle jedoch häufig in Frameworks enthalten.

Nach der Art ihrer Erweiterung und Anwendung wird zwischen den Typen Black-Box- und White-Box-Framework unterschieden. White-Box-Frameworks werden mit den Mitteln objektorientierter Programmiersprachen wie Vererbung und Überschreiben an spezielle Aufgaben angepasst. Sie sind flexibler erweiterbar, jedoch schwieriger zu verwenden, weil dazu umfangreiches Wissen erforderlich ist. Black-Box-Frameworks stellen Komponentenschnittstellen zur Verfügung, die durch Komposition und Delegation erweitert werden können. Sie erfordern einen geringeren Einarbeitungsaufwand, weil eine bessere Kapselung ihrer Implementierung vorliegt. Sie sind jedoch schwieriger zu entwickeln, weil ihre Schnittstellen für einen weiten Anwendungsbereich stabil sein müssen.

Eine Zuordnung von Application Frameworks zu diesen beiden Typen erfolgt nicht streng, es existiert ein fließender Übergang zwischen den Typen. Meist werden Frameworks zuerst als White-Box-Frameworks entwickelt und „reifen“ während häufiger Nutzung und Weiterentwicklung zu Black-Box-Frameworks.

Erweiterbarkeit von Application Frameworks wird durch sog. Hooks erreicht [Froehlich et al. 1999]. Sie stellen von Framework-Entwickler vorgesehene Erweiterungsstellen für Entwurf und Implementierung dar, zum Beispiel durch Parameter oder Unterklassen. Mit Hilfe von strukturierten Texten als Hook-Beschreibungen können Experten einer Anwendungsdomäne Anforderungen an die Frameworkentwicklung beschreiben, ohne über detaillierte Kenntnisse objektorientierter Programmiersprachen verfügen zu müssen.

Die als Hot Spots bezeichneten Variabilitätspunkte in Frameworks stellen vordefinierte Verfeinerungen für die Entwicklung von Softwaresystemen dar [Pree1999]. Sie werden durch Mittel wie Hook-Methoden, Design Patterns (siehe 2.1.3) oder abstrakte Klassen implementiert. Hot Spot Cards sind eine Möglichkeit für die Beschreibung der Variabilität eines Frameworks. Sie werden durch strukturierte Texte beschrieben und stellen ein Kommunikationsmittel zwischen Frameworkentwicklern und Anwendungsentwicklern dar. Daraus werden durch Aufteilung und Verfeinerung Hot Spots entwickelt, die auch rekursiv aus weiteren Hot Spots zusammengesetzt werden können.

Für die erfolgreiche Entwicklung von Application Frameworks – insbesondere für die Entwicklung der benötigten Variabilität in Form von Hot Spots – ist eine Auswertung von Domänenwissen (siehe 2.2.1) notwendig, damit weitere Anforderungen an ein Framework erkannt werden können. Dazu wird zunächst Wissen als Beziehungen zwischen Begriffen in sog. Knowledge Graphs modelliert, aus denen durch Verfeinerung und Abstraktion sog. Knowledge Domains entwickelt werden [Aksit et al. 1999]. Die Definition der Variabilität eines Frameworks auf Basis dieser Modelle wird nur wenig methodisch unterstützt, hier wird in der Literatur auf die Erfahrung der Entwickler und die Unterstützung durch Experten verwiesen

Bei der Schaffung und Wiederverwendung von Application Frameworks erfolgt typischerweise eine Zweiteilung der Tätigkeiten in Framework-Entwicklung und Anwendungsentwicklung. Die Framework-Entwicklung erarbeitet in einem iterativen Vorgehen ein Application Framework mit Variationspunkten entsprechend den Anforderungen der Domäne. Anwendungsentwickler verwenden das Framework durch Anpassung, Erweiterung und Konkretisierung und entwickeln daraus ein Softwaresystem. Zur Vermeidung von Fehlern bei Erweiterung und Verfeinerung der Frameworks werden durch die Framework-Entwickler Erläuterungen und Beispiele bereitgestellt, die häufig in Form von sog. Rezepten und Kochbüchern formuliert werden. Durch eine geeignete Beschreibung der vorgesehenen Erweiterungen kann die Anwendungsentwicklung in begrenztem Umfang durch Werkzeuge unterstützt werden [Ivanov 1999]. Sind Änderungen über das beim Framework-Entwurf vorgesehene Maß hinaus notwendig, müssen Änderungen an der Architektur der Anwendung oder ein Refactoring des Frameworks erfolgen.

Die Qualität eines Application Frameworks wird während der iterativen Weiterentwicklung schrittweise verbessert. Der Erfolg der Weiterentwicklung hängt stark von der Qualifikation der Framework-Entwicklers ab, weil eine methodische Unterstützung weitgehend fehlt. Die mit Änderungen an Frameworks verbundenen Risiken sind hoch, da eine Vielzahl von Einsatzfällen berücksichtigt werden müssen. Während der Veränderungen entstandene Fehler beeinträchtigen die Weiterentwicklung der Frameworks.

Application Frameworks weisen einige wesentliche Vorteile auf [Fayad et al. 1999]:

- Durch *Modularisierung* wird die Implementierung in gewissem Umfang durch stabile Schnittstellen gekapselt.
- Die *Wiederverwendbarkeit* eines größeren Teils des Entwicklungsaufwands wird durch verallgemeinerte Komponenten und die Nutzung von Domänenwissen erreicht.
- Eine *Erweiterbarkeit* der Frameworks ist durch Hook-Methoden erreichbar.

Das industrielle Potential der Wiederverwendung von Frameworks ist deutlich größer als das der Komponenten [Bosch 2000]. Dank der zunehmenden Entwicklung von methodischer Unterstützung sind die erreichten Ergebnisse kontinuierlich besser geworden. Trotzdem decken Frameworks nur einen geringen Anteil eines Softwaresystems ab. Außerdem ist die praktische Wirkung von Methoden für die Weiterentwicklung und damit die Evolution von Frameworks noch zu gering; in der industriellen Praxis werden Kapselung, Entkopplung und Strukturierung entsprechend der Anforderungen nicht genügend erreicht. Die wesentlichen Probleme und Nachteile von Application Frameworks sind nach [Fayad et al. 1999]:

- *Hoher Entwicklungsaufwand*. Die erforderliche hohe Qualität und die Erweiterbarkeit für verschiedene Anwendungsbereiche erfordern einen noch höheren Aufwand als für komplexe Softwaresysteme.
- *Hoher Lernaufwand*. Die Nutzung eines objektorientierten Application Frameworks erfordert beträchtlichen Aufwand vom Entwickler, bis er effektiv Softwaresysteme entwickeln kann. Ob sich ein solcher Aufwand lohnt, ist nicht von vornherein sicher. Anwendungsentwickler benötigen die Unterstützung der Framework-Entwickler bei Anpassung und Refactoring. Wegen fehlender Einheitlichkeit der Frameworks ist der Aufwand für Lernen und Unterstützung bei jedem Framework neu erforderlich.

- *Mangelnde Integrierbarkeit mit anderen Frameworks*. Jedes Application Framework enthält eine Struktur und Erweiterungsmöglichkeiten. Eine Interaktion mit anderen Frameworks erfordert meist Änderungen der Struktur, die nicht vorgesehen sind.
- *Mangelnde Wartbarkeit*. Änderungen an einem Framework betreffen immer auch die Softwaresysteme, die mit ihnen erstellt sind. Migrationen müssen nachvollzogen werden. Bei der manuellen Nutzung eines Frameworks unterlaufen den Anwendungsentwicklern aufgrund der Komplexität der Aufgabe häufig Fehler.
- *Aufwendige Validierung und Test*. Verallgemeinerte Lösungen wie Frameworks sind schwieriger mit konkreten Fällen zu testen. Außerdem ist der Test der für eine Anwendung vorgenommenen aufwendiger als bei normaler Software, weil die Struktur des Steuerflusses vom Framework vorgegeben wird.
- *Komplexität*. Variabilität und abstrakte Elemente erfordern zusätzlichen Quellcode, wodurch die Komplexität von Frameworks höher ist als die von Komponenten mit vergleichbaren Aufgaben.
- *Fehlende Unterstützung durch Methoden und Werkzeuge*. Für die Entwicklung von Application Frameworks gibt es keine etablierten Standards und wenige Methoden und Werkzeuge. Insbesondere für die planmäßige Entwicklung ausgehend von Domänen- und Anforderungsanalyse fehlen Methoden und Beschreibungsmittel, die Werkzeugunterstützung erlauben. Dies wirkt sich sowohl bei der Entwicklung als auch bei der Anwendung von Frameworks aus.

Diese Probleme und Mängel treffen mit der Eigenschaft von Application Frameworks zusammen, einen iterativen Prozess zur Weiterentwicklung und Reifung zu erfordern. Als Folge besteht ein hohes Risiko, dass die Entwicklung anstelle zu einer stabilen Architektur und zu hoher Softwarequalität zum Effekt des Architectural Decay führt, der eine langfristige Nutzung oder eine weitere Evolution des Frameworks verhindert.

Für eine Verbesserung dieser Situation wird mehr methodische Unterstützung für die Erstellung und Weiterentwicklung von Frameworks benötigt, insbesondere für deren Planung aufgrund von Anforderungen der Domäne [Bosch 2000] sowie für die Entwicklung von Variabilität und deren Konkretisierung. Auch Kompositionstechniken können die Framework-Anwendung unterstützen [Aksit et al. 1999]. Die Kernidee der Vorfertigung von Systemen mit Variabilität wird durch die Produktlinien-Methoden weiterentwickelt und auch mit der Methodik ALEXANDRIA verfolgt. Der Gedanke der Black-Box-Frameworks wird von den Komponenten-Architekturen mit Plug-in-Schnittstellen aufgegriffen, der als Erweiterung der Methodik vorgeschlagen wird.

### 2.1.3 Wiederverwendung etablierter Prinziplösungen als Patterns

Patterns stellen eine Möglichkeit dar, Entwurfswissen und Prinziplösungen in klar verständlicher Form zu beschreiben. Sie ermöglichen eine Wiederverwendung auf der Ebene des Entwurfs, weil sie etablierte Lösungsprinzipien beschreiben, die auch zur Erfüllung nichtfunktionaler Anforderungen geeignet sind.

Die Beschreibung der Prinzipien lehnt sich an Formen an, die bei Lösungsprinzipien auf anderen Gebieten üblich sind. Nach dem Abstraktionsgrad - bezogen auf den Quellcode - lassen sich drei Klassen von Patterns unterscheiden:

- Design Patterns [Gamma et al. 1996] beschreiben Prinziplösungen für Beziehungen zwischen Klassen und Objekten.

- Architectural Patterns sind Beschreibungen von Prinziplösungen, die eher dem Grobentwurf und der Architektur zuzuordnen sind [Buschmann et al. 1996].
- Idioms sind Beschreibungen der prinzipiellen Benutzung von Programmiersprachelementen.

Patterns sind Entwurfshilfsmittel in Form natürlichsprachlicher und halbformaler Beschreibungen. Bis auf die Idioms enthalten sie keinen Quellcode einer Programmiersprache, sondern beschreiben Lösungen mit Hilfe von strukturiertem Text und Diagrammen des Softwareentwurfs, ergänzt durch Beispiele. Die Wiederverwendung des in Pattern beschriebenen Wissens erfolgt dadurch, dass ein Entwickler seine Problemstellung mit den Prinziplösungen vorliegender Patterns vergleicht und ein oder mehrere Patterns auswählt. Dazu muss er die Lösungsprinzipien der Patterns verstehen. Das zur Benutzung ausgewählte Pattern muss er anschließend zur aktuellen Problemstellung in Bezug setzen und die Struktur des Patterns für die konkrete Situation der Problemstellung umsetzen.

Die Vorteile von Patterns bestehen in der Unterstützung der Softwareentwicklung durch Mittel zur Erhöhung der Variabilität, durch Vereinfachung der Dokumentation und durch Vereinheitlichung von Lösungen. Als Nachteil ist zu nennen, dass sie keinen wesentlichen Beitrag zur Vorfertigung leisten. Auch ihre automatisierte Anwendung ist nicht möglich, weil weitergehende Werkzeuge auf der Basis formalisierter Beschreibungen in ihrer Wirksamkeit dadurch begrenzt sind, dass Patterns immer von einem Kontext abhängen, der potentiell unbegrenzt ist und dessen Umfang von der Erfahrung von Entwerfer und Anwender eines Patterns abhängt, wie [Züllighoven et al. 1998] feststellt. Bei ihrer manuellen Implementierung entstehen häufig Fehler. Ihre Nutzung ist von der Beherrschung der jeweiligen natürlichen Sprache abhängig.

Der wichtige Beitrag von Patterns für Variabilität und Flexibilität besteht in der Bereitstellung von Prinziplösungen für Entwurf und Implementierung, die bestimmte funktionale und nichtfunktionale Eigenschaften aufweisen. In der Produktlinien-Methodik ALEXANDRIA werden Patterns vor allem für das Refactoring (siehe 6.1.2 und 8.3) und für Variabilität zur Laufzeit (siehe 7.1) verwendet.

### 2.1.4 Vorfertigung mittels Programm-Generatoren und Baukasten-Systemen

Komponenten können sehr bequem wiederverwendet werden, wenn ihre Nutzung durch Werkzeuge automatisiert wird. Auf einer Basis einer Beschreibung von Verhalten und Struktur wird bei Generator-Ansätzen Quellcode aus Bausteinen konfiguriert oder generiert [Czarnecki et al. 2000]. Erfolgreich konnten solche Ansätze für bestimmte, eng begrenzte Domänen wie Nutzerschnittstellen (GUI) oder Steuerungsprogrammierung angewendet werden. Für graphische Nutzerschnittstellen wird aus einer graphischen oder textuellen Spezifikation Programmcode generiert.

Wartbarkeit und Flexibilität sind bei diesen Ansätzen sehr gut, dafür ist die Portabilität gering. Die Entwicklung von Software ist bezüglich Plattform, Leistungsfähigkeit und Herstellerbindung unmittelbar vom Generator-Werkzeug abhängig. Die Generator-Werkzeuge sind nur teilweise so erweiterbar, dass eigene Bausteine in Lösungen eingebunden werden können. Die Leistungsfähigkeit und Lebensdauer der damit erstellten Softwaresysteme sind ebenfalls von den betreffenden Werkzeugen abhängig.

Ansätze der Generativen Programmierung benutzen ebenfalls Generatoren. Solche Ansätze werden in Abschnitt 2.3 unter den Aspekten von Komposition und Zerlegung untersucht.

### 2.1.5 Software-Produktlinien

Anwendungsorientierung als Leitgedanke bei der Schaffung wiederverwendbarer Komponenten verbessert deren Wiederverwendungsgrad. Bei der Objekttechnologie stellt Anwendungsorientierung zwar ebenfalls ein wichtiges Ziel dar [Booch 1991][Coad Yourdon 1992], es wird bei der Strukturierung von Implementierungen jedoch meist zugunsten technischer Kriterien vernachlässigt. Im Werkzeug-Material-Ansatz WAM [Züllighoven et al. 1998] steht die Anwendungsorientierung im Vordergrund, damit die Anpassbarkeit und Benutzbarkeit von Software verbessert wird.

Die Verstärkung des Leitgedankens der Anwendungsorientierung bei Wiederverwendungs-Methoden führt zur Entwicklung von *Software-Produktlinien*. Eine Produktlinie ist eine Familie von Produkten mit Gemeinsamkeiten bezüglich Aufbau, Herstellung oder Kundenkreis. Der Begriff wurde in der Betriebswirtschaftslehre geprägt (siehe [Kotler Bliemel 1999] S. 680). Für die Softwareentwicklung wird dieser Begriff etwas enger gefasst, Software-Produktlinien bezeichnen Familien von Softwaresystemen, die Ähnlichkeiten in Funktionalität und technischer Lösung aufweisen.

Software-Produktlinien-Methoden haben die koordinierte Entwicklung einer Gruppe ähnlicher Softwaresysteme zum Ziel. Sie bauen auf Wiederverwendungstechniken und auf Domain-Engineering-Methoden auf. Produktlinien-Architekturen und Systemfamilien sind synonyme Bezeichnungen für diesen Ansatz.

Entsprechend der Grundidee der Software-Produktlinie werden Bestandteile, die für alle Systeme einer Produktlinie benötigt werden, als gemeinsamer Kern implementiert. Er wird auch als Referenzarchitektur oder als wiederverwendbare Plattform bezeichnet. Zur Schaffung verschiedener Produkte wird der Kern durch variable Teile ergänzt. Die variablen Teile erfüllen jeweils eine Teilmenge von Anforderungen. Die Systeme werden durch Zusammenfügen von Kern und spezifischen Bestandteilen anhand jeweils zutreffender Anforderungen erzeugt (Abb. 3). Diese konsequente methodische Auswertung von Anforderungen stellt einen wesentlichen Fortschritt gegenüber den vorher genannten Ansätzen dar: Wiederverwendung mit Komponententechnologie erfolgt nur auf der Ebene von Quellcode; bei Frameworks ist zwar auch der Entwurf Gegenstand der Wiederverwendung, seine Erarbeitung aus Anforderungen wird jedoch meist dem Domänen-Experten übertragen.

Der wirtschaftliche Vorteil des Produktlinien-Ansatzes liegt in der schnelleren und weniger aufwendigen Implementierung neuer Produkte, wenn der Kern bereits wesentliche Funktionalität enthält und die variablen Teile flexibel kombinierbar sind.

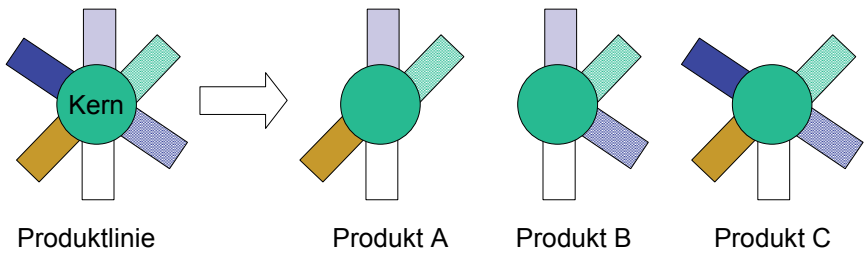



*Abb. 3: Aufbau von Produktlinie und Produkten aus Kern und variablen Teilen*

Produktlinien-Methoden bilden die Grundlage der vorliegenden Arbeit. Dieser Lösungsansatz erweitert die oben vorgestellten Methoden um Langfristigkeit, Planung der

Wiederverwendung, Variabilität und Zerlegung. Verschiedene Beiträge zu Produktlinien werden in den Abschnitten 2.2 bis 2.6 betrachtet.

Es sind eine Reihe von Produktlinien-Methoden bekannt, die teilweise aufeinander aufbauen (Abb. 4) und die in dieser Arbeit in unterschiedlicher Weise berücksichtigt werden. Der grundsätzliche Ablauf der Entwicklung von Produktlinie und Produkten ist bei den Methoden ähnlich:

- Erfassen von gemeinsamen Anforderungen einer Gruppe von Softwaresystemen und Modellierung der Anforderungen als sog. Features (siehe 2.2.1),
- Implementieren dieser gemeinsamen Anforderungen als gemeinsamer Kern aller Produkte,
- Beschreiben der Anforderungen an ein Softwaresystem als Konfiguration von gemeinsamen und spezifischen Anforderungen,
- Implementieren der spezifischen Anforderungen an das Softwaresysteme in Form von variablen Teilen, so dass Kern und variable Teile zusammen die Anforderungen an dieses System erfüllen,
- Komposition der Konfiguration von Kern und variablen Teilen

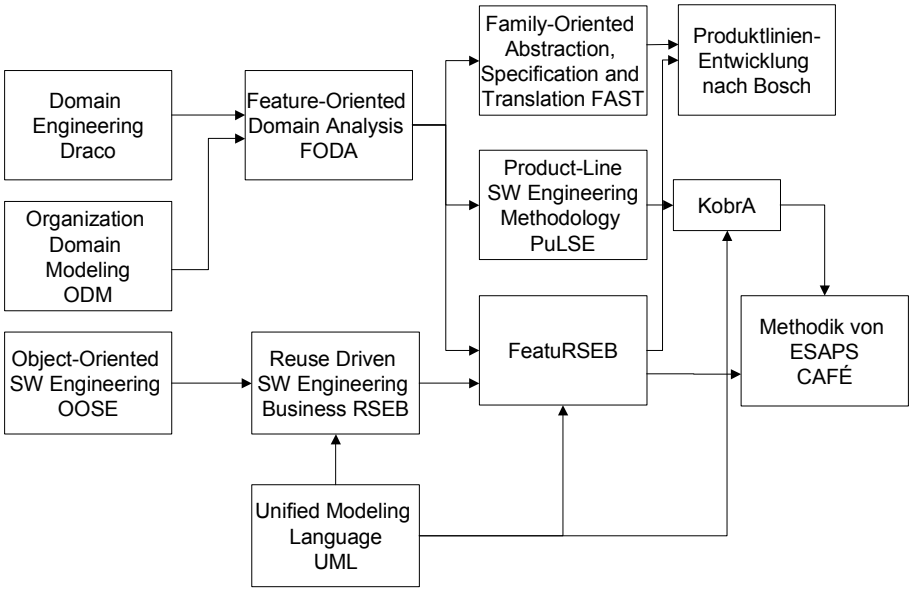



*Abb. 4: Entwicklungsfolge von Produktlinien-Methoden*

Die Methoden Draco, ODM und FODA sind der Domänenanalyse zuzuordnen, auf sie wird in Abschnitt 2.2.1 eingegangen.

Bei Reuse Driven Software Engineering Business RSEB [Jacobson et al. 1997] handelt es sich um eine aus der OOSE-Methode hervorgegangene Vorgehensweise, die eine objektorientierte Entwicklung von Produktlinien auf der Basis der Unified Modeling Language UML [UML 2001] zum Ziel hat. Wiederverwendung wird hier anhand von Use Cases organisiert. Wegen des hohen Anteils informaler Ausdrucksmittel ist Werkzeugunterstützung dieser Vorgehensweise nur sehr begrenzt möglich.

Die Methode FAST, Family-Oriented Abstraction, Specification and Translation definiert ebenfalls Phasen, Aktivitäten und Ergebnisse von Entwicklungsprozessen für

Produktlinien. Dazu werden Prozesse durch Zustandsmodelle beschrieben. Die Umsetzung und Anpassung dieser Prozesse auf eine konkrete Modellierungstechnik und Notation für die Softwarearchitektur wird jedoch nicht vorgenommen. FAST wurde von AT&T entwickelt und wird dort erfolgreich genutzt [WeissLai 1999]. Für FAST fehlen über die Prozessbeschreibung hinausgehende Informationen, die eine Anwendung ermöglichen würden.

Die Product-Line Software Engineering Methodology PuLSE beschreibt den Entwicklungsprozess von Produktlinien mit Phasen, Ergebnissen und Aktivitäten auf abstraktem Niveau, ohne eine konkrete Modellierungstechnik und Notation vorzugeben.

KobrA konkretisiert dies für objektorientierte Entwicklung mit der UML, es ist eine Methode auf der Basis von PuLSE. PuLSE und KobrA sind Produkte des Fraunhofer-Instituts für Experimentelles Software Engineering, die in Zusammenhang mit Beratungsdienstleistungen vermarktet werden. Für KobrA stehen erst seit kurzem weitergehende Informationen zur Verfügung, doch beziehen sich diese vor allem auf die Einbettung in Entwicklungsprozess, Organisation und Umfeld [Atkinson et al. 2002]. Dabei bleiben die Aussagen zum Prozess sehr allgemein und lassen wenig Fortschritte gegenüber dem Stand der Technik erkennen, beispielsweise in den Bereichen Qualitätsmanagement, Konfigurationsmanagement sowie im Software-Entwicklungsprozess ohne Mehrgliedrigkeit. Die Darstellung von Variabilität im Objektmodell erfolgt bei KobrA mittels UML-Stereotypen. Dieser Ansatz führt nach Erfahrungen des Autors zu mangelnder Übersichtlichkeit und geringer Ausdrucksfähigkeit von Objektmodellen (siehe auch 2.2.2). Außerdem erfordert der KobrA durch die Komponenten- und Framework-Orientierung eine manuelle Implementierung von Produkten, denn eine automatisierte Umsetzung der Objektmodelle zu Produkten ist wegen fehlender formaler Definition der Semantik der Variationspunkte nicht möglich. PuLSE und KobrA werden deshalb im Rahmen der vorgestellten Arbeiten nicht weiter verfolgt.

FeatuRSEB (gesprochen Featured RSEB) ist eine Weiterentwicklung des RSEB. RSEB wird durch die Integration von Konzepten der Featuremodellierung aus FODA (siehe 2.2.1) erweitert um Bezüge zu wiederverwendbaren Elementen in Modell und Implementierung. Damit wurden einige Schwächen von RSEB beseitigt, die sich bei der industriellen Erprobung in der Domäne der Telekommunikation zeigten [Jacobson et al. 1997][Griss et al. 1998]. Für FeatuRSEB liegen umfassende Informationen in publizierter Form vor, die eine Bewertung und Nutzung erlauben. Der Beitrag von FeatuRSEB zum Fortschritt besteht darin, dass durch die Zusammenführung von Domain Engineering mit objektorientierter Modellierung und Wiederverwendung die Möglichkeiten der Analyse und Beschreibung von Gemeinsamkeiten und Unterschieden um Modell- und Implementierungsmethoden ergänzt werden. Diese Zusammenführung von Methoden wird als Ausgangspunkt für die vorgestellten Arbeiten benutzt. Wesentliche Eigenschaften sollen deshalb hier genannt und weiter unten bewertet werden. Wichtige Punkte sind:

- Anforderungen an Produkte und Produktlinie werden durch Use-Case-Modelle der UML beschrieben
- Die Strukturierung der Anforderungen nach Gemeinsamkeit und Variabilität erfolgt durch ein Featuremodell
- Architektur und Komponenten werden mittels objektorientierter Analyse- und Entwurfsmethoden entwickelt. Die Ergebnisse werden mittels Objektmodell beschrieben.

- Die Implementierung von Produkten und Produktlinie erfolgt mit objektorientierten Mitteln.

FeatuRSEB ist wegen dieser Eigenschaften für Produktlinien in solchen Domänen geeignet, die mittels Use Cases beschrieben und in Objekte gegliedert werden können. Da dies in sehr vielen Domänen möglich ist, kann FeatuRSEB als nicht domänenspezifische, allgemein anwendbare Methode bezeichnet werden.

Die Methoden zur Produktlinienentwicklung von Bosch [Bosch 2000] unterstützen vor allem die Architekturentwicklung. Die Beiträge zur Architekturentwicklung werden weitgehend als Basis der hier entwickelten Methoden angewendet. Die Hinweise zu evolutionärer Prozessgestaltung haben die Gestaltung der Entwicklungsprozesse (siehe 3.3) beeinflusst. Defizite bestehen im Fehlen von Unterstützung für die Automatisierung der Implementierung und einem Mangel an Unterstützung der Separation of Concerns (siehe 2.3) als Basis für die Komposition.

ESAPS und CAFÉ sind zwei aufeinander aufbauende Europäische Verbundprojekte aus dem *Eureka Sigma!2023*-Programm der EU. Zu beiden Projekten liegt eine Reihe von Veröffentlichungen vor [ESAPS][CAFE]. Die Veröffentlichungen sind sehr allgemein gehalten. Sie erlauben keine unmittelbare Anwendung der Methoden in eigenen Projekten; enthaltene Bewertungen der Ergebnisse lassen sich nicht im Detail nachvollziehen. Zahlreiche weitere Ergebnisse sind nur für die Projektpartner freigegeben und nicht öffentlich zugänglich; sie liegen dem Autor nicht vor. Deshalb können die in diesen Projekten entwickelten Methoden nur an wenigen Stellen in die Bewertung einbezogen und weiter verwendet werden.

Die als Bestandteile der Methodik ALEXANDRIA entwickelten Methoden im Bereich Anforderungsanalyse und Featuremodellierung wurden teilweise in Zusammenarbeit mit einem Projektbeteiligten von ESAPS und CAFÉ erarbeitet. Auf diesem Wege konnten einige wenige Ergebnisse der Projekte hier genutzt werden. Dazu gehören das Prozessmodell (Abb. 6 und Abb. 7, S. 21) und Bewertungen von hier behandelten Methoden aus der Sicht von Großunternehmen. Eigene Arbeiten zum Datenmodell der Repositories für Anforderungen (siehe 4.1), zum Scoping (siehe 4.6) und zu Werkzeugen der Featuremodellierung (siehe 4.7) konnten in das Projekt CAFÉ eingebracht werden.

**Unterscheidung zwischen Produktlinien- und Produkt-Engineering**
In FeatuRSEB wird ein Vorgehensmodell angewendet, das durch eine Unterscheidung zwischen der Entwicklung gemeinsamer Teile (Produktlinien-Engineering) und der Entwicklung einzelner Produkte (Produkt-Engineering) gekennzeichnet ist, wie es in Abb. 5 in Anlehnung an [Jacobson et al. 1997] dargestellt ist. Dabei werden drei Typen von Entwicklungsprozessen unterschieden: das Produktlinien-Engineering besteht aus dem Anwendungsfamilien-Engineering, bei dem Anforderungen an Produktlinie und Architektur behandelt werden, und dem Komponentensystem-Engineering, bei dem die Komponenten der Produktlinie entwickelt und gewartet werden; daneben gibt es das Produkt-Engineering, bei dem konkrete Produkte auf der Basis von Komponenten und Prozessen des Produktlinien-Engineering entwickelt werden.

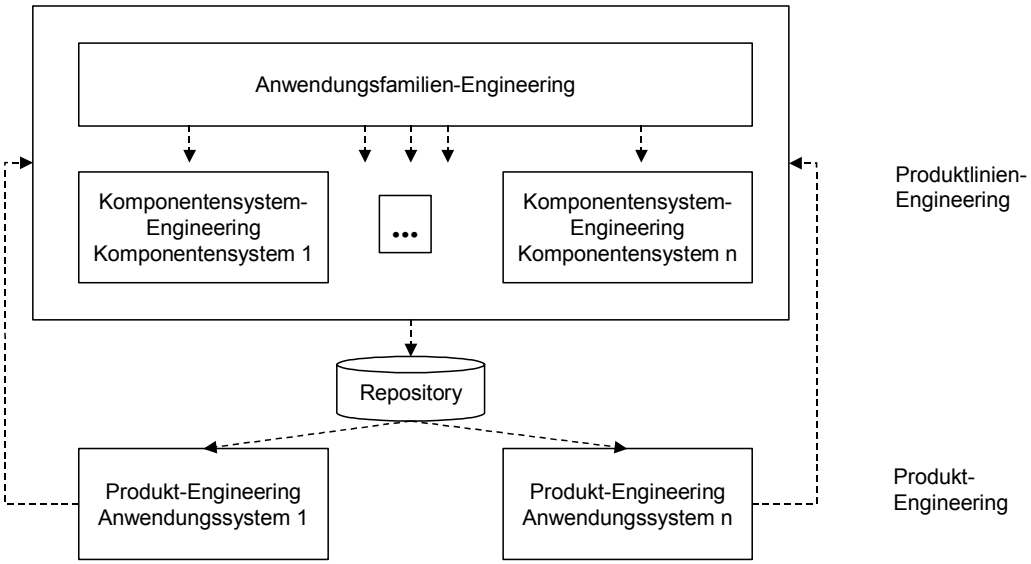



*Abb. 5: Vorgehensmodell in FeatuRSEB*

Die Aufteilung in Anwendungsfamilien-Engineering und Komponentensystem-Engineering erfolgt, weil die Architektur einer Produktlinie in FeatuRSEB aus einzelnen Subsystemen besteht, die als Komponentensysteme bezeichnet werden. Das Anwendungsfamilien-Engineering beinhaltet die Analyse der Anforderungen an die Produktlinie und die Festlegung der Gliederung und der Schnittstellen der Komponentensysteme. Das Komponentensystem-Engineering umfasst Entwurf und Implementierung der verschiedenen Komponentensysteme.

Abb. 6 zeigt zum Vergleich das Vorgehensmodell von ESAPS, das ähnliche Aktivitäten enthält, jedoch in anderer Darstellung. Auch hier wird eine Trennung in Produktlinien-Engineering (obere Reihe) und Produkt-Engineering (untere Reihe) vorgenommen. Die Aktivitäten Anforderungsanalyse, Modellierung (Entwurf) sowie Implementierung sind zwar auch in FeatuRSEB enthalten, gegenüber Abb. 5 werden sie aber hier separat dargestellt. Bei beiden Vorgehensmodellen werden sie in iterativ-inkrementeller Folge ausgeführt. Die sechs Hauptaktivitäten in Abb. 6 [ESAPS] werden jetzt kurz erörtert, weil sie in der Methodik ALEXANDRIA ebenfalls enthalten sind.

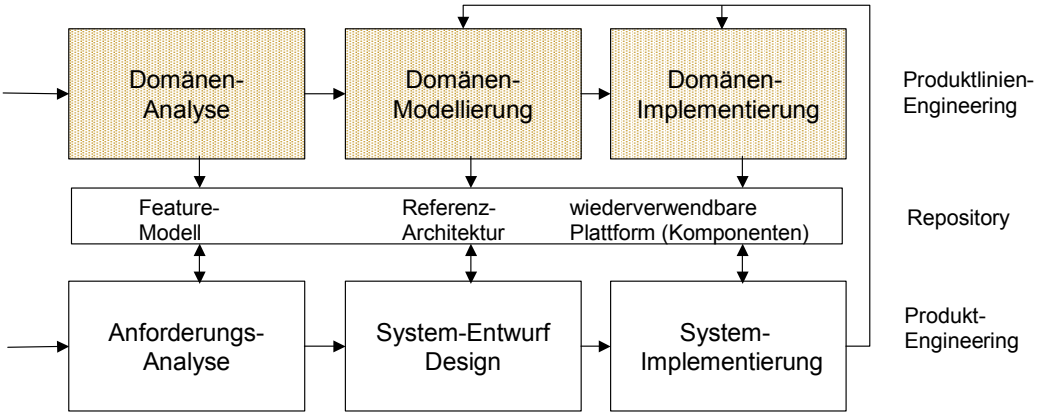



*Abb. 6: Prozessmodell in ESAPS*

Zur *Analyse und Beschreibung von Anforderungen im Produktlinien-Engineering* gehören teilweise die gleichen Techniken und Tätigkeiten wie bei der konventionellen Anforderungsanalyse, wie das Erfassen und Erfragen der Anforderungen. Über die konventionellen Anforderungsanalyse hinaus sind jedoch Unterschiede zwischen gemeinsamen (für jedes Produkt der Produktlinie zutreffenden) und variablen (nur für einige Produkte zutreffenden) Merkmalen zu erfassen. Dazu wird in ESAPS genauso wie in FeatuRSEB das Featuremodell aus FODA benutzt, auf das in Abschnitt 2.2.1 genauer eingegangen wird. Es erlaubt einen Überblick über Gemeinsamkeiten und Unterschiede zwischen Produkten der Produktlinie, wobei auf Details von Anforderungen, Architektur und Implementierung verwiesen wird. Das Featuremodell wird zur Strukturierung der Anforderungen, zur Abschätzung des Aufwands für Produkte, zur Konfiguration von Produkten sowie zur Weiterentwicklung der Produktlinie benutzt.

*Analyse und Entwurf der Produktlinie* beginnen in FeatuRSEB mit der Ableitung der Architektur der Produktlinie aus dem Use-Case-Modell. Diese Aktivität gehört zum Anwendungsfamilien-Engineering (siehe Abb. 5). Die Architektur besteht aus Schichten, die aus Komponentensystemen bestehen. Anhand der Definition der externen Schnittstellen entwirft das Komponentensystem-Engineering Objektmodelle mit Klassen und Interaktionen. Auf die geforderte Variabilität ist besonderes Augenmerk zu legen, für deren Umsetzung werden Standardlösungen wie Design Patterns empfohlen. Klassen der Objektmodelle werden mittels trace-Beziehungen mit den dazugehörigen Use Cases verknüpft.

Die *Implementierung der Komponentensysteme der Produktlinie* erfolgt bei FeatuRSEB durch ein Framework in einer objektorientierten Programmiersprache. Gegenüber konventioneller Softwareentwicklung liegt hierbei ein Schwerpunkt in der Umsetzung der geforderten Variabilität. Nach erfolgreichem Test werden Objektmodell und Quellcode im Repository bereitgestellt. Bei ESAPS und CAFE sind neben Frameworks auch andere Arten der Implementierung erwähnt, wie Komponenten-Architekturen (siehe 2.2.3).

Die *Analyse und Beschreibung von Anforderungen* des Kunden *im Produkt-Engineering* wird anhand der bereits im Anforderungs- und Featuremodell der Produktlinie erfassten Anforderungen durchgeführt. Der Kunde kann aus den enthaltenen Features auswählen, daraufhin wird die dazugehörige Anforderungsbeschreibung des Produkts zusammengestellt. Bei der Zusammenstellung sind die im Featuremodell hinterlegten Regeln zu berücksichtigen. Solche Regeln werden in FeatuRSEB in Form von Notizen ausgedrückt. Decken die in der Produktlinie vorhandenen Anforderungen nicht alle Kunden-

anforderungen ab, gibt es mehrere Varianten, mit nicht erfüllten Anforderungen umzugehen:

- Erweiterung der Produktlinie um die neuen Anforderungen;
- individuelle Realisierung der Anforderungen für den Kunden;
- den Kunden mit Kosten- und Zeit-Argumenten zum Verzicht auf diese Anforderungen bewegen.

Bei *Analyse und Entwurf des Produkts* können die im Repository enthaltenen Verknüpfungen von Use Cases zu Komponentensystemen genutzt werden, damit anhand der Features Klassen für das Objektmodell des Produkts ausgewählt werden können. Dies ist allerdings nur dann möglich, wenn eine geeignete Zerlegung Überschneidungen verhindert. Gegebenenfalls ist eine Spezialisierung oder Parametrisierung wiederverwendeter Klassen durchzuführen. Für individuelle Anforderungen sind neue Klassen zu entwerfen und in das Objektmodell zu integrieren.

*Implementierung und Inbetriebnahme des Produkts* erfolgen unter Nutzung der Implementierung der betreffenden Komponentensysteme oder Frameworks der Produktlinie. Individuell entworfene Klassen werden hier integriert. Test, Auslieferung und Inbetriebnahme des Produkts unterscheiden sich nicht wesentlich von konventionell entwickelten Software-Systemen.

Wesentliche Inhalte der einzelnen Aktivitäten und die dazugehörigen Notationen werden in den folgenden Abschnitten 2.2 und 2.4 genauer untersucht, soweit sie Beiträge zu den hier entwickelten Arbeiten leisten.

Abb. 7 zeigt die Weiterentwicklung des Prozessmodells im Projekt CAFÉ, die zusätzlich zu den genannten auch Aktivitäten im Umfeld der eigentlichen Softwareentwicklung enthält [CAFE].

Als Bewertung der in diesem Abschnitt vorgestellten Produktlinien-Methoden wird eingeschätzt, dass sie wichtige Beiträge zur Gliederung der Aktivitäten sowie zum Umfeld der eigentlichen Entwicklung der Software leisten, weil sie Wiederverwendungstechniken mit Domänenanalyse-Techniken verknüpfen und so Wiederverwendung auf Anforderungs-Ebene ermöglichen. Deshalb werden diese Grundideen sowie das Prozessmodell von FeatuRSEB in der Produktlinien-Methodik ALEXANDRIA angewendet (siehe Kap. 3).

Als Techniken für eine Implementierung von Variabilität werden von den genannten Methoden lediglich die Auslagerung variabler Anteile in Komponenten sowie Variabilität mittels Frameworks mit Vererbung und Design Patterns genutzt. Beide Techniken weisen erhebliche Nachteile auf:

Sollen variable Teile in Komponenten ausgelagert werden, müssen diese Komponenten überschneidungsfrei und kombinierbar sein (siehe 2.2.3). Die Aufgabe der Zerlegung in überschneidungsfreie Komponenten wird jedoch von diesen Methoden nicht gelöst. Ein Ausweg ist die Schaffung sehr feingranularer Komponenten in entsprechend höherer Anzahl, mit dem Effekt eines hohen zusätzlichen Aufwands für Verwaltung und Integration.

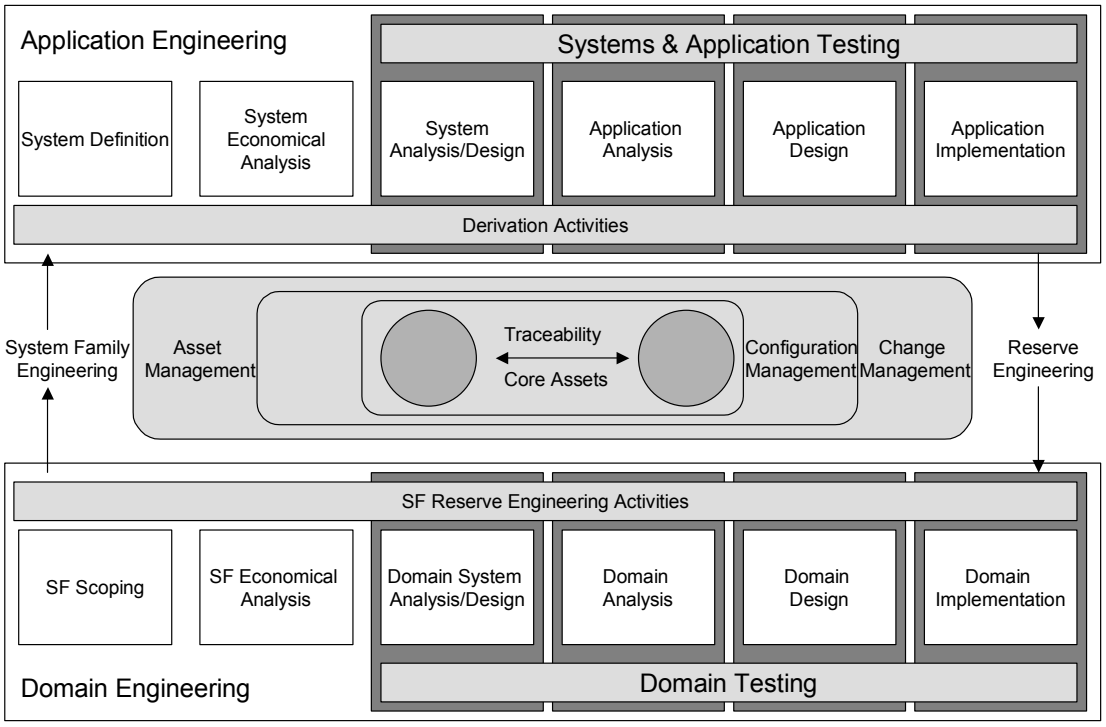



*Abb. 7: CAFÉ-Prozessmodell*

Die Implementierung von Variabilität mittels Vererbung und Design Patterns führt zu einer Vielzahl zusätzlicher Elemente und einem starken Zuwachs der Komplexität. Dadurch ist die Skalierbarkeit der betreffenden Methoden begrenzt, wie die Untersuchungen von FeatuRSEB (siehe 10.3.1) gezeigt haben.

## 2.2 Variabilität in Modell und Implementierung

Wird eine Software so erweitert, dass sie in flexibler Weise anstelle genau einer Aufgabe eine gewisse Menge verwandter Aufgaben erfüllen kann oder an verschiedene Bedingungen angepasst werden kann, so bezeichnet man diese Eigenschaft als *Variabilität*. Die Eigenschaft ist beispielsweise bei Standardsoftware für einen Massenmarkt dazu nötig, dass Kunden solche Software an ihre Bedürfnisse anpassen können. Variabilität kann dazu verwendet werden, zukünftig benötigte Funktionalität als Vorfertigung zu implementieren und auf diese Weise zur Flexibilität von Softwaresystemen beizutragen.

Können wiederverwendbare Komponenten *mit* Variabilität zur Erfüllung geänderter Anforderungen genutzt werden, kann eine größere Anzahl verschiedener Anforderungen kurzfristig erfüllt werden als durch Komponenten ohne Variabilität. Die Wirksamkeit von Vorfertigung und Komposition für die Flexibilität steigt dadurch.

Den entscheidenden Faktor für einen erfolgreichen Einsatz der Variabilität stellt das Verhältnis von notwendigem Aufwand zum Nutzen dar. Architekturen und Modelle haben großen Einfluss auf den Aufwand, wenn sie helfen können, die Komplexität von Lösungen gering zu halten. Wegen des Aufwands kommt außerdem der Planung von Variabilität bei der Vorfertigung von Komponenten große Bedeutung zu. Ohne eine Planung und ohne Entscheidung für die richtige Entwicklungsrichtung könnte der hohe Aufwand zur Schaffung von Variabilität – erhöhter Entwicklungsaufwand und komplizierte Architektur – später nicht ausreichend genutzt werden. Geeignete Techniken der

Anforderungsanalyse müssen zur Definition der Entwicklungsziele genutzt werden, damit bei der Gestaltung von Variabilität eine Anwendungsorientierung erreicht wird.

### 2.2.1 Variabilität von Anforderungen in der Domänenanalyse und im Featuremodell

Methoden und Beschreibungsmittel zur Analyse von Anforderungen an Problemlösungen in einem Anwendungsgebiet, einer sog. Domäne, werden von der *Domänenanalyse* bereitgestellt. Von besonderem Interesse sind für Produktlinien die Möglichkeiten zur Beschreibung von Gemeinsamkeiten und Variabilität von Softwaresystemen für diese Domäne. Bereits die frühen Methoden der Domänenanalyse wie Draco [Neighbors 1980] ermöglichen eine solche Analyse und Beschreibung von Gemeinsamkeiten und Unterschieden von Anforderungen. Außerdem strukturieren sie das Wissen über eine Domäne. Häufig werden dazu strukturierte Beschreibungen mit natürlichsprachlichen Elementen oder Graphen mit Begriffsknoten genutzt, wie z.B. bei den Wissensmodellen in KADS [Wielinga et al. 1992]. KARL [Fensel 1995] erweitert diese Modelle zwecks Auswertbarkeit um formale Elemente. Den verschiedenen Methoden ist gemeinsam, dass sie mit einer Beschreibung in einem Domänenmodell [Czarnecki et al. 2000] Hinweise auf zukünftige Aufgabenstellungen innerhalb einer Domäne liefern. Werden sie für die Planung der Wiederverwendung eingesetzt, können sie wesentliche Mängel ausgleichen, die den Erfolg von Techniken der Wiederverwendung vorher begrenzt haben.

Die erfolgreiche und umfassende Nutzung der Domänenanalyse-Methoden bei der Wiederverwendung ist in der Vergangenheit wegen des sehr hohen Aufwands gescheitert. Der Aufwand für die Domänenanalyse ist in den seltensten Fällen wirtschaftlich zu rechtfertigen, denn ein Unternehmen will meist nur einige Anwendungen einer Domäne realisieren und steht durch die Konkurrenz zu anderen Anbietern unter Zeit- und Kostendruck. Zur Verringerung dieses Nachteils ergänzen neuere Methoden wie Organization Domain Modeling ODM [Simos et al. 1996] die Analyse und die Wissensbeschreibung um ein mehrstufiges *Scoping*. Scoping bezeichnet Entscheidungen zur Abgrenzung eines Verantwortungsbereichs. Solche Entscheidungen beziehen sich bei der Domänenanalyse darauf, welche Anforderungen weiter betrachtet werden, indem sie verfeinert, modelliert oder in einem Softwaresystem implementiert werden.

Für die methodische Entwicklung und die Werkzeugunterstützung der Produktlinien-Entwicklung werden klar definierte und einfach strukturierte Modelle benötigt. Die Methode Feature-Oriented Domain Analysis FODA [Kang et al.1990] führt *Featuremodelle* zur Beschreibung von gemeinsamen und spezifischen Anforderungen ein. Features sind Merkmale, die aus Kundensicht Systeme einer Domäne voneinander unterscheiden. Featuremodelle sind Domänenmodelle, die Anforderungen strukturieren, durch Features zusammenfassen und in Beziehung zueinander setzen. Eine bestimmte Auswahl aus der Menge der Features beschreibt dann ein bestimmtes Produkt der Domäne. Featuremodelle haben sich als gut geeignet für die Beschreibung von Anforderungen an eine Produktlinie erwiesen. Sie werden in FeatuRSEB (siehe 2.2.1) zur Beschreibung von Variabilität von Anforderungen genutzt. Czarnecki und Eisenecker entwickelten das Featuremodell in nützlicher Weise weiter [Czarnecki et al. 2000]. Featuremodelle nach FeatuRSEB und [Czarnecki et al. 2000] werden in dieser Arbeit weiterentwickelt und als zentrale Notation für Features, Anforderungen und Entwurfsentscheidungen verwendet.

**Definitionen und graphische Darstellung von Features und ihren Beziehungen**
Für Features liegen verschiedene Definitionen vor; ein Feature ist danach „ a property of a domain concept, which is relevant to some domain stakeholders and is used to discriminate between concept instances“ ([Czarnecki et al. 2000], S. 744). Übertragen auf die Produktlinien-Entwicklung bezieht sich das *domain concept* auf die Produktlinie, *concept instances* bezeichnet Produkte aus der Produktlinie. *Features* sind danach alle Eigenschaften, nach denen aus Kundensicht Produkte unterschieden werden können. Weitere Interessenten (engl.: stakeholder) wie Entwickler haben andere Sichtweisen, die sich meist nicht eindeutig trennen lassen. Entsprechend unscharf bleiben die Definitionen für Features. Beim Ansatz FORM [Kang et al. 1998] wird eine Unterscheidung in Sichten vorgenommen, die jedoch keine eindeutige Zuordnung definiert und deshalb hier nicht verfolgt wird.

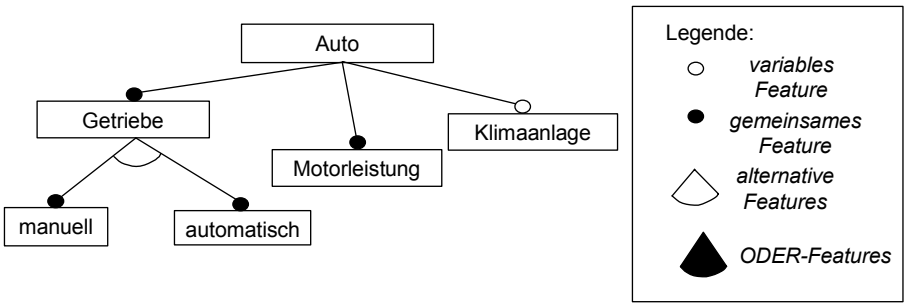



*Abb. 8: Beispiel eines Featuremodells*

Features stellen Knoten in einem Featurediagramm dar, deren Beziehungen durch Kanten dargestellt werden. Nach den meisten Definitionen stehen Features in hierarchischer Beziehung zueinander, die untergeordnete und übergeordnete Features verbindet. Die Hierarchiebeziehungen müssen widerspruchsfrei und frei von Zyklen sein. Abb. 8 zeigt ein Beispiel mit der Darstellung nach [Czarnecki et al. 2000]. Der oberste Knoten der Hierarchie kennzeichnet den sog. Konzeptknoten, der für die Produktlinie steht. Die Hierarchiebeziehung ist eine gerichtete Kante mit ausgezogener Linie, ein ausgefüllter Kreis weist zu einem gemeinsamen Feature, das zum gemeinsamen Kern der Produktlinie gehört. Leere Kreise, Kreisbögen zur Gruppierung für alternative und ODER-Features kennzeichnen variable Features, die individuelle Anforderungen für einzelne Produkte repräsentieren. Diese Unterscheidung drückt Regeln aus, nach denen die Festlegung von Features für ein Produkt der Produktlinie erfolgen kann:

- Gemeinsame Features, die nicht Teil einer Gruppierung sind, müssen immer in einem Produkt enthalten sein, wenn das übergeordnete Feature für ein Produkt festgelegt ist.
- Variable Features können ausgewählt werden, wenn das übergeordnete Feature für ein Produkt festgelegt ist.
- Aus einer Gruppe alternativer Features muss genau ein Feature ausgewählt werden, sobald das übergeordnete Feature für ein Produkt festgelegt ist.
- Aus einer Gruppe von ODER-Features muss mindestens ein Feature ausgewählt werden, sobald das übergeordnete Feature für ein Produkt festgelegt ist; es können jedoch auch mehrere Features ausgewählt werden.

Durch Kombinationsmöglichkeiten von als gemeinsam gekennzeichneten Features mit variablen Features in verschiedenen, auch gemischten Gruppierungen werden in [Czarnecki et al. 2000] eine Reihe von Konfigurationen beschrieben. Dabei kommt es jedoch bei verschiedenen Kombinationen von Gruppierungen zu Mehrdeutigkeiten der graphischen Darstellung, wie in [Riebisch et al. 2002a] untersucht wird. Gegenüber [Czarnecki et al. 2000] ist eine Änderung der graphischen Syntax erforderlich, damit eindeutige Darstellungen entstehen.

Ein anderer Ansatz zur graphischen Darstellung von Featuremodellen verwendet graphische Elemente des UML-Klassendiagramms zur Darstellung von Features und ihren Beziehungen [Clauß 2001]. Symbole für Klassen, Aggregations- und Vererbungsbeziehungen werden dabei mit einer neuen Semantik benutzt. Dieser Ansatz birgt die Gefahr, dass die Darstellung nicht oder falsch verstanden wird, zumal Software-Designer als potentielle Nutzer die UML-Elemente in ihrer eigentlichen Semantik bei der täglichen Arbeit anwenden. Der Ansatz wird hier nicht weiter verfolgt, da die gewählte Darstellung die wichtige Funktion eines Kommunikationsmittels nicht ausreichend erfüllen kann, auch wenn die Nutzbarkeit vorhandener graphischer UML-Editoren einen Vorteil verspricht.

Neben hierarchischen Beziehungen bestehen zwischen Features meist weitere Beziehungen zur Beschreibung von Abhängigkeiten, die Auswahlmöglichkeiten von Features einschränken. Im Featuremodell werden diese Beziehungen durch eine gestrichelte Linie dargestellt, ein Pfeil zeigt die Richtung der Abhängigkeit an (Abb. 9). Die Art der Abhängigkeit wird durch einen Stereotyp an der Linie angegeben: << require>> oder <<exclude>>. Beziehungen des Typs require von einem Feature A zu einem Feature B geben an, dass das Feature B immer ausgewählt werden muss, wenn Feature A ausgewählt wurde. Beziehungen des Typs exclude zwischen einem Feature C und einem Feature D geben an, dass eine Auswahl von C die Auswahl von D ausschließt. Diese Beziehungen sind immer unidirektional.

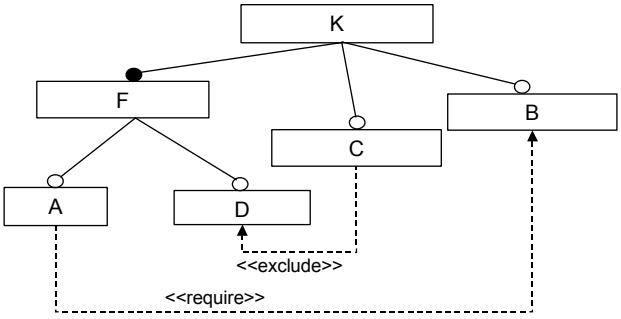



*Abb. 9: Featuremodell mit Abhängigkeiten*

Abhängigkeitsbeziehungen müssen widerspruchsfrei sein. Dazu gehört, dass gemeinsame Features nicht in Abhängigkeitsbeziehung zu variablen Features stehen dürfen und dass ein Feature keine require-Beziehung zu verschiedenen Features haben darf, die nicht gleichzeitig ausgewählt werden können – etwa weil es sich um Alternativen handelt oder wegen einer exclude-Abhängigkeit zwischen ihnen.

Bei der Anwendung der Featuremodelle zeigt sich, dass die Definition von Syntax und Semantik der Featuremodelle zu unscharf ist. Dieser Mangel macht sich bei der Definition der hierarchischen Gliederung von Features in Featuremodellen bemerkbar. Bei der praktischen Anwendung fällt dem Entwickler häufig die Entscheidung schwer, ob es

sich bei der Beziehung zwischen zwei Features um (hierarchische) Unterordnung oder um eine require-Abhängigkeit handelt. Bei genauerer Untersuchung wird klar, dass der Charakter der Hierarchiebeziehung noch nicht genau genug definiert ist, damit eindeutige Darstellungen von Featuremodellen erreichbar sind.

Außerdem wird eine genauere Definition für Features und Feature-Typen dazu benötigt, dass Featuremodelle methodenbasiert ausgewertet werden können. Weitere Informationen beispielweise über Bezüge zu anderen Modellen oder über Bindungszeitpunkte von Variabilität werden in den Featuremodellen nach [Czarnecki et al. 2000] als Notiz in Form natürlichsprachlicher Texte ergänzt. Für eine Nutzung solcher Informationen durch Methoden und Werkzeuge sind diese zu strukturieren und mittels formaler Syntax und Semantik zu formulieren. Lösungen für die genannten Aufgaben werden in Kapitel 4 mit einer präzisierten Definition für Featuremodelle angegeben.

**Wechselwirkungen zwischen Features**
Abhängigkeiten zwischen Features bestehen häufig darin, dass sie gleiche Bereiche des Systemverhaltens in unterschiedlicher Weise beeinflussen. Die Kombination solcher Features führt dann zu Wechselwirkungen, wobei häufig ein aus Sicht der beteiligten Features unerwünschtes Systemverhalten auftritt. Diese gegenseitige Beeinflussung wird als *Feature Interaction* bezeichnet. Sie wurde zuerst in der Telekommunikations-Domäne untersucht. Es existieren verschiedene Ansätze mit den Zielen, während des Entwurfs Interaktionen zu erkennen, aufzulösen und letztendlich zu vermeiden. Eine Einführung zu diesem Thema ist in [Zave 1999] enthalten; eine Übersicht und Bewertung der Ansätze wird von [Calder et al. 2002] gegeben. Die Nutzung solcher Ansätze schafft eine wichtige Voraussetzung für eine Feature-orientierte Zerlegung und Komposition nach den in dieser Arbeit vorgestellten Methoden. Diese Ansätze werden deshalb unverändert in die ALEXANDRIA-Methodik einbezogen.

### 2.2.2 Variabilität in objektorientierten Modellen

Im Entwurf muss unterschieden werden zwischen Elementen, die in allen Produkten benötigt werden und solchen für nur einzelne Produkte einer Produktlinie. Diese Unterscheidung im Featuremodell wird durch die Kennzeichnung gemeinsamer und variabler Anforderungen und deren Bezug zu den Elementen der Implementierung veranschaulicht.

FeatuRSEB basiert auf objektorientierten Methoden für Modellierung und Implementierung (siehe 2.2.1). Objektorientierte Methoden und Modelle sind jedoch nur für die Entwicklung und Beschreibung genau einer Lösung verwendbar. Das zeigt sich darin, dass Varianten in Modellen nicht unmittelbar beschreibbar sind und dass die Zerlegung nicht auf die Schaffung flexibel kombinierbarer Komponenten ausgerichtet ist, sondern auf eine Schaffung von Komponenten gemäß der Gliederung der Problemdomäne. Objektorientierte Modelle müssen für eine Integration in FeatuRSEB um eine Kennzeichnung variabler Teile erweitert werden, damit zwischen gemeinsamen und variablen Modellelementen unterschieden werden kann. Außerdem müssen die objektorientierten Methoden bei der Implementierung eine solche Zerlegung der Lösung unterstützen, die frei von Überschneidungen ist; dieser Punkt wird in Abschnitt 2.3 untersucht.

Für die Erweiterung objektorientierter Modelle um Variabilität schlägt die Methode FeatuRSEB die Einführung von Variationspunkten vor (Abb. 10, [Böllert 2002]). Die grau hinterlegten Komponenten C, D und E realisieren variable Anforderungen, Komponente A eine gemeinsame. Mit den Mitteln von FeatuRSEB können nicht alle Elemente in gemeinsame und variable zerlegt werden, so dass im Modell gemischte Ele-

mente enthalten sind. B ist eine gemischte Komponente mit b als Variationspunkt, der als gefüllter Kreis dargestellt wird. Hier kann in Abhängigkeit von den variablen Anforderungen 2 und 3 die Komponente D oder C eingefügt werden.

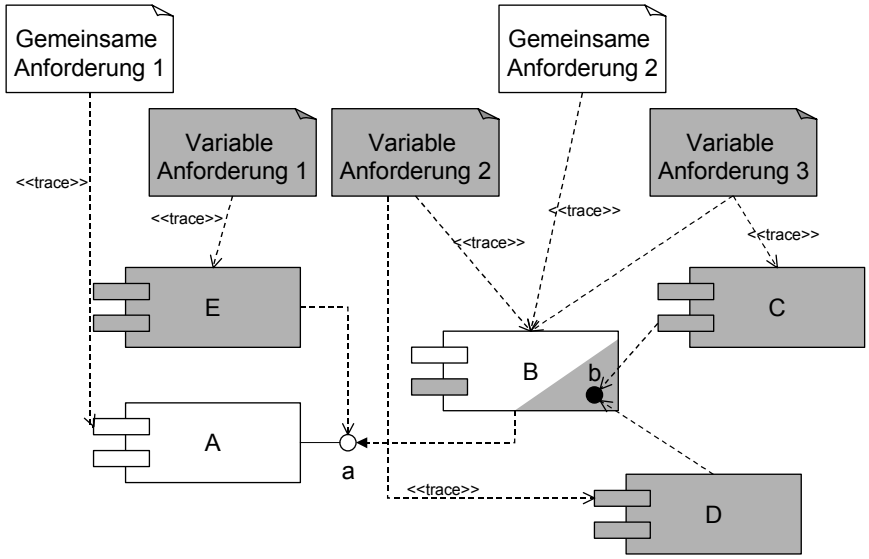



*Abb. 10: Darstellung von Variabilität mittels Variationspunkten*

Diese Darstellung eignet sich zur Veranschaulichung von Variabilität, wie sie in einer Framework-Architektur umgesetzt werden kann. Der Bezug eines Variationspunkts zu Features wird dabei nur grafisch dargestellt. Genauere Informationen zu Ablauf und Ergebnis einer Kombination oder Komposition sind im Modell nicht angebbar. Damit kann das Modell nicht zur Steuerung der Konfigurierung eines Produkts einer Produktlinie genutzt werden. Es muss nach einer Darstellung mit der Möglichkeit zusätzlicher Attribute gesucht werden.

Die UML als Standard-Modellierungssprache bietet als Möglichkeit der Erweiterung die Spezifizierung von Modellelementen mittels Stereotypen, Tagged Values und Constraints an [UML 2001]. Verschiedene Ansätze zur Erweiterung der UML nutzen diese Ergänzungen zur Kennzeichnung, wie auch der eigene Ansatz in [Riebisch et al. 2000]. Ein Stereotyp <<variant>> kennzeichnet ein variables Modellelement, im Beispiel in Abb. 11 eine Komponente. Der Tagged Value {feature= PIN Check} dieses Beispiels stellt durch ein hinzugefügtes Paar aus Schlüssel und Wert die Referenz zum dazugehörigen variablen Feature im Featuremodell her - hier zu einem Feature namens PIN Check.

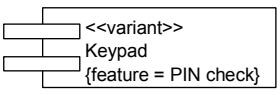



*Abb. 11: Beispiel einer variablen Komponente mit Stereotyp und Tagged Value*

Die Erfahrungen bei der Nutzung dieser Erweiterung zeigen Mängel bezüglich der Übersichtlichkeit der Modelle. Insbesondere bei umfangreichen Modellen sind die Zuordnungen zu Features für den Bearbeiter schwer erfassbar. Außerdem ist keine Kombination des Stereotyps <<variant>> mit den weiteren Stereotypen möglich, die

während der Verfeinerung der UML- Modelle üblicherweise benutzt werden. Hier liegt eine der Beschränkungen der Methode KobrA (siehe 2.1.5) , die Stereotype zur Kennzeichnung nutzt. Referenzen variabler Modellelemente zu Features werden dort in Tabellen dargestellt, die als Entscheidungsmodell bezeichnet werden.

Als ein weiterer Ansatz zur Unterscheidung von Entwurfselementen wurde von einigen Werkzeugherstellern die Hervorhebung mittels Farbattributen untersucht, wie beispielweise für die Unterscheidung von Aspekten in [OTW24]. Eine solche Kennzeichnung von Modellelementen führt zu guter Übersichtlichkeit, wenn nur wenige Unterscheidungen vorzunehmen sind. Durch die geringe Anzahl deutlich unterscheidbarer Farben stößt dieser Ansatz bei komplexen Modellen schnell an seine Grenzen; für die Unterscheidung zwischen Features ist er aus diesem Grund ungeeignet.

Zusammenfassend wird festgestellt, daß FeatuRSEB gegenüber den anderen untersuchten Produktlinien-Methoden eine Komposition von Softwaresystemen durch die Kopplung zwischen Featuremodellierung und objektorientierten Entwurfsmethoden unterstützt. Die erwähnten Mehrdeutigkeiten in Featuremodellen stehen einer weitergehenden Werkzeugunterstützung noch entgegen. Außerdem ist die Entwicklung überschneidungsfreier Komponenten in FeatuRSEB nicht gelöst (weitere Ansätze dazu werden in Abschnitt 2.3 untersucht). Die Darstellung mit Variabilitätspunkten ist zu ungenau für eine werkzeuggestützte Komposition.

Diese Punkte – Komposition, Mehrdeutigkeiten, Werkzeugunterstützung, überschneidungsfreie Komponenten – werden deshalb im Rahmen der ALEXANDRIA-Methodik behandelt. Die präzisierte Definition und die Erweiterung des Featuremodells um auswertbare Abhängigkeiten (siehe 4.1) und das Konzept der explizit beschriebenen Traceability-Links (siehe 4.2) bieten Steuerungsmöglichkeiten für die Komposition, die im Hypermodule automatisiert ausführbar sind (siehe 5.3). Überschneidungsfreie Komponenten sind durch eine Zerlegung nach dem Hyperspace-Ansatz (siehe 2.3.4) erreichbar; variable Modellelemente werden durch Kapselung in Hyperslice-Komponenten (siehe 5.2.2) zusammengefasst und beschrieben.

### 2.2.3 Variabilität in Software-Architekturen und Quellcode

Variabilität von Software soll zu Lösungen führen, die schnell und mit geringem Aufwand eine Veränderung von Eigenschaften zulassen – meist zwischen einer bestimmten Anzahl von Varianten. Varianten können sich beispielweise auf Unterschiede im Verhalten oder in den verarbeiteten Daten beziehen. Variabilität bezieht sich auf vorbereitete Ansatzstellen, an denen das Verhalten der Software geändert werden kann.

Dabei gibt es verschiedene Möglichkeiten für den Zeitpunkt der Änderung und damit der Entscheidung für eine Variante. In der Literatur [Bosch 2000][PLP-SEI] wird der Akt der Entscheidung auch als Bindung der Variabilität und der Zeitpunkt als *Bindungszeitpunkt* bezeichnet. Andere Begriffe hierfür sind Konkretisierung oder Spezialisierung einer Produktlinie. Aus der Vielzahl der verwendeten Bezeichnungen für Bindungszeitpunkte ist hier vor allem eine Unterscheidung zwischen frühem und spätem Bindungszeitpunkt wichtig.

Ein *früher Bindungszeitpunkt* der Variabilität liegt vor, wenn für variable Teile vor der Lauffähigkeit eines Systems entschieden wird, welche Variante wirksam wird. Das entstehende Produkt enthält dann zur Laufzeit keine Variabilität mehr. Frühe Bindungszeitpunkte werden vor allem genutzt, wenn schlanke Produkte benötigt werden, etwa wegen beschränkter Ressourcen. Dieser Bindungszeitpunkt liegt auch vor, wenn eine spätere Entscheidung für andere Varianten unerwünscht ist, beispielweise weil ein

Hersteller eine Unterscheidung der Varianten in preisgünstige Einsteiger- und hochwertige Profi-Produkte vornimmt. In der Einsteiger-Variante soll keine zusätzliche, deaktivierte Funktionalität enthalten sein, die eventuell von Fachleuten aktiviert werden könnte, was den Absatz der hochwertigen Produkte behindern würde.

Als *später Bindungszeitpunkt* wird eine Entscheidung für Varianten nach der Übersetzung in ein lauffähiges System bezeichnet. Das lauffähige System enthält dann noch Variabilität. Eine Auswahl zwischen Varianten ist beispielweise während der Installation dadurch möglich, dass Komponenten ausgetauscht oder durch Parameter-Funktionen aktiviert werden. Auch die Variantenauswahl zur Laufzeit wird als später Bindungszeitpunkt bezeichnet, was zum Beispiel durch Einfügen von Plug-in-Komponenten, durch Parametrisierung eines Produkts oder durch Aktivierung von Systemfunktionen erfolgen kann. Späte Bindungszeitpunkte werden u.a. dann angewendet, wenn der Kunde oder Betreiber Anpassungsmöglichkeiten erhalten soll oder wenn ein Hersteller mit einem Produkt einen größeren Markt abdecken möchte. So enthalten fast alle Standardsoftware-Produkte Variabilität mit spätem Bindungszeitpunkt.

Für die Implementierung von Variabilität in Software existieren verschiedene Techniken, die oft verschiedene Bindungszeitpunkte unterstützen:

*Vererbung.* Ist eine Komponente als Klasse in einer objektorientierten Programmiersprache implementiert, so kann Vererbung als White-Box-Technik zur Spezialisierung genutzt werden, damit die Komponente an ihre Aufgabe angepasst wird. Der Produzent kann die Spezialisierung dadurch steuern, dass er für abstrakte Operationen eine Anpassung erzwingt oder durch als `final` definierte Operationen diese verhindert. Die erfolgreiche Nutzung von Vererbung setzt ein genaues Verständnis der Komponente voraus. Eine Erweiterung der Spezialisierung durch Vererbung erfordert tiefere Klassenhierarchien sowie mehr abstrakte Klassen und Operationen. Die Funktionalität wird dadurch über mehrere Komponenten verteilt, was zu Problemen bei Verständlichkeit und Testbarkeit führt. Dies trägt zu den Nachteilen der Vererbung bei; insbesondere werden Wartbarkeit und damit auch Flexibilität und Evolution beeinträchtigt. Die Vererbung kann in Zusammenhang mit frühem oder spätem Bindungszeitpunkt verwendet werden.

*Template-Instanzierung*. Gelegentlich müssen Komponenten bezüglich der verwendeten Typen oder logischer Bedingungen parametriert werden. Einige Programmiersprachen ermöglichen die Nutzung von Templates als Schemata mit abstrakten Code-Bestandteilen. Diese Schemata werden durch Parameter instanziert, damit konkrete, lauffähige Komponenten entstehen. Templates werden insbesondere zur Anpassung an (Daten-)Typen verwendet. Die generative Technik des Template Metaprogramming basiert darauf [Czarnecki et al. 2000]. Ihre Anwendung beeinflusst die Wartbarkeit positiv, ihr Einsatzbereich ist allerdings bezüglich Aufgaben und Programmiersprachen begrenzt. Diese Möglichkeit ist wegen der Manipulation des Quellcodes nur für einen frühen Bindungszeitpunkt – vor der Übersetzung in ausführbaren Code – geeignet.

*Generierung*. Dieser Mechanismus wurde bereits als Form der Wiederverwendung unter 2.1.4 genannt. Er basiert auf der Anwendung eines Komponenten-Generators, der aufgrund einer Spezifikation in einer domänen- oder komponentenspezifischen Sprache eine Komponente bereitstellt, die in Softwaresysteme einbezogen werden kann. Damit kann ebenfalls Variabilität in den Grenzen des Generators erreicht werden. Wartbarkeit und Flexibilität sind bei diesem Mechanismus sehr gut, allerdings ist die Entwicklung bezüglich Plattform, Leistungsfähigkeit und Support vom Generator und dessen Produzenten abhängig. Die übliche Form der Generierung ist nur für einen frühen Bindungs-

zeitpunkt geeignet, weil der Generator den Quellcode bearbeitet, bevor eine Software gestartet wird.

*Erweiterungen*. Als Erweiterungen werden Variationspunkte bezeichnet, an denen der Produzent verschiedene Varianten von Verhalten bereitstellt. Für diese Art der Variabilität in Software existieren eine ganze Reihe von Lösungen. Sie reichen von einfachen Fallunterscheidungen anhand von Parametern über spezielle Architekturmittel, zusätzlichen Code, Vererbung oder zusätzliche Schnittstellen bis zur Abstraktion der Steuerung von Verhalten durch sogenannte Reflexions [Challa 1998]. Allen Erweiterungs-Maßnahmen ist gemeinsam, dass sie zusätzlichen Quellcode zur „Umschaltung" erfordern, der keinen direkten Beitrag zu Funktionalität einer Lösung leistet. Es ist zusätzlicher Aufwand notwendig, die Lösung wird komplizierter. Die Wartbarkeit verringert sich dadurch; die Wahrscheinlichkeit von Folgefehlern durch mangelnde Verständlichkeit steigt. Diese Möglichkeit erlaubt einen späten Bindungszeitpunkt, könnte aber auch für einen frühen Bindungszeitpunkt genutzt werden. Für einige dieser Erweiterungs-Maßnahmen existieren Prinziplösungen, die in Form von Design Patterns beschrieben sind. Es gibt jedoch große Unterschiede zwischen den Auswirkungen verschiedener Design Patterns auf Qualitätsmerkmale wie Flexibilität und Wartbarkeit, wie in [Bosch 2000] untersucht wird. Eine Reihe von Patterns trägt zur Modularisierung und Entkopplung bei und verbessert dadurch die Wartbarkeit, andere Patterns sollen speziell die Variabilität erhöhen.

*Modularisierung und Konfiguration*. Unterschiedliche Teile einer Lösung – sowohl im Modell als auch in der Implementierung – werden in separate, unabhängige Moduln ausgelagert und nur die für alle Varianten gemeinsamen Teile bleiben in einem Modul zusammen. Bezogen auf Variabilität bedeutet dies, dass Variationspunkte in Komponenten-Schnittstellen verlagert werden. Eine solche Zerlegung ist jedoch oft nicht möglich, weil es häufig schwierig ist, Unabhängigkeit der Varianten zu erreichen (siehe Abschnitt 2.3). Diese Möglichkeit weist bezüglich der Wartbarkeit die besten Eigenschaften auf. Sie wird häufig für einen frühen Bindungszeitpunkt benutzt, kann aber bei zur Laufzeit ladbaren Komponenten auch einen späten Bindungszeitpunkt ermöglichen.

Wegen der Bedeutung für Produktlinien soll auf diese Technik der Schaffung von unabhängigen Komponenten etwas näher eingegangen werden. Als wichtiger Vertreter für diesen Ansatz der Implementierung von Variabilität wird das Plug-in-Konzept von Eclipse [Eclipse 2003] untersucht. Dieses Konzept weist zunächst alle Eigenschaften der Komponententechnologie auf (siehe 2.1.1). Die einzelnen Komponenten sind darüber hinaus durch die Bereitstellung einer Plug-in-Schnittstelle variabel kombinierbar und bauen aufeinander auf. Ein Kern mit grundlegenden Elementarfunktionen bietet die Basis für Komponenten, die gemeinsam eine Plattform für die flexible Integration von weiteren Komponenten und Werkzeugen bieten. Jede Komponente kann wiederum eigene Plug-in-Schnittstellen anbieten, die neue Kombinationsmöglichkeiten und Varianten erlauben. Es muss nicht bei der Entwicklung einer Komponente definiert werden, welche Kombinationen sie ermöglicht, sondern ihre Schnittstelle kann später durch neue Komponenten und deren Schnittstellen erweitert werden. Jede Komponentenschnittstelle wird durch ein sog. Manifest deklariert, das deren Kopplungsmöglichkeiten zu anderen Komponenten ausdrückt, indem es eine Menge sogenannter Extension Points als Ansatzstellen für Erweiterungen durch andere Komponenten und eine Menge von Erweiterungen anderer Plug-in-Schnittstellen beschreibt. Zu jedem Extension Point kann eine Schnittstelle in Form eines Application Program Interface API gehören. Jede weitere Komponente kann andere Komponenten erweitern und eigene APIs definieren. Jede Komponente umfasst selbst die Beschreibung der notwendigen Voraussetzungen

für ihren Einsatz, wodurch ein hohes Maß an Variabilität möglich ist. Bei dem hier vorgestellten Konzept von Eclipse können neue Komponenten nur vor dem Start eines Systems hinzugefügt werden, späte Bindung von Variabilität ist damit nicht erreichbar. Andere Komponentenmodelle wie COM unterstützen auch Bindung zur Laufzeit.

Fasst man die Bewertung der in diesem Abschnitt gegenübergestellten fünf Techniken für eine Umsetzung von Variabilität zusammen, so bieten Modularisierung und Konfiguration die meisten Vorteile, weil sie keine zusätzliche Komplexität erfordern. Die Schnittstellen der Komponenten müssen jedoch geeignet definiert sein, damit durch einfachen Austausch Variabilität mit geringem Aufwand erreichbar wird. Die verschiedenen Komponentenansätze wie COM, JavaBeans, Eclipse und andere unterstützen zwar prinzipiell eine solche Austauschbarkeit; es muss jedoch eine Zerlegung gefunden werden, die Überschneidungen vermeidet. Gelingt eine überschneidungsfreie Zerlegung nicht, können bei der Integration Widersprüche entstehen, wenn Software-Produkte zusammengesetzt werden sollen. Ansätze zur Zerlegung werden im Abschnitt 2.3 auf Möglichkeiten der Schaffung von Überschneidungsfreiheit untersucht. Als Bestandteil der ALEXANDRIA-Methodik wird der Hyperspace-Ansatz (siehe 2.3.4) zur Bildung überscheidungsfreier Komponenten genutzt (siehe Kap. 5). Die Schaffung von Variabilität durch Definition erweiterbarer (Plug-in-) Schnittstellen für Komponenten wird durch die Methodik nicht unmittelbar unterstützt, sie stellt jedoch eine wichtige Ergänzung dar.

## 2.3 Zerlegung zwecks Komposition und Beherrschung der Komplexität

Die Erstellung von Softwaresystemen durch Wiederverwendung flexibel kombinierbarer Komponenten ist nur erreichbar, wenn diese Komponenten so zerlegt sind, dass jede Komponente genau *eine* Anforderung implementiert. Diese Zerlegung ist deshalb von großer Bedeutung, weil Anforderungen meist über viele Teile einer Software verteilt implementiert werden, und die meisten Komponenten mehrere Anforderungen erfüllen.

Diese Zerlegung soll am Beispiel einer Komponente zur Verwaltung von Bestellungen diskutiert werden. Eine solche Komponente weist neben den funktionalen Eigenschaften auch Eigenschaften bezüglich Datenspeicherung, Robustheit und Zeitverhalten auf. Diese Komponente kann nur dann zur Schaffung eines neuen Produkts wiederverwendet werden, wenn alle Anforderungen *gleichzeitig* erfüllt werden; bei höheren Anforderungen beispielweise an das Zeitverhalten kann sie nicht verwendet werden. Wegen der Vielfalt betroffener Eigenschaften tritt der Fall der Erfüllung *aller* Anforderungen selten ein. Zwecks besserer Wiederverwendbarkeit wird angestrebt, dass eine Komponente möglichst nur von einer Anforderung beeinflusst wird. Eine solche Arte der Zerlegung von Software in Komponenten wird als *überschneidungsfreie Zerlegung* bezeichnet.

Ansätze zu Beherrschen von Komplexität durch Zerlegung von Problemlösungen sind Bestandteil aller Lösungsmethoden. Bei der Softwareentwicklung diente die Zerlegung anfangs vor allem dazu, durch Schaffung von Komponenten und durch Reduktion von Abhängigkeiten zwischen ihnen die Komplexität für den Entwickler beherrschbar zu machen [Parnas 1972]. Neben der Kapselung ist hier das Prinzip der Separation of Concerns (dt. etwa: Trennung von Belangen) von Bedeutung. Der Ausdruck geht auf Dijkstra zurück, der die Vorteile der Entflechtung der verschiedenen Problemlösungen für Verständlichkeit und Weiterentwicklung hervorhebt [Dijkstra76]. Die Zerlegung soll hier vor allem das Verstehen der Lösung vereinfachen. Änderungen sollen dadurch vereinfacht werden, dass durch Entkopplung die Anzahl derjenigen Teile verringert wird, auf die sich Änderungen eines anderen Teils einer Lösung auswirken. Für Evolu-

tion im Sinne des Erreichens einer Weiterentwicklung von Software sind beide Wirkungen der Zerlegung – Verständlichkeit und Entkopplung – von großer Bedeutung.

Bei den meisten Entwurfsmethoden erfolgt eine Zerlegung nur nach *einem* Kriterium, andere Aspekte bleiben über die Software verteilt. Die objektorientierte Modellierung sieht eine Zerlegung in Klassen oder Module vor, die meist entsprechend der Hauptanforderungen oder Zuständigkeitsbereiche gegliedert werden. Eine weitere mögliche Zerlegung ist die in Architektur, Komponenten und System, wie von RSEB vorgeschlagen [Jacobson et al. 1997]. Eine dritte Zerlegung in Unternehmensziele, Organisation, Prozess und Technologie wird von [Bass et al. 1998] für die Architekturentwicklung vorgeschlagen. [Bosch 2000] schlägt eine weitere Zerlegungsart entsprechend den Entwicklungsphasen in Entwicklung, Einsatz und Evolution vor, die vor allem für Architekturprinzipien wertvoll ist.

Wegen der oben erwähnten Abhängigkeit der Wiederverwendbarkeit der Komponenten von Anforderungen und Architektur ist auf jeden Fall eine *mehrfache Zerlegung* erforderlich - zusätzlich zu der Zerlegung in Klassen oder Module – die von den etablierten Entwurfsmethoden nicht unterstützt werden. Eine solche als „Fine-grained extension model“ bezeichnete Zerlegung ([Bosch 2000] S. 273) wird von Techniken der Generativen Programmierung unterstützt, die deshalb in diesem Abschnitt untersucht werden.

Unter der Bezeichnung *Generative Programmierung* werden Techniken der Zerlegung von Quellcode in Verbindung mit Möglichkeiten der automatisierten Komposition bereitgestellt, die durch Quellcode-Manipulation durch einen sog. Generator erreicht werden [Generative]. Dadurch soll eine Zerlegung von Quellcode auch unter solchen Bedingungen erreicht werden, bei denen konventionelle Modularisierungstechniken aufgrund stark miteinander verwobener Problemlösungen nicht einsetzbar sind. Bei der Generativen Programmierung soll durch die Technik der sog. Multiple Separation of Concerns (MSOC) eine gleichzeitige Zerlegung nach mehreren Kriterien erfolgen [Ossher et al. 2001]. Damit lassen sich feingranulare Komponenten erreichen. Der Quellcode bleibt für die verschiedenen Anforderungen getrennt, die Bestandteile werden entsprechend der gewünschten Eigenschaften durch ein Generator-Werkzeug zu einem lauffähigen System zusammengeführt. Die Methoden und Techniken konzentrieren sich stark auf Mittel der Programmiersprachen, wie bei Template Metaprogramming [Czarnecki et al. 2000].

### 2.3.1 Aspektorientierte Programmierung

Aspektorientierte Programmierung ist eine Technik der Generativen Programmierung. Multiple Separation of Concerns wird hierbei dadurch erreicht, dass spezifische Aufgaben aus den Klassen ausgelagert und in sog. Aspekt-Moduln (kurz: Aspekte) implementiert werden [Kiczales et al. 1997]. Ein Aspekt implementiert die Ergänzung des Verhaltens einer oder mehrerer Klassen in Bezug auf einen Concern. Die Bindung eines Aspekts an die betreffenden Klassen erfolgt bei der Übersetzung. Bei der Zerlegung in Aspekte ist es wichtig, dass eine Orthogonalität der Aspekte erreicht wird, denn ein Aspekt kann nur Klassen, jedoch keine anderen Aspekte ergänzen. Mit Aspect/J steht eine Implementierung dieser Technik für die Programmiersprache Java zur Verfügung [Kiczales et al. 2001][AspectJ]. Der Ansatz ist sehr stark auf die Implementierung und damit auf Programmiersprach-Konzepte ausgerichtet. Die Modellierung wird bisher nur für strukturelle Eigenschaften betrachtet, für die Modellierung von Verhalten existieren keine Vorschläge. Untersuchungen von Macke [Macke 2001] sowie Erfahrungen bei der Entwicklung der ALEXANDRIA-Methodik zeigten, dass die geforderte Orthogonalität der Aspekte zu verringerter Verständlichkeit und Wartbarkeit führt.

### 2.3.2 GenVoca

Im GenVoca-Ansatz [Batory et al.1997] werden Concerns als GenVoca-Komponenten bezeichnet und in sog. Realms (deutsch: Gebiet, Reich) gegliedert. Im Gegensatz zu Aspekten erfolgt hier die Integration zu einem Software-System durch Übereinanderlagern der Realms als Schichten. Dabei ergänzen die oberen Schichten die unteren um weitere Funktionalität. Variabilität lässt sich dadurch erreichen, dass einzelne Schichten weggelassen oder hinzugefügt werden. Die Schichtenarchitektur eines Softwaresystems wird durch eine GenVoca-Grammatik formal beschrieben. Die Grammatik wird von einem Generator verarbeitet und Realms und Komponenten werden zu einem System zusammensetzt. Für den Bau eines solchen Generators sowie die Implementierung von GenVoca-Komponenten steht die Jakarta-Tool Suite zur Verfügung [Jakarta]. Der Ansatz ist ähnlich wie die Aspektorientierte Programmierung auf den Quellcode fokussiert. Die Modellierung von Realms und GenVoca-Komponenten wird durch die bekannten Methoden nicht unterstützt. Nach eingehender Untersuchung [Kleinschmidt 2002] wird der Ansatz GenVoca wegen seiner Begrenzungen durch starke implizite Abhängigkeiten zwischen den Schichten, wegen der stagnierenden Weiterentwicklung sowie wegen seiner unzureichenden Dokumentation in der ALEXANDRIA-Methodik nicht weiter verfolgt.

### 2.3.3 Intentional Programming

Intentional Programming als eine weitere Technik der Generativen Programmierung wendet Transformationen zur Implementierung einer Lösung an. Eine Folge von solchen Transformationen (sog. Enzymes) überführt abstrakte Konzepte (sog. Intentions) zu konkreten, lauffähigen Software-Lösungen [Simonyi 1995]. Die programmiersprachlichen Mittel der Konzepte und Transformationen müssen so gestaltet werden, dass sie möglichst wenige Abhängigkeiten voneinander aufweisen, so dass bei ihrer Interaktion keine Inkonsistenzen auftreten. Ein wesentlicher Nachteil dieser Technik besteht darin, dass es für einen Entwickler sehr schwer ist, für umfangreichere Lösungen die Forderung der Unabhängigkeit zu erfüllen und gleichzeitig abstrakte Konzepte zur Verfügung zu stellen, mit denen eine Anwendungsorientierung erreicht werden kann. Deshalb wird diese Technik hier nicht weiter verfolgt.

### 2.3.4 Hyperspaces

Im Hyperspace-Ansatz wird die Zerlegung aus der aspektorientierten Programmierung in Klassen und Aspekte auf beliebig viele Zerlegungen in Dimensionen erweitert [Ossher et al. 2001][Hyperspace]. In der auch als Subject Oriented Design beschriebenen Weise [Harrison Ossher 1993] ist eine mehrdimensionale Separation of Concerns möglich [Clarke et al. 1999][Tarr et al. 1999] . Der Ansatz definiert ein allgemeines, abstraktes Modell, nach dem eine *gleichzeitige* Zerlegung einer Lösung auf mehrere Arten möglich ist. Er unterscheidet sich damit von bisherigen Methoden, bei denen eine Zerlegung nur *nacheinander* möglich ist. Der Ansatz ermöglicht mit der Zerlegung die Erstellung unabhängiger, feingranularer, komponierbarer Komponenten. Er löst damit das Überschneidungsproblem, das Wiederverwendung von Komponenten und Komposition behindert (siehe auch Abschnitt 2.1). Sein Schwerpunkt richtet sich auf die Behandlung der Struktur eines Systems. Betrachtungen des Verhaltens zur Laufzeit ist zunächst nicht vorgesehen; die Abbildung von Instanzen und Objekten zur Laufzeit fehlt. Der Ansatz bietet deshalb wenig Unterstützung für Aufgabengebiete, bei denen solche Aspekte im Vordergrund stehen.

Der Hyperspace-Ansatz ist unabhängig von Zerlegungsmethoden, Modellierungsarten und Programmiersprachen; er muss vor einer Anwendung erst noch konkretisiert wer-

den. Mit Hyper/J existiert eine Implementierung dieses Ansatzes für die Programmiersprache Java [Tarr et al. 2002] (siehe 2.3.5).

Der Hyperspace-Ansatz ermöglicht eine Zerlegung in überschneidungsfreie Komponenten, die eine eindeutige Zuordnung von Komponenten zu Anforderungen erlaubt. Durch die Zerlegung trägt er zur Verständlichkeit und wartbarkeit bei und unterstützt damit das Ziel der evolutionären Entwicklung. Seine Konzepte der Komposition der Komponenten ermöglichen Flexibilität bei der Schaffung neuer Produkte. Diese Vorteile haben dazu geführt, dass er als wesentliche Basis der Methodik ALEXANDRIA verwendet wurde (siehe Kap. 3 und 6). Er wird dazu hier kurz vorgestellt.

Für das Verständnis des Ansatzes sind die drei Begriffe Hyperspace, Hyperslice und Hypermodule wichtig. Der Hyperspace bezeichnet den (mehrdimensionalen) Raum aller möglichen Zerlegungsarten eines Systems. Das Ergebnis einer Zerlegung wird als Komponente in einem Hyperslice gekapselt, damit es getrennt betrachtet und bearbeitet werden kann. Ein lauffähiges System entsteht aus einer Vereinigung ausgewählter Zerlegungsprodukte und wird als Hypermodule bezeichnet. Die Begriffe werden in den folgenden drei Abschnitten kurz erläutert.

**Identifikation von Belangen und Zerlegung im Hyperspace**
Für eine mehrdimensionale Zerlegung in Belange (Separation of Concerns) ist zunächst zu klären, welche Belange als Zerlegungskriterien in Frage kommen. Für objektorientierte Methoden beispielsweise erfolgt eine Zerlegung in Klassen und Objekte. Für die featuregetriebene Entwicklung einer Produktlinie muss die Zerlegung entsprechend der Features erfolgen, damit die spätere Komposition ebenfalls durch Features gesteuert werden kann. Weitere Zerlegungskriterien können auch die fachliche Zuordnung von Komponenten zwecks Arbeitsteilung und die Aufteilung nach Ablaufumgebung im Fall eines heterogenen Systems sein. Alle Zerlegungskriterien bilden als Belange den Hyperspace. Abb. 12 zeigt das Beispiel eines Hyperspace einer Internet-basierten Anwendung als mehrdimensionalen Raum. Jede Dimension stellt eine mögliche Art der Zerlegung dar. Eine Dimension ist dabei in diskrete (d.h. nicht kontinuierliche) Einheiten unterteilt, die jeweils einen Belang repräsentieren. Die Reihenfolge der Belange in einer Dimension hat dabei keine Bedeutung. Der Nullpunkt einer Dimension wird als 0-Belang (engl. none concern) bezeichnet.

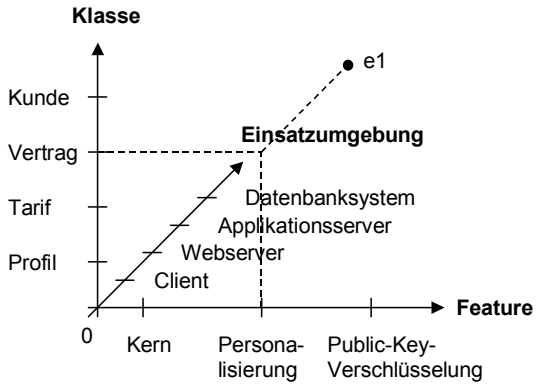



*Abb. 12: Beispiel für einen Hyperspace mit Zuordnung eines Elements*

Die Zerlegung eines Systems erfolgt gleichzeitig nach allen relevanten Kriterien. Dabei entsteht eine Menge von Elementen als Zerlegungsprodukte. Welche Elemente hier möglich sind, wird durch die konkrete Abbildung des Ansatzes auf eine Modellierungs-

oder Programmiersprache festgelegt. Im Fall von Hyper/J gehören Pakete, Klassen, Attribute und Operationen zu diesen Elementen. Bei der Komposition wird dann noch zwischen atomaren und zusammengesetzten Elementen unterschieden, für die eine unterschiedliche Behandlung erforderlich ist. In Hyper/J sind Attribute und Operationen atomar, aus diesen werden Klassen und daraus wiederum Pakete als zusammengesetzte Elemente gebildet.

Die Zuordnung eines Elements zu Belangen erfolgt durch Zuordnung zu einem Punkt im Hyperspace. Die Koordinaten des Punktes geben dann die gleichzeitige Zuordnung eines Elements zu genau einem Belang jeder der Dimensionen an. Ist ein Element für eine Dimension ohne Bedeutung, sozusagen „ohne Belang“, wird es dem 0-Belang dieser Dimension zugeordnet. Bei zusammengesetzten Elementen sind enthaltene Elemente implizit dem gleichen Belang zugeordnet, es sei denn, es wurde eine andere Zuordnung für einzelne von ihnen festgelegt. In Abb. 12 ist das Element `e1` den Belangen `Vertrag`, `Personalisierung` und `Applikationsserver` zugeordnet. Es gehört damit zur Klasse Vertrag, die auf dem Applikationsserver abläuft und das funktionale Merkmal Personalisierung umsetzt.

**Kapselung eines Belangs im Hyperslice**

Ein Hyperspace ermöglicht die Zuordnung von Elementen zu Belangen, die wiederum Dimensionen der Zerlegung zugeordnet sind. Für die Entwicklung und Bearbeitung von Lösungen ist es erforderlich, alle Elemente eines Belangs zusammenzufassen und zu kapseln. Dadurch entstehen überschneidungsfreie Komponenten, die damit eine Komposition in beliebigen Kombinationen zulassen. Für die Kapselung entsprechend den Belangen werden Hyperslices verwendet. Hyperslices werden weitgehend unabhängig voneinander gehalten, damit sie kombinierbar sind.

Ein Hyperslice kapselt Elemente aus einem oder mehreren Belangen einer oder mehrerer Dimensionen. Je nach konkreter Abbildung des Ansatzes wird ein geeignetes Modul-Konstrukt für die Abbildung von Hyperslices verwendet. In Hyper/J sind dies Pakete, die dann alle betroffenen Elemente zusammenfassen.

Für die Abgrenzung, Bearbeitung und spätere Komposition sind Bezüge zwischen den Hyperslices notwendig, was durch deklarative Vollständigkeit als Eigenschaft der Hyperslices erreicht werden soll. Damit soll trotz der Bezugnahme eine Unabhängigkeit zwischen den betroffenen Hyperslices erreicht werden. Deklarative Vollständigkeit bedeutet hier, dass alle Elemente, auf die innerhalb eines Hyperslices Bezug genommen wird, auch in diesem definiert werden. Die Definition muss zumindest soweit erfolgen, dass die Referenz gültig ist. Wenn beispielweise im Fall von Hyper/J eine Operation einer Klasse aufgerufen werden soll, die in einem Hyperslice nicht enthalten ist, so muss diese Klasse zumindest als abstrakte Klasse mit der betreffenden - mindestens abstrakten oder leeren - Operation eingefügt werden, damit eine deklarative Vollständigkeit erreicht wird. Alle Details, die nicht benötigt werden, werden weggelassen. Eine Unabhängigkeit ist damit soweit gegeben, dass dieser Hyperslice nicht direkt von dem anderen Hyperslice abhängt, in dem die Operation oder Klasse implementiert ist. Eine gewisse Abhängigkeit ist trotzdem unvermeidbar, denn irgendein anderer Hyperslice muss die Operation oder Klasse mit einem passenden Verhalten implementieren, damit ein vollständiges System erzeugt werden kann. In Hyper/J würde im Fall fehlender Implementierung der Operation eine Ausnahme `UnimplementedError` hervorgerufen.

**Integration von Belangen im Hypermodule**
Hypermodule werden zum Zusammenfügen der durch Zerlegung in Belange entstandenen Hyperslices zu einem kompletten System eingeführt. Ein Hypermodule fasst eine Menge von Belangen zusammen. Welche Belange dabei wie zusammenzufügen sind, wird durch Integrationsmethoden beschrieben, die in dem Hypermodule für die Elemente der Belange definiert werden. Stehen im einfachsten Fall alle Elemente orthogonal zueinander, ist nur ein Zusammenfügen ohne Integration erforderlich, die Angabe von Integrationsmethoden kann entfallen. Auch ein Hypermodule ist deklarativ vollständig, es ist damit unabhängig von anderen Hypermodules und Hyperslices.

In den meisten Fällen gibt es einige semantisch übereinstimmende Elemente. Durch die Forderung nach deklarativer Vollständigkeit existiert beispielweise eine Klasse gleichen Namens in zwei zu integrierenden Belangen. Dann muss eine Integrationsmethode definiert werden, die Angaben darüber enthält, welche Elemente übereinstimmen und wie sie zu integrieren sind. Im Beispiel muss die nur deklarierte Operation an die implementierte Operation gebunden werden. Da die konkrete Integrationsmethode von der konkreten Modellierungs- oder Programmiersprache abhängt, enthält der Hyperspace-Ansatz dazu keine weiteren Festlegungen.

Der Hyperspace-Ansatz erfordert folgende Festlegungen bei der Umsetzung auf eine konkrete Modellierungs- oder Programmiersprache:

> *Elemente*. Die zu verwendenden Elemente sind zu definieren. Es ist zu entscheiden, welche davon atomar und welche zusammengesetzt sind.
>
> *Hyperspace*. Der Hyperspace mit Belangen und Dimensionen ist zu beschreiben. Dazu ist festzulegen, wie die Elemente den Belangen zugeordnet werden.
>
> *Hyperslices* sind zu definieren, insbesondere bezüglich der Art der Kapselung und der Beschreibung. Die Kriterien der deklarativen Vollständigkeit sind ebenfalls festzulegen.
>
> *Hypermodule* und die dazugehörigen *Integrationsmethoden* sind zu definieren. Dabei ist insbesondere über ihre Mächtigkeit zu entscheiden.
>
> *Werkzeug*. Es ist ein Werkzeug zu entwickeln, das die in Hypermodules beschriebenen Integrationen durchführt.

Der Hyperspace-Ansatz ermöglicht eine simultane Zerlegung eines Systems nach verschiedenen Zerlegungsarten (engl. Multiple Separation of Concerns) in feingranulare Komponenten, die getrennte Kapselung der dabei entstehenden unabhängigen Komponenten und die Komposition der Komponenten entsprechend einer Auswahl von Belangen. Die getrennte Kapselung und die weitgehende Unabhängigkeit verspricht einen Gewinn an Wartbarkeit. Der Ansatz unterstützt im Gegensatz zu den anderen genannten Ansätzen über die Implementierung hinaus auch Anforderungsanalyse und Modellierung durch eine Herstellung der Bezüge zu Belangen. Der Ansatz ermöglicht die Verringerung der Komplexität der Komponenten, weil die Variabilität durch mehrdimensionale überdeckungsfreie Zerlegung statt mittels Entwurfsmustern realisiert wird. Was fehlt, ist die Steuerung von Zerlegung und Komposition durch Anforderungen und Unterstützung für die Modellierung. Der Ansatz wird deshalb im Rahmen der Methodik ALEXANDRIA weiter verfolgt: für die Zerlegung und Komposition von Modellen wird die Modellierungssprache Hyper/UML entwickelt (siehe Kap. 5). Auf der Basis der Komposition dieser Modelle und des Hyper/J-Quellcodes realisiert die Methode Hyper-FeatuRSEB die featuregesteuerte Produktentwicklung (siehe Kap. 6).

### 2.3.5 Hyper/J

Hyper/J bietet als Programmiersprache und mit einem Kompositionswerkzeug eine Umsetzung des Hyperspace-Ansatzes auf die Programmiersprache Java; es stellt damit ein Beispiel dar, wie die abstrakten Bestandteile des Hyperspace-Ansatzes umgesetzt werden können. Da Hyper/J die Entwicklung überschneidungsfreier Quellcode-Komponenten ermöglicht, unterstützt es die Komposition. Das Konzept von Hyper/J wird in der ALEXANDRIA-Methodik als Vorbild für die Entwicklung von Hyper/UML verwendet; die Programmiersprache und das Kompositionswerkzeug werden zur Erzeugung lauffähiger Produkte einer Produktlinie genutzt.

Die Gliederung des Abschnitts folgt der Gliederung der beim Hyperspace-Ansatz zu konkretisierenden Elemente. Für weitergehende Informationen zu Hyper/J sei auf [Tarr et al. 2002] verwiesen. Eine zusammenfassende Darstellung wird beispielweise in [Czarnecki et al. 2000] und [Böllert 2002] gegeben.

*Elemente*. Hyper/J enthält alle Elemente der Programmiersprache Java: Paket, Interface, Klasse, Attribut, Klasseninitialisierung, Konstruktor und Operation. Die Elemente sind durch Namen innerhalb von Namensräumen identifizierbar und eindeutig referenzierbar. Paket, Interface und Klasse werden als zusammengesetzte Elemente definiert, alle anderen Elemente sind atomar. Auch Operationen werden bisher als atomar definiert, da der zu erwartende Nutzen den zusätzlichen Aufwand durch die Betrachtung einzelner Anweisungen bisher nicht rechtfertigt.

*Hyperspace*. Die Definition eines Hyperspace in Hyper/J enthält eine Liste von Zuordnungen von Elementen zu Belangen der einzelnen Dimensionen. Der Name von Elementen wird vollständig, d.h. mit Namensraum angegeben. Für Operationen und Konstruktoren wird außerdem die Signatur angegeben. Existiert bezüglich einer Dimension keine Zuordnung, wird der Belang mit `None` angegeben.

*Hyperslice*. Als Hyperslices werden Pakete der Sprache Java benutzt. Deklarative Vollständigkeit bezieht sich auf die Programmiersprach-Definition, soweit sie vom Compiler geprüft wird. Damit wird sichergestellt, dass alle Verweise innerhalb eines Hyperslice liegen. Benötigt ein Element ein anderes aus einem anderen Hyperslice, so wird das benötigte Element innerhalb des Hyperslice soweit deklariert, wie das für eine syntaktische Korrektheit erforderlich ist. Benötigte Klassen und Schnittstellen werden als leere Schnittstellen oder als abstrakte leere Klassen deklariert. Benötigte Attribute werden mit ihrem Typ deklariert und erhalten keine Initialisierung. Benötigte Operationen und Konstruktoren werden als leere Hülle implementiert, mit einer einzigen Anweisung zum Erzeugen der Ausnahme `UnimplementedError` für den Fall des (fehlerhaften) Aufrufs dieser Hülle ohne vorherige Integration.

*Hypermodule*. Ein Hypermodule enthält eine Liste aller zu integrierenden Hyperslices sowie eine Liste aller vorzunehmenden Integrationen. Dazu wird jeweils die Integrationsmethode sowie Typ und Name der zu integrierenden (gleichnamigen) Elemente angegeben. Ergebnis ist ein neues Paket mit dem Namen des Hypermodules, das die resultierenden Elemente enthält.

*Integrationsmethoden*. Hyper/J verfügt über die beiden grundlegenden Integrationsmethoden Verschmelzen und Ersetzen. Das Ergebnis der Integrationsmethode Verschmelzen wird durch die beiden Integrationsmethoden Ordnen und Summieren genauer bestimmt. Die Semantik der Integrationsmethoden war nicht Bestandteil des Hyperspace-Ansatzes, sie wird deshalb im Folgenden angegeben.

Eigene Erfahrungen bei der Nutzung von Hyper/J für Produktlinien zeigen, dass die Definition von Integrationsmethoden eine Abwägung zwischen Möglichkeiten und Risiken erfordert. Sehr mächtige Integrationsmethoden erlauben dem Entwickler weitgehende Flexibilität bei der Steuerung der Komposition, gleichzeitig ist es für den Entwickler der Hyperslices schwierig, die Auswirkungen einer Integration vorherzusehen und zu berücksichtigen.

Die Integrationsmethode Verschmelzen (engl. mergeByName) vereinigt zwei oder mehr Elemente gleichen Typs miteinander. Im Fall zusammengesetzter Elemente verschmelzen auch darin enthaltene Elemente gleichen Typs jeweils miteinander; im entsprechenden Hypermodule kann jedoch auch eine andere Integrationsmethode für diese Elemente angegeben werden.

Im Fall des Verschmelzens zweier Klassen verschmelzen enthaltene Attribute und Operationen mit gleichen Namen und Typen ebenfalls zu jeweils einem Attribut beziehungsweise einer Operation miteinander. Leere oder abstrakte Operationen werden dabei nicht mit verschmolzen, sie entfallen.

Eine verschmolzene Operation führt beim Aufruf nacheinander die Anweisungen aus, die beide Ausgangs-Operationen enthielten. Die Reihenfolge ist dabei nicht definiert. Will man eine bestimmte Reihenfolge beim Verschmelzen erreichen, muss dazu für die in den Operationen enthaltenen Elemente durch die Integrationsmethode Ordnen (engl. order relationship) eine Reihenfolge mittels `before` und `after` definiert werden.

Die Reihenfolge des Verschmelzens zweier Operationen bestimmt auch über die Datenverarbeitungsfolge und den Rückgabewert der resultierenden Operation. Der Eingabewert wird dem zuerst ausgeführten Methodenteil übergeben. Dessen Rückgabewert wird, falls möglich, dem nachfolgenden Teil als Eingabewert übergeben. Der Rückgabewert wird immer von dem zuletzt ausgeführten Methodenteil geliefert. Wird das Ergebnis der beiden ursprünglichen Operationen gewünscht, kann dies durch die Integrationsmethode Summieren (engl. summary relationship) gesteuert werden, so dass Summen, logische Verknüpfungen oder Zusammensetzungen als Arrays aus beiden ursprünglichen Rückgabewerten erhalten werden können.

Die Integrationsmethode Ersetzen (engl. override relationship) ersetzt ein Element vollständig durch ein anderes gleichen Typs. Da dabei implizite Bezüge anderer Elemente auf das betroffene Element beeinträchtigt werden können, muss ein Entwickler die Auswirkungen dieser Integrationsmethode sehr viel genauer prüfen als bei der Integrationsmethode Verschmelzen. Wegen der Gefahr der Entstehung von Fehlern soll diese Integrationsmethode nur in besonderen Fällen eingesetzt werden.

*Werkzeug*. Mit Hyper/J wird ein Werkzeug zur Verarbeitung der Hypermodule-Definition und zur Durchführung der Integration mitgeliefert. Es verarbeitet dabei die Hyperspace-Definition und die Hyperslices in Form von Java-Bytecode-Dateien (class-Dateien). Es liefert ein Hypermodule-Paket, das den Bytecode der integrierten Klassen enthält und das mit jeder virtuellen Maschine ausgeführt werden kann. Es fehlen jedoch Werkzeuge zur Fehlersuche in Hyperslices, zur Fehlersuche während der Integration sowie zur Navigationsunterstützung bei der Entwicklung von Hyperslices.

Hyper/J erlaubt durch Zerlegung in Hyperslices und Integrationsmethoden in Hypermodules die Implementierung von Variabilität ohne Überschneidungen zwischen Moduln. Durch die deklarative Vollständigkeit ist eine (teilweise) Unabhängigkeit der zerlegten Teile erreichbar. Auf dieser Basis ist die Komplexität eines Systems deutlich geringer als bei der Realisierung mit Design Patterns, wie in Abschnitt 10.3 gezeigt wird. Das

Werkzeug Hyper/J weist in der gegenwärtig vorliegenden Version noch zahlreiche Beschränkungen und Mängel auf, wie [Tarr et al. 2002] zu entnehmen ist. Die Beschränkungen lassen eine Nutzung als Prototyp zu, erlauben jedoch noch keinen industriellen Einsatz.

### 2.3.6 Weitere generative Techniken

Zur Verarbeitung und Komposition von Quellcode existieren noch weitere Techniken. Transformationswerkzeuge auf der Basis von Graph-Transformationen ermöglichen Veränderungen an Quellcode entsprechend vorgegebener formaler Spezifikationen. Sie werden mit Erfolg für Code-Veränderungen bei Migrationsaufgaben und bei der Behebung häufiger Fehler in Alt-Systemen verwendet [Baxter 2002]. Solche Veränderungen sind jedoch nur auf niedrigem Abstraktionsgrad möglich, eine Nutzung zur Umsetzung von Variabilität und Flexibilität für ganze Softwaresysteme wird derzeit noch nicht unterstützt.

Auch Frameprozessoren und XSLT Stylesheet-Prozessoren können Quellcode verarbeiten, obwohl sie eigentlich für die Verarbeitung von logischen Ausdrücken beziehungsweise XML-Code entwickelt wurden [Eisenecker Schilling 2002]. Sie können so angewendet werden, dass jeder Verarbeitungsschritt eine ähnliche Veränderung bewirkt wie eine Schicht der GenVoca-Architektur. Für diese Techniken gilt die gleiche kritische Bewertung wie für GenVoca (siehe 2.3.2), da auch sie durch die verbleibenden Abhängigkeiten zwischen Teilen und Schichten geringe Unterstützung für deren Entkopplung leisten. Darüber hinaus ist kritisch zu bemerken, dass die Wartung und Weiterentwicklung der eigentlichen Steueranweisungen sehr schwierig ist, weil Verständlichkeit (in Bezug auf Auswirkungen von Änderungen) und Testbarkeit unzureichend sind. Die erforderliche Zerlegung fördert Langlebigkeit und Evolutionsunterstützung nicht. Diese Ansätze werden deshalb hier nicht weiter verfolgt.

## 2.4 Gestaltung von Entwicklungsprozessen für Evolution und Komposition

Erfahrungen des Autors mit der Wiederverwendung in der industriellen Nutzung zeigen, dass die Entwicklung (software-)technischer Methoden allein nicht für Effektivitätsverbesserungen in der Praxis ausreicht. Solche Methoden enthalten Vorschriften für die Arbeitsweise von Mitarbeitern, die Softwareentwicklung ausführen. Innerhalb eines Unternehmens existieren immer Regelungen zu Arbeitsteilung und -organisation. Die softwaretechnischen Methoden können nur dann erfolgreich angewendet werden, wenn sie zu den Unternehmens-Regelungen passen. Für Methoden der Produktlinien-Entwicklung gilt dies besonders, weil sie stark mit den Unternehmenszielen koordiniert und von vielen Personen in einem Unternehmen umgesetzt werden müssen. Insbesondere die Regelungen zum *Entwicklungsprozess* haben großen Einfluss darauf, dass die Ziele Evolution und Komposition erreicht werden, weil sie grundsätzliche Vorgaben für Arbeitsteilung, Reihenfolge von Aktivitäten, Kommunikation zwischen den Entwicklern sowie für die Führungstätigkeit enthalten und damit die Arbeit der Entwickler in vielfältiger Weise beeinflussen. Es ist deshalb zu untersuchen, wie Prozessmodelle konventioneller Softwareentwicklung oder alternative Ansätze zu den Zielen dieser Arbeit beitragen können.

### 2.4.1 Konventionelle, projekt-orientierte Softwareentwicklung als Ausgangssituation

Die in dieser Arbeit entwickelten Methoden sollen technologische Fortschritte gegenüber der gegenwärtigen Praxis der Softwareentwicklung ermöglichen. Deren wesentlichen Zielstellungen der Entwicklungsprozesse werded deshalb kurz dargestellt. Die

damit verbundene Einschätzung stammt aus Erfahrungen des Autors aus verschiedenen industriellen Projekten und wird durch die Ausführungen von [Bosch 2000] und [Brown et al. 1998] gestützt. Softwareentwicklung wird demzufolge gegenwärtig in den meisten Organisationen in einer Weise durchgeführt, die durch die folgende Priorisierung der Ziele gekennzeichnet ist:

- *Entwicklung eines einzelnen Systems.* Nahezu überall wird die Softwareentwicklung in Form von Projekten organisiert, deren Ziel jeweils *ein* Softwaresystem ist, das an einen oder viele interne oder externe Kunden ausgeliefert wird. Die Qualitätsmerkmale des Softwaresystems werden meist stärker beachtet als Qualitätsmerkmale des Entwicklungsprozesses.
- *Fokussierung auf Auslieferung.* Alle Entscheidungen werden dem Ziel der Bereitstellung zu einem festgelegten Zeitpunkt untergeordnet. Solche Entscheidungen auf den Gebieten der Softwareentwicklung und des Projektmanagements führen häufig zu negativen Voraussetzungen für anschließende Weiterentwicklung und Wartung der Software.
- *Mangelnde Zielsetzung zu Qualitätsmerkmalen.* Trotz vieler Ansätze und Methoden zur kundenorientierten Durchsetzung von Qualitätsmerkmalen für Software [DIN 66272] erfolgt nur sehr selten eine zielgerichtete Beeinflussung des Softwareentwicklungsprozesses. Innere Qualitätsmerkmale wie Wartbarkeit werden kaum beachtet.
- *Vernachlässigung von Weiterentwicklung und Wartung.* Es wird wenig Augenmerk auf langfristige Aufgaben wie Weiterentwicklung der Software und Wartung gelegt. Die für diese Aufgaben wesentlichen technischen Maßnahmen und Qualitätsmerkmale werden nicht unter wirtschaftlicher Zielsetzung betrachtet.
- *Fehlende Koordination und Variantenbewertung bei der Entscheidungsfindung.* Wirtschaftliche Entscheidungen werden oft anhand von Kennzahlen getroffen, die softwaretechnischen Einflüssen unterliegen. Die Entscheider aus dem Bereich des Managements verfügen jedoch meist nicht über genügend Kenntnisse der Softwaretechnik zur Prüfung der Plausibilität der Kennzahlen. Entscheidungen technischen Inhalts, zum Beispiel über eine Architektur und über Entwurfsalternativen werden dagegen von Mitarbeitern mit softwaretechnischem Arbeitsschwerpunkt getroffen, die selten eine Bewertung der wirtschaftlichen Auswirkungen vornehmen können. Die Entscheidungen aus beiden Bereichen werden nicht ausreichend miteinander koordiniert.
- *Unausgewogenes Verhältnis zwischen Regelungen und Motivation.* Wenn besondere Motivation und starkes Engagement der Bearbeiter zum Erreichen von Zielstellungen nötig sind, werden statt psychologisch ausgerichteter Führungsmethoden häufig organisatorische Maßnahmen eingesetzt. Führen diese Maßnahmen zu einem starken Entzug von Verantwortung und Kompetenzen, sind sie ungeeignet oder beteiligen sie die Bearbeiter unzureichend an der Entscheidungsfindung, kann dies zur Demotivation der Mitarbeiter führen und das effektive Erreichen solcher Ziele wie Qualitätsorientierung, langfristige Entwicklungsziele, Prozessoptimierung und Know-How-Transfer verhindern.

Diese Merkmale konventioneller Softwareentwicklung führen zu typischen Problemen, zu deren Behebung geeignete Methoden helfen können. Diese Probleme wurden in Zusammenhang mit dem Begriff *Softwarekrise* schon oft beschrieben. Ein Beispiel für eines dieser Probleme ist die Kosten- und Terminüberschreitung durch eine große An-

zahl der Projekte. Methoden der Führung von Entwicklungsprozessen und des Projektmanagements orientieren sich deshalb sehr stark auf die Vermeidung dieses Problems; dazu gehören Vorgehensmodelle wie das V-Modell [V-Modell] und Methoden der Planung und Vorhersage. Diese Methoden helfen durch eine starke Orientierung auf kurzfristige Termineinhaltung. Doch das führt gleichzeitig zum Wegfall von Aktivitäten, die bei geringem Zusatzaufwand große Effekte bei späterer Weiterentwicklung und Wiederverwendung bewirken würden, wie eine Bewertung der Softwarearchitektur auf ihre Wartbarkeit und die Dokumentation von Entwurfsentscheidungen. Die langfristigen Ziele Evolution und Flexibilität werden dadurch vernachlässigt.

### 2.4.2 Prozessmodelle für Evolution und langfristige Entwicklung

Das sogenannte Wasserfall-Modell der Softwareentwicklung [Royce 1970] trennt die Aktivitäten der Softwareentwicklung in eine strenge Folge aus Anforderungsanalyse, Entwurf, Implementierung und Inbetriebnahme. Dieses Prozessmodell war lange Zeit bestimmend für die industrielle Praxis der Softwareentwicklung. Es weist eine Reihe gravierender Nachteile auf, wie das Voraussetzen statischer Anforderungen, das Fehlen einer parallelen Bearbeitung verschiedener Aufgaben sowie das Negieren iterativer oder inkrementeller Vorgehensweisen. Alle Tätigkeiten nach der Auslieferung werden als *eine* Phase Wartung zusammengefasst, womit der Eindruck einer homogenen Aktivität erzeugt wird. Die Vielfalt der Aktivitäten während der Wartung und der Bedarf nach besonderer Unterstützung durch Werkzeuge, Methoden und Management wird dabei nicht sichtbar. Auch wenn das Wasserfall-Modell inzwischen in den Vorgehensmodellen der meisten Unternehmen durch iterative Elemente ergänzt wurde, wirken sich die genannten Nachteile beim Projektmanagement der Softwareentwicklung weiterhin aus. Als besonders nachteilig für Vorfertigung und Evolution erweist sich, dass das Reverse Engineering mangelhaft unterstützt wird.

#### Iterative Prozesse

Obwohl Boehms Spiralmodell [Boehm 1988] bereits seit langem bekannt ist, hat es bisher beim Projektmanagement der meisten industriellen Großprojekte das Wasserfall-Modell nur ergänzen und nicht verdrängen können. Das Spiralmodell ermöglicht eine iterativ-inkrementelle Vorgehensweise. Damit sind eine schrittweise Entwicklung, der Bau von Prototypen und Folgen von Produkt-Versionen beschreibbar. Allerdings erfordert das Projektmanagement wesentlich höheren Aufwand zur Steuerung des Entwicklungsfortschritts und zur Analyse des erreichten Standes der Fertigstellung. Vorfertigung und Evolution werden durch dieses Vorgehensmodell besser unterstützt, wenn auch konkrete Aktivitätsbeschreibungen für Wartung und Reverse Engineering nicht im Modell enthalten sind.

Eine Weiterentwicklung mit einer Verstärkung des iterativen Charakters stellen die Agilen Prozesse [Ambler 2002] dar. Ein besonders bekannter Ansatz ist hierbei das eXtreme Programming XP [Beck 1999]. Dieser Ansatz erlaubt für kleine hoch qualifizierte Teams unter einigen Voraussetzungen eine Softwareentwicklung in hochgradig iterativen Prozessen mit sehr kurzen Entwicklungszyklen. Dadurch lassen sich nicht nur Risiken der Entwicklung vermeiden, sondern durch ständige Vereinfachung während des Refactoring bleibt die Wartbarkeit der Software erhalten, obwohl der Aufwand für die Dokumentation verringert wird. Durch automatisierte Tests und intensive Kommunikation wird gesichert, dass Code-Standards erhalten bleiben.

Der Ansatz von XP unterstützt die Ziele der ALEXANDRIA-Methodik wie Evolution und Langlebigkeit sehr gut, wenn die Voraussetzungen wie hohe Qualifikation der Entwickler und geringe Teamgröße erfüllt sind. Bei starker Fluktuation der Entwickler oder

heterogenen, großen Teams kann die Kommunikation und Koordination nicht mehr in ausreichendem Maße sichergestellt werden. Während der Softwareentwicklung werden die inneren Qualitätsmerkmale durch das Streben nach Einfachheit positiv beeinflusst. Gleichzeitig wird das Ziel der Flexibilität der Systeme durch die kurzen Iterationen besser als bei konventioneller Softwareentwicklung erreicht, weil Änderungen aufgrund von Kunden-Anforderungen kurzfristig und mit geringem Aufwand umgesetzt werden.

**Prozesse der Vorfertigung und Wiederverwendung**
Für eine schrittweise Entwicklung, Vorfertigung und Wiederverwendung sind mehrere Entwicklungsprozesse zu koppeln, die jeweils iterativ ablaufen. Solche Prozesse können mit Erweiterungen des Spiralmodells, wie beispielweise dem Prozessmodell EOS mit hierarchisch gestaffelten Entwicklungsprozessen modelliert werden [Hesse 1998]. Im rechten Teil von Abb. 13 wird jeder einzelne Entwicklungsprozess durch eine separate Spirale dargestellt. Eine fertiggestellte Komponente aus Prozess $X_2$ fließt in die Entwurfsphase eines Softwaresystems mit Prozess S ein, und kann dadurch bei der Entwicklung der Software-Architektur berücksichtigt werden. Andere Entwicklungsprozesse $X_n$ und $K_n$ liefern Prototypen, Testwerkzeuge und Bewertungshilfsmittel. Wird die Koordination zwischen diesen Prozessen sowohl für Entwicklungs- als auch für Projektmanagement-Entscheidungen gewährleistet, erlaubt es ein solches Vorgehensmodell, miteinander verbundene Entwicklungsprozesse zu planen und zu steuern. Tätigkeiten der Weiterentwicklung und der Einbindung existierender Software werden jedoch nicht weiter unterstützt als beim Spiralmodell nach Boehm. Es ist zu erwarten, das der Aufwand für das Projektmanagement größer ist als beim Spiralmodell.

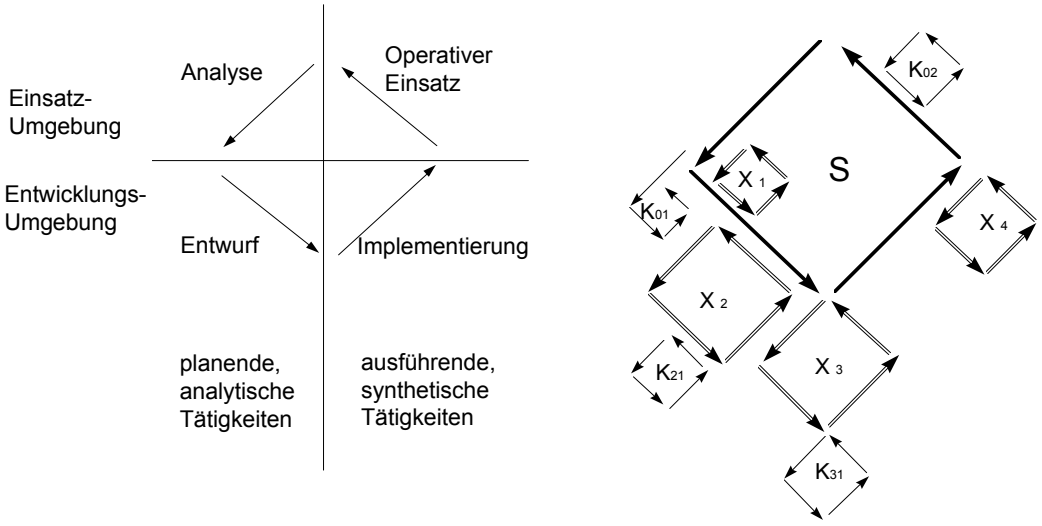



*Abb. 13: Hierarchisch gestaffelte Entwicklungsprozesse bei EOS*

Ein in seiner Staffelung ähnliches Prozessmodell ist bei der Methode FeatuRSEB (Abb. 5, S. 20) und den Produktlinien-Methoden (Abb. 6, S. 21) anzutreffen. Bei diesen Entwicklungsprozessen werden die Aspekte der Koordination zwischen den Teilprozessen für Vorfertigung und für Anwendungsentwicklung besonders betrachtet. Beispiele hierfür sind die Berücksichtigung der langfristigen Strategie einer Produktlinie bei der Anforderungsdefinition eines neuen Produktes. Ein derartiges Vorgehensmodell mit mehreren koordinierten Prozessen wird als Basis der in dieser Arbeit entwickelten Methoden verwendet. Es findet in Abschnitt 3.3 Anwendung.

**Modellierung der Prozesse während der Wartung von Software**
Für das Ziel, eine Weiterentwicklung kontinuierlich aufrecht zu erhalten, muss stärkeres Augenmerk auf die Tätigkeiten der Wartung gelegt werden. Dazu ist das Stufen-Phasenmodell von Rajlich und Bennet [Rajlich Bennet 2000] sehr gut geeignet, weil es die „Wartung“ aufteilt. Es stellt den Lebenszyklus eines Softwaresystems in 5 Stufen dar, die sich durch den Grad an Wartbarkeit und durch den Anteil an Wartungs- und Refactoring-Tätigkeiten unterscheiden (Abb. 14):

- *Erste Entwicklung (Initial Development).* Softwareentwickler erstellen eine erste lauffähige Version eines Softwaresystems.
- *Evolution.* Entwickler erweitern die Fähigkeiten des Systems, damit die Anforderungen der Nutzer erfüllt werden.
- *Pflege (Servicing).* Entwickler beheben Fehler und nehmen einfache funktionale Änderungen vor.
- *Weiterbetrieb (Phaseout).* Die Herstellerfirma hat entschieden, keine Pflege mehr durchzuführen, sondern das System noch solange mit Gewinn zu nutzen wie möglich.
- *Stilllegung (Closedown).* Die Herstellerfirma zieht das System vom Markt zurück und verweist die Kunden auf ein Nachfolgesystem, falls ein solches existiert.

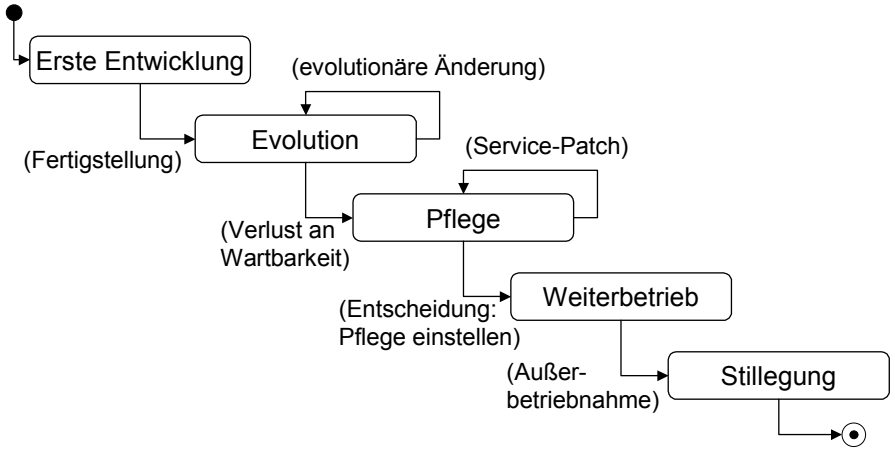



*Abb. 14: Stufen-Modell für den Lebenszyklus von Softwaresystemen als UML-Zustandsdiagramm*

Betrachtet man die Qualität der Entwicklungsdokumente, den Kenntnisstand der Entwickler und die Wartbarkeit der Software genauer und stellt diesen die Möglichkeiten der Weiterentwicklung gegenüber, wird deutlich, dass die Stufe Evolution nach Abschluss der Entwicklung als einzige die Voraussetzungen für umfassende Anpassungen des Softwaresystems an geänderte Anforderungen und damit für eine wirksame Verlängerung der Lebensdauer bietet. In allen nachfolgenden Stufen ist die Wartbarkeit der Software (und damit die Fähigkeit der Entwickler, Änderungen vorzunehmen) soweit gesunken, dass beispielweise Änderungen an Geschäftsprozessen oder eine Erweiterung um neue Produktmerkmale nicht mehr möglich sind. Daraus ergibt sich die Erkenntnis, dass Softwaresysteme in der Stufe Evolution gehalten werden müssen, solange sie die Risikostufe „geschäftskritisch“ aufweisen. Es wird deutlich, dass Softwaresysteme in den Stufen Pflege und Weiterbetrieb nicht mehr an Anwenderforderungen angepasst werden können und dass ihre Stilllegung nur eine Frage der Zeit ist. Reengineering-

Maßnahmen (bestehend aus Reverse Engineering und Refactoring, siehe 2.5 und 2.6) stellen die einzige Möglichkeit dar, ein Softwaresystem wieder in die Stufe Evolution zurückzuführen. Sie sind jedoch mit hohen Kosten und großen Risiken bezüglich Fehlern und Qualitätsmängeln verbunden.

Das Stufen-Modell zeigt die Bedeutung von Maßnahmen, die evolutionäre Änderungen erlauben, den Erhalt der Kenntnisse über Architektur und Funktion ermöglichen und zum Erhalt der Wartbarkeit eines Systems beitragen. Es beschreibt – über ein iteratives Prozessmodell wie das Spiralmodell hinaus – Entwicklungsstufen, die sonst nur als Wartung zusammengefasst werden. Die Fokussierung des Entwicklungsprozesses auf die Auslieferung wird überwunden. Die Bedeutung der Weiterentwicklung für die Wirtschaftlichkeit wird verdeutlicht. Durch Bewertung auch der Wartungskosten erhalten die inneren Qualitätsmerkmale wie Wartbarkeit und Portabilität [DIN 66272] mit Einflussgrößen wie Verständlichkeit, Strukturiertheit, Modularität und Angemessenheit der Architektur stärkeres Gewicht.

### 2.4.3 Fokussierung auf Qualitätsmerkmale und Prozessverbesserung

Neben der traditionell starken Orientierung auf die Einhaltung von Terminen und Budgets sind eine Reihe von Maßnahmen des Projekt- und Qualitätsmanagements auf das Erreichen von Qualitätsmerkmalen für Software gerichtet, häufig in Zusammenhang mit Vorgaben für Produktivität und Kundenzufriedenheit. Dazu gehören die analytischen und konstruktiven Maßnahmen des Qualitätsmanagements, wie sie beispielsweise zum Capability Maturity Model CMM [Thaller 1993], zur Methode BOOTSTRAP [BOOTSTRAP] und zum Maßnahmenpaket der European Foundation for Quality Management EFQM [EFQM] gehören.

Die Kunden achten vor allem auf die äußeren Qualitätsmerkmale Funktionalität, Benutzbarkeit, Zuverlässigkeit und Effizienz [DIN 66272], deshalb bieten die genannten Maßnahmen vor allem Unterstützung für deren Erfüllung. Die Umsetzung, d.h. das tatsächliche Verbessern dieser Merkmale, muss jedoch durch die Entwickler manuell, mit wenig Modell- und Werkzeugunterstützung vorgenommen werden.

Eine direkte Unterstützung für Evolution und Komposition fehlt bei diesen Maßnahmen, weil hierfür das innere Qualitätsmerkmal Wartbarkeit [DIN 66272] größere Bedeutung erhalten müsste. Wartbarkeit wird als Ziel gering priorisiert, weil es sich nicht unmittelbar auf die Kundenzufriedenheit auswirkt, weil es quantitativ relativ schlecht bewertbar ist und weil das Management häufig kurzfristige Ziele bevorzugt, oft wegen Mangel an Verständnis, Kontrolle und Einflussmöglichkeiten gerade auf innere Qualitätsmerkmale.

Die oben genannten Maßnahmen bringen eine starke Fokussierung auf die Verbesserung der Entwicklungsprozesse mit sich. Durch die Idee der „Best Practice“ sollen Erfolge dadurch wiederholbar werden, dass ein vorgezeichneter Idealweg der Weiterentwicklung der Entwicklungsprozesse verfolgt wird. Damit bieten sie das Potential, Maßnahmen zur Verbesserung der Wartbarkeit auf dem Weg der „Best Practice“ ebenfalls zu unterstützen. Die Akzeptanz und Motivation der Mitarbeiter, diesem vorgezeichneten Entwicklungsweg zu folgen und die vorgeschlagenen Methoden in ihre Arbeitsweise zu übernehmen, stellt eine wesentliche Voraussetzung für den Erfolg solcher Maßnahmen dar. Diese Maßnahmen werden deshalb in Verbindung mit der ALEXANDRIA-Methodik verwendet, Abschnitt 9.3 zeigt die dafür notwendige Schwerpunktsetzung.

### 2.4.4 Entscheidungsfindung und -dokumentation

**Koordination wirtschaftlicher und technischer Aspekte bei der Entscheidungsfindung**

Während der Entwicklungsprozesse sind eine Vielzahl von Entscheidungen zu treffen, wie über den Umfang und den Inhalt von Entwicklungsaufgaben oder über anzuwendende technische Lösungen. In den meisten größeren Unternehmen sind Entscheidungskompetenzen und Qualifikation durch hierarchische Organisationsformen voneinander getrennt.

Das obere Management trifft die Entscheidungen über die langfristige Strategie beispielsweise bezüglich der Produktentwicklung allein. Die Entscheidungsträger verfügen dabei typischerweise über Qualifikation auf betriebswirtschaftlichem Gebiet. Kenntnisse über Softwareentwicklung oder über die Steuerungsmöglichkeiten liegen bei diesem Personenkreis äußerst selten vor.

Die Entscheidungen über technische Fragen der Softwareentwicklung werden von den Entwicklern häufig allein getroffen. Dies trifft auch auf strategische Entscheidungen wie die über eine Architektur oder über Entwurfsalternativen zu, die Auswirkungen auf die langfristige Produktentwicklung haben. Eine Abstimmung über die strategischen Ziele zwischen beiden Entscheidungsträgern kommt wegen fehlender Kommunikation in größeren Organisationen meist nicht zustande. Zudem verfügen die meisten Entwickler über wenig Verständnis für wirtschaftliche Zusammenhänge und Marktstrategien.

Eine Lösung dieses Konfliktes könnte beispielsweise durch Ausgleich von Kompetenzen und Qualifikationen erfolgen. Hierfür kann im Rahmen dieser Arbeit keine Lösung angeboten werden. Für die Herstellung der Kommunikation bei gemeinsamer Entscheidungsfindung werden in den Abschnitten 4.6 und 8.3.3 Lösungsansätze entwickelt.

**Variantenbewertung bei Entscheidungen und deren Dokumentation**

Zur Vermeidung von Fehlern und zur Behandlung von Risiken sollten Entscheidungen immer nach einer Bewertung möglicher Alternativen erfolgen. In der Praxis werden Entscheidungen oft nur durch ihr Ergebnis oder überhaupt nicht dokumentiert, obwohl viele Vorgehensmodelle und Prozessbeschreibungen, wie das V-Modell [V-Modell] und das CMM [Thaller 1993], dies in unterschiedlichem Umfang fordern. Zur Dokumentation einer Entscheidung gehört danach sowohl der Anlass einer Entscheidung, ermittelte Alternativen und deren Bewertung als auch die getroffene Entscheidung und deren Konsequenzen. Zur Bewertung sind die Anforderungen aus der Anwendungsdomäne oder von den Kunden wesentlich. In den Entscheidungsmodellen von KobrA (siehe 2.1.5 und [Atkinson et al. 2002]) werden Entscheidungen durch Textschemata beschrieben, die aus folgenden Bestandteile zusammengesetzt sind:

- Fragestellung: textuell, mit Bezug zu Problembereich und Anforderungen
- Menge möglicher Lösungsalternativen
- Referenzen zu betroffenen Elementen und Variationspunkten
- Auswirkung der Entscheidung für mögliche Antworten auf betroffene Elemente
- Referenzen zu davon abhängigen Entscheidungen
- Auswirkung auf abhängige Entscheidungen

Eine solches Textschema ist geeignet, Entscheidungen nachvollziehbar zu beschreiben und spätere Revisionen zu vereinfachen. Damit unterstützt es die Ziele Evolution und

Flexibilität. Es wird den Entscheidungsprozessen dieser Arbeit zugrundegelegt, beispielweise in den Abschnitten 4.6 und 8.3.3 für Scoping, Koordination und Evolution.

### 2.4.5 Ansätze mit stärkerer Fokussierung auf die Motivation

In Arbeiten auf den Gebieten softwaretechnischer Methoden und Modellierung werden häufig vor allem technische Aspekte der Problemlösung betrachtet. Dabei darf nicht außer Acht gelassen werden, dass die Problemlösung bei der Softwareentwicklung hohe Kreativität, großes Abstraktionsvermögen und viel Ausdauer erfordert. Entscheidungen haben oft weitreichende Auswirkungen auf die nachfolgenden Entwicklungsabschnitte. Dafür sind Motivation und Engagement der Entwickler eine unabdingbare Voraussetzung. Die meisten Methoden und Maßnahmen nehmen auf Motivation und Engagement der Entwickler jedoch wenig Rücksicht.

Mit Total Quality Management TQM [Mellis et al. 1996] existiert eine Führungsmethode, die bei ihrer Ausrichtung auf Produktivität und langfristige Kundenzufriedenheit auch die Motivation der Mitarbeiter berücksichtigt. Zu dieser Methode gehören Prinzipien wie faktenbasierte Entscheidungen, Risikominimierung durch schrittweise Veränderungen und Beteiligung aller Mitarbeiter an Verbesserungen, die zur Vermeidung der genannten Gefahren beitragen können. Diese Methode sollte deshalb bei der Einführung, Anpassung und Aufrechterhaltung der in dieser Arbeit vorgestellten softwaretechnischen Methoden angewendet werden. Darauf wird in Abschnitt 9.2 Bezug genommen.

## 2.5 Reverse Engineering und Beschreiben von Struktur und Information

Reverse Engineering ist notwendig, wenn die Wartbarkeit eines Systems soweit gesunken ist, dass Änderungen und Weiterentwicklung nicht mehr möglich sind und wenn Anforderungsbeschreibungen und Modelle nicht (mehr) zur Verfügung stehen. Gemeinsam mit einer anschließenden Refactoring-Maßnahme erlauben sie die Rückführung eines Softwaresystems aus den Phasen Pflege, Weiterbetrieb oder Stillegung des Stufen-Modells (siehe Abb. 14, S. 44) in die Phase Evolution.

**Analyse**

Bei der Analyse und Bewertung von Methoden des Reverse Engineering gilt das Interesse hier vorrangig den Methoden zur Analyse der Software, obwohl das Reverse Engineering von Daten ebenfalls eine wichtige Aufgabe darstellt. Die meisten bekannten Methoden und Techniken beziehen sich auf den Quellcode, weil dieser in derartigen Fällen die einzige zuverlässige Informationsquelle darstellt. Diese Techniken nutzen vor allem die syntaktischen Informationen, die im Quellcode enthalten sind. Dazu gehören Techniken zur Zerlegung in Subsysteme, zur Begriffssynthese, zum Mustervergleich, zur Analyse von statischen und dynamischen Abhängigkeiten sowie Objektorientierte Metriken und Techniken der Software-Visualisierung. Gut geeignete Verfahren zur dynamischen Analyse sind die Profiling- und Visualisierungsverfahren von [Systä 1999] und [Walker et al. 1998]. Eine Zusammenstellung und Bibliographie solcher Verfahren ist in [Müller et al. 2000] enthalten. Als wichtiges Vorgehen für die Analyse hat sich das Aufstellen und überprüfen von Hypothesen erwiesen, wie es Teil der meisten Vorgehensweisen ist, wie z.B. [Weide et al. 1995].

**Verknüpfung zwischen unterschiedlich abstrakten Informationen**

Im Quellcode fehlen jedoch zahlreiche Informationen, die für die Erweiterung eines Softwaresystems um neue Anforderungen oder für die Umgestaltung ihrer Architektur benötigt werden. Dafür werden Informationen über abgebildete Geschäftsprozesse, über

Entwurfsziele, -entscheidungen und Abhängigkeiten sowie über frühere Anforderungen benötigt. Für Veränderungen wird Wissen über verschiedene Aspekte der Architektur und über die Auswirkungen der Änderungen benötigt. Zwischen dem Abstraktionsgrad der verfügbaren Informationen aus dem Quellcode und der benötigten Informationen besteht eine Lücke, die das Verständnis behindert.

Zum Schließen dieser Lücke sind neben Informationen vor allem Abstraktionen erforderlich. Dies erfordert hohen kognitiven Aufwand der Entwickler. Analysen von existierendem Quellcode sind schwierig und aufwendig, sie weisen große Risiken auf. Für die Erkennung von Abstraktionen und das damit verbundene Verständnis durch die Entwickler ist Traceability als Verknüpfung zwischen den verschiedenen Abstraktionsebenen von Quellcode, Architektur und Anforderungen sehr wichtig. Dadurch hergestellte Bezüge helfen, schrittweise Zuordnungen zwischen Elementen dieser Abstraktionsebenen herzustellen und zusammengehörende Elemente zu identifizieren. In der Literatur gibt es einige Ansätze zu solchen Analysen, die als *Program Comprehension* und als *Architectural Recovery* bezeichnet werden [Müller et al. 2000]. Diese Methoden haben bisher keine große Akzeptanz in der industriellen Praxis finden können. Für weitere Ansätze wie das Erschließen von Wissen aus der Analyse von Bezeichnern oder die Erkennung von Strukturen durch statistische Auswertungen mit Metriken sind bisher keine erfolgreichen Anwendungen bekannt. Diese Ansätze stellen jedoch einen wichtigen Ausgangspunkt für Weiterentwicklungen dar. Sie werden deshalb als Grundkonzept der Reverse-Engineering-Methode in Abschnitt 8.1 verwendet.

Ausgehend vom hohen Aufwand und den großen Risiken des Reverse Engineering ist bei der Definition von Methoden und Vorgehensweisen des Forward-Engineering darauf zu achten, dass die benötigten Informationen gar nicht erst verloren gehen. Es sind Maßnahmen zur Aufrechterhaltung der Konsistenz der Modelle und der Dokumentation vorzusehen. Diese Forderung fließt in das entwickelte Vorgehensmodell in Abschnitt 3.3 sowie in die Definition von Traceability-Links in Abschnitt 4.2 ein.

**Traceability**
Traceability wurde als Konzept zur Herstellung von Beziehungen zwischen Anforderungen verschiedener Ebenen in das Requirements Engineering eingeführt, damit Anforderungen besser verstanden und Auswirkungen von Änderungen besser verfolgt werden können [Hull et al. 2002]. Dick entwickelt einfache Traceability-Beziehungen zwischen verschiedenen Elementen durch Verknüpfungen weiter, die er als Rich Traceability bezeichnet und die eine bessere Strukturierung und besser auswertbare Beschreibungen ermöglichen [Dick 2000]. Diese Konzepte eignen sich gut zur Verknüpfung zwischen den verschiedenen Abstraktionsebenen, die während evolutionärer Weiterentwicklung zu verstehen und deren Elemente bei Veränderungen konsistent zueinander zu halten sind. Dieses Konzept wird deshalb in Abschnitt 4.2 zur Verknüpfung zwischen allen Artefakten der Produktlinien-Entwicklung ausgebaut und durch OCL-Ausdrücke definiert.

## 2.6 Einbeziehen von Refactoring und Migration in die Weiterentwicklung

*Refactoring* („Neu-Schneiden“) bezeichnet das Verändern der Struktur einer Software bei (weitgehender) Beibehaltung ihrer Funktion, beispielweise zur Anpassung an eine Architektur oder zum Zerlegen in Komponenten [Fowler 1999]. *Migration* bezeichnet das Überführen und die Einpassung von Software in eine neue Umgebung. Durch die Adaption dieses Begriffs aus Biologie und Soziologie, wo er wiederholte Wanderungsbewegungen bezeichnet (lat. migrare – wandern, wegziehen) [Duden 2000], wird der wiederkehrende Charakter solcher Veränderungen an der Software betont. Häufig wer-

den damit Anpassungen der internen Struktur einer Komponente an eine sich wiederholt ändernde Architektur bezeichnet.

Beide Begriffe beschreiben Techniken, die zur Fortsetzung der Evolution (siehe Stufen-Modell, S. 44) benötigt werden. Prinzipiell gelten die gleichen Techniken sowohl für sog. Legacy-Systeme, für Komponenten von Dritten, die in ein Softwaresystem einbezogen werden sollen, als auch für Softwaresysteme, die sich in der Stufe Evolution befinden, aber deren Architektur geänderten Anforderungen angepasst werden soll. Als Legacy Software („geerbte Software“) werden Altsysteme bezeichnet, deren Weiterentwicklung schwierig oder unmöglich scheint, die aber trotzdem wichtige Aufgaben zu erfüllen haben. Bezogen auf das Stufen-Modell aus Abb. 14 (S. 44) haben sie die Stufe Evolution schon verlassen. Ihre Wartbarkeit durch den Effekt des Architectural Decay häufig gering, zudem sind Dokumentation und Architektur meist mangelhaft oder fehlen. Oft existiert nur noch der Quellcode als verlässliche Informationsquelle.

Auch wenn Refactoring und Migration die Evolution existierender Software unterstützen, beeinflussen die angewandten Techniken die weitere Flexibilität und Evolution ebenfalls. Sie sind danach zu bewerten, wie sie sich auf die Wartbarkeit auswirken, insbesondere auf Verständlichkeit, Erweiterbarkeit, Konfigurierbarkeit und Komplexität der Resultate.

**Techniken**

Techniken zu Änderungen der betroffenen Komponenten können in Black-Box- und White-Box-Techniken eingeteilt werden [Bosch 2000]. *Black-Box-Techniken* passen nur die Schnittstelle von Komponenten an, ohne Veränderung der Inhalte. Als Voraussetzung dafür reichen im wesentlichen Informationen über die Schnittstelle aus. Ein typisches Beispiel stellt die Methode des Wrapping dar. Ein Wrapper kapselt eine anzupassende Komponente dadurch, dass er die gesamte Interaktion mit ihr selbst abwickelt. Das erlaubt dem Wrapper, die Kommunikation mit dieser Komponente zu überwachen und zu steuern, dass ihr Verhalten gegenüber der Schnittstelle so verändert wird, dass sie die aktuellen Anforderungen der Architektur erfüllt. Dies kann zu hohem Aufwand führen, insbesondere wenn das Verhalten einer anzupassenden Komponente stark von der jeweiligen Vorgeschichte abhängt, und deshalb eine umfangreiche Interaktion zwischen dem Wrapper und der Komponente erforderlich wird. Neben potentiellen Problemen mit dem Zeit- und Ressourcenbedarf dieser Technik wird die Korrektheit stark von der Verfügbarkeit von Informationen über die Komponente bestimmt. Durch die Kapselung im Wrapper entsteht eine neue, vollständig gekapselte Komponente, was die Wartbarkeit positiv beeinflusst. Wrapper weisen im allgemeinen keine weitere Flexibilität auf.

*White-Box-Techniken* erfordern das Verständnis (siehe 2.5) und den Zugriff auf die interne Struktur. Dabei sind Teile der internen Spezifikation, Struktur und Implementierung zu verändern durch Entfernen, Überschreiben oder Ergänzen. Zwei wesentliche Techniken sind die sogenannte Copy-Paste-Nutzung und die Vererbung [Bosch 2000]. Weitere Techniken auf Quellcode-Ebene werden in [Fowler 1999] beschrieben.

Die *Copy-Paste-Nutzung* kann auch als einfachste Form der Wiederverwendung betrachtet werden; sie wird von [Sametinger 97] als Code Scavenging bezeichnet. Der Teil einer Komponente, der die benötigte Funktionalität enthält, wird kopiert und als neue Komponente genutzt. Nach dem Kopieren werden Veränderungen zwecks Passfähigkeit mit der Architektur vorgenommen. Auch wenn diese Technik von Entwicklern insbesondere unter Zeitdruck äußerst häufig angewendet wird, hat sie schwerwiegende Nachteile. Werden die kopierten Quellcode-Teile nicht genau analysiert, wird das Ver-

halten nur zum Teil verstanden, was zu einer Häufung von Mängeln bezüglich Funktionalität und Robustheit führt. Es entstehen mehrere Kopien von Quellcode-Teilen, was den Wartungsaufwand vervielfacht. Meist entstehen durch Referenzen zusätzliche Abhängigkeiten, wodurch die Komplexität der Software steigt. Durch das bloße Übernehmen von Strukturen wird oft die Architektur beeinträchtigt; Verständlichkeit, Flexibilität und letztendlich Wartbarkeit werden verringert.

*Die Vererbung* ist als weitere White-Box-Technik der Anpassung für objektorientierte Programmiersprachen verfügbar. Durch Vererbung kann die Schnittstelle einer Klasse so verändert werden, dass sie den Anforderungen entspricht. Wenn es nicht durch Möglichkeiten der Programmiersprache verhindert werden kann – wie durch die Modifier `private` und `protected` in C++ und Java – werden dabei Teile der ursprünglichen Schnittstelle sowie Informationen über den internen Aufbau nach außen gegeben. Die dann fehlende Kapselung beeinflusst die Wartbarkeit negativ. Da kein Code dupliziert wird, steigt der Wartungsaufwand nicht in gleichem Maße. Probleme der Verständlichkeit und Testbarkeit resultieren jedoch aus der Aufteilung von Funktionalität über mehrere Klassen und beeinträchtigen die Wartbarkeit.

**Vorgehensweise**

Für das Vorgehen bei Refactoring und Migration gibt es eine Reihe von Ansätzen, die auf Erfahrungen aus größeren Migrationsprojekten beruhen und nach geringer Anpassung hier benutzt werden können. Die Reengineering-Methoden des SEI [Reeng-SEI] beschreiben das Vorgehen mit Planung, Entscheidungen und Risikobewertung, enthalten aber nur wenige Aussagen für die tatsächliche Durchführung von Veränderungen an der Software. Die aus Erfahrungen mit Refactoring bei Extreme Programming aufbereiteten Methoden von [Fowler 1999], [Roberts 1999] und [Kerievsky 2002] umfassen Zusammenstellungen von Vorgehensweisen für feingranulare Veränderungen, meist auf der Ebene des Quellcodes. Wichtige Forderungen zur Verminderung der Risiken sind die Bereitstellung von Testfällen zur Prüfung auf Fehler und iteratives, planmäßiges Vorgehen in kleinen Schritten. Diese Vorgehensweisen werden in den vorgestellten Arbeiten übernommen, wenn Refactoring und Migration bei iterativ-inkrementeller Entwicklung, bei der Evolution der Produktlinie sowie bei der Einbindung existierender Systeme angewendet werden.

Als Bewertung kann eingeschätzt werden, dass die angeführten Refactoring- und Migrationstechniken sehr gut zur Einbindung in die Methodik ALEXANDRIA geeignet sind. Dazu werden in den Abschnitten 8.2 und 8.3 Anpassungen vorgenommen und Schwerpunkte des Einsatzes bei der Produktlinien-Entwicklung angeben.

# 3 Software-Produktlinien-Methodik ALEXANDRIA

*Im folgenden Kapitel wird die Konzeption der vom Autor entwickelte Methodik der Entwicklung von Software-Produktlinien vorgestellt. Es wird auf Ziele und Beiträge eingegangen, anschließend werden die einzelnen Bestandteile und ihre Einordnung in den Entwicklungsprozess im Überblick vorgestellt. Die folgenden Kapitel 4 bis 1 behandeln diese Bestandteile der Methodik im Detail.*

## 3.1 Grundlegende Ausrichtung

Die Methodik ALEXANDRIA ist das Ergebnis eines Forschungsprojekts an der TU Ilmenau zur Entwicklung von softwaretechnischen Methoden für flexible und langlebige Softwaresysteme. Die Methodik soll durch kurzfristige Produktbereitstellung und langfristige Aufrechterhaltung der Entwicklungsfähigkeit zum wirtschaftlichen Erfolg des Produzenten beitragen.

Die entwickelten Methoden sollen den Entwicklungsprozess unterstützen und steuern. Sie sollen geeignet sein, dabei die Anwenderorientierung der Software zu fördern. Sie sollen weitgehend Modelle zur Entwicklung von Software nutzen. Die Forschungsarbeiten sollen zur Konsolidierung der Methoden der Softwaretechnik beitragen, deshalb sind bewährte Methoden unter Nutzung ihrer Vorzüge so weit wie möglich zu integrieren. Die Methoden sollen das Beherrschen der Komplexität großer Softwaresysteme unterstützen und sich für einen Einsatz in der industriellen Praxis eignen. Zu ihren Zielen soll deshalb die praktikable Anwendbarkeit, die Möglichkeit der Werkzeugunterstützung und die Berücksichtigung existierender Software gehören. Die Methoden sind in Zusammenarbeit mit Industriepartnern zu evaluieren.

## 3.2 Ziele und Schwerpunkte

Bei den Forschungsarbeiten konnte von Erfahrungen mit Methoden der Wiederverwendung, der Komponententechnologie, der Entwicklung von Application Frameworks sowie mit der Optimierung von Software-Entwicklungsprozessen ausgegangen werden.

Die Methoden der Wiederverwendung sollen um das fehlende Element der planmäßigen, anforderungsgetriebenen Entwicklung ergänzt werden (siehe 2.1). Deshalb wurde unter Einbeziehung von Domänenanalyse-Techniken an Produktlinien-Methoden gearbeitet. Die Featuremodellierung erwies sich als eine geeignete Technik, mit der die Entwicklung von Produktlinien geplant und gesteuert werden kann. Erste Konzepte dazu wurden in [Riebisch et al. 1999] und [Streitferdt et al. 2000] veröffentlicht. Die Featuremodellierung ist darüber hinaus auch für die evolutionäre Weiterentwicklung und für die Ableitung von Produkten aus der Produktlinie geeignet und wurde deshalb als zentrales Modell eingeführt. Dazu wurde von der FeatuRSEB-Methode (siehe Abschnitt 2.1.5) die Analyse, die Featuremodellierung und der zweistufige Entwicklungsprozess für Produktlinien übernommen, sie wurde durch eine präzisierte Definition und durch eine Ergänzung der Modelle um OCL-ähnliche Ausdrücke ergänzt (siehe Kap. 4 und 6). Dadurch wurde eine bessere Auswertbarkeit erreicht, die weitergehende Werkzeugunterstützung erlaubt. Abb. 15 stellt die verschiedenen Beiträge der Methodik ALEXANDRIA und ihren Bezug zu vorhandenen Methoden und Techniken dar.

Für die flexible Kombinierbarkeit der Bestandteile einer Produktlinie ist eine geeignete Technik der Implementierung notwendig, die eine Zerlegung in unabhängige, überschneidungsfreie Komponenten erlaubt. Die Implementierung mittels Objektorientie-

rung, Komponententechnologie und Application Frameworks, wie sie von anderen Produktlinien-Methoden vorgeschlagen werden, können eine solche Zerlegung nicht erreichen (siehe 2.1.5) oder eignen sich wegen mangelnder Skalierbarkeit nicht für große Produktlinien. Die Techniken der Generativen Programmierung bieten durch eine mehrfache Zerlegung (Multiple Separation of Concerns) und durch dazugehörige Kompositionswerkzeuge die besser geeignete Implementierungstechniken und führen zu geringerer Komplexität der Produktlinien (siehe 2.3). Nach Untersuchung und Vergleich der Generativen Techniken erwies sich der Hyperspace-Ansatz (siehe 2.3.4) als am besten geeignet.

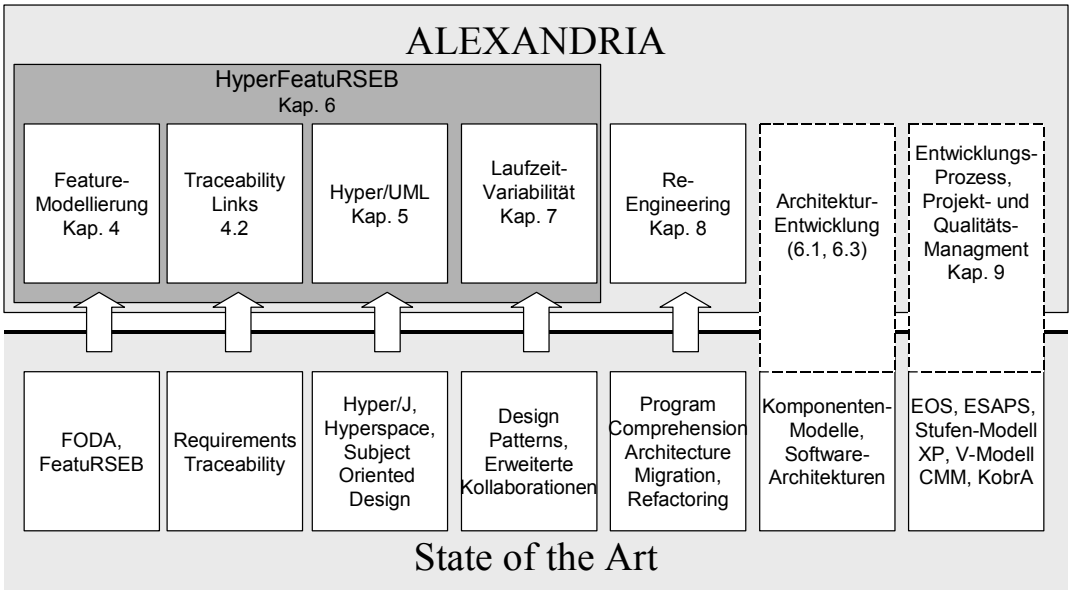



*Abb. 15: Beiträge und Schwerpunkte der Methodik ALEXANDRIA*

Zur Modellierung von Produktlinien muss die UML um Möglichkeiten der Darstellung von Variabilität erweitert werden, wie in Abschnitt 2.2.2 gefordert. Zwecks Komposition auf der Abstraktionsebene von Modellen wurde der Hyperspace-Ansatz auf UML-Modelle überführt, indem Hyper/UML entwickelt wurde (siehe Kap. 5 und [Böllert 2002]). Damit kann die sehr fehleranfällige und aufwendige manuelle Anpassung der Implementierung durch eine Komposition ersetzt werden. Die Methode HyperFeatuRSEB beschreibt, wie die Steuerung der Zerlegung und der Komposition mit Featuremodellen vorgenommen wird (siehe Kap. 6). Zur Verbindung von Featuremodellen mit Anforderungen, Entwurf und Implementierung wurden die aus dem Requirements Engineering bekannten Traceability-Links zu einem Konzept weiterentwickelt, das eine Verknüpfung der verschiedenen Abstraktionsebenen der Softwareentwicklung zwecks Verständlichkeit und Werkzeugunterstützung erlaubt (siehe 4.2). Sie ermöglichen, die Abhängigkeiten zwischen Komponenten bei der Auswahl von Features zu beachten, den Aufwand und die Auswirkungen von Änderungen bei der Weiterentwicklung zu ermitteln und die Klarheit und damit die Wartbarkeit einer Produktlinie zu erhöhen. Bei der automatischen Generierung von Testmitteln und Dokumentation werden die Traceability-Links ebenfalls ausgewertet.

Durch Komposition von Modellen und Quellcode nach dem Hyperspace-Ansatz lässt sich Variabilität für einen frühen Bindungszeitpunkt implementieren. Variabilität für einen späten Bindungszeitpunkt wird durch Abstraktionen und Erweiterungen ermöglicht, die jedoch einer automatisierten Anwendung nicht zugänglich sind, weil die dabei

meist benutzten Design Patterns dies nicht unterstützen (siehe 2.2.3). In der Methodik ALEXANDRIA wird deshalb das Konzept der Erweiterten Kollaborationen zur werkzeuggestützten Anwendung solcher Design Patterns genutzt. Laufzeit-Variabilität lässt sich dadurch während der Komposition nach dem Hyperspace-Ansatz implementieren (Kap. 7). Zur Steuerung der Laufzeit-Variabilität werden die in den Featuremodellen enthaltenen Beziehungen ausgewertet.

Für das Einbeziehen existierender Software – sowohl von Legacy-Systemen als auch von am Markt erhältlichen Komponenten – sind deren Verständnis, deren Anpassung und ihre Integration erforderlich. Existierende Methoden der Program Comprehension, des Refactoring und der Migration (siehe 2.5 und 2.6) beziehen sich meist auf Quellcode, für die Weiterentwicklung innerhalb einer Produktlinie werden jedoch auch Informationen zu Anforderungen, Variabilität und Architektur benötigt. Es wurde eine Reverse-Engineering-Methode auf der Basis von Featuremodellen entwickelt, die die Features zur Verbindung zwischen den verschiedenen Abstraktionsebenen benutzt und außerdem Traceability-Links ermittelt (siehe Kap. 1). Die Refactoring- und Migrationsmethoden werden durch Vorgabe der für Produktlinien-Architekturen der notwendigen Schwerpunkte angepasst.

Die Software-Architektur einer Produktlinie entscheidet wesentlich über deren Eigenschaften sowie über die Eigenschaften daraus erstellter Produkte. Die vorhandenen Methoden für die Entwicklung und Beschreibung solcher Architekturen sind geeignet, die oben genannten Ziele zu erreichen. Die Methodik ALEXANDRIA unterstützt die Vorgehensweise nach Bosch, die Plug-in-Architekturen sowie weitere Ansätze, sie ist jedoch nicht an eine bestimmte Methode gebunden. Die Methoden der Architekturentwicklung werden bei den Aktivitäten der Entwicklung und Erweiterung von Produktlinien benutzt (siehe 6.1 und 6.3).

Die Ziele der Methodik können nur erreicht werden, wenn ihre Aktivitäten in einen geeigneten Entwicklungsprozess eingebunden werden. Dazu eignen sich iterativ-inkrementelle Entwicklungsprozesse wie der von EOS (siehe 2.4), der in Teilprozesse auf mehreren Ebenen gegliedert wird (siehe 3.3). Langfristige Ziele erfordern eine Steuerung der softwaretechnischen Methoden durch geeignete Techniken des Projekt- und Qualitätsmanagements. Solche Techniken werden nicht im Rahmen der Methodik ALEXANDRIA entwickelt, in Kap. 9 werden jedoch Schwerpunkte für die Anwendung existierender Techniken genannt, damit die Methodik erfolgreich eingesetzt werden kann.

## 3.3 Entwicklungsprozess

Die Entwicklung und Evolution von Produktlinien erfordert eine Trennung zwischen der Entwicklung der Teile der Produktlinie und der Entwicklung einzelner Produkte auf der Basis der Produktlinie. Aufbauend auf den Methoden FeatuRSEB und ESAPS (siehe Abschnitt 2.1.5) wird deshalb bei der Methodik ALEXANDRIA ein mehrteiliger Entwicklungsprozess vorgesehen. Abb. 16 zeigt eine Übersicht über die Hauptaktivitäten dieses Entwicklungsprozesses. Das Produktlinien-Engineering als wichtigster Teilprozess erfasst Anforderungen an die Produktlinie, strukturiert sie in einem Featuremodell und entwickelt daraus ein System überschneidungsfreier Komponenten, die auf einer Referenzarchitektur beruhen. Als zweiter Teilprozess existiert der Produkt-Engineering-Prozess. Jeder solcher Teilprozess entwickelt ein Produkt. Er nutzt das Featuremodell, die Referenzarchitektur und die Komponenten der wiederverwendbaren Plattform der Produktlinie.

Die Weiterentwicklung und die Einbeziehung existierender Elemente wird durch diese beiden Prozesse nicht unterstützt. Deshalb wird als dritter Teilprozess ein Reengineering-Prozess zur Einbindung existierender Komponenten und Systeme etabliert. Er richtet sich an Featuremodell, Referenzarchitektur und Komponenten der Produktlinie aus und trägt zu ihrer Weiterentwicklung bei.

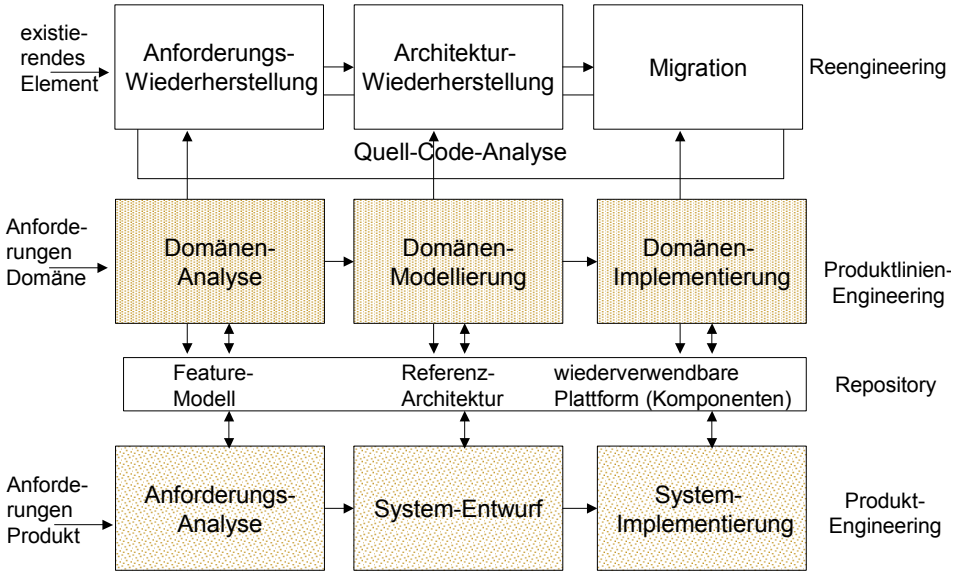



*Abb. 16: Teilprozesse der Produktlinien-Entwicklung bei der Methodik* ALEXANDRIA *im Überblick*

Die wirtschaftlich erfolgreiche Entwicklung und Aufrechterhaltung einer Produktlinie erfordert weitere Aktivitäten der zielgerichteten Steuerung und Beeinflussung des Entwicklungsprozesses, die in Abb. 16 nicht dargestellt sind. Dazu gehören die Maßnahmen des Qualitätsmanagements sowie die Planung und Entscheidungsfindung anhand wirtschaftlicher Kriterien. Die Beiträge von ALEXANDRIA konzentrieren sich zwar auf das Gebiet der Softwaretechnik, zusätzlich werden Lösungen zur Unterstützung der genannten Aktivitäten durch ein geeignetes Management gezeigt und Schwerpunkte gesetzt, die eine Nutzung vorhandener Management-Methoden für Produktlinien ermöglichen. So werden Richtlinien zum Scoping [Riebisch et al. 2001a] und eine Methode zur modellbasierten Entwicklung von Testfällen entwickelt [Riebisch et al. 2002b]. Die Ergebnisse werden in Abschnitt 4.6 und in Kapitel 9 in den Bezug zur Entwicklung und Evolution von Produktlinien gestellt.

### 3.3.1 Produktlinien-Engineering

Der Prozess des Produktlinien-Engineering umfasst die Aktivitäten der Analyse, Modellierung und Implementierung einer wiederverwendbaren Plattform, die gemeinsame Softwareteile für Produkte einer Produktlinie bereitstellt. Dabei stehen die Eigenschaften der Erweiterbarkeit und Wartbarkeit der Produktlinie im Vordergrund. Dies zielt darauf, dass Änderungen mit geringem Aufwand umgesetzt werden können und dass eine lange Lebensdauer der Produktlinie erreicht wird. Dadurch wird die Amortisierung der bei der Entwicklung der Produktlinie geleisteten Investitionen ermöglicht. Außerdem können die Anwendungsprozesse der Domäne auf lange Zeit mit Software-Produkten unterstützt werden.

Der Prozess verläuft iterativ-inkrementell: bei jeder Abfolge der Aktivitäten Analyse, Modellierung und Implementierung werden weitere Komponenten bereitgestellt, Ände-

rungen vorgenommen oder Erweiterungen umgesetzt. Abb. 17 stellt diese Abfolge von Aktivitäten und deren Ergebnisse dar.

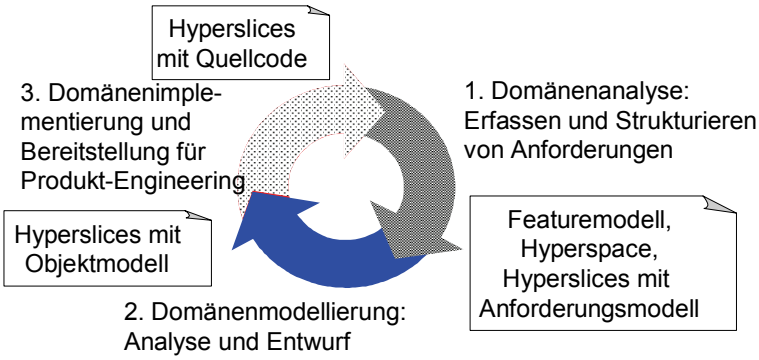



*Abb. 17: Ablauf des Prozesses des Produktlinien-Engineering im Überblick*

**Domänenanalyse**
Die Aktivitäten der Domänenanalyse basieren auf der Methode FeatuRSEB, die um eine Zerlegung nach dem Hyperspace-Ansatz erweitert wurde. Sie enthalten die Erfassung, die Beschreibung und das Strukturieren der Anforderungen an die Produktlinie. Nachdem die Anforderungen mit vorhandenen Techniken der Anforderungs- und Domänenanalyse ermittelt wurden, wird durch einen ersten Scoping-Schritt entschieden, welche Anforderungen aus der Domäne von der Produktlinie erfüllt werden sollen. Dabei wird festgelegt, welche Variabilität die Produktlinie aufweisen soll. Wie in FeatuRSEB werden die Anforderungen daraufhin in gemeinsame und variable Features abgebildet und im Featuremodell dargestellt. Zur Vorbereitung der Entwicklung überschneidungsfreier Komponenten nach dem Hyperspace-Ansatz wird zusätzlich ein Hyperspace definiert, in dem alle variablen Features auf Belange einer Dimension abgebildet werden.

Die Anforderungen werden in einem Anforderungsmodell definiert, das entsprechend den üblichen objektorientierten Methoden aus Use-Case-Modell, Klassenmodell und Schnittstellenmodell bestehen kann [Hitz Kappel 2003]. Die Modelle werden jedoch entsprechend der Features in variable und gemeinsame Teile zerlegt, die getrennt voneinander in Hyperslices gekapselt werden. Dazu wird anstelle von UML als Modellierungssprache Hyper/UML eingesetzt, die in Kapitel 5 vorgestellt wird. Eine detaillierte Beschreibung der Aktivitäten folgt in Abschnitt 4.1 und 4.3. Modelle für die Ableitung von Tests werden in gleicher Weise wie das Anforderungsmodell in Hyperslices zerlegt und stehen für die Komposition von Tests für Produkte zur Verfügung.

**Rolle des Featuremodells**
Das Featuremodell erhält wegen der Feature-gesteuerten Komposition eine besondere Bedeutung im Produktlinien-Engineering und in dessen Kopplung mit Produkt- und Reengineering.

- Es bietet einen Überblick über die Anforderungen an die Produktlinie. Damit unterstützt es die Kommunikation zwischen Kunden und Entwicklern. Die Weiterentwicklung wird durch bessere Verständlichkeit der Auswirkung von Änderungen ebenfalls unterstützt.
- Es unterscheidet zwischen gemeinsamen und variablen Features. Damit steuert es die Zerlegung der Anforderungsmodelle entsprechend dem Hyperspace-Ansatz.

- Es stellt Abhängigkeiten und Wechselwirkungen zwischen Features dar, die Abhängigkeiten zwischen Anforderungen, zwischen Elementen des Entwurfs und zwischen Komponenten der Implementierung wiederspiegeln können.
- Es dient als Ausgangspunkt der Anforderungsanalyse des Produkt-Engineering, weil Features ausgewählt werden und daraufhin eine Anforderungsbeschreibung konfiguriert wird. Aufgrund dieser Auswahl erfolgt anschließend die Komposition des gewünschten Produkts.
- Es wird beim feature-basierten Reverse Engineering (siehe 8.1) angewendet, damit Zusammenhänge zwischen den verschiedenen Abstraktionsebenen Anforderungen, Architektur und Quellcode einfacher erkannt werden können.

Das Featuremodell, seine Weiterentwicklung gegenüber den Vorläufern und seine Anwendung ist Gegenstand des Kapitels 4.

**Domänenmodellierung**
Die Umsetzung der in Hyperslices gekapselten Anforderungsmodelle in Objektmodelle und Architektur unterscheidet sich je nach vorgesehenem Bindungszeitpunkt der Variabilität:

- Für einen frühen Bindungszeitpunkt wird eine Komposition der Modelle nach Hyper/UML vorgenommen. Dazu muss während der Modellierung eine Zerlegung nach Features durchgeführt werden. Für jedes Feature entsteht, entsprechend den Hyperslices im Anforderungsmodell ein Hyperslice des Objektmodells. Dabei bilden die Objektmodelle in Hyper/UML die Referenzarchitektur. Die Modellierungssprache Hyper/UML wird in Kapitel 5 vorgestellt. Die Komposition zum Objektmodell eines Produkts erfolgt durch ein Hyper/UML-Werkzeug. Soweit Dokumentationen erforderlich sind, die nicht aus dem Objektmodell abgeleitet werden können, können diese ebenfalls in Hyperslices zerlegt werden und stehen für eine Komposition ähnlich der des Objektmodells bereit.
- Für einen späten Bindungszeitpunkt, d.h. zur Setup- oder Laufzeit, werden Variationspunkte im Modell mittels Entwurfsmustern und Komponenten realisiert, wie dies in Kapitel 7 vorgestellt wird. Die Konfiguration zu einem Produkt erfolgt mittels eines Konfigurator-Werkzeugs, das Informationen des Featuremodells auswertet.

**Referenzarchitektur**
Das Ergebnis der Modellierung der Produktlinie wird als Referenzarchitektur bezeichnet. Sie beschreibt die Architektur der Produktlinie - als Zusammenwirken von Komponenten, ihren Schnittstellen und den Abhängigkeiten zwischen ihnen - mit Darstellung der Variabilität entsprechend dem Featuremodell. Die Referenzarchitektur realisiert die funktionalen und nichtfunktionalen Anforderungen an die Produktlinie entsprechend Anforderungs- und Featuremodell und enthält dazu Vorgaben für Architekturstil und Komponentenmodell der Komponenten. Je nach benötigten Bindungszeitpunkten existieren für einzelne Features Objektmodelle als Zerlegung in Form von Hyperslices, als Zerlegung in unabhängige Komponenten oder als Objektmodell mit Variationspunkten.

**Domänenimplementierung**
In der Aktivität Domänenimplementierung werden die Komponenten der Produktlinie implementiert. Dabei wird ebenso wie bei der Domänenmodellierung der Bindungszeit-

punkt der einzelnen Features berücksichtigt. Für einen frühen Bindungszeitpunkt werden die Hyperslices der Referenzarchitektur, die Hyper/UML-Modelle enthalten, in Hyperslices mit Quellcode in Hyper/J umgesetzt (siehe Abschnitt 6.1). Ihre spätere Komposition zu möglichen Produkten durch das Hyper/J-Werkzeug ist bereits bei der Implementierung zu beachten.

Bei der Realisierung eines späten Bindungszeitpunkts durch Konfiguration von Komponenten werden die Objektmodell-Komponenten in unabhängige Quellcode-Komponenten umgesetzt. Für die Ableitung eines konkreten Produkts erzeugt ein Konfigurationsgenerator entsprechende Konfigurations-Steueranweisungen, wie etwa in Form von Setup-Files, die dann die Auswahl der benötigten Quellcode-Komponenten steuert.

Für eine Bindung zur Laufzeit ist eine gemeinsame Implementierung der gesamten Produktlinie mit Konfigurationsvariablen erforderlich. Die Konfigurationsvariablen steuern die Variabilität unter Nutzung von Design Patterns, die bei der späteren Komposition in die Implementierung einbezogen werden (siehe 7.1). Zur Ableitung von Produkten durch entsprechende Konfigurations-Steueranweisungen, beispielweise in Form von Config-Files ähnlich denen vieler monolithischer Programmsysteme, existiert ein Konfigurationsgenerator, der auf der Basis der Feature-Konfiguration dieses Produkts die Quellcode-Komponenten zusammenfügt.

Die als wiederverwendbare Plattform bezeichnete Implementierung der Produktlinie besteht aus der Menge der Hyperslices mit Quellcode in Hyper/J, aus einer Menge von implementierten Komponenten oder aus einer gemeinsamen Implementierung mit Variabilität in Form von Konfigurationsvariablen.

Im Repository werden Verweise zwischen Anforderungsmodell, Featuremodell, Referenzarchitektur und den Komponenten als Traceability Links gespeichert. Jeweils ein solcher Link verbindet eine Anforderung mit dem entsprechenden Feature, dem dazugehörigen Architektur- beziehungsweise Objektmodellelement oder dem jeweiligen Element der Implementierung. In der Implementierung kann es sich dabei um eine Komponente, eine Klasse oder eine Operation handeln. Die Traceability Links werden bei Modellierung und Implementierung erstellt und bei Erweiterung der Produktlinie oder der späteren Konfiguration von Produkten verwendet (siehe 4.2).

### 3.3.2 Produkt-Engineering

Der Prozess des Produkt-Engineering ist eng an den Prozess und die Ergebnisse des Produktlinien-Engineering gekoppelt (siehe auch Abb. 16 auf S. 54). Der Prozess existiert für jedes Produkt und verläuft ebenfalls iterativ-inkrementell. Abb. 18 zeigt die Aktivitäten und Ergebnisse im Überblick.

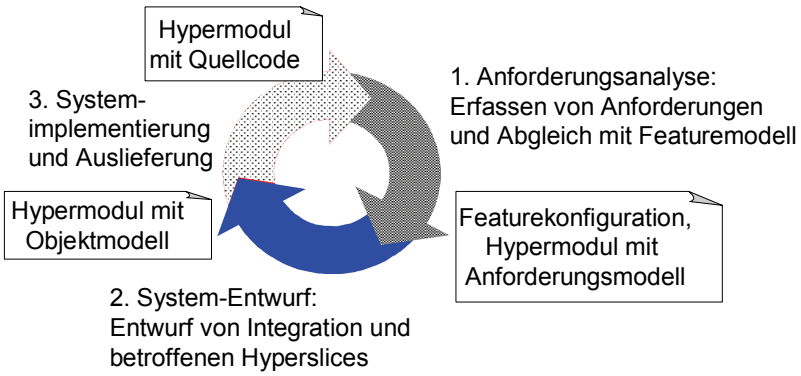



*Abb. 18: Ablauf des Produkt-Engineering im Überblick*

**Anforderungsanalyse**
Die Anforderungsanalyse für ein Produkt aus einer Produktlinie weist gegenüber der konventionellen Anforderungsanalyse die Besonderheit auf, das sie nach zwei Seiten ausrichtet: einerseits werden die Wünsche des Kunden ermittelt und präzisiert, andererseits werden die für die Produktlinie erfassten Anforderungen berücksichtigt. Als wesentliches Kommunikationsmittel wird dabei das Featuremodell verwendet.

Für das zu erzeugende Produkt wird ein Hypermodule angelegt. Zuerst wird entschieden, welche Features des Featuremodells für den Kunden zutreffen. Daraufhin werden aus dem zerlegten Anforderungsmodell der Produktlinie die entsprechenden Hyperslices im Hypermodule zu einem Anforderungsmodell des Produkts mittels Hyper/UML-Komposition zusammengefügt. Für die ausgewählten Features ist zu entscheiden, ob sie im Produkt erst beim Setup oder zur Laufzeit instanziert werden sollen. Die in diesem Schritt ebenfalls zu erstellenden Testfallbeschreibungen für das Produkt können in gleicher Weise durch Komposition erzeugt werden wie das Anforderungsmodell.

Bei der Modellierung der Anforderungen wird von Kunden und Entwickler gemeinsam versucht, eine weitgehende Beschränkung auf vorhandene Möglichkeiten der Produktlinie zu erreichen. Es kann vorkommen, dass sich nicht alle Anforderungen des Kunden auf diese Weise realisieren lassen, zum Beispiel weil einige Anforderungen nicht in dieser Kombination oder überhaupt nicht durch vorhandene Bestandteile der Produktlinie abgedeckt werden. In einem solchen Fall wird mittels Scoping entschieden, ob diese Anforderungen als Ergänzungen nur für dieses Produkt, als Erweiterung der Produktlinie oder überhaupt nicht in Zusammenhang mit der Produktlinie erfüllt werden. Zu ergänzende Anforderungen werden in einem separaten Hyperslice zusammengefasst. Weitere Informationen zu Anforderungsanalyse und Scoping enthält Kapitel 4.

**Systementwurf**
Im Hypermodule des Produkts existiert vom vorangegangenen Schritt bereits das Anforderungsmodell. Entsprechend der ausgewählten Features werden die dazugehörenden Hyperslices des Objektmodells aufgenommen. Die Auswahl der Hyperslices erfolgt anhand der Traceability-Links. Bei der Komposition dieser Hyperslices zum Objektmodell des Produkts wird je nach Bindungszeitpunkt unterschieden, ob Variabilität enthalten sein soll, die erst später zu instanzieren ist.

Für alle Variabilitäten mit frühem Bindungszeitpunkt werden lediglich die ausgewählten Hyperslices integriert. Innerhalb des Hypermodules des zu entwickelnden Produkts werden dazu die Integrationsmethoden definiert, die zur Erzeugung eines Objektmodells in Hyper/UML führen. Für späte Bindungszeitpunkte werden mittels Design Pat-

terns, entsprechenden Komponenten-Schnittstellen oder weiteren Maßnahmen Variationspunkte in das Objektmodell aufgenommen.

Anforderungen außerhalb der Produktlinie werden manuell als Ergänzungen des Hyper/UML-Objektmodells innerhalb eines separaten Hyperslices erstellt und integriert. Die Aktivitäten des Systementwurfs werden in Abschnitt 6.2.2 dargestellt. Die Komposition von Objektmodellen mit Hyper/UML ist Gegenstand von Kapitel 5.

**Systemimplementierung**
Die Implementierung des Objektmodells erfolgt ebenfalls nach dem Hyperspace-Ansatz durch Integration von Hyperslices mit Quellcode beispielsweise in Hyper/J. Die Hyperslices werden anhand der Traceability-Links von Objektmodell und Featuremodell ausgewählt. Für die Implementierung des Produkts wird ein Hypermodule definiert und für die Hyperslices werden geeignete Integrationsmethoden festgelegt.

Soweit Variationspunkte für spätere Bindungszeitpunkte zu integrieren sind, werden Variabilitäten mittels Design Patterns oder anderen Mitteln implementiert. Die Instanzierung dieser Variabilitäten mittels Konfiguration zu einem Produkt erfolgt mittels eines Konfigurator-Werkzeugs, das auf dem Featuremodell basiert. Die Aktivitäten der Implementierung werden in Abschnitt 6.2.3 sowie in Kapitel 7 dargestellt.

### 3.3.3 Reengineering

Produkt- und Produktlinien-Engineering stellen Entwicklungsprozesse dar, die von Anforderungen ausgehen und eine Implementierung zum Ergebnis haben. Bereits vorhandene Implementierungen werden dabei nicht betrachtet. Die Evolution als Weiterentwicklung von Software erfordert deren Überarbeitung, was auch als Reengineering bezeichnet wird. Reengineering wird sowohl bei der Erweiterung einer Produktlinie um neue Anforderungen (siehe 6.3.1) als auch bei der Aufnahme existierender Produkte oder Komponenten (siehe 6.3.2) angewendet.

Beim Reengineering werden Entwicklungsschritte wiederholt, damit Änderungen vorgenommen werden können. Abb. 12 (S. 35) stellt in der obersten Reihe von Aktivitäten das Reengineering in Bezug zu Produkt- und Produktlinien-Engineering.

Bei der *Migration* ist die Struktur existierender Software-Elemente zu verändern und diese sind in den Bestand der wiederverwendbaren Plattform aufzunehmen (siehe 8.3). Dazu sind vorher Entscheidungen über notwendige Veränderungen an Architektur und Anforderungen dieser Software-Elemente zu treffen. Wenn – was oft der Fall ist – Informationen über zugrundeliegende Anforderungen und Architektur nicht ausreichen, damit Komposition und Integration durchgeführt werden können, müssen diese wiedergewonnen werden. Dazu sind Aktivitäten umgekehrt zu normalen Entwicklungsrichtung notwendig, die als Reverse Engineering bezeichnet werden. Für die *Anforderungs-Wiederherstellung* wurde eine Methode entwickelt, die das Featuremodell zum Vergleich mit der Produktlinie und zur Verifikation der ermittelten Anforderungen nutzt (siehe 8.1). Die *Architektur-Wiederherstellung* beinhaltet die Entwicklung einer neuen Architektur und einer Zerlegung, wobei beides vom Hyperspace-Engineering koordiniert wird (siehe 8.2).

## 3.4 Fallbeispiel

Während der Entwicklung der hier vorgestellten Methoden wurden verschiedene Projekte zur Entwicklung von Software-Produktlinien durchgeführt (siehe 10.2). Dabei sollten die Möglichkeiten und Grenzen dieser Methoden bei der praktischen Anwen-

dung analysiert und bewertet werden. Dazu wurden Kooperationsprojekte mit verschiedenen Industriepartnern etabliert, die Softwaresysteme als Produktlinien entwickeln.

Während der Kooperationsprojekte zeigte sich, dass wesentliche Teile der Domänenanalyse-Ergebnisse und der Referenzarchitekturen vertraulichen Charakter haben. Für die jeweiligen Unternehmen sind mit diesen Informationen strategische Entscheidungen für die zukünftige Entwicklung und für den Wettbewerb verbunden, weshalb die Informationen nicht für wissenschaftliche Arbeiten verwendet und nicht veröffentlicht werden dürfen. Deshalb wurden Fallstudien durchgeführt, die nicht direkt aus einem industriellen Umfeld stammen, aber trotzdem in Komplexität, Umfang und Charakter eine Bewertung der Methoden erlauben.

Als eine solche Fallstudie wurde an der TU Ilmenau eine Software-Produktlinie im Rahmen studentischer Arbeiten entwickelt, das Digitale Video-Projekt [DVP]. Gegenstand dieses Projektes ist die Entwicklung eines Digitalen Videosystems als Software-Produktlinie. Diese Produktlinie ist für die Methodenentwicklung geeignet: Seine Komplexität und sein Umfang erreicht eine Größenordnung, bei der sich die wesentlichen Eigenschaften der Methodik ermitteln lassen. Es enthält Variabilität und Komposition, eine Einbindung existierender Komponenten ist erforderlich. Für eine Durchführung im Rahmen studentischer Arbeiten weist diese Produktlinie weitere vorteilhafte Merkmale auf: Sie ist vom Umfang her gerade noch in dieser Organisationsform bearbeitbar; viele benötigten Komponenten sind frei oder mit geringem Aufwand verfügbar; die Ergebnisse sind für die Beteiligten verwendbar, so dass die Entwickler selbst als Kunden in Frage kommen.

Beispiele aus diesem Projekt werden in der vorliegenden Arbeit zur Vorstellung der Methoden verwendet, deshalb wird es in diesem Abschnitt kurz skizziert. Angaben zum Umfang der Produktlinie sind Abschnitt 10.2.4 enthalten.

Die Produktlinie des Digitalen Video-Projekts umfasst Geräte zur Aufnahme und Wiedergabe von Videofilmen. Sie verwendet als Hardware handelsübliche Komponenten für PCs, die zusätzlich zur üblichen Ausstattung eine digitale Satelliten-Empfangseinheit, einen Videosignal-Ausgang zum Anschluss eines Fernsehgerätes sowie eine Infrarotschnittstelle für die Steuerung durch eine Fernbedienung umfassen. Abb. 19 zeigt den Aufbau eines typischen Produkts als UML-Verteilungsdiagramm. Für die Software der Geräte werden das Betriebssystem Linux sowie die Open-Source-Software vdr [Schmidinger 2002] und linuxTV.org [Metzler Metzler 2002] für die Basisfunktionalität eines Videorekorders sowie die DVB-Treiber [Metzler Metzler 2002] für die Übertragung von Video-Datenströmen an weitere Geräte verwendet.

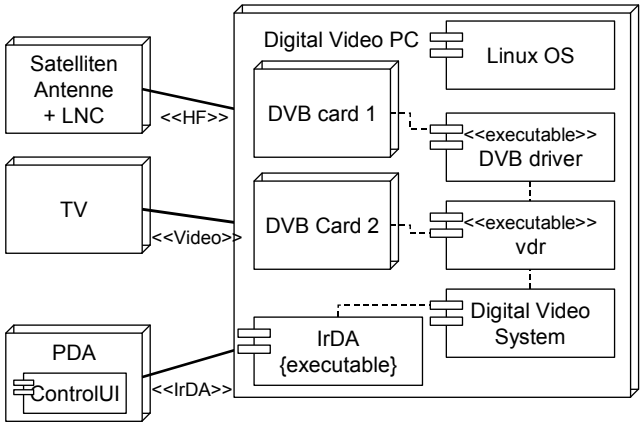



*Abb. 19: Aufbau eines digitalen Videorekorders auf PC-Basis*

Die verschiedenen Videorekorder dieser Produktlinie umfassen als gemeinsame Features die Aufnahme und Speicherung von Videos auf einer Festplatte und alle Funktionen, die ein üblicher Videorekorder aufweist. Als variable Features sind Funktionen zur Nachbearbeitung der Aufzeichnung wie Videoschnitt, Nachvertonung, Ausblenden von Werbung oder Logos verfügbar, weiterhin Funktionen zur Weiterleitung an andere über Netzwerkverbindung angeschlossene Geräte als sog. Video Broadcasting, Funktionen zur komfortablen Steuerung nach Inhalten oder Stichworten sowie zur komfortablen Verwaltung und Wiedergabe gespeicherter Videos ähnlich einer Bibliothek. Die Bedienung ist neben Tastatur und Maus wahlweise über eine Infrarot-Fernbedienung oder über einen Personal Digital Assistent PDA möglich. Die Anzeige von Menüs und Statusinformationen erfolgt durch Einblendung auf dem Fernseh-Bildschirm, auf der Anzeige des PDA sowie in eingeschränkter Form am Gerät. Ein Teil der Features ist in Abb. 23 (S. 70) als Featurediagramm dargestellt. Die variablen Teile der Produktlinie werden als Komponenten implementiert, ein Teil der Variabilität weist einen späten Bindungszeitpunkt auf.

# 4 Steuerung von Wiederverwendung und Evolution mit Featuremodellen

*In diesem Kapitel werden Featuremodelle weiterentwickelt und in die Methodik ALEXANDRIA eingeordnet, damit sie als zentrales Modell bei der Entwicklung und Evolution von Produktlinien eingesetzt werden können. Zur besseren maschinellen Auswertbarkeit wird die Definition der Featuremodelle in Abschnitt 4.1 präzisiert. Die Elemente der Featuremodelle werden mittels Traceability-Links mit anderen Modellen verknüpft. Es wird dargestellt, wie die Featuremodelle bei der Analyse, bei Erweiterungen von Produktlinien sowie bei der Produktentwicklung angewendet werden.*

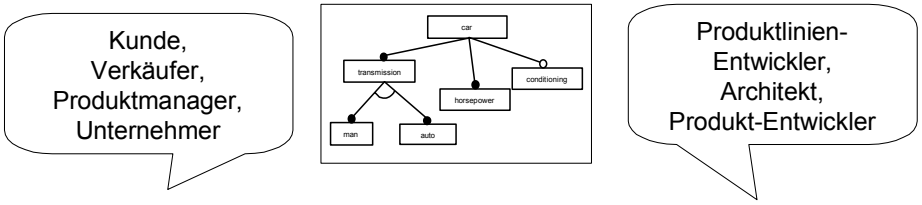
Die Entwicklung von Software-Lösungen für eine spätere Wiederverwendung erfordert die Berücksichtigung zukünftiger Anforderungen. Da zukünftige Anforderungen nicht vorhersagbar sind, versucht die Domänenanalyse die (gegenwärtigen) Anforderungen an alle Systeme einer Domäne zu erfassen und zur Planung zu nutzen. Aufgrund eines ungünstigen Verhältnisses zwischen Aufwand und Nutzen konnte sich die Domänenanalyse jedoch in der Industrie nicht durchsetzen. Die Entwicklung von Softwareproduktlinien verhilft Domänenanalyse-Techniken in eingeschränktem Umfang zum Einsatz. Produktlinien basieren auf der Wiederverwendung von Gemeinsamkeiten von Systemen einer Domäne. Die Produktlinien benötigen hierfür eine Darstellung gemeinsamer und unterschiedlicher Anforderungen, wie sie von Domänenanalyse-Techniken geliefert werden kann. Das Featuremodell hat sich als geeignete Darstellungsform dafür erwiesen, seine Anwendbarkeit leidet allerdings unter ungenauer und teilweise mehrdeutiger Definition von Syntax und Semantik, wie in Abschnitt 2.2.1 und in [Riebisch et al. 2002a] dargestellt ist.

Über die Analyse hinaus unterstützen Featuremodelle im Rahmen der hier entwickelten Methoden eine Vielzahl von Aufgaben bei der Produktlinien-Entwicklung, die in Abb. 20 aufgelistet sind. Dabei soll deutlich werden, dass ein Featuremodell sowohl die Sicht der Kunden auf Anforderungen darstellt, als auch die Sicht der Entwickler unterstützt, weil es Verweise auf die interne Struktur einer Produktlinie und eines Produkts liefert.

Die Anforderungsanalyse für neue Produkte wird an vorhandenen Features ausgerichtet, damit eine möglichst weitgehende Übereinstimmung der Anforderungen mit bereits implementierten Teilen der Produktlinie erzielt wird. Entsprechend der dabei aufgestellten Produkt-Anforderungen werden dann Produkte weitgehend durch Komposition oder Konfiguration erstellt, wie in Abschnitt 6.2 dargestellt wird.

Die Nutzung von Featuremodellen wird bei der Erläuterung verschiedener Aktivitäten und Prozesse deutlich gemacht:

- Modellierung und Zerlegung richten sich an Features aus (siehe Kapitel 6)
- Die Komposition wird durch die Feature-Auswahl dadurch gesteuert, dass das Featuremodell Abhängigkeiten zwischen Features ausdrückt (siehe Kapitel 5 und 6)
- Features stellen für Wartung und Refactoring die Brücke zwischen Anforderungen und Architektur dar (siehe Abschnitt 6.3 und Kapitel 1), weil sie durch Traceability-Links verbunden sind (siehe Abschnitt 4.2)
- Die Planung der Weiterentwicklung der Produktlinie wird mittels Scoping anhand des Featuremodells gesteuert (siehe Abschnitt 4.6)




- Überblick über die Anforderungen
- Unterscheidung zwischen gemeinsamen und variablen Features
- Darstellung von Abhängigkeiten zwischen Features
- Auswahl von Features zur Definition neuer Produkte
- Orientierung und Entscheidung der Weiterentwicklung der Produktlinie an Features
- Gegenwert von Investitionen: Optionen für zukünftige Produkte

- Definition des Hyperspace aufgrund variabler Features
- *Separation of Concerns* aufgrund variabler Features bei Zerlegung von Anforderungs- und Objektmodell sowie Quellcode
- Eindeutige Zuordnung zwischen Features und Hyperslices der Produktlinie
- Darstellung von Abhängigkeiten zwischen Hyperslices im Featuremodell
- Steuerung der Komposition eines Produkts durch eine Konfiguration von Features

*Abb. 20: Aufgaben, bei denen das Featuremodell die Entwicklung von Produktlinien unterstützt*

## 4.1 Modellierung von Features und ihren Beziehungen in Featuremodellen

Die starke Einbindung des Featuremodells in die produktlinienbasierte Entwicklung erfordert eine Behebung der in Abschnitt 2.2.1 genannten Mängel und die Entwicklung einer Darstellung, die einer methodischen und maschinellen Auswertung zugänglich ist. Für die dementsprechende Weiterentwicklung des Featuremodells sind folgende Forderungen zu stellen:

- Für Features und die Beziehungen – insbesondere die Hierarchiebeziehung – und die Abhängigkeiten zwischen ihnen sind Syntax und Semantik präzise zu definieren, damit sie eindeutig erstellt und mittels Werkzeugen ausgewertet und gepflegt werden können
- Bei der Gruppierung von Features bestehende Mehrdeutigkeiten sind zu beseitigen
- Für die Erstellung von Featuremodellen ist eine methodische Vorgehensweise aufzustellen
- Für Beziehungen zu Elementen anderer Modelle wie Anforderungsmodell, Objektmodell sind Syntax und Semantik präzise zu definieren

### 4.1.1 Definition von Feature und Featuremodell

Vergleiche von Featuremodellen aus der praktischen Anwendung zeigten, dass es unterschiedliche Typen von Features gibt, die eine differenzierte Behandlung erfordern. Dadurch entstehen sehr häufig mehrdeutige Darstellungen, die trotzdem den meisten vorhandenen Definitionen entsprechen. Die Definition für Features bei Czarnecki und Eisenecker lautet beispielsweise: „A feature is a property of a domain concept, which is relevant to some domain stakeholder and is used to discriminate between concept instances“ ([Czarnecki et al. 2000] S. 755). Die Gemeinsamkeit mit verschiedenen weiteren Definitionen über Feature und Featuremodelle in anderen Publikationen besteht darin, dass ein Feature eine Produkteigenschaft ausdrückt, die ein Kunde zur Unterscheidung zwischen zwei Produkten einer Domäne nutzen kann und für die er bereit ist,

einen (Mehr-)Preis zu bezahlen. Die Definition in [Böllert 2002] ergänzt die Unterscheidung zwischen gemeinsamen und variablen Features:

> Ein Feature „ist eine Eigenschaft einer Produktlinie aus Sicht der Kunden. Ein gemeinsames .. (Feature) ist eine Eigenschaft, die alle Produkte einer Produktlinie aufweisen. Ein variables ... (Feature) ist eine Eigenschaft, worin sich Produkte einer Produktlinie unterscheiden können."

Während der industriellen Anwendung von Featuremodellen in verschiedenen Domänen wurden unterschiedliche Typen von Features zusammengetragen: Relevante Produkteigenschaften können in Funktionen, Prozessschritten und Datenelementen bestehen. Es kann sich aber auch um Schnittstellen, die einen Standard abdecken, oder um von Dritten bereitgestellte Komponenten handeln, die einen speziellen Kundennutzen versprechen. Auch nichtfunktionale Eigenschaften wie Mengen-, Geschwindigkeits- oder Qualitätsmerkmale können als Feature wichtig sein, ebenso wie Möglichkeiten späterer Erweiterung oder Flexibilität. Als Ergebnis dieser Untersuchungen wird folgende konkretere Definition formuliert:

**Definition**: Ein *Feature* repräsentiert ein Produktmerkmal, das wertvoll für Kunden ist und zur Unterscheidung zwischen verschiedenen Produkten einer Domäne genutzt wird. Es wird durch einen einzelnen Ausdruck beschrieben. Ein Merkmal aller Produkte einer Produktlinie wird als *gemeinsames Feature* (engl.: mandatory feature) bezeichnet. Ein Merkmal, in dem sich Produkte einer Produktlinie unterscheiden, ist ein *variables Feature* (engl.: variable feature, optional feature). Es gibt drei Kategorien von Features:

- *Funktionale Features* (engl.: functional features) beschreiben Verhaltens-Aspekte oder die Art und Weise der Interaktion mit einem Produkt.
- *Schnittstellen-Features* (engl.: interface features) drücken die Konformität eines Produkts mit einer vorgegebenen oder Standard-Schnittstelle aus oder bezeichnen ein Subsystem.
- *Parameter-Features* (engl.: parameter features) beschreiben Eigenschaften in Bezug auf die Umgebung, die quantitativ oder als Aufzählung angebbar sind, oder nichtfunktionale Eigenschaften.

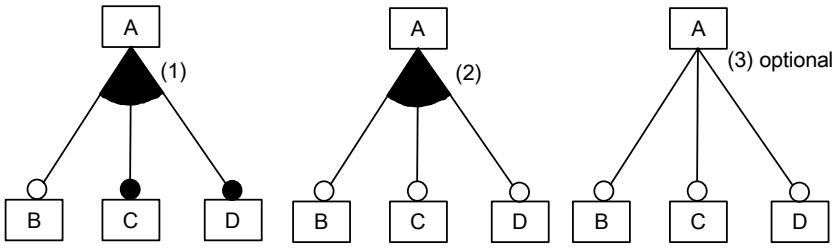
Ein variables Feature wird ergänzend dadurch beschrieben, zu welchem Zeitpunkt der Produkterstellung oder Benutzung die Entscheidung für das Aktivieren dieser Eigenschaft getroffen wird (sog. *Bindungszeitpunkt*, engl.: binding time). Statt eines variable Features der genannten Kategorien kann auch lediglich eine *Option* auf spätere Entscheidung für einen Kunden wertvoll sein und deshalb als Feature modelliert werden.

Bezeichnungen für Features müssen aus Kundensicht gewählt werden. Dabei kann es sich um sehr detaillierte oder sehr technisch orientierte Bezeichnungen handeln, wenn eine Produktauswahl nach solchen technischen Merkmalen erfolgt. Beispiele dafür sind Features wie „FireWire-Schnittstelle nach IEEE 1394" einer Produktlinie für Digitalkameras oder „DDR266 RAM" für PCs. Über die Bezeichnung wird eine Beziehung zu einem Kontext hergestellt.

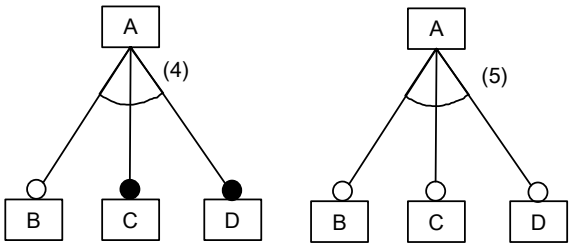
**Beziehungen zwischen Features**

Bei der Definition von Syntax und Semantik der Beziehungen zwischen Features nach [Czarnecki et al. 2000] bestehen Mehrdeutigkeiten. Diese betreffen die Gruppierung von Hierarchiebeziehungen durch Alternative, OR oder XOR. Entsprechende Fallunter-

scheidungen und Normierungen sind in [Riebisch et al. 2002a] angegeben, Abb. 21 zeigt ein Beispiel mit der Symbolik nach Abb. 8, S. 25.




Identische Sachverhalte in verschiedenen Darstellungen mit der Gruppierung „OR“




Identische Sachverhalte in verschiedenen Darstellungen mit der Gruppierung „Alternative“

*Abb. 21: Beispiele für Mehrdeutigkeiten bei der graphischen Darstellung nach [Czarnecki et al. 2000]*

Zur Vermeidung dieser Mehrdeutigkeiten und zur Erhöhung der Verständlichkeit bei der Gruppierung zwischen variablen Features werden Vielfachheiten eingeführt, wie sie in ähnlicher Form in Klassendiagrammen der UML und in Entity-Relationship-Modellen ERM üblich sind. Eine Kennzeichnung der Gruppierung als Alternative, XOR und OR ist durch Angabe einer Vielfachheit sowie durch Kennzeichnung der gruppierten Features als gemeinsam und variabel möglich. Abb. 22 (Seite 70) zeigt im rechten Teil eine graphische Darstellung dieser neu eingeführten Gruppierung. Ein Beispiel der Anwendung der Gruppierung mit einer Vielfachheit ist im rechten Teil von Abb. 23 (Seite 70) enthalten. Dort wird eine Gruppierung mit Vielfachheit „1“ genutzt, die eine alternative Auswahlmöglichkeit der Features `Netzwerkanschluss` und `Modem` ausdrückt.

Bei der Untersuchung der Hierarchiebeziehungen in Featuremodellen im Rahmen der in 10.2 genannten industriellen Projekte wurde deutlich, dass die Featuremodelle vorrangig für die Entscheidung über Produkte genutzt werden müssen. Die Anordnung eines Features in der Hierarchie drückt aus, wie stark sein Einfluss auf die weiteren Entscheidungen über andere Features eines Produktes ist; Features mit großem Einfluss beispielweise auf die Architektur von Produkten sind weit oben in der Hierarchie anzuordnen. Die Hierarchie wird damit aus Sicht des Anbieters und Verkäufers eines Produkts gestaltet und gibt die Entscheidungsfolge wieder. Für den Kunden spielt die Featurehierarchie meist keine Rolle; wie in den durchgeführten Projekten und Fallstudien (siehe 10.2) deutlich geworden ist, liegen seine Anforderungen an ein Produkt in den meisten Fällen als Liste von Features vor.

Die verschiedenen Kategorien von Beziehungen zwischen Features und die besondere Bedeutung der Entscheidungsfolge für die Gestaltung der Hierarchie wird in einer neuen Definition des Featuremodells ausgedrückt. Diese Definition widerspiegelt die Rolle von Featuremodellen bei der Entwicklung und Anwendung von Produktlinien:

**Definition**: Ein *Featuremodell* stellt Anforderungen im Überblick dar und modelliert die Variabilität einer Produktlinie. Es wird für die Definition eines Produkts durch Kunden genutzt. Ein Featuremodell ist ein Graph mit Features als Knoten und Featurebeziehungen als Kanten. Features werden durch Hierarchiebeziehungen geordnet, zusätzlich zu denen weitere Beziehungen existieren. Folgende Kategorien von Beziehungen können unterschieden werden:

- *Hierarchiebeziehungen* dienen zur Steuerung des Entscheidungsprozesses über neue Produkte durch Kunden. Die Hierarchie widerspiegelt die *Entscheidungsfolge* bei der Definition von Produkten. Ein Feature wird um so näher an der Wurzel einer Hierarchie angeordnet, je wichtiger es für die Entscheidung über andere Features ist, und je höher sein Einfluss auf die Architektur der Produktlinie ist.
- Beziehungen der Generalisierung und Spezialisierung sowie der Aggregation von Features werden durch die *Verfeinerungsbeziehung* (eng.: refinement relation) ausgedrückt. Wenn eine solche Beziehung mit einer Hierarchiebeziehung zusammenfällt, wird sie in der graphischen Darstellung unterdrückt.
- Abhängigkeiten zwischen variablen Features, die eine Auswahl beeinflussen, werden durch require-, exclude- oder als *Gruppierungsbeziehung* beschrieben. Eine require-Beziehung erfordert die gleichzeitige Auswahl eines anderen variablen Features, eine exclude-Beziehung schließt diese aus. Wenn eine require-Beziehung mit einer Hierarchiebeziehung zusammenfällt, wird sie in der graphischen Darstellung unterdrückt. Eine Gruppierung beschreibt durch Vielfachheiten Auswahlmöglichkeiten von Features mit gleichem Vorgänger in der Hierarchie: „0..1“ erlaubt die Auswahl von keinem oder einem Feature der Gruppe, „1“ erfordert die Auswahl von genau einem (Alternative), „1..*“ erfordert die Auswahl von mindestens einem Feature, mehrere sind möglich. Abwandlungen wie „2..5“ sind ebenso möglich [Riebisch et al. 2002a]. Für die Beschreibung komplizierterer Abhängigkeiten werden require- oder exclude-Beziehungen mit formalen Ausdrücken beschrieben, die in einer graphischen Darstellung hinterlegt werden.

Wenn für den Aufbau einer Hierarchie Zwischenknoten sinnvoll sind, werden diese als abstrakte Features, sog. *Konzeptknoten* (engl.: concept) eingefügt, denen keine konkrete Implementierung eines Features entspricht. Der Wurzelknoten einer Hierarchie ist immer ein Konzeptknoten.

Features und Beziehungen werden durch weitere ergänzende Informationen präzisiert, die einer formalen Syntax und Semantik folgen sollten, jedoch auch mit natürlicher Sprache ausgedrückt werden können. Features werden außerdem durch sog. *Traceability-Links* mit Elementen anderer Modelle verbunden (siehe 4.2). Traceability-Links des Typs „ImplementedBy“ verbinden Features mit dazugehörigen Modell- und Implementierungs-Elementen, während Trace-Links des Typs „RequiredBy“ Features mit Anforderungen (in Use-Case-Modellen, Klassenmodellen, Interface-Modellen) verbinden. Für variable Features sind *Vorgaben* für Auswahl und Bindungszeitpunkt für den Fall zu hinterlegen, dass durch Kunden bei der Produktdefinition keine Entscheidung getroffen wird.

In Abschnitt 4.1.3 werden die Darstellungen unterschiedlicher Fälle von Featurebeziehungen im Featuremodell untersucht. Dabei wird verdeutlicht, wie im Featuremodell

die Entscheidungsfolge als wichtigste Beziehung behandelt wird, der alle übrigen Kategorien von Featurebeziehungen untergeordnet werden.

**Beschreibung mittels Constraint Language**

Die Featuremodelle und insbesondere die Beziehungen zwischen Features sollen durch Methoden und Werkzeuge ausgewertet werden, damit Abhängigkeiten zwischen Features visualisiert, damit Produkte als Konfigurationen von Features definiert, damit diese Konfigurationen auf ihre Gültigkeit geprüft und damit für Parameter-Features gültige Wertebereiche der Parameter in Abhängigkeit von anderen Features ermittelt werden können. Für UML-Modelle werden für ähnliche Zwecke Beschreibungen von Beziehungen zwischen Modellelementen mit Ausdrücken der Object Constraint Language OCL, einem standardisierten Bestandteil der UML, vorgenommen [UML 2001]. Solche Ausdrücke können von vielen UML-Modellierungswerkzeugen ausgewertet werden.

Ausdrücke mit OCL wären gut zur hier benötigten Beschreibung von Beziehungen geeignet. Dazu ist die OCL jedoch zu erweitern, weil sie wegen ihrer Beschränkung auf UML keinen Zugriff auf Featuremodelle und Features erlaubt. Eine derartige Erweiterung wird von Streitferdt [Streitferdt 2003] definiert. Mit den Ausdrücken in dieser OCL-Erweiterung werden Regeln und Bedingungen in Form von Invarianten ausgedrückt. Solche Ausdrücke werden für alle Beziehungen der oben angegebenen Definitionen definiert. Als Beispiel werden hier Beschreibungen für eine require- und eine exclude-Beziehung von einem Feature F1 zu F2 in den beiden folgenden Ausdrücken angegeben. Der Ausdruck `selected()` bewertet dabei, ob ein Feature einer Produktlinie für ein Produkt ausgewählt wurde:

```
context F1, F2
inv exclude:
   (selected(F1) implies not selected(F2))

context F1, F2
inv require:
   (selected(F1) implies selected(F2))
```

Zusätzlich zu den Beziehungen der oben angegebenen Definitionen wird eine *erweiterte Beziehung zwischen Features* definiert, die direkt einen OCL-Ausdruck oder eine natürlich-sprachliche Erläuterung enthalten kann. Diese Beziehung wird insbesondere für kompliziertere Abhängigkeiten zwischen mehreren Features oder für die Ermittlung von Parameter-Wertebereichen für Parameter-Features angewendet.

Im Beispiel drückt sie die Gültigkeit des Parameter-Features ServerRAMsize für die Größe des RAM-Speichers eines Produkts der Produktlinie Digitaler Videorecorder aus, die von der Anzahl von Clients für Video-on-Demand abhängt:

```
context ServerRAMsize:Number, ClientPC:Number
inv mathematical:
    ServerRAMsize:Number = ClientPC:Number * 256 + 256
```

Ausdrücke in dieser OCL-Erweiterung werden dem graphischen Featuremodell und seinen Beziehungen hinterlegt. Sie werden durch die Werkzeuge (siehe 4.7 und 6.4) ausgewertet, während beispielsweise Features für neue Produkte ausgewählt, Produkte durch Komposition erstellt oder Abhängigkeiten zwischen Features untersucht und bearbeitet werden. Die Ausdrücke stellen eine Präzisierung der graphisch oder natürlichsprachlich angegebenen Beziehungen in einem Featuremodell dar. Sie werden meist

von Entwicklern erstellt, für einige Fälle ist auch eine Generierung durch Werkzeuge möglich. Dem Aufwand für ihre Erstellung steht der Nutzen bei der werkzeuggestützten Prüfung von Produkt-Konfigurationen und bei den weiteren Auswertungen von Featuremodellen gegenüber.

### 4.1.2 Graphische Darstellung in Featurediagrammen

Zwecks besserer Verständlichkeit werden Featuremodelle bereits bei FODA und FeatuRSEB graphisch dargestellt, wie in Abb. 8 (S. 25) an einem Beispiel und der Legende gezeigt wurde. Für die Erweiterungen der oben angegebenen Definition müssen die graphischen Darstellungen um neue Elemente erweitert werden.

Bei der Festlegung neuer graphischer Elemente sind die Prinzipien für ergonomische Darstellung bezüglich Übersichtlichkeit und Verständlichkeit anzuwenden, wie sie unter anderem bei [HCI] zusammengestellt wurden. Außerdem sind spezielle Anforderungen zu berücksichtigen, die sich aus dem Charakter der Featuremodelle ergeben:

- Es ist eine weitgehende Ähnlichkeit zu verwandten graphischen Ausdrucksmitteln wie UML und ERM anzustreben. Damit wird eine gute Akzeptanz und ein geringer Einarbeitungsaufwand erreicht. Es darf jedoch nicht zur Definition einer geänderten Semantik für bekannte Elemente kommen, weil dies die Verständlichkeit beeinträchtigen würde.
- Syntax und Semantik der graphischen Darstellungselemente sollen die formale Definition der Modellelemente selbst unterstützen und Mehrdeutigkeiten vermeiden.
- Unvollständige Informationen müssen darstellbar sein, da Featuremodelle Anforderungen abbilden, die zumindest während des Erstellungsprozesses teilweise unvollständig sind.
- Stufenweises Ausblenden von Teilen der Informationen soll Arbeiten auf unterschiedlichen Niveaus von Detaillierung und Vollständigkeit unterstützen.

Abb. 22 zeigt die graphische Darstellung der oben definierten Featuremodell-Elemente. Hierarchiebeziehungen und allgemeine Features mit einer Kennzeichnung als gemeinsam und variabel sollten in jedem Diagramm dargestellt werden. Zur schrittweisen Anreicherung mit Details kann optional die Angabe der Feature-Kategorie, die Hervorhebung von Konzept-Features, die Angabe von Gruppierungen und Abhängigkeiten, wie `require` und `exclude` sowie von Verfeinerungsbeziehungen erfolgen. Eine hierarchische Verfeinerung in Unter-Diagrammen ist an Konzept-Features möglich. Für die Gruppierung folgt die Vielfachheit `m..n` der oben angegebenen Definition in Anlehnung an die UML. Weitergehende Informationen können als Notiz zu Features und zu Beziehungen angegeben werden. In graphischen Editoren sind diese Informationen über das Notiz-Symbol erreichbar. Hier sind natürlich-sprachliche Texte möglich. Insbesondere zur Beschreibung von Abhängigkeiten fehlt die maschinelle Auswertbarkeit solcher Texte. Deshalb werden hierfür OCL-Ausdrücke eingeführt, die eine maschinelle Prüfung dieser Abhängigkeiten erlauben.

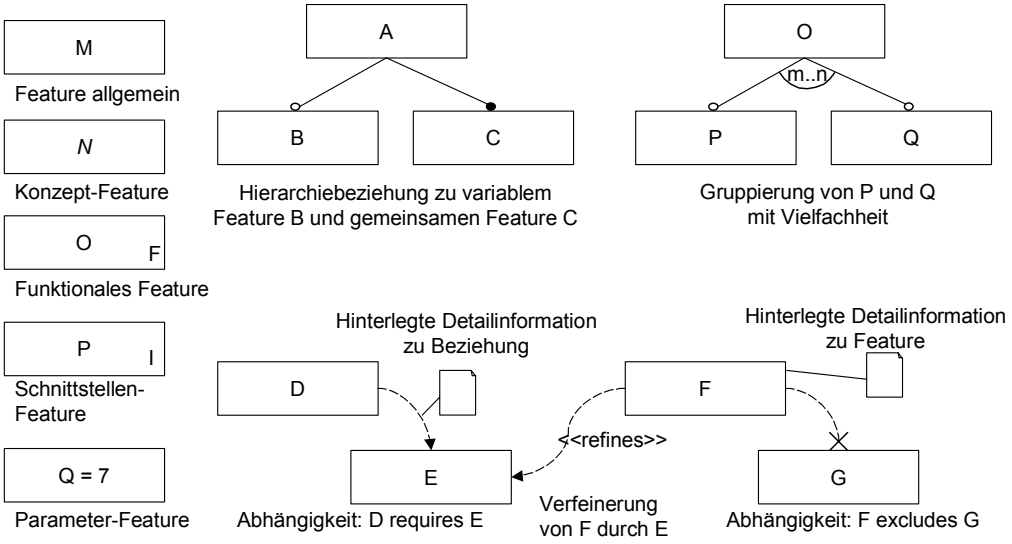



*Abb. 22: Graphische Darstellungselemente für Featuremodelle*

Eine graphische Darstellung von Traceability-Links „ImplementedBy“ und „Required-By“ ist nicht vorgesehen, da diese eine Verbindung zu anderen Diagrammen herstellen.

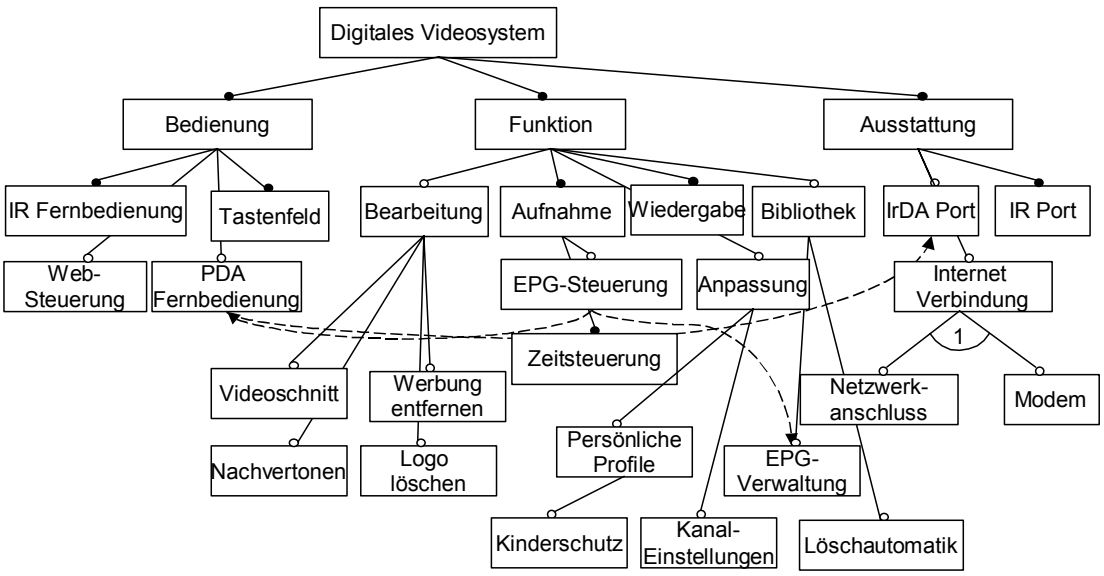



*Abb. 23: Featuremodell des Digitalen Videosystems (Ausschnitt)*

Für die Verwendung und Pflege von Featuremodellen ist die Nutzung graphischer Editoren sinnvoll, auf die in Abschnitt 4.7 eingegangen wird. Besondere Vorteile ergeben sich bei Aufteilung von Featuremodellen in hierarchisch gegliederte Diagramme und durch schrittweises Ein- und Ausblenden von untergeordneten Teilen einer Feature-Hierarchie.

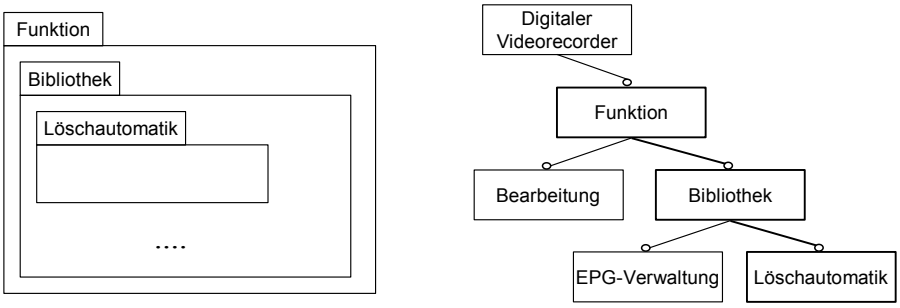
### 4.1.3 Wechselwirkungen zwischen Feature-Beziehungen in Featuremodellen

Die Definition des Featuremodells in Abschnitt 4.1.1 hebt die Darstellung der Entscheidungsfolge im Featuremodell gegenüber den anderen Beziehungen hervor, weil das Featuremodell vorrangig zur Entscheidung über neue Produkte im Rahmen einer Pro-

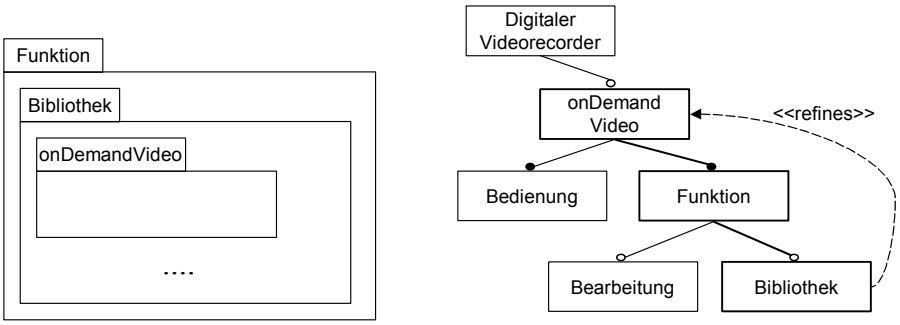
duktlinie verwendet wird. Die Modellierung der weiteren Beziehungen wird für die Weiterentwicklung der Produktlinie und für Entwicklungsentscheidungen benötigt, ihre graphische Darstellung wird der der Entscheidungsfolge untergeordnet. Gemäß Definition von 4.1.1 werden diese weiteren Beziehungen zusätzlich zur Hierarchie - und damit außerhalb von ihr - dargestellt.

Eine Verfeinerungsbeziehung zwischen Features, die mit der Entscheidungsfolge übereinstimmt, wird in der graphischen Darstellung nicht separat gezeichnet. Sie kann jedoch trotzdem im Repository gespeichert werden, damit sie für Analyse und Auswertung zur Verfügung steht. In der graphischen Darstellung wird in einem solchen Fall nur die Hierarchiebeziehung im Featuremodell dargestellt. Der obere Teil von Abb. 24 stellt diesen Fall als Verfeinerung von Paketen und die resultierende Darstellung im Featuremodell dar, wobei sich das Beispiel auf Ausschnitte des Featuremodells aus Abb. 23 (S. 70) bezieht. Besteht zwischen Verfeinerungsbeziehung und Entscheidungsfolge ein Unterschied, wird nur die Entscheidungsfolge zwischen Features durch die Hierarchiebeziehung ausgedrückt. Die Verfeinerungsbeziehung wird durch eine zusätzliche Beziehung außerhalb der Feature-Hierarchie beschrieben, die entsprechend der Abb. 22 (S. 70) als *refines*-Beziehung graphisch dargestellt wird. Die Entscheidungsfolge wird damit in der graphischen Darstellung durch das wichtigere Ausdrucksmittel wiedergegeben, weil deren Bedeutung für die Anwendung des Featuremodells größer ist als die der Verfeinerung. In Abb. 24 wird im Fall 2 das Feature *onDemand Video* wegen seiner großen Bedeutung für die nachfolgenden Entscheidungen weit oben in der Hierarchie des Featuremodells angeordnet, obwohl dieses Feature eine Verfeinerung - hier ein Teil - des Features *Bibliothek* darstellt. Die Verfeinerungsbeziehung von *Bibliothek* zu *onDemand Video* wird im Featuremodell graphisch nur durch eine ergänzende *refines*-Beziehung ausgedrückt. In vielen Fällen wird die *refines*-Beziehung wegen ihrer geringeren Bedeutung in der graphischen Darstellung ausgeblendet oder weggelassen.

Zusätzlich zu den bisher genannten Abhängigkeiten zwischen Features können gegenseitige Beeinflussungen des Verhaltens bestehen (engl. Feature Interaction), wenn Features gleichzeitig vorhanden sind. Der Unterschied besteht darin, dass Feature Interaction zur Abhängigkeit der *Verhaltensspezifikation* eines Features vom Vorhandensein anderer führt, während Abhängigkeit in der oben beschriebenen Weise die *Zulässigkeit* eines Features beschreibt. Es existieren eine Reihe von Arbeiten zum Thema Feature Interaction (siehe 2.2.1), die die Erkennung und Vermeidung solcher Beeinflussungen zum Ziel haben.




**Fall 1**: Übereinstimmung zwischen Verfeinerungsbeziehung und Entscheidungsfolge




**Fall 2**: Unterschied zwischen Entscheidungsfolge und Verfeinerungsbeziehung

*Abb. 24: Darstellung von Unterschieden zwischen Entscheidungsfolge und im Featuremodell*

In der vorliegenden Arbeit werden Features zur Zerlegung in Belange nach dem Hyperspace-Ansatz genutzt, wobei eine Dimension des Hyperspace dazu entsprechend den Features in Belange gegliedert wird. Voraussetzung dafür ist, dass die Features sich in ihrer Spezifikation nicht gegenseitig beeinflussen. Durch die Auflösung eventuell vorhandener Feature Interaction muss diese Voraussetzung vor der Zerlegung geschaffen werden. Gemäß [Lorentsen et al. 2002] existieren drei Typen von Wechselwirkungen, die sich auch überlagern können:

1. Ein Feature benutzt ein anderes – Use Interactions
2. Gemeinsamer Zugriff von Features auf eine (begrenzt vorhandene) Ressource, höherpriorisierte Features entziehen anderen die Ressource
3. Ein Feature verhindert ein anderes, beispielweise durch Blockieren der Ausführung

Wechselwirkungen von Typ 2 können oft durch exclude-Abhängigkeiten der Implementierung beschrieben werden. In allen anderen Fällen müssen derartige Wechselwirkungen aus den Spezifikationen der Features entfernt werden, beispielsweise durch Entwicklung einer geeigneten Service-Architektur. Tritt bei der Entwicklung des Featuremodells oder bei der Verfeinerung der Verhaltensbeschreibung eine gegenseitige Beeinflussung mit anderen Features auf, sind entsprechende Ansätze zur Feature Interaction anzuwenden, bevor mit der Aufstellung des Featuremodells fortgefahren werden kann. In den meisten Fällen führt eine weitere Zerlegung der betreffenden Features zur Auflösung dieser Beeinflussungen, wie in Abschnitt 4.4.1 dargestellt wird.

## 4.2 Verknüpfung von Modellen mittels Traceability-Links

Der Aufwand für das Verstehen einer Software durch den Entwickler stellt einen großen Teil am Gesamtaufwand für die Durchführung von Änderungen an dieser Software dar (siehe 1.1). Die Beschreibung von Beziehungen und Abhängigkeiten innerhalb der spielt dafür eine wichtige Rolle: sie sind erforderlich für das Verstehen der Auswirkun-

gen einer Änderung und für das Identifizieren der zu verändernden Elemente. Die Verständlichkeit von Software – definiert durch das Qualitätsmerkmal Klarheit [DIN 66272] – hängt stark davon ab, wie gut ein Entwickler Bezüge zwischen verschiedenen Abstraktionen herstellen kann, beispielsweise zwischen einer Anforderung, einem Element einer Architektur und einer implementierten Komponente.

Traceability-Links werden im Requirements Engineering verwendet, um Beziehungen zwischen verschieden abstrakten Wissensebenen und Anforderungen auszudrücken (siehe 2.5). Diese Beziehungen werden vom Entwickler erkannt oder hergestellt. Viele Konzepte für Repositories von CASE-Tools enthalten ähnliche Links für die Verknüpfung zwischen Modellelementen. Diese Konzepte sind sehr gut für eine Verknüpfung zwischen Anforderungen, Features, Entwurf und Implementierung geeignet, damit die oben genannten Forderungen erfüllt werden können. Zusätzlich zu den bereits genannten Zielen sollen die Traceability-Links die folgenden Aktivitäten unterstützen:

- Werkzeuggestützte Aktualisierung von Dokumenten nach Änderungen
- Abschätzung des Aufwands von Änderungen
- Prüfung der vollständigen Durchführung von Änderungen
- Generieren von Produkten als Konfigurationen von (variablen) Komponenten
- Prüfen der Gültigkeit von Konfigurationen
- Einbinden des Featuremodells als Kommunikationsmittel zwischen Anforderungen und Architektur

**Umsetzung**

Solche Traceability-Links sind in der Definition der Beziehungen des Featuremodells enthalten (siehe 4.1.1) Ihre Umsetzung innerhalb des Repositories einer Entwicklungsumgebung beziehungsweise einer Produktlinie erfordert das Herstellen von Beziehungen zwischen Elementen der verschiedenen Modelle. Für eine Verknüpfung von Modellelementen mit der Implementierung wurde ein Ansatz entwickelt, der auf Referenzen von Dokumentationswerkzeugen beruht [Sametinger Riebisch 2002]. Beziehungen zwischen Elementen verschiedener Modelle werden innerhalb des Repositories unter Nutzung von XML hergestellt [Streitferdt 2003], indem Ausdrucksmittel wie `xpath` und `xpointer` benutzt werden. Abb. 25 zeigt eine solche Verknüpfung von Elementen verschiedener Modelle am Beispiel.

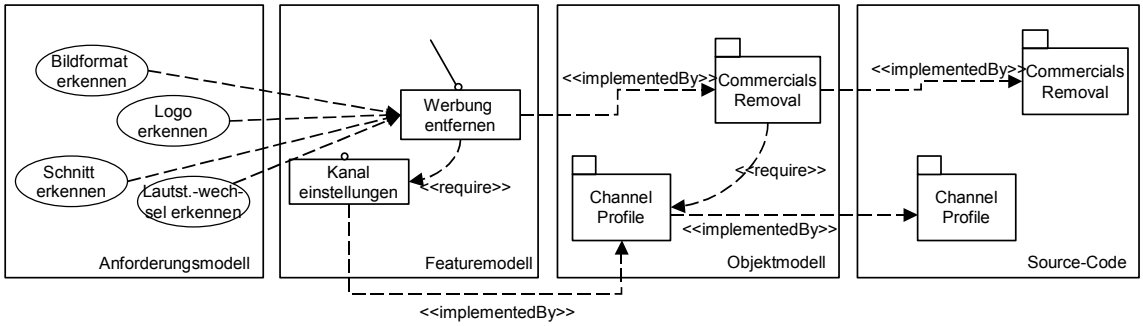



*Abb. 25: Darstellung einer Abhängigkeit zwischen variablen Entwurfs-Teilen (Objektmodell) als require-Beziehung zwischen den entsprechenden Features (Featuremodell)*

**Projektion von Abhängigkeiten**

Traceability-Links können bei der Definition neuer Produkte und bei der Weiterentwicklung einer Produktlinie dazu benutzt werden, Abhängigkeiten zwischen variablen Features auszuwerten. Dabei kann es sich zum Beispiel um *require*- oder *exclude*-

Abhängigkeiten handeln (siehe 4.1.1). Sie resultieren häufig aus Anforderungen an eine Produktlinie, oft sind sie jedoch auch durch Unvereinbarkeiten oder Abhängigkeiten zwischen Objektmodell-Elementen oder zwischen Elementen der Implementierung verursacht.

Das Beispiel in Abb. 25 zeigt im Featuremodell eine *require*-Abhängigkeit vom Feature *Werbung entfernen* zum Feature *Kanaleinstellungen*. Diese Abhängigkeit entstammt nicht den Anforderungen, sondern der *require*-Abhängigkeit zwischen den entsprechenden Komponenten des Objektmodells. Sie wird in das Featuremodell projiziert. In diesem Beispiel resultiert die *require*-Beziehung daraus, dass der Entwurf der Klassen für das Entfernen von Werbeeinblendungen in Videosequenzen auf Klassen im Objektmodell des Senders, wie Bildformat, Senderlogo und ähnliches Bezug nimmt.

Eine solche Projektion von Modell oder Implementierung in das Featuremodell wird benötigt, damit eine Entscheidung für ein Produkt anhand einer Auswahl von Features getroffen werden kann und damit die Gültigkeit von Feature-Konfigurationen für Produkte geprüft werden kann. Diese Projektion wird durch die Traceability-Links ermöglicht, die eine Verbindung zwischen den Features und den dazugehörigen Elementen in Objektmodell und Implementierung herstellen.

In diesem Fall wird die *require*-Abhängigkeit zwar zwischen zwei Features im Featuremodell dargestellt, doch sie spiegelt nur die Abhängigkeit der entsprechenden Modell- oder Implementierungs-Komponenten wieder. In ähnlicher Weise werden auch Abhängigkeiten zwischen den Hyperslices der Implementierung im Featuremodell wiedergespiegelt. Je nach ihrer Quelle werden die Abhängigkeiten unterschiedlich behandelt:

- Die Abhängigkeit resultiert aus Anforderungen: Die Abhängigkeitsbeziehung wird ins Featuremodell aufgenommen mit Traceability-Link auf die entsprechende Anforderung im Anforderungsmodell
- Die Abhängigkeit resultiert aus dem Objektmodell: Die Abhängigkeitsbeziehung wird im Featuremodell abgebildet mit einem Traceability-Link auf die verursachenden Entwurfselemente
- Die Abhängigkeit resultiert aus der Implementierung: Die Abhängigkeitsbeziehung wird im Featuremodell abgebildet mit einem Traceability-Link auf die verursachenden Implementierungs-Elemente

Der Wartung und Konsistenzsicherung der Abhängigkeiten kommt große Bedeutung zu. Durch werkzeuggestützte, automatisierte Aktualisierung kann sie unterstützt werden. Die im Featuremodell nur abgebildeten Abhängigkeiten widerspiegeln die verursachenden Abhängigkeiten, bei Änderungen werden sie durch Kopplung der Werkzeuge aktualisiert. Als Verweis und zur Aktualisierung werden die Traceability-Links zwischen Feature und entsprechendem Element genutzt.

**Unterstützen des Verfolgens von Zusammenhängen**

Das Verstehen von Zusammenhängen bei Änderungen komplexer Systeme wird durch Verknüpfung von Informationen verschiedener Abstraktionsgrade vereinfacht (siehe 2.5). Traceability-Links können vom Entwickler zur Navigation innerhalb von Modellen verschiedener Abstraktionsgrade genutzt werden. Beispielweise können von einem Feature ausgehend abhängige Elemente in Objektmodell und Implementierung in einem Editor ausgewählt, dargestellt und bearbeitet werden. Durch Traceability-Links referenzierte Modellelemente können hervorgehoben und in Listen dargestellt werden. Die

Listendarstellung referenzierter Elemente wird darüber hinaus dazu benutzt, dass die vollständige Umsetzung einer Änderung geprüft werden kann.

**Erstellen von Traceability-Links**
Ein Traceability-Link beschreibt eine Beziehung, die von einem Entwickler während einer Problemlösung hergestellt oder während einer Reengineering-Aktivität erkannt wurde. Der Link wird im Repository des dabei verwendeten Entwicklungswerkzeugs aufgezeichnet und stellt einen Teil der Dokumentation dar. Der betreffende Entwickler prüft und bestätigt die betreffende Beziehung.

**Pflege von Traceability-Links bei Veränderungen**
Bei Veränderungen an Modellen und Implementierung müssen die Traceability-Links angepasst werden. Bei jeder Änderung werden die davon potentiell betroffenen Traceability-Links angezeigt und ebenfalls verändert. Die Veränderungen an Software folgen meist grundlegenden Abläufen, wie sie für Veränderungen an Quellcode als Refactoring beschrieben wurden [Fowler 1999]. In vergleichbarer Weise sind Refactorings für Traceability-Links zu sammeln. Dies ist Gegenstand laufender Arbeiten. Für Erstellung, Verwaltung, Pflege und Nutzung von Traceability-Links ist die Unterstützung durch Entwicklungswerkzeuge essentiell, damit effektive Entwicklungsunterstützung erreicht wird und die Konsistenz der Links sichergestellt werden kann.

## 4.3 Identifikation und Modellierung von Anforderungen im Featuremodell

Featuremodelle für die Entwicklung und Weiterentwicklung einer Produktlinie werden im Schritt Domänenanalyse der Methode HyperFeatuRSEB (siehe 6.1.1) erstellt, der zum Produktlinien-Engineering gehört. Bei der Anforderungsanalyse im Produkt-Engineering werden diese Featuremodelle angewendet, wie im Abschnitt 4.5 dargestellt wird.

Die Erfassung der Anforderungen an Produkte innerhalb der betreffenden Domäne erfolgt weitgehend mit klassischen Mitteln der Anforderungsanalyse. Der bei früheren Ansätzen der Domänenanalyse (siehe 2.2.1) aufgetretene sehr hohe Aufwand wird dadurch reduziert, dass bereits während der Analyse mittels Scoping-Entscheidungen eine Konzentration auf die wichtigsten Anforderungen erfolgt. Da solche Entscheidungen Wissen erfordern, das erst während der Analyse gewonnen wird, verläuft der Analyse- und Entscheidungsprozess zyklisch. Die eigentliche Domänenanalyse ist in üblicher Weise eine Folge von Aktivitäten zur Erfassung von Informationen (meist als Elicitation bezeichnet), deren Vervollständigung und Strukturierung durch Modellierung und ihrer Verifikation. Nach jeder Aktivität erfolgt zunächst eine Scoping-Entscheidung darüber, welche Anforderungen und Modellelemente weiter betrachtet werden, bevor in der nächsten Aktivität weiterer Aufwand investiert wird. Nicht weiter zu betrachtende Modellelemente werden nicht gelöscht; sie werden für eventuelle spätere Anwendung aufbewahrt, wobei sie mit dem erreichten Stand der Bearbeitung gekennzeichnet werden. Die Aktivitätenfolge einer Iteration ist in Abb. 26 als UML-Aktivitätsdiagramm dargestellt. Die tatsächliche Reihenfolge der Elicitation- und Modellierungsaktivitäten hängt stark von der Art und den Quellen der Informationen ab; diese Aktivitäten sind deshalb in der Abbildung als nebenläufig dargestellt.

Als Ergebnis dieses Prozesses entsteht zum einen das Featuremodell und zum anderen eine verfeinerte Anforderungsbeschreibung in Form eines Anforderungsmodells. Dieses besteht - wie für objektorientierte Vorgehensweisen üblich - aus Use-Case-Diagrammen einschließlich dazugehöriger textueller Use-Case-Beschreibungen, aus Aktivitätsdiagrammen sowie teilweise aus Klassendiagrammen zur Darstellung von Objekten der

Domäne. Auf welcher dieser Beschreibungen der Schwerpunkt liegt, hängt von den darzustellenden Eigenschaften der Domäne ab.

Die Elemente des Anforderungsmodells werden Features zugeordnet. Für jedes Feature wird zunächst ein Hyperslice angelegt. Dann wird für jedes Modellelement entschieden, zu welchem Feature es gehört, und es wird in den betreffenden Hyperslice übernommen. Dabei entsteht schrittweise ein vollständiges Anforderungsmodell für die Produktlinie, das entsprechend der Features in überschneidungsfreie Komponenten zerlegt ist. Die Komponenten sind in der in Kapitel 5 gezeigten Weise in Hyper/UML formuliert und in Hyperslices gekapselt. Dadurch wird eine spätere Komposition der Anforderungsmodelle für Produkte ermöglicht.

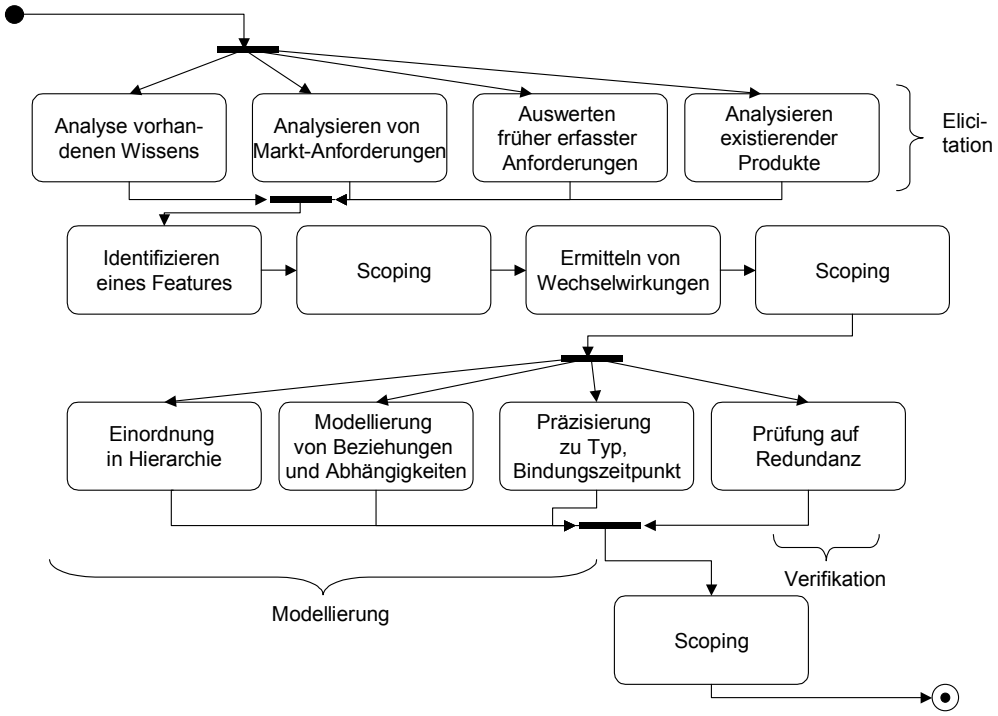



*Abb. 26: Aktivitäten innerhalb einer Iteration bei der Erstellung von Featuremodellen*

Zum *Analysieren vorhandenen Wissens* als wesentlichem Teil der Elicitation gehört die Nutzung der gesamten Vielfalt möglicher Informationsquellen, wie sie bei der konventionellen Anforderungsanalyse und der Domänenanalyse betrieben wird. Besonderes Augenmerk ist auf Informationsquellen und Erfahrungsträger zu richten, die Aussagen über weitere Eigenschaften einer Domäne liefern können. Das *Analysieren existierender Produkte,* auch von Konkurrenten, sowie das *Analysieren von Markt-Anforderungen* stellt einen weiteren Teil der Elicitation dar. Die etablierten Mittel der Marktforschung fließen hier ein. Erfasste Informationen werden schrittweise zur Vervollständigung des Anforderungsmodells genutzt. Während des anschließenden Scoping werden u.a. Gemeinsamkeiten und Unterschiede von Anforderungen bewertet.

Für das als Beispiel verwendete Digitale Video-Projekt wurden Beschreibungen verschiedener Videorecorder und Videobearbeitungsgeräte analysiert, die am Markt erhältlich sind. Außerdem gehörten Informationen über Merkmale des vdr-Projekts [Schmidinger 2002] und des linux.tv-Projekts [Metzler Metzler 2002] zu den Informationsquellen. Zusätzlich wurde eine Kundenumfrage durchgeführt [Kubali 2003].

Das *Auswerten früher erfasster Anforderungen,* die aus verschiedenen Gründen noch nicht realisiert wurden, ist für die Entwicklung einer Produktlinie ebenfalls bedeutsam. Möglicherweise können im Fall geringerer Entwicklungskosten je Produkt auch Kundengruppen bedient werden, die bei früherer Entwicklung von Einzel- oder Standardprodukten nicht erreichbar waren. Das Scoping nach dieser Aktivität dient dazu, frühere Entscheidungen zu überprüfen.

Im Fall des Digitalen Video-Projekts wurde beispielsweise das Feature on-Demand-Video zunächst nicht weiter verfolgt. Es wurde entschieden, keinen weiteren Aufwand für Analyse und Modellierung der dazu gehörenden Anforderungen zu investieren. In einer späteren Version wurde diese Entscheidung revidiert; das Feature und die entsprechenden Anforderungen wurden mit leicht veränderten Details zur Produktlinie hinzugefügt.

Bei der Auswertung der Elicitation-Ergebnisse und dem *Identifizieren von Features* kommt der Vergabe von Namen in einer für Kunden verständlichen und unterscheidbaren Form große Bedeutung zu. Es sind Begriffe zu verwenden, wie sie von Kunden als Anforderungen genutzt werden. Häufig treten Synonyme auf, die eine Zusammenlegung mehrerer Features zulassen. Bei der Entscheidung oder im anschließenden Scoping kann eine Kategorie und ein Bindungszeitpunkt für einzelne Features festgelegt werden, wenn bereits genügend Informationen vorliegen.

Das *Ermitteln von Wechselwirkungen* mit anderen Features nutzt Informationen aus bereits durchgeführten Untersuchungen der Features für eine eventuelle Zusammenlegung. Insbesondere ist zu ermitteln, welche Features von anderen beeinflusst werden oder sogar abhängig sind. Über Features mit wichtigem Einfluss auf andere Features muss bei einer Produkt-Definition häufig zuerst entschieden werden, weshalb diese in der Hierarchie eines Featuremodells nahe der Spitze angeordnet werden müssen. In Einzelfällen ist dazu auch der Einfluss von Features auf die Architektur der Produktlinie zu bewerten. Die Bewertung kann beim Scoping auch dazu führen, dass ein Teil der Features nicht weiter betrachtet zu werden braucht.

Das *Einordnen in die Hierarchie* erfolgt so, dass für die Features eine sinnvolle Entscheidungs-Reihenfolge für Produkte entsteht, damit die am weitesten reichenden Entscheidungen zuerst getroffen werden. Häufig ergeben sich dabei Unterschiede zu den beim Entwurf üblichen Hierarchien, die durch schrittweise Verfeinerung entstehen. Im Beispiel (Abb. 23, S. 70) hat das Feature on-Demand Video Server weitreichenden Einfluss auf die Architektur eines Produkts und somit auf andere Features; es wurde im Featuremodell weit oben angeordnet, wie das auch von der Definition für Hierarchiebeziehungen eines Featuremodells festgelegt ist. Für Entwerfer wäre eine ganz andere Einordnung dieses Features durchaus naheliegend, zum Beispiel als Verfeinerung von Bedienungsfunktionen. Wird eine Darstellung von Verfeinerungsbeziehungen benötigt, kann diese im Modell durch <<refines>> graphisch oder als Ausdruck dargestellt werden (siehe auch 4.1.3).

Bei der Einordnung in eine Hierarchie sind noch weitere Gesichtspunkte zu beachten, wenn auch mit geringerer Wichtung. So sollte versucht werden, alternativ oder in anderer Gruppierung wählbare Features in einer Hierarchie einem gemeinsamen, ggf. abstrakten Feature unterzuordnen, damit die Gruppierungsbeziehung angewendet werden kann. Ist das nicht erreichbar, kann die Gruppierung auch außerhalb der Hierarchie durch eine Kombination von require- und exclude-Beziehungen nachgebildet werden, die mit OCL verknüpft sind, damit die gleichen Konsistenzbedingungen und Auswahlregeln dargestellt werden. Für spätere Veränderungen am Featuremodell wirkt sich

allerdings nachteilig aus, dass die Alternative oder die Gruppierung in einem solchen Fall für den Entwickler nicht mehr im graphischen Modell erkennbar ist.

Die *Modellierung von Beziehungen und Abhängigkeiten* zwischen Features fügt weitere Informationen zu Feature- und Anforderungsmodell hinzu. Die Analyse von Abhängigkeiten kann zur weiteren Verfeinerung von Features führen. Wegen der Bedeutung der Abhängigkeiten wird diese Aktivität separat in Abschnitt 4.4 behandelt.

Die weitere *Präzisierung* durch Anreichern des Featuremodells mit genaueren Informationen dient dazu, diese Informationen bei der späteren Implementierung der Produktlinie nutzen zu können. In der graphischen Standarddarstellung des Featuremodells werden sie meist ausgeblendet, damit die Verständlichkeit nicht beeinträchtigt wird. Zu diesen Informationen gehören beispielweise Featuretyp, Bindungszeitpunkt und Beziehungstypen. Die Präzisierung der Beziehungen sollte durch OCL-Konstrukte erfolgen, damit werkzeuggestützt auswertbare Modelle erreicht werden. Da die Präzisierung einen hohen Aufwand erfordert, wird sie nur für tatsächlich zu implementierende Features durchgeführt.

Eine *Prüfung auf Redundanz* dient zur Ermittlung von Features, die ggf. zusammenzulegen oder zu verfeinern sind. In einer Hierarchie mehrfach auftretende Features werden dadurch zusammengelegt, dass ihre hierarchische Anordnung verändert wird. Diese Aktivität gehört zur Verifikation, die in konventioneller Weise noch weitere Prüfungen der Anforderungsmodelle zur Sicherstellung ihrer Qualität umfasst.

Das zwischen den Aktivitäten durchgeführte *Scoping* dient der Entscheidung über Einordnung und weitere Behandlung der Features. Ein Feature kann als Bestandteil der Produktlinie realisiert werden, oder es wird zwar realisiert, jedoch nur für ein einzelnes Produkt und nicht als Bestandteil der Produktlinie, oder es wird nicht weiter verfolgt (siehe 4.6). Die Bindungszeitpunkte variabler Features sind nach Möglichkeit bereits während der Elicitation, spätestens jedoch während des abschließenden Scoping festzulegen. Ablauf und Einflussgrößen des Scoping werden in Abschnitt 4.6 untersucht.

## 4.4 Ermittlung und Wartung von Abhängigkeiten zwischen Features

Die Darstellung der Abhängigkeiten hat große Bedeutung für die Nutzung des Featuremodells bei der Feature-Auswahl, Zerlegung, Komposition und Prüfung. Abhängigkeiten erhöhen die Komplexität einer Produktlinie. Eine explizite Darstellung erleichtert die Evolution der Produktlinie, weil Wechselwirkungen für die Entwickler sichtbar werden und so die Verständlichkeit verbessert wird. Von besonderer Bedeutung sind hier require- und exclude-Beziehungen sowie Gruppierungen.

### 4.4.1 Minimieren von Abhängigkeiten

Zur Erstellung von Featuremodellen gibt es meist mehrere mögliche Varianten. So können Hierarchiebeziehungen auch als require-Beziehungen dargestellt werden, oder Features können weiter verfeinert und dabei zerlegt werden, damit neue Features entstehen. Eine weitere Zerlegung ist insbesondere dann von Vorteil, wenn Features sich gegenseitig im Verhalten beeinflussen (oft als Feature Interaction bezeichnet, siehe 4.1.3) oder wenn zahlreiche Abhängigkeiten bestehen, die eine Entwicklung neuer Produkte einschränken.

Durch die Zerlegung eines Features in Unter-Features kann erreicht werden, dass die Beeinflussung im Sinne von Feature Interaction sich nur noch auf ein Unter-Feature bezieht, das dann durch require- oder exclude-Beziehungen mit dem beeinflussenden Feature verbunden wird. Werden alle verbleibenden Teile des Featuremodells dadurch

frei von Beeinflussungen, konnte die Feature Interaction aufgelöst werden. Für kompliziertere, aber nur sehr selten vorkommende Beeinflussungen sei auf weiterführende Arbeiten zu Feature Interaction verwiesen. Solche Ansätze sind nicht Gegenstand der vorliegenden Arbeit, eine Klassifikation und Bewertung bekannter Verfahren ist in [Calder et al. 2002] enthalten.

Wenn mehrere Features durch require-Beziehungen mit einem Feature verbunden sind, das in neuen Produkten mit nur wenigen Abhängigen verwendet werden soll, ist eine Verfeinerung sinnvoll. Häufig gelingt es, durch Zerlegung die require-Beziehungen auf die Unter-Features aufzuspalten. Wird eines der erhaltenen Unter-Features für ein neues Produkt ausgewählt, sind weniger andere Features erforderlich. Ein Beispiel hierzu ist in [Halle 2001], S. 91 enthalten.

Bei der Verfeinerung ist der resultierende Aufwand für das Refactoring oder die Erstellung von Objektmodell und Implementierung zu beachten. Je feiner und vielfältiger die Variabilität einer Produktlinie gestaltet wird, desto größer ist der notwendige Aufwand. Zur Abwägung von Aufwand und Nutzen ist eine Darstellung der Kosten für das betreffende Feature sinnvoll, wie weiter unten dargestellt wird.

### 4.4.2 Auflösung von Abhängigkeiten

Häufig sind Abhängigkeiten Anlass für Änderungen an Architektur und Implementierung, beispielsweise damit exclude-Beziehungen zwischen Features aufgelöst (siehe Abschnitt 6.3.1) oder damit Interaktionen zwischen Features verändert werden. Nach Auflösung der Ursachen beispielweise durch Veränderung an zwei Moduln mit widersprechenden Zustandsübergängen, kann die entsprechende exclude-Beziehung zwischen ihnen entfernt werden; nach Aktualisierung des Featuremodells anhand der Traceability-Links zwischen den betroffenen Features und diesen Moduln existiert auch im Featuremodell keine exclude-Beziehung mehr, beide betroffenen Features können gleichzeitig in einem Produkt enthalten sein.

Solange Änderungen bearbeitet werden, können nicht nur Entwurf und Modell, sondern auch die Abhängigkeiten zwischen Features zeitweise inkonsistent sein. Werkzeuge müssen solche Inkonsistenzen zulassen, aber andererseits Prüfungen zu deren Auffinden und Beheben anbieten können.

## 4.5 Featuremodelle bei einer Anforderungsanalyse für Produkte

Die zentrale Rolle des Featuremodells bei Anforderungsanalyse und Modellentwurf von *Produktlinien* wird in den Abschnitten 4.3 und 6.1.2 deutlich gemacht. Bei der Anforderungsanalyse für jedes *Produkt* einer Produktlinie (siehe Abb. 16, S. 54) kommt dem Featuremodell eine ebenso große Bedeutung zu. Es ist dafür verantwortlich, dass die Formulierung von Anforderungen so gesteuert wird, dass Referenzarchitektur und wiederverwendbare Plattform möglichst weitgehend (wieder-) verwendet werden können. Je weniger Abweichungen von der Produktlinie ein Produkt aufweist, desto kostengünstiger und schneller kann das Produkt entwickelt werden.

Die Anforderungen an ein Produkt werden anhand des Featuremodells formuliert. Abweichungen von der betreffenden Produktlinie werden dabei als Abweichungen von Features betrachtet. Für die Verknüpfung von Analyse- und Entscheidungs-Aktivitäten hat sich bei der praktischen Anwendung ein iteratives Vorgehen bewährt, das aus Zyklen von Hypothesen-Erstellung und -Verifikation besteht. In jedem Zyklus werden einige Anforderungen und Features definiert und Entscheidungen über deren Ausgestaltung getroffen.

Das Featuremodell drückt Anforderungen nur sehr allgemein aus. Wesentlich genauere Informationen sind im Anforderungsmodell der Produktlinie enthalten, das bereits vorliegt. Es wurde während des Schrittes Domänenanalyse entwickelt (siehe 6.1.1 und 4.3) und in Hyperslices entsprechend der Features der Produktlinie zerlegt. Während der Anforderungsanalyse für ein Produkt wird dessen Anforderungsmodell aus diesen Hyperslices zusammengesetzt, entsprechend der Entscheidung über Features des Produkts. Anhand des erstellten Anforderungsmodells können Unterschiede zwischen Anforderungen des Kunden und Eigenschaften der Produktlinie im Detail festgestellt und entschieden werden. Die dafür benötigten Anforderungen werden mit den Elicitation-Verfahren der konventionellen Anforderungs- und Domänenanalyse (siehe auch 2.2.1), durch Auswertung vorhandener Dokumente und Formulare sowie bei Interviews mit speziellen Fragetechniken ermittelt.

### 4.5.1 Ausrichtung der Anforderungsanalyse am Featuremodell

Das *Ziel* der Anforderungsanalyse für Produkte aus einer Produktlinie unterscheidet sich von der für die konventionelle Einzelfertigung. Neben gemeinsamen Zielen, wie Erfassen, Verstehen, Strukturieren und Prüfen der Anforderungen eines Kunden geht es hier zusätzlich darum, einen möglichst großen Teil eines Produktes aus wiederverwendbaren, vorgefertigten Teilen aufzubauen. Damit dieses Ziel erreicht wird, sind nicht nur ständige Vergleiche zwischen erfassten Anforderungen und existierenden Lösungen nötig, es müssen auch Kompromisse von Seiten des Kunden eingegangen werden.

Diese Kompromiss-Entscheidungen sind das Ergebnis von Verhandlungen zwischen Hersteller und Kunden und damit zwischen Auftraggeber und Auftragnehmer. Der Kunde wünscht ein Feature (und damit die Realisierung einer Anforderung), der Hersteller macht ein Angebot für Zeitdauer und Kosten. Je genauer eine Anforderung mit der Produktlinie übereinstimmt, desto schneller und kostengünstiger wird die Herstellung. Der Kunde sollte bereit sein, von Teilen seiner Anforderungen abzuweichen. Solche Abweichungen können in Details des Verhaltens realisierter Funktionen oder im Verzicht auf ganze Funktionen bestehen. Kompromisse sind auch bezüglich der Kombination mit anderen Features und bezüglich des Bindungszeitpunkts erforderlich, wenn die Aufgaben im Rahmen einer Produktlinie gelöst werden sollen.

Der Entwicklungsaufwand für neue oder abgewandelte Features ist um so geringer zu erwarten, je geringer die Unterschiede zum vorhandenem Featuremodell sind. Abb. 27 gibt eine qualitative Bewertung des Entwicklungsaufwands für eine Anforderung und einen Kunden in Abhängigkeit von der Abweichung wieder; die Bewertung wurde vom Autor aus den Erfahrungen industriellen Projekten abgeleitet. Bei minimaler Abweichung von vorgefertigten Lösungen fällt geringer Mehraufwand an. Etwas größer wird der Aufwand bei geringen Ergänzungen von Features. Höherer Aufwand ist erforderlich, wenn für das Entfernen von Features ein Refactoring von Modellen und Quellcode notwendig ist. Extrem hoher Aufwand kann erforderlich sein, wenn Architekturänderungen erforderlich werden. Zum Vergleich wurden die Aufwendungen für das betreffende Feature in einem individuellen Softwareprodukt (das außerhalb einer Produktlinie entwickelt wird) und in einem Massenprodukt dargestellt. Der Aufwand für die individuelle Entwicklung ist höher als in der Produktlinie; hier zeigt sich der erwähnte Zeit- und Kostenvorteil. Bei starker Abweichung von der Produktlinie übersteigt der Änderungsaufwand den Aufwand für eine (individuelle) Neuentwicklung, eine aus Erfahrungen mit dem Refactoring bekannte Tatsache. Verglichen mit dem Massenprodukt, dessen Entwicklungsaufwand sich auf eine Anzahl von Kunden verteilt, ist der Auf-

wand für das betreffende Feature als Teil der Produktlinie höher; dies ist durch den Mehraufwand für die Variabilität bedingt.

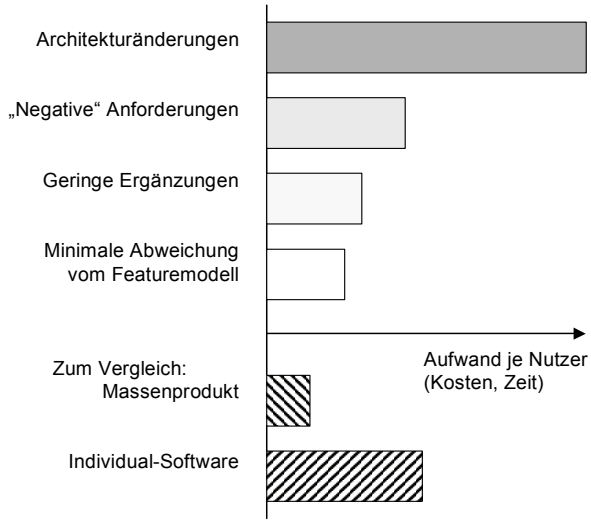



*Abb. 27: Kosten- und Zeitaufwand zur Realisierung einer Anforderung je nach Abweichung von der Produktlinie*

Fordert der Kunde trotz solcher Zeit- und Kostenvorteile eine Abweichung von der Produktlinie, so muss der Hersteller entscheiden, ob er ein Feature als Ergänzung außerhalb der Produktlinie oder als deren Erweiterung realisiert. Das Treffen einer solchen Entscheidung wird in Abschnitt 4.6 erläutert, ihre Umsetzung in den Abschnitten 6.3 und 8.2.

### 4.5.2 Entscheidung über variable Features

Gemeinsame Features sind Bestandteil jedes Produkts. Für variable Features ist zu entscheiden, ob sie in einem Produkt enthalten sein sollen oder nicht. Damit entspricht die Festlegung seiner Eigenschaften im Featuremodell dem Weglassen eines variablen Features (bei Verzicht) oder der Markierung als notwendiges Feature (bei Auswahl). Diese Entscheidungen sind in Abb. 28 als UML-Aktivitätsdiagramm dargestellt. Der dargestellte Ablauf wird für jedes Feature durchlaufen.

Die Entscheidung über ein variables Feature wird als Bindung der Variabilität bezeichnet. Je nach dem Zeitpunkt der Festlegung variabler Eigenschaften wird von verschiedenen Bindungszeitpunkten gesprochen, wie in Abschnitt 2.2.3 erläutert wurde. Wird bereits während der Anforderungsanalyse entschieden, dass ein Feature nicht benötigt wird, liegt ein früher Bindungszeitpunkt vor. Das betreffende variable Feature wird aus dem Featuremodell für das Produkts entfernt. Soll das spätere Treffen einer Entscheidung über ein Feature ermöglicht werden, etwa während der Installation oder zur Laufzeit des Produkts (später Bindungszeitpunkt), bleibt das Feature als variabel im Featuremodell enthalten, sein Bindungszeitpunkt wird im Featuremodell hinterlegt. Getroffene Entscheidungen werden mit den Informationen entsprechend Abschnitt 4.6.1 dokumentiert.

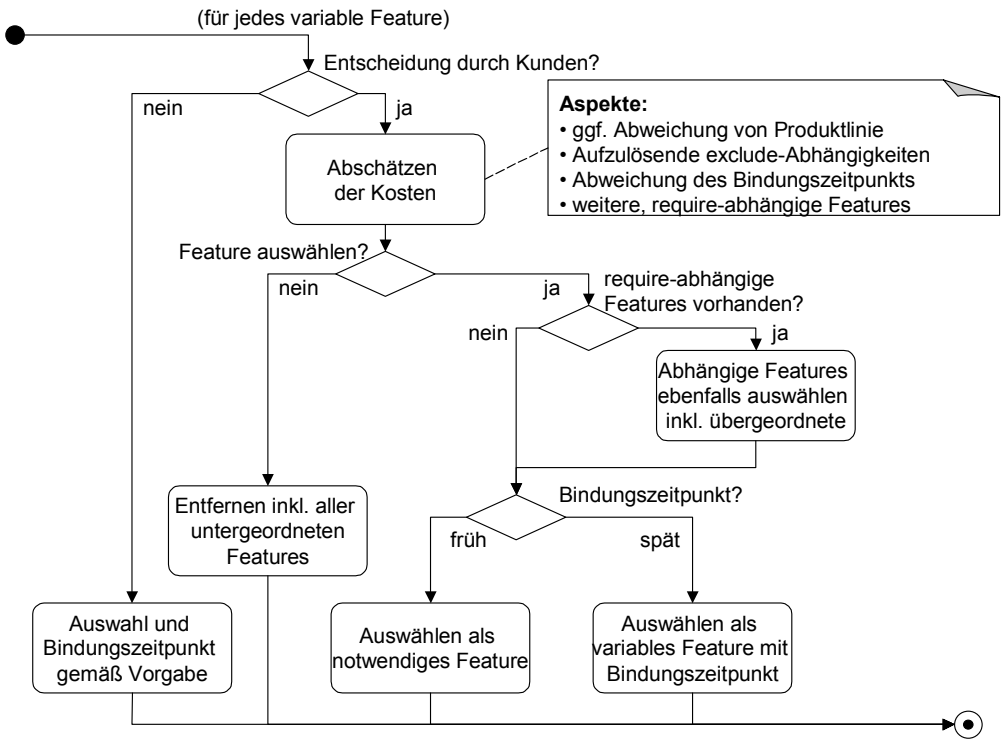



*Abb. 28: Entscheidungen über ein variables Feature bei Produkt-Anforderungsdefinition*

Da Features im Featuremodell nach dem Grad der Auswirkung eines Features auf andere angeordnet sind, erfolgen die Entscheidungen in der Feature-Hierarchie von oben nach unten. Der genaue Ablauf hängt stark von der Kunden-Hersteller-Beziehung ab. Überwiegt die Nachfrage und hat der Kunde genaue Vorstellungen vom Produkt bis hin zu technischen Details, werden Entscheidungen in der Reihenfolge gefällt, die der Kunde vorgibt. Überwiegt das Angebot und dem Kunden werden Features und ihr Nutzen vorgestellt, sollte nach der Reihenfolge vorgegangen werden, die das Featuremodell durch seine Hierarchie vorgibt. Möchte der Kunde für nachfolgende oder verfeinerte Entscheidungen keine Aussagen machen, sind diese Entscheidungen anhand der im Featuremodell hinterlegten Vorgaben zu treffen. In Abschnitt 4.1.1 wurden dafür Vorgaben für variable Features als Bestandteil des Featuremodells definiert.

## 4.6 Entscheidung über die Evolution anhand des Featuremodells

Die Evolution einer Produktlinie erfolgt immer aufgrund einer unternehmerischen Entscheidung des Herstellers, denn sie erfordert Investitionen, die sich erst in der Zukunft auszahlen. Eine solche Entscheidung erfolgt meist auf Anforderungen und Anregungen von Kunden hin, die neue Features, erweiterte Variabilität oder erweiterte Kombinationsmöglichkeiten wünschen. Die unternehmerische Entscheidung bezieht sich darauf, ob eine Abweichung von der Produktlinie als künftiger Bestandteil der Produktlinie, außerhalb der Produktlinie oder ggf. überhaupt nicht realisiert wird. Aspekte der Softwaretechnik tragen zur Entscheidung bei, weil sie wichtige Fakten zum erforderlichen Aufwand liefern, der sich aus dem Grad der Abweichung, dem eigentlichen Entwicklungsaufwand und den Risiken ergibt.

Zu den unternehmerischen Aspekten bei der Entscheidung über neue Features einer Produktlinie gehören:

- Der Return on Investment, beeinflusst von Aufwand, Grad der Übereinstimmung, erwarteter Nachfrage nach diesem Feature oder dieser Konfiguration,

- Die Strategie der zukünftigen Entwicklung der Produktlinie, insbesondere Marktvolumen, Schwerpunkte der Kundenwünsche, Investitionsbereitschaft,
- Die angestrebte Übereinstimmung mit Kundenwünschen, damit eine Produktlinie für die Kunden attraktiv wird, wenn möglichst viele der gewünschten Features vorgefertigt werden, damit sie schnell und günstig angeboten werden können.

Die softwaretechnischen Aspekte umfassen

- Maßnahmen zur Verbesserung von Wartbarkeit und Portabilität, die bei Erweiterungen möglich wären,
- Nutzen und Aufwand des Refactoring der bestehenden Teile der Produktlinie,
- Einfluss der Änderung auf die Qualität der Architektur, insbesondere auf Verständlichkeit und Konsistenz,
- Einfluss der Änderung auf die Komplexität der Produktlinie: je mehr Features sie umfasst, desto aufwendiger ist die Entwicklung jedes einzelnen (weiteren) Features.

Wenn die Entscheidung für die Aufnahme eines neuen Features und damit für eine Evolution der Produktlinie getroffen werden soll, werden anhand der Abhängigkeiten im Featuremodell zunächst Abweichungen ermittelt, aus denen dann der Entwicklungsbedarf abgeleitet und Veränderungen an der Produktlinie vorgenommen werden (siehe Abschnitt 6.3.1).

Der Grad der Abweichung neuer Anforderungen von der Produktlinie wird zunächst anhand des Featuremodells bestimmt. Dabei sind die nach Abb. 27 (S. 81) diskutierten Fälle zu unterscheiden. Entwicklungsaufwand und Risiken werden dadurch ermittelt, dass vorhandene Komponenten der Produktlinie mit Anforderungs-, Objektmodell und Implementierung analysiert werden. Danach kann eine präzise Aufwandsschätzung vorgenommen werden.

Bei der Aufwandsschätzung werden Wechselwirkungen zwischen Kombinationen von Features meist außer Acht gelassen. Da eine Schätzung selbst auch Aufwand erfordert, wird sie, wie in größeren Projekten üblich, erst vorgenommen, wenn die Entscheidung für die Durchführung einer Erweiterung sehr wahrscheinlich ist. Aufwandsschätzungen für verschiedene Kombinationen von Features unter Berücksichtigung ihrer Abhängigkeiten und Wechselwirkungen werden in der Praxis nicht vorgenommen, da der Aufwand dafür zu hoch ist; eine gröbere Abschätzung aufgrund des Featuremodells reicht für die Festlegung einer bestimmten Kombination meist aus.

Die unternehmerische Entscheidung des Herstellers über Entwicklungsaufträge für neue Features wird häufig als Scoping bezeichnet, in Anlehnung an ähnliche Entscheidungen bei der Anforderungsanalyse konventioneller Softwareentwicklung. Auf die Entscheidungsfindung wird in [Riebisch et al. 2001a] eingegangen, in Abschnitt 6.2.1 wird sie in die Produktentwicklung eingeordnet. Aus Sicht des Herstellers ist zu entscheiden, ob das betreffende Feature als Bestandteil des Kerns, als variables Feature, oder als Ergänzung außerhalb der Produktlinie zu entwickeln ist. Das Scoping kann folgende Wertung für die Entwicklung innerhalb der Produktlinie zum Ergebnis haben:

1. Feature realisieren
2. Feature vorbereiten (insbesondere bei Architekturentscheidungen)
3. Feature später realisieren

4. Feature nicht realisieren

Auch in den Fällen 2, 3 und 4 könnte einem Kunden das Feature außerhalb der Produktlinie bereitgestellt werden, eine Evolution der Produktlinie fände jedoch nicht statt.

Bei einer Entscheidung über ein Feature müssen neben diesem aktuell betrachteten Feature auch Einflüsse potentieller zukünftiger Features betrachtet werden. Einflüsse auf die Entwicklung der Architektur sind hier besonders wichtig, wie zum Beispiel durch nicht-funktionale Anforderungen. Solche Anforderungen können zu einer Wertung als Fall 2 führen. Sowohl eine zu geringe als auch eine zu starke Berücksichtigung zukünftiger Features kann zum Misserfolg einer Produktlinie führen, wie die Erfahrungen des Autors zeigen und [Dikel et al. 1997] bestätigt.

### 4.6.1 Dokumentation von Entscheidungen

Entscheidungen erfolgen nach der Prüfung und Bewertung möglicher Alternativen. Bei späteren Änderungen der Gegebenheiten sind Entscheidungen häufig zu revidieren. Liegt dann eine *Dokumentation* von Alternativen und ihrer (früheren) Bewertung vor, können die Entscheidungen mit geringerem Aufwand und größerer Sicherheit getroffen werden. Durch eine solche Dokumentation wird somit die Evolution unterstützt. Dies trifft für alle Entscheidungen zu, auch für Fragestellungen über das Scoping hinaus, wie über

- Variabilität und Bindungszeitpunkt (siehe 4.6),
- objektorientierte Zerlegung, Separation of Concerns und Architekturprinzipien (siehe 6.1.2),
- Erweiterung der Produktlinie (siehe 6.3),
- Reverse Engineering und Refactoring (siehe 8.1 und 8.2),
- Regelungen des Managements und Gestaltung des organisatorischen Umfelds (siehe 9.2).

In Anlehnung an bekannte Entscheidungsmodelle (siehe Abschnitt 2.4.4) werden in dieser Arbeit Entscheidungen durch folgende Bestandteile charakterisiert und dokumentiert:

- Fragestellung: textuell, mit Bezug zu den Anforderungen
- Alternativen als Menge möglicher Antworten: Auflistung textueller Beschreibungen oder von Modellelementen
- Bewertung der Alternativen aufgrund der Anforderungen: Parameter oder Text
- Referenzen zu betroffenen Elementen und Variationspunkten in Anforderungsmodell, Objektmodell und Implementierung: Links
- Auswirkung der Entscheidung für mögliche Antworten auf betroffene Elemente: Links mit Parametern oder Text
- Referenzen zu davon abhängigen Entscheidungen: Links
- Auswirkung auf abhängige Entscheidungen: Links mit Parametern oder Text

Für die Visualisierung von Alternativen, Bewertungen und Auswirkungen haben sich bei kleinen Datenmengen tabellarische Darstellungen bewährt, wie beispielweise die Cross-Referenz-Tabellen in Abschnitt 8.1. Zwecks Wartbarkeit dieser Dokumentation werden die Referenzen durch Links zu betroffenen Modell- oder Implementierungs-

Elementen im Repository mit doppelter Verkettung abgebildet, damit bei Änderungen alle betroffenen Elemente direkt auffindbar sind. Die Konsistenz dieser Links wird durch die Werkzeugunterstützung sichergestellt.

## 4.7 Werkzeugunterstützung

Die Nutzung von Featuremodellen bei der Entwicklung und Evolution von Produktlinien ist in der Praxis nur dann mit wirtschaftlichem Erfolg möglich, wenn sie durch Werkzeuge unterstützt wird. Wegen der Komplexität der Featuremodelle und ihrer wichtigen Rolle für die Produktlinien-Entwicklung sind Auswertungen und Maßnahmen zur Konsistenzsicherung erforderlich. Die Werkzeuge zur Featuremodellierung sind mit den weiteren Werkzeugen der Produktlinien-Entwicklung zu verbinden. Sie müssen folgende Leistungen bieten:

- Graphische Bearbeitung und Darstellung der Featuremodelle entsprechend den Definitionen in Abschnitt 4.1.1 und 4.1.2 mit Möglichkeiten der hierarchischen Verfeinerung, wobei auch unvollständige Informationen abgelegt werden sollen.
- Auswertung der Informationen des Modells mit der definierten Syntax und Semantik
- Prüfung der Konsistenz der Featuremodelle
- Speicherung der Informationen in einer für andere Werkzeuge auswertbaren Form oder Bereitstellung entsprechender Schnittstellen.
- Speicherung von Verweisen in Form von Traceability-Links zwischen Anforderungs-, Feature-, Objektmodell und Implementierung im Repository und den einzelnen Modellelementen
- Aufzeichnung und Pflege der Traceability-Links
- Abbildung von Abhängigkeiten zwischen Features einschließlich der aus Objektmodell und Implementierung abgeleiteten Beziehungen
- Prüfung und Darstellung von Abhängigkeiten sowie Auswertung von Abweichungen zu neuen Anforderungen als Grundlage von Aufwandsschätzungen
- Auflistung der variablen Features zur Entscheidungsfindung bei der Produktdefinition unter Berücksichtigung von Abhängigkeiten

Für viele dieser Anforderungen sind Werkzeuge verfügbar oder wurden im Rahmen der Arbeiten geschaffen. Das Werkzeug AmiEddi [AmiEddi 2002] ermöglicht die graphische Erstellung und Bearbeitung von Featuremodellen, speichert die Informationen jedoch in einem eigenen Datenformat. Zwecks Integration mit anderen Werkzeugen wurde im Rahmen der Arbeiten zu ALEXANDRIA eine XML-Schnittstelle geschaffen, die eine ausreichend flexible Kopplung auch in einer heterogenen Werkzeugumgebung erlaubt. Abb. 29 zeigt die graphische Nutzerschnittstelle dieses Werkzeugs.

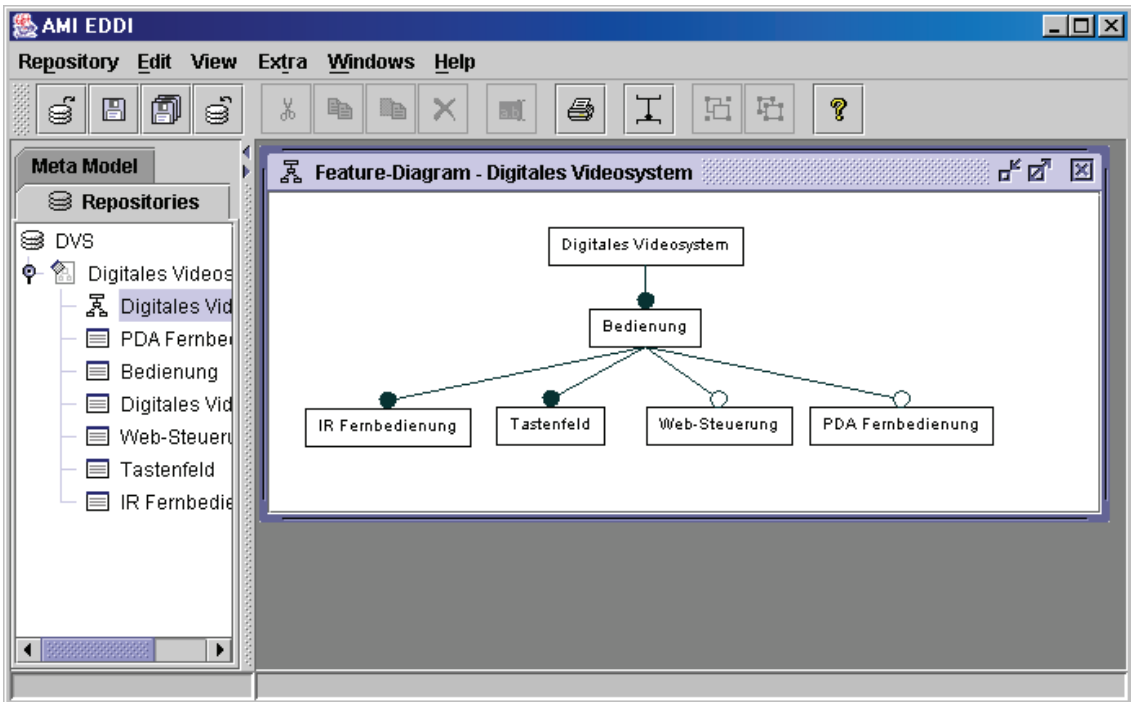



*Abb. 29: Graphische Nutzerschnittstelle von AmiEddi 1.3*

AmiEddi erlaubt eine Darstellung von Vielfachheiten entsprechend der in Abschnitt 4.1.1 vorgenommenen Definition. Eine Unterscheidung nach Featuretypen sowie weitere Beziehungen zusätzlich zu den Hierarchiebeziehungen sind bisher nicht vorgesehen. Außerdem fehlt bei AmiEddi die Modellierung und Auswertung von Abhängigkeiten, die Beschreibung von Beziehungen zu anderen Modellen sowie die Auswahl von Features bei der Anforderungsanalyse von Produkten. Gegenwärtig wird ein Nachfolger dieses Werkzeugs entwickelt, der Erweiterungen des Modells ermöglichen soll und damit solche Beziehungen darstellen kann [Bednasch 2002].

Für die Modellierung von Abhängigkeiten sind gegenwärtig eigene Werkzeuge in Arbeit [Streitferdt 2003], die eine formale Beschreibung und Auswertung von Abhängigkeiten auf der Basis der OCL zulassen. Ein OCL-Interpreter wertet diese Beschreibungen aus, prüft Produktdefinitionen auf ihre Gültigkeit und ermittelt Abweichungen von einer Produktlinie.

Für die Anforderungsanalyse von Produkten und die dabei notwendige Konfiguration von Features wurde ein Produkt-Konfigurator entwickelt, der über eine konkrete Produktlinie hinaus allgemein verwendbar ist. Er wird in Abschnitt 6.4 vorgestellt. Dieses Werkzeug muss noch um eine XML-Schnittstelle zum Einlesen von Änderungen an Featuremodellen ergänzt werden.

Die Darstellung und Auswertung der Traceability-Beziehungen zwischen Featuremodell und Anforderungsmodell, Objektmodell und Implementierung wurde unter Nutzung des Werkzeugs javadoc [Javadoc] im Quellcode erreicht, für die anderen Modelle wurden entsprechende Referenzen definiert [Sametinger Riebisch 2002].

# 5 Komposition von Modellen mit Hyper/UML

*In diesem Kapitel wird Hyper/UML als Umsetzung des Hyperspace-Ansatzes auf die Modellierungssprache UML vorgestellt. Hyper/UML ermöglicht die modellbasierte Zerlegung und Komposition von Komponenten. Dadurch wird Flexibilität erreicht, weil schnell und einfach neue Produkte erstellt werden können. Durch die Schaffung feingranularer Komponenten wird die Wiederverwendbarkeit und die Wartbarkeit verbessert. Neben den Elementen von Hyper/UML und ihrem Bezug zur UML werden die Integrationsmethoden für die verschiedenen Typen von Elementen vorgestellt, mit denen die Komposition durchgeführt wird.*

Der Hyperspace-Ansatz (siehe 2.3.4) ermöglicht durch seine spezielle Zerlegung die Schaffung feingranularer Komponenten, die eine gute Wartbarkeit, hohe Flexibilität und gute Wiederverwendbarkeit aufweisen. Der Hyperspace-Ansatz ist ein abstraktes Konzept, weil er von konkreten Modellierungs- und Programmiersprachen unabhängig ist. Damit er tatsächlich verwendet werden kann, muss er in konkrete Ausdrucksmittel und Werkzeuge umgesetzt werden. Mit Hyper/J (siehe 2.3.5) existiert eine Umsetzung für die Programmiersprache Java. Das Generator-Werkzeug von Hyper/J ermöglicht eine automatisierte Durchführung der Komposition von Quellcode-Komponenten.

Die Entwicklung von Produktlinien und ihren Komponenten muss mit Hilfe von Modellen und Architekturen erfolgen. Die Vorteile modellbasierter Entwicklung bestehen unter anderem darin, dass auch größere und komplexere Lösungen von den Entwicklern verstanden und überprüft werden können und dass eine Arbeitsteilung bei der Entwicklung erreicht werden kann. Die UML stellt die für Modellierung in der objektorientierten Softwareentwicklung etablierte und standardisierte Modellierungssprache dar. Sie weist jedoch keine Möglichkeiten zur Darstellung von Variabilität auf. Überschneidungsfreie Komponenten lassen sich mit ihrer Hilfe im allgemeinen nicht entwickeln.

Mit Hyper/UML wird eine Umsetzung des Hyperspace-Ansatzes auf die UML in der Version 1.4 vorgestellt. Damit kann eine Zerlegung und Komposition auf Modellebene durchgeführt werden. Hyper/UML bietet gemeinsam mit Hyper/J die Basis für die featuregesteuerte Komposition von Produktlinien auf Modellebene, die als Element der Methode HyperFeatuRSEB im Abschnitt 6.1.2 vorgestellt wird. Die so erstellten Softwaresysteme erfüllen die Forderung nach Schlankheit: sie weisen in Architektur und Quellcode keine Teile auf, die nur wegen der Variabilität eingefügt wurden, aber nicht zur Erfüllung der Anforderungen beitragen. Damit unterscheiden sie sich von Systemen, die mit Wiederverwendungstechniken wie Vererbung oder mit Application Frameworks erstellt wurden.

Wie in Abschnitt 2.3.4 bereits dargestellt wurde, ermöglicht der Hyperspace-Ansatz eine als Multiple Separation of Concerns bezeichnete Zerlegung von Software in beliebig viele Belange. Dadurch wird das Überschneidungsproblem gelöst, das die Wiederverwendung von Komponenten behindert (siehe 2.1). Die Belange als Kriterien einer solchen Zerlegung werden in Form eines Zerlegungsraums, des sogenannten Hyperspace, definiert. Bei der Zerlegung werden die Elemente eines Systems den Belangen entsprechend aufgeteilt, zu denen sie gehören. Diese aufgeteilten Elemente werden in Komponenten zusammengefasst, die als Hyperslices bezeichnet werden. Dabei entsteht je Belang genau eine Komponente. Dieser Sachverhalt wird in der Methode HyperFea-

tuRSEB (siehe Kap. 6) dazu genutzt, je variablem Feature einer Produktlinie eine unabhängige Komponente zu bilden.

Werden UML-Modellelemente nach diesem Ansatz auf Hyperslices aufgeteilt, ist es möglich, die Forderung aus Abschnitt 2.2.2 zu erfüllen und Variabilität mit UML-Mitteln auszudrücken, weil dadurch überschneidungsfreie Komponenten gebildet werden, die mittels automatisierter Komposition zu einem System zusammengefügt werden können. Modularisierung und Entkopplung der Hyperslices tragen zu Verständlichkeit und Änderbarkeit bei. Das Ergebnis der Komposition ist in einem sogenannten Hypermodule enthalten, zu dem alle resultierenden Elemente der einbezogenen Hyperslices gehören, die für ein lauffähiges Softwaresystem benötigt werden.

Zur Umsetzung des Hyperspace-Ansatzes auf die UML muss die Hyper/UML folgende Konzepte realisieren:

- Veränderung der Elemente der UML, damit die Modelle zerlegt und als Hyperslices beschrieben werden können.
    - Anpassung der Elemente der UML an die Zuordnung zu Hyperslices.
    - Schaffung von UML-Elementen für Hyperspace, Hyperslice und Hypermodule.
- Entwicklung von Methoden, die eine Komposition der Hyperslices zu Hypermodules durchführen und alle betroffenen UML-Elemente behandeln können.
- Kopplung der Komposition von Hyperslices der Hyper/UML mit automatisierter Komposition der jeweils dazugehörigen Implementierung in Hyper/J sowie Steuerung der Komposition durch Features in Featuremodellen.
- Unterstützung der Erweiterungen durch UML-Werkzeuge.

Für Erweiterungen der Modellierungssprache UML sind die Mittel Stereotypes, Tagged Values, Constraints und Erweiterungen des Metamodells vorgesehen. Die Erweiterungsmöglichkeiten durch Stereotypes, Tagged Values und Constraints erlauben nicht, die vom Hyperspace-Ansatz geforderten Eigenschaften zu erreichen. Deshalb sind Erweiterungen des Metamodells der UML erforderlich, auch wenn dieser Weg den Nachteil aufweist, dass existierende Werkzeuge angepasst werden müssen, damit die Modell-Erweiterungen bearbeitet werden können.

Das UML-Metamodell enthält Klassen für alle Elemente, aus denen ein UML-Modell aufgebaut werden kann. Jedes Modellelement ist eine Instanz einer Klasse des Metamodells. Damit ist jedes mittels UML formulierte Modell eine Instanz des Metamodells. Hyper/UML nutzt das UML-Metamodell sowie die Object Constraint Language OCL, die als Bestandteil der UML ebenfalls standardisiert ist. Abb. 30 stellt diese Abhängigkeit der Metamodelle als Beziehungen zwischen Paketen von Metamodell-Elementen dar.

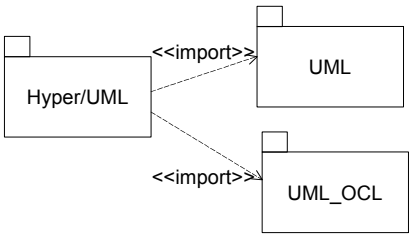



*Abb. 30: Das Metamodell Hyper/UML als Erweiterung des Metamodells der UML*

Eine Klasse des Metamodells definiert über Attribute, Assoziationen, Regeln und Operationen (Methoden) die Eigenschaften der betreffenden Modellelemente. Attribute legen fest, welche Eigenschaften Modellelemente haben können. Assoziationen beschreiben die Möglichkeiten, die die Entwickler bei der Verknüpfung von Modellelementen durch Beziehungen haben. Dabei definieren Regeln die Gültigkeit der Modelle. Regeln wurden mittels OCL formuliert.

## 5.1 Erweiterung der UML-Elemente

Zunächst ist zu untersuchen, welche Elemente des Metamodells der UML für Hyper/UML zu betrachten und zu erweitern sind. Die vollständige Umsetzung des Hyperspace-Ansatzes auf UML erfordert grundsätzlich alle Diagrammarten, deshalb kommen zunächst sämtliche Ausdrucksmittel der UML in Frage. Die Metamodell-Klassen für die Elemente Beziehung, Instanz, Komponente, Knoten und Subsystem werden jedoch aus den folgenden Gründen nicht für Hyper/UML erweitert:

*Beziehungen* verbinden Modellelemente miteinander. Sie werden in Hyper/UML als Teil des Zustands eines Modellelements beschrieben. Sie werden deshalb nicht als eigenständige Modellelemente ins Metamodell aufgenommen. Assoziationen bilden jedoch eine Ausnahme und werden deshalb aufgenommen, denn ein Assoziationsende kann durch ein Attribut der verbundenen Klasse dargestellt werden.

*Instanzen* sind konkrete Ausprägungen von Modellelementen zur Laufzeit des modellierten Systems. Instanzen werden in Objektdiagrammen, Interaktionsdiagrammen und Aktivitätsdiagrammen dargestellt. Der Hyperspace-Ansatz umfasst keine Beschreibung von Systemen zur Laufzeit. Zur Laufzeit werden Instanzen nur von Klassen gebildet, die durch Komposition gebildet wurden. Da bei der Komposition von Hyper/UML-Modellen wieder UML-Modelle entstehen, werden Instanzen nicht in Hyper/UML abgebildet.

*Komponenten und Knoten* bezeichnen in der UML einen implementierten, physisch vorhandenen Systembestandteil. Die Abhängigkeiten von Komponenten und Knoten werden in Komponenten- und Verteilungsdiagrammen der UML modelliert. In der bisherigen Anwendung des hier vorgestellten Ansatzes spielten Komponenten für Zerlegung und Komposition keine Rolle, sie wurden deshalb noch nicht in Hyper/UML abgebildet. Möglicherweise werden sie benötigt, wenn aus Produktlinien Varianten sowohl mit frühen als auch mit späten Bindungszeitpunkten abgeleitet werden, oder wenn Produktlinien für verteilte Systeme modelliert werden sollen. Letzteres ist Gegenstand weiterführender Arbeiten.

*Subsysteme* werden in der UML zum Modellieren von Schnittstellen zu solchen Systemen genutzt, die im betreffenden Modell nicht weiter benötigt werden. Da in den bisher betrachteten Fällen immer alle Bestandteile bei Zerlegung und

Komposition betrachtet wurden, wurde auf die Aufnahme von Subsystemen in Hyper/UML bisher verzichtet, für zukünftige Arbeiten ist eine Abbildung jedoch sinnvoll.

Alle verbleibenden UML-Elemente werden in Hyper/UML abgebildet. Sie sind in Tabelle 1 mit ihren entsprechenden Metamodell-Klassen und -Paketen dargestellt.

*Tabelle 1: Elemente in Hyper/UML und ihr Bezug zum UML-Metamodell*

| Hyper/UML-Elemente | UML-Metamodell-Klasse | UML-Metamodell-Paket |
|---|---|---|
| Pakete | Package | Model Management |
| Modelle | Model | |
| Klassen | Class | Foundation |
| Schnittstellen | Interface | |
| Attribute | Attribute | |
| Assoziationen (zwischen Klassen) | AssociationEnd | |
| Operationen | Operation | |
| Akteure | Actor | Behavioral Elements::Use Cases |
| Use-Cases | UseCase | |
| Zustandsautomaten | StateMachine | Behavioral Elements::State Machines |
| Zustände | SimpleState | |
| Signale | Signal | Behavioral Elements::Common Behavior |
| Aktivitätsgraphen | ActivityGraph | Behavioral Elements::Activity Graphs |
| Aktivitätszustände | ActionState, SubactivityState | |

Weiter ist zu untersuchen, bei welchen Elementen es sich um atomare und bei welchen es sich um zusammengesetzte Elemente handelt, da diese im Metamodell und bei der Komposition unterschiedlich zu behandeln sind. Durch Zusammensetzung von Elementen entsteht ein Baum, der in Abb. 31 dargestellt ist. Atomare Elemente enthalten keine weiteren Elemente, wie Assoziationen, Attribute, Operationen, Akteure und Aktivitätszustände. Sie sind deshalb Blätter des Baums.

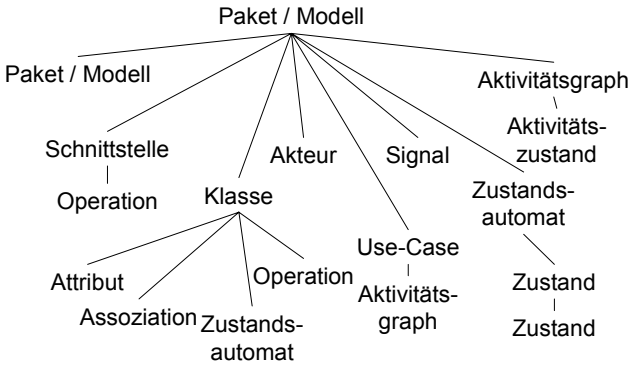



*Abb. 31: Hierarchie zusammengesetzter und atomarer Elemente in Hyper/UML*

Die Metamodellklassen für atomare Elemente werden in Hyper/UML von PrimitiveUnit abgeleitet. Metamodellklassen für zusammengesetzte Elemente werden von CompoundUnit abgeleitet, wie in Abb. 32 dargestellt ist. Hier wird von den Zuordnungen in der UML 1.4 etwas abstrahiert, damit Abhängigkeiten von zukünftigen Änderungen des Standards verringert werden, wie in [Böllert 2002] erläutert wurde.

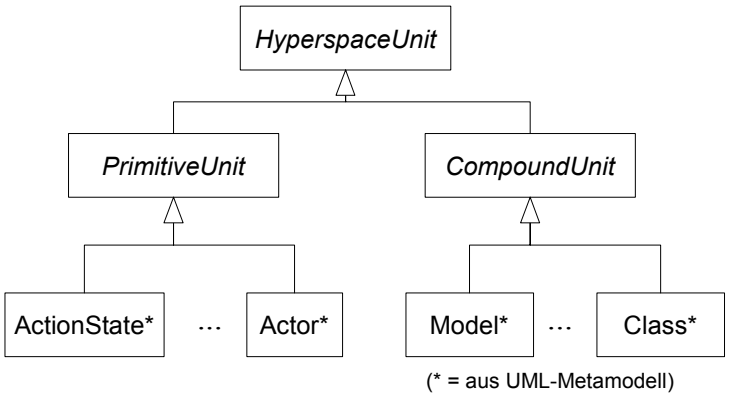



*Abb. 32: Atomare und zusammengesetzte Elemente im Hyper/UML-Metamodell*

Für zusammengesetzte und atomare Elemente sind keine zusätzlichen Regeln erforderlich. Für die Navigation durch die Hierarchie von Hyper/UML-Elementen durch die Integrationsmethoden werden jedoch einige Operationen benötigt, die hier nur erwähnt werden. Ihre Definition mit OCL-Syntax ist in Anhang A von [Böllert 2002] enthalten.

| | |
|---|---|
| CompoundUnits.units() | liefert alle in einem zusammengesetzten Element enthaltenen Elemente zurück |
| HyperspaceUnit.allParents() | liefert die Elementhierarchie zurück, in der sich das Element befindet |
| HyperspaceUnit.parent() | liefert für ein Element eines zusammengesetzten Elements dieses zusammengesetzte Element zurück |

## 5.2 Erweiterung des Metamodells

Für die Kapselung von Belangen, für die Zuordnung von Belangen zu Elementen sowie für die Komposition von Elementen werden die Modellelemente Hyperspace, Hyperslice und Hypermodule des Hyperspace-Ansatzes einschließlich der Integrationsmethoden benötigt (siehe 2.3.4). Die UML bietet lediglich die Modellelemente *Subsystem* und *Package* für die Kapselung von Elementen an, die dafür nicht ausreichen. Im Metamodell von Hyper/UML werden deshalb Erweiterungen des UML-Metamodells vorgenommen. Im folgenden werden Syntax, Regeln und Operationen der neuen Elemente

erläutert. Die OCL-Definitionen der Regeln werden aus Gründen der Lesbarkeit nur als Text wiedergeben, die OCL-Ausdrücke sind im Anhang A von [Böllert 2002] enthalten.

### 5.2.1 Hyperspace

Die Zerlegung eines Systems nach dem Hyperspace-Ansatz erfolgt nach Dimensionen. Ein Hyperspace ordnet die bei dieser Zerlegung identifizierten Elemente den Belangen der Dimensionen zu (siehe 2.3.4). In Hyper/UML wird ein Hyperspace durch die gleichnamige Metamodellklasse definiert, die durch Assoziationen mit ihren Dimensionen in Beziehung steht. Abb. 33 stellt den betreffenden Ausschnitt des Metamodells dar. Die Dimensionen bestehen aus Concern-Objekten, die Belange repräsentieren. In Anlehnung an Hyper/J besitzt jede Dimension einen 0-Belang (NoneConcern). Die Zuordnung zum 0-Belang drückt aus, dass ein Element in einer Dimension für keinen Belang von Interesse ist. Hyper/UML-Elemente sind Instanzen der Metamodellklasse HyperspaceUnit und damit entsprechend der Zerlegung den Belangen eines Hyperspaces zugeordnet.

Die Definition von Regeln beschreibt die Gültigkeit eines Hyper/UML-Modells. Sie folgen den Forderungen des Hyperspace-Ansatzes:

- Die Belange einer Dimension haben unterschiedliche Bezeichner.
- Pro UML-Modell existiert nur ein Hyperspace; die Metamodell-Klasse Hyperspace hat nur ein Objekt. Dies ist notwendig, da der Ansatz mehrere Hyperspaces pro System nicht vorsieht.
- Die Dimensionen eines Hyperspace haben unterschiedliche Bezeichner.
- Jedes Element ist in jeder Dimension nur einem Belang explizit zugeordnet, wie dies im Hyperspace-Ansatz vorgesehen ist. Diese Regel wird in der Klasse HyperspaceUnit verankert.

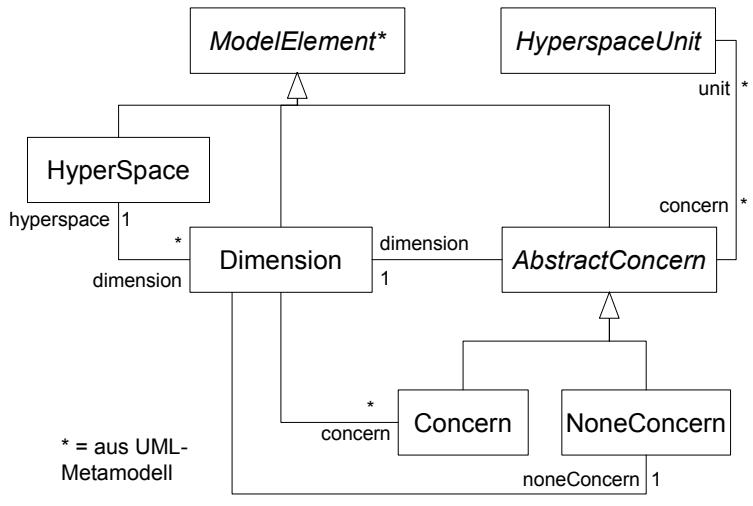



*Abb. 33: Hyperspace im Hyper/UML-Metamodell*

Einige Operationen sind notwendig, damit eine Prüfung dieser und weiterer Regeln möglich ist:

| | |
|---|---|
| Dimension.allConcerns() | liefert die Belange einer Dimension zurück |
| HyperspaceUnit.concernInDimension() | liefert den Belang zurück, dem das Element in der betreffenden Dimension zugeordnet ist. Diese Zuordnung kann explizit oder auch implizit über die Elementhierarchie erfolgen. Ist keine Zuordnung erfolgt, ist das Element dem 0-Belang zugeordnet. |

### 5.2.2 Hyperslice

Hyperslices fassen alle Elemente eines Modells zu einer Komponente zusammen, die zu einem Belang eines Hyperspace gehören. Damit sind sie die entscheidenden Elemente, die eine Zerlegung und Kapselung von Modellen ermöglichen. Sie sollen möglichst wenig Abhängigkeiten voneinander aufweisen, damit eine getrennte Bearbeitung und Betrachtung von Belangen ermöglicht wird. Deshalb müssen Hyperslices deklarativ vollständig sein, d.h. benötigte Elemente werden innerhalb eines Hyperslices bis zu dem Umfang deklariert, den die Regeln für gültige UML-Modelle fordern.

Wenn wie im Beispiel in Abb. 34 über eine Assoziation `outputDevice` auf die Klasse `Device` Bezug genommen und eine Operation `show()` aufgerufen werden soll, die im betreffenden Hyperslice `PdaRemoteControl` nicht enthalten ist, würden im oberen Fall die Assoziation die Forderung nach deklarative Vollständigkeit verletzen. Damit eine deklarative Vollständigkeit erreicht wird, muss diese Klasse zumindest mit der betreffenden Operation eingefügt werden, so dass die Assoziation und der Aufruf dann auf Elemente innerhalb des Hyperslices `PdaRemoteControl` verweisen. Alle Details der Klasse, die nicht benötigt werden, werden weggelassen. In diesem Beispiel wird `Device` nur als abstrakte Klasse mit der benötigten Operation als abstrakter Methode innerhalb des Hyperslice `PdaRemoteControl` definiert. Erst durch die Komposition der Klassen `Device` beider Hyperslices steht eine Operation `show()` zur Verfügung, die das benötigte Verhalten aufweist (siehe auch Abb. 40, S. 103). Eine gewisse Unabhängigkeit ist dadurch gegeben, dass der eine Hyperslice nicht direkt von dem anderen abhängt, in dem die Operation und die Klasse modelliert ist. Eine darüber hinausgehende Abhängigkeit ist trotzdem unvermeidbar, denn irgendein anderer Hyperslice muss die Operation aus diesem Hyperslice und ihr Verhalten modellieren, damit ein System vollständig beschrieben ist.

Die Definition von Hyperslices als Spezialisierung von Packages (Abb. 35) wird durch folgende Regel präzisiert, deren Einhaltung durch Werkzeuge geprüft werden kann, weil sie mittels OCL formuliert wurde:

- Hyperslices können nicht ineinander geschachtelt werden, da sie voneinander unabhängige Module darstellen sollen. Eine Schachtelung würde die Hyperslices in eine hierarchische Beziehung zueinander setzen, wobei Abhängigkeiten zwischen oberer und unteren Ebenen entstehen würden.

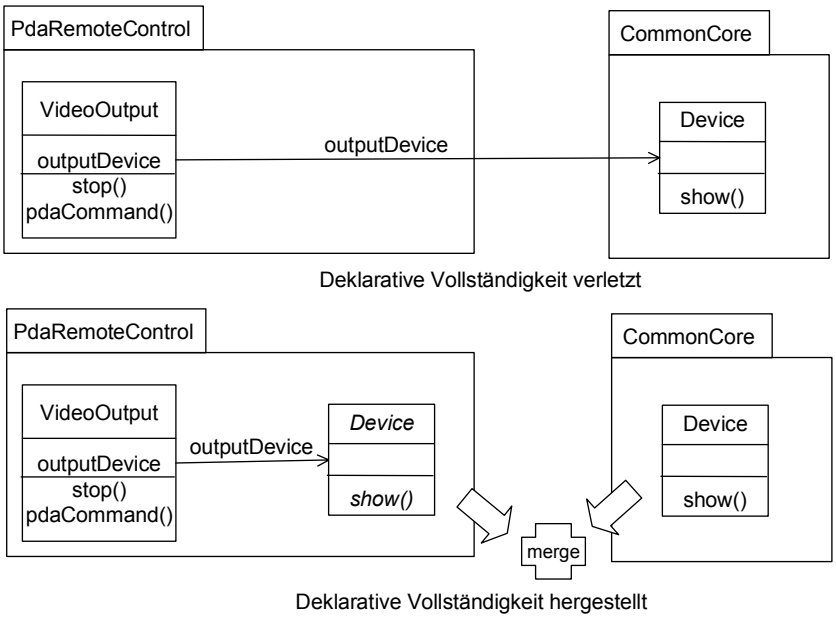



*Abb. 34: Beispiel für deklarative Vollständigkeit*

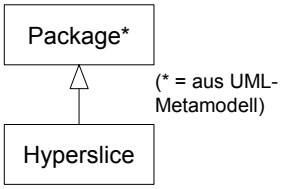



*Abb. 35: Hyperslice im Hyper/UML-Metamodell*

### 5.2.3 Hypermodule

Die Komposition der in Hyperslices zerlegten Belange zu einem kompletten System wird von Hypermodules übernommen. Ein Hypermodule fasst eine Menge von Belangen zusammen und definiert Integrationsmethoden für deren Elemente. Abb. 36 zeigt schematisch den Aufbau eines Hypermodule.

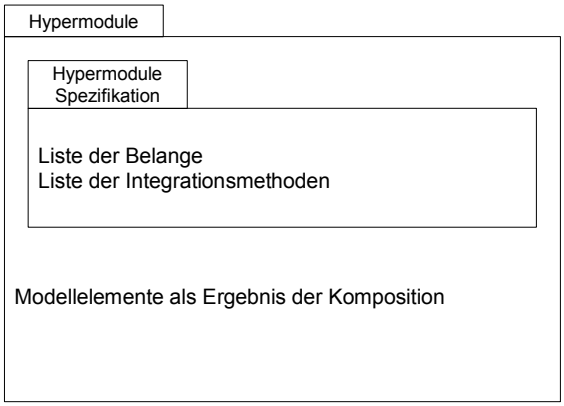



*Abb. 36: Bestandteile eines Hypermodule (schematisch)*

Die Definition im Metamodell ist in Abb. 37 dargestellt. Hypermodules erben von Hyperslices, beispielweise die Forderung nach deklarativer Vollständigkeit. Ein Hy-

permodule steht in Beziehung zu den zu integrierenden Belangen. Die Entwickler können für jeden Hypermodule spezielle Integrationsmethoden angeben, die als Unterklassen von IntegrationRelationship im Metamodell enthalten sind. Die Integrationsmethoden werden aufgrund ihrer Bedeutung in den folgenden Abschnitten separat behandelt.

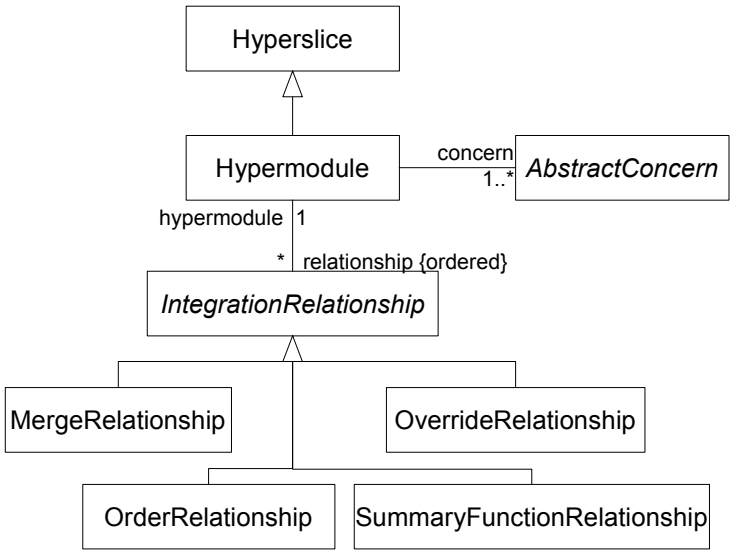



*Abb. 37: Hypermodule im Hyper/UML-Metamodell*

In Anlehnung an Hyper/J wurden im Rahmen der hier vorgestellten Arbeiten für Hyper/UML die Integrationsmethoden Verschmelzen, Ordnen, Summieren und Ersetzen umgesetzt, die als Klassen MergeRelationship, OrderRelationship, SummaryFunctionRelationship und OverrideRelationship im Metamodell enthalten sind (Abb. 37). Die Entscheidung für genau diese vier Integrationsmethoden wurde aufgrund der durchgeführten Fallstudien [Halle 2001][Böllert 2002] getroffen. Sie stellt einen Kompromiss dar: Einerseits ist die Bereitstellung mächtiger Integrationsmöglichkeiten Voraussetzung für flexible Kompositions- und Zerlegungsweisen bei Entwicklung und Anwendung von Produktlinien. Andererseits müssen die Integrationsmethoden möglichst einfach gehalten werden, damit die Entwickler von Hyperslices die Auswirkungen ihrer Arbeit auf potentielle Produkte abschätzen können.

Die Integrationsmethode Verschmelzen (siehe 5.4) führt gleichartige Modellelemente zusammen. Sie wird in der Methode HyperFeatuRSEB als Standard-Integrationsmethode angewendet (siehe 6.1.2). Die Integrationsmethode Ordnen (siehe 5.5) dient der Vergabe einer Reihenfolge beim Verschmelzen von Modellelementen mit Sequentialität. Die Integrationsmethode Summieren (siehe 5.6) wird benutzt, wenn Rückgabewerte von verschmolzenen Operationen zusammenzuführen sind. Die Integrationsmethode Ersetzen (siehe 5.7) wird zum Überschreiben von Elementen benutzt. Die Ergänzung des Ansatzes um weitere Integrationsmethoden ist möglich und vorgesehen, wenn zukünftige Erfahrungen und Anforderungen dies sinnvoll erscheinen lassen. Die Integration mittels Kollaborationen (siehe 7.1.2) stellt eine solche Ergänzung dar, die Laufzeit-Variabilität ermöglicht.

Die Integrationsmethoden werden einem Hypermodule als geordnete Liste beigefügt, weil die Reihenfolge der Ausführung für die Integrationsmethoden Verschmelzen und Ersetzen von Bedeutung ist (siehe 2.3.4). Die Festlegung der Integrationsmethoden und ihrer Reihenfolge obliegt dem Entwickler bei der Festlegung der zutreffenden Dimensionen, wie in Abschnitt 6.1.2 näher erläutert wird.

Zur Definition des Hypermodules gehören außerdem die Integrationsmethoden für Hyper/UML; sie werden zwecks besserer Gliederung dieser Arbeit in den Abschnitten 5.4 bis 5.7 erläutert. Dabei wird zur Vermeidung von Wiederholungen bei der Beschreibung der Integrationsmethoden für die verschiedenen Typen von Modellelementen weitgehend ein einheitliches Beschreibungsschema verwendet:

*Vorbedingungen*: Die Vorbedingungen enthalten Anforderungen an die zu integrierenden Elemente. Eine häufige Forderung besteht beispielsweise darin, dass die Elemente in bestimmten Eigenschaften übereinstimmen müssen. Die Erfüllung dieser Anforderungen ist Aufgabe der Entwickler, sie können dabei und bei der Prüfung der Erfüllung von Werkzeugen unterstützt werden.

*Nachbedingungen*: Die Nachbedingungen beschreiben das Ergebnis der Integration dieser Elemente und damit auch deren Durchführung. Zur Illustration werden häufig Beispiele aus der Produktlinie „Digitales Video-System“ verwendet.

*Ausnahmen*: Mögliche Fehlerquellen können trotz Prüfung der Vorbedingungen dazu führen, dass die Integration nicht wie vorgesehen durchgeführt werden kann. In solchen Fällen erfolgt ein Abbruch der Integration. Die Entwickler müssen die Fehlerquellen dann beseitigen, wobei sie von Werkzeugen unterstützt werden können.

## 5.3 Ablauf der Komposition eines Systems

Hyperslices enthalten die Modellelemente als Ergebnis einer Zerlegung nach Belangen (Separation of Concerns). Die Komposition dieser Hyperslices zu einem kompletten Softwaresystem erfolgt innerhalb eines Hypermodules. Will ein Entwickler ein Softwaresystem durch eine solche Komposition erzeugen, definiert er ein Hypermodule, in dem er die relevanten Belange und die Art ihrer Integration festlegt.

Jeder Hyperslice ist einem Belang eindeutig zugeordnet, außerdem ist er durch die deklarative Vollständigkeit von anderen Hyperslices entkoppelt. Durch die Integration entsteht ein Hypermodule, das alle integrierten Elemente enthält und das ebenfalls deklarativ vollständig sein soll, es darf keine Regel des UML- und des Hyper/UML-Metamodells verletzen. Ob das Hypermodule bezüglich der Aufgabenstellung korrekt ist, kann allerdings mit den derzeit verfügbaren Beschreibungsmitteln nicht geprüft werden. Möglicherweise führen die Arbeiten zu formalen Beschreibungsmitteln und zu ausführbaren UML-Modellen, beispielweise im Umfeld der Model-Driven Architecture MDA [MDA] hier zu Forstschritten. Derzeit sind Tests zur Prüfung der Korrektheit notwendig. Auf Maßnahmen zur Generierung von Testfällen und zum Test wird in den Abschnitten 6.1.3 und 9.3 eingegangen.

Die Integration der Modellelemente erfolgt in zwei Schritten, im dritten Schritt wird aus dem entstehenden Modell dann ein lauffähiges System erzeugt:

**Schritt 1: Übernehmen aller relevanten Elemente in das Hypermodule**

Die durch die festgelegten Belange definierten Hyperslices werden in das betreffende Hypermodule kopiert. Ihre Elemente werden entsprechend der Namensräume und der Paketzuordnung in eine gemeinsame Namensraum-Hierarchie eingeordnet. Kopierte Hyperslices werden zu gewöhnlichen UML-Paketen, da eine Schachtelung von Hyperslices wegen der Forderung nach Unabhängigkeit unzulässig ist.

**Schritt 2: Integration der Elemente mit Integrationsmethoden**
Die Integrationsmethoden beschreiben, in welcher Weise die Zusammenführung gleicher oder überdeckender Elemente erfolgt, damit Konflikte zwischen gleichen Elementen aufgelöst werden. Im Fall einer orthogonalen Zerlegung würden bei der Komposition von Hyperslices keine Konflikte entstehen, da deren Elemente disjunkt wären. Eine solche Zerlegung wird jedoch im Hyperspace-Ansatz nicht gefordert. Außerdem sind durch die Forderung nach deklarativer Vollständigkeit der Hyperslices zumindest die Deklarationen von assoziierten Elementen wie beispielweise Klassen in mehreren Hyperslices enthalten. Zur Beschreibung von Integrationsmethoden gehören die Angabe der betreffenden Elemente aus verschiedenen Hyperslices sowie ihre Beziehungen zueinander (Vorbedingungen), die Art der Verknüpfung und die resultierenden Elemente (Nachbedingungen) und Folgen von abweichenden Bedingungen (Ausnahmen). Die folgenden Abschnitte beschreiben Syntax, Regeln und Semantik der Integrationsmethoden jeweils für die verschiedenen Hyper/UML-Elemente oder für gleichartige Gruppen von ihnen.

**Schritt 3: Überführung in ein lauffähiges System**
Als Ergebnis der Komposition entsteht ein Hypermodule, das alle vorher festgelegten Belange abdeckt und die Anforderungen an ein gültiges UML-Modell erfüllt. Hyper/UML wurde in Abstimmung mit Hyper/J entwickelt, damit eine gleichartige Komposition von Objektmodell und (Java-) Quellcode möglich ist. Für alle Hyper/UML-Modellelemente werden dazu entsprechende Implementierungen in der Programmiersprache Hyper/J geschaffen, üblicherweise in engem zeitlichen Zusammenhang mit der Erarbeitung der Modelle. Dieser Quellcode wird genauso wie die Modellelemente in Hyperslices zerlegt und gekapselt.

Eine durchgängige Komposition eines lauffähigen Systems ist dadurch möglich, dass der Entwickler auch für Hyper/J einen Hypermodule definiert, die Belange vom Hypermodule für Hyper/UML überträgt und die notwendigen Integrationsmethoden definiert. Das Hyper/J-Werkzeug übernimmt anschließend die Überführung der Modellelemente in Code. Dieser Code wird von den üblichen Java-Werkzeugen abgearbeitet, womit ein ausführbares Softwaresystem vorliegt.

## 5.4 Integrationsmethode Verschmelzen

Die Integrationsmethoden steuern die Komposition von Modellelementen, die in mehr als einem zu integrierenden Hyperslice auftreten. Die Integrationsmethode Verschmelzen vereinigt zwei oder mehr Modellelemente gleichen Typs miteinander. Sie wird im Metamodell durch die Klasse MergeRelationship umgesetzt (Abb. 38). Sie stellt die mächtigsten Möglichkeiten zur Komposition bereit, da Eigenschaften wie auch Einschränkungen u.ä. von Elementen des gleichen Typs vereinigt werden. Dazu müssen die Wechselwirkungen zwischen Elementen verschiedener Hyperslices vom Entwickler geprüft und ggf. gesteuert werden. Beispiele für das Verschmelzen sind in den folgenden Abschnitten enthalten.

Die Semantik der Integrationsmethode unterscheidet sich je nach Typ der Modellelemente. Deshalb werden zunächst die Syntax sowie der allgemein gültige Teil der Regeln und der Semantik erläutert. Abb. 38 zeigt die graphische Syntax der Umsetzung der Integrationsmethode, die mit mehreren Hyperslice-Elementen verbunden ist. Diese können in die zwei Kategorien Zielelemente (targetUnit) und Quellelemente (sourceUnit) eingeteilt werden. In einem Zielelement sind nach dem Verschmelzen die Bestandteile des oder der Quellelemente enthalten; letztere sind anschließend nicht mehr enthal-

ten. Der Zugriff einer nachfolgenden Integrationsmethode auf ein verschmolzenes (und anschließend entferntes) Element ist damit nicht mehr möglich.

Sind die zu verschmelzenden Elemente aus anderen zusammengesetzt, wird für diese die Integrationsmethode in gleicher Weise angewandt. In den Quellementen enthaltene gleichartige Elemente sind also anschließend im Zielelement als verschmolzene Elemente enthalten, während alle weiteren zusätzlich aufgenommen werden. Von dieser Regel kann abgewichen werden, wenn der Entwickler für das betreffende Hypermodule die Integrationsmethode Ersetzen für Elemente bestimmter Typen angibt.

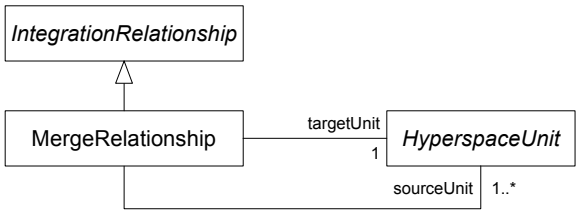



*Abb. 38: Integrationsmethode Verschmelzen im Metamodell von Hyper/UML*

Folgende Regeln beschreiben die Integrationsmethode genauer. Diese Regeln werden durch die unten genannten Vorbedingungen sowie durch die Vorbedingungen für die einzelnen Elementtypen (siehe 5.4.1 bis 5.4.7) ergänzt, wobei jeweils das Beschreibungsschema aus Abschnitt 5.2.3 angewendet wird. Die Regeln werden hier als Text angegeben, für die OCL-Ausdrücke sei auf Anhang A von [Böllert 2002] verwiesen:

- Das Zielelement ist nicht gleichzeitig Quellelement, da ein Element nicht mit sich selbst verschmolzen werden darf.
- Die zu verschmelzenden Elemente sind den im Hypermodule angegebenen Belangen zugeordnet, da anderenfalls unbeteiligte Elemente in die Integration einbezogen würden.
- Die zu verschmelzenden Elemente gehören zu den in Hyper/UML definierten Typen. Für die einzelnen Typen werden nachfolgend weitere Regeln definiert.
- Die zu verschmelzenden Elemente sind vom gleichen Typ. Elemente unterschiedlicher Typen können nicht verschmolzen werden.

Die folgenden Operationen sind für die Prüfung der Regeln notwendig. Dabei werden mit ModelElement und Operation auch Klassen des UML-Metamodells erweitert.

MergeRelationship.units() gibt die zu verschmelzenden Elemente zurück

ModelElement.correspondsTo()
: prüft, ob das Modellelement mit dem übergebenen Element übereinstimmt, und liefert in diesem Fall true zurück. Wenn für die im Folgenden beschriebenen Modellelemente weitere Regeln gelten, wird die Operation in den betreffenden Unterklassen überschrieben.

Operation.correspondsTo()
: prüft, ob die Operation mit der übergebenen Operation übereinstimmt, und liefert in diesem Fall true zurück. Zusätzlich zu ModelElement wird hier die Signatur überprüft.

Die Semantik der Integrationsmethode Verschmelzen ist für Einschränkungen, Eigenschaftswerte und Abhängigkeiten von Elementen im Wesentlichen gleich. Unterschiede sind in den unten folgenden Abschnitten dargestellt.

**Vorbedingungen**
Den zu verschmelzenden Elementen ist kein Eigenschaftswert mit gleichem Namen, aber unterschiedlichem Wert zugeordnet. Anderenfalls würde ein Konflikt auftreten, der nicht durch Regeln lösbar wäre. Hätte beispielsweise eine Klasse `Settings` des Hyperslices `CommonCore` den Eigenschaftswert `persistence=false` und die Klasse `Settings` eines weiteren Hyperslices `PersonalProfiles` den Wert `persistence=true`, so wäre nicht klar, welchen Wert die integrierte Klasse haben soll.

**Nachbedingungen**
Das als Ergebnis entstehende Ziel-Element behält seinen Namen, soweit vorher einer vorhanden war. Die Namen von Quellelementen werden ignoriert. Referenzen auf Quellelemente werden an den Namen des Ziel-Elements angepasst. Besondere Maßnahmen zur Erfüllung dieser Nachbedingung sind vor allem für Referenzen im Text notwendig, wie in Kommentaren und Einschränkungen.

Eigenschaftswerte der Quellelemente werden dem Zielelement zusätzlich zugeordnet. Durch Kopieren entstandene redundante Eigenschaftswerte werden entfernt.

Einschränkungen der Quellelemente werden dem Zielelement zusätzlich zugeordnet, was einer UND-Verknüpfung der Einschränkungen entspricht.

Abhängigkeiten der Quellelemente von anderen Modellelementen werden dem Zielelement zusätzlich zugeordnet. Abhängigkeiten anderer Modellelemente von Quellelementen gehen statt dessen vom Zielelement aus. Durch Kopieren entstandene redundante Abhängigkeiten werden entfernt.

**Ausnahmen**
Widersprechen sich Einschränkungen, sind UML-Modelle ungültig. Für das Verschmelzen können solche Widersprüche zwischen Einschränkungen auftreten, die für das Zielelement gelten. In einem solchen Fall bricht die Integration mit einer Fehlermeldung ab; die Entwickler des Hypermodules und / oder der Hyperslices müssen den Konflikt lösen. Die Widerspruchsfreiheit von Einschränkungen kann nur dann von Werkzeugen überprüft werden, wenn sie in einer formalen Sprache wie OCL formuliert sind.

### 5.4.1 Verschmelzen von Paketen, Modellen und Hyperslices

Das Verschmelzen von Hyperslices führt zur Komposition von Softwaresystemen, weil alle anderen Modellelemente in den Hyperslices enthalten sind und ebenfalls sukzessive verschmolzen werden, sofern für sie keine abweichenden Integrationsmethoden spezifiziert sind.

Für die Semantik beim Verschmelzen dieser Modellemente kommen zu den unter 5.4 genannten Vor- und Nachbedingungen und Ausnahmen vor allem Regeln zur Hierarchie enthaltener Elemente hinzu.

**Vorbedingungen**
Pakete sind meist Teil einer Hierarchie von Enthaltenseins-Beziehungen. Durch das Verschmelzen von Paketen verschiedener Hierarchieebenen können Widersprüche entstehen, weshalb zu verschmelzende Pakete einer identischen Hierarchieebene ange-

hören müssen. Diese Bedingung ist erfüllt, wenn eine beliebige Integrationsmethode diese Identität herstellt, beispielweise durch das Verschmelzen übergeordneter Pakete. Deshalb wird die Bedingung unter Einbeziehung aller Integrationsmethoden geprüft.

Für Modelle und Hyperslices gilt die gleiche Bedingung.

Das Hinzufügen eines variablen Features soll dies illustrieren. Im Beispiel in Abb. 39 ist der Hyperslice `LogoRemoval` in einem Paket `Editing` enthalten, das als Bestandteil des Pakets `VarPlugins` Teil der Produktlinie ist, die im Paket `DVPL` zusammengefasst ist. Soll `LogoRemoval` mit dem Hyperslice `RemoveMovingLogo` verschmolzen werden, muss dieser ebenfalls im Paket `Editing` enthalten und in der gleichen Weise in die Pakethierarchie eingeordnet sein, damit die Vorbedingungen erfüllt werden.

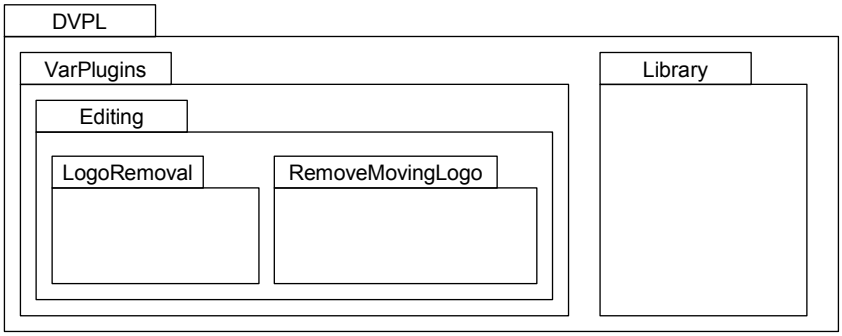



*Abb. 39: Pakethierarchie des Beispiels*

**Nachbedingungen**
Ein Zielpaket enthält zusätzlich die Elemente der Quellpakete, wobei gleichnamige Elemente miteinander verschmolzen werden. Das Gleiche gilt für Hyperslices und Modelle.

### 5.4.2 Verschmelzen von Klassen und Schnittstellen

Für die Semantik der Integrationsmethode Verschmelzen sind die unter 5.4 genannten Vor- und Nachbedingungen und Ausnahmen in Hinblick auf Sichtbarkeit, Zusammensetzungen und Vererbungsbeziehungen zu konkretisieren.

**Vorbedingungen**
Miteinander zu verschmelzende Klassen müssen alle gleichermaßen abstrakt oder konkret, passiv oder aktiv sein. Eine aktive Klasse ist dadurch gekennzeichnet, dass alle ihre Instanzen in ihrem eigenen Laufzeitprozess (genauer: Thread) ablaufen.

**Nachbedingungen**
Eine Zielklasse enthält zusätzlich Attribute, Assoziationen, Operationen, Zustandsautomaten und Schnittstellen der Quellklassen, wobei übereinstimmende Attribute, Assoziationen, Operationen, Zustandsautomaten und Schnittstellen miteinander verschmolzen werden.

Eine Zielklasse erbt zusätzlich von denjenigen Klassen, von denen die Quellklassen erben. Unterklassen der Quellklassen erben jetzt von der Zielklasse. Dabei werden redundante Generalisierungsbeziehungen entfernt. Für Schnittstellen und Realisierungsbeziehungen als spezielle Generalisierung gilt die gleiche Semantik; zusätzlich

werden Operationen entfernt, die unter Einschluss der gesamten Vererbungshierarchie redundant definiert sind.

Nehmen Modelle der UML auf Instanzen Bezug (wie bei Interaktionen und in Aktivitätsgraphen), so sind statt Instanzen von Quellklassen solche der jeweiligen Zielklasse relevant.

**Ausnahmen**
Vererbungshierarchien der UML dürfen keine Zyklen enthalten. Entstehen durch das Verschmelzen von Klassen oder Schnittstellen Zyklen in der Vererbungshierarchie, führt dies zu einem Fehler. Entsteht eine Vererbungsbeziehung mit mehreren Oberklassen (Mehrfachvererbung), so führt dies zu Fehlern bei der Umsetzung in Hyper/J. Spätere Umsetzungen des Hyperspace-Ansatzes in andere Programmiersprachen können Mehrfachvererbung erlauben, wenn die Programmiersprache dies vorsieht.

Die UML erlaubt nicht, dass in einer Oberklasse definierte Attribute und Assoziationen in einer Unterklasse nochmals unter gleichem Namen definiert werden. Führt eine Verschmelzung zu einem solchen Ergebnis, resultiert daraus ein Fehler.

In der UML muss bei der Realisierung von Schnittstellen durch eine Klasse diese alle definierten Operationen implementieren. Werden als Ergebnis einer Verschmelzung nicht mehr alle Operationen implementiert (beispielweise nach Umordnung von Realisierungsbeziehungen), führt dies zu einem Fehler. Allerdings könnten andere Integrationsmethoden des betreffenden Hypermodules dieses Problem beheben.

Abb. 40 (S. 103) zeigt als Beispiel die Verschmelzung der Klasse `VideoItem`, die in drei Hyperslices enthalten ist. Die Attribute und Operationen dieser Klassen werden dabei ebenfalls verschmolzen.

### 5.4.3 Verschmelzen von Attributen, Assoziationen und Operationen

Attribute, Assoziationen und Operationen von Klassen stellen einen wesentlichen Teil von Eigenschaften und Verhalten in objektorientierten Modellen dar, entsprechend hoch ist ihre Bedeutung für eine erfolgreiche Komposition von Software. Die für die Integrationsmethode Verschmelzen allgemein definierte Semantik muss nur um wenige Punkte ergänzt werden, die sich auf Gleichartigkeit von Attributen, Assoziationen und Operationen beziehen.

**Vorbedingungen**
Attribute, Assoziationen und Operationen müssen immer im Zusammenhang mit ihren Klassen verschmolzen werden, weil sie von anderen Modellelementen auch immer über ihre Klassen referenziert werden. Losgelöst von einer Klasse wäre der Aufruf einer Operation und der Zugriff auf eine Assoziation oder auf ein Attribut ungültig.

Zu verschmelzende Attribute müssen gleiche Eigenschaften aufweisen. Dazu gehören Typ (Instanzen- oder Klassenvariablen), Änderbarkeit (changeable, frozen, addOnly), Initialwert (falls vorhanden), Vielfachheit und Ordnung.

Zu verschmelzende Assoziationen müssen auf Teilnehmer gleichen Typs verweisen. Sie müssen alle allgemeinen Typs, vom Typ Aggregation oder vom Typ Komposition sein. Außerdem müssen Änderbarkeit, Navigationsrichtung, Vielfachheit und Ordnung übereinstimmen.

**Nachbedingungen**
Zielattribute übernehmen den Initialwert, wenn mindestens ein Quellattribut einen solchen definiert. Lesende oder schreibende Zugriffe auf Quellattribute beziehen sich nun auf das betreffende Zielattribut. Für Zugriffe auf Assoziationen gilt die gleiche Semantik.

Zieloperationen enthalten die Implementierungen der betreffenden Quelloperationen. Die Reihenfolge der Ausführung beim Aufruf kann vom Entwickler durch eine Integrationsmethode Ordnen festgelegt werden, anderenfalls ist sie nicht definiert. Die Parameterauswertung erfolgt in der Reihenfolge der Implementierungen: die Implementierung aus der ersten Quelloperation erhält die vom Aufrufer übergebenen Parameter und übergibt die Resultate an die Implementierung der zweiten Quelloperation, die sie dann an die nächste übergibt, und so weiter. Der Aufrufer erhält folglich den letzten Wert der Parameter zurück. Für Rückgabeparameter kann dieses Verhalten durch die Integrationsmethode Summieren im Hypermodule so verändert werden, dass die Resultate mehrerer Teile zurückgegeben werden (siehe Abschnitt 5.6).

Zieloperationen erhalten einen Defaultwert, wenn mindestens eine Quelloperation einen solchen definiert.

Aktionen, die zur Laufzeit eine Quelloperation aufrufen würden, rufen nach der Verschmelzung die betreffende Zieloperation auf (CallAction). Solche Aktionen sind zum Beispiel Nachrichten zwischen Objekten bei Interaktionen, in Zustandsautomaten entry-, exit-Aktionen oder Aktionen von Zuständen oder Transitionen.

Ereignisse, die von einer Quelloperation ausgelöst werden, werden nach der Verschmelzung von der betreffenden Zieloperation ausgelöst. Transitionen in Zustandsautomaten schalten demzufolge aufgrund der Ausführung von Zieloperationen.

Abb. 40 zeigt als UML-Klassendiagramm ein Beispiel für das Verschmelzen von drei Hyperslices, die jeweils eine Klasse `VideoOutput` enthalten. Es zeigt leicht vereinfacht, wie bei Produkten der Produktlinie Digitaler Videorekorder die Bildausgabe am Fernsehgerät durch die Features PDA-Fernsteuerung und Infrarot-Fernsteuerung beeinflusst wird. Die Operation `pdaCommand()` blendet Menüs und Bildschirmmasken in den Video-Ausgabestrom ein, wenn diese durch Fernbedien-Kommandos des PDA aktiviert werden. Die Operation `onScreenDisplay()` blendet Menüs in den Video-Ausgabestrom ein, die durch Befehle der Infrarot-Fernbedienung aktiviert wurden. Die Operationen `start()`, `stop()` und `selectVideoItem()` steuern die Anzeige eines Videos oder eines Hintergrundbildes für den Bereitschaftszustand des Gerätes.

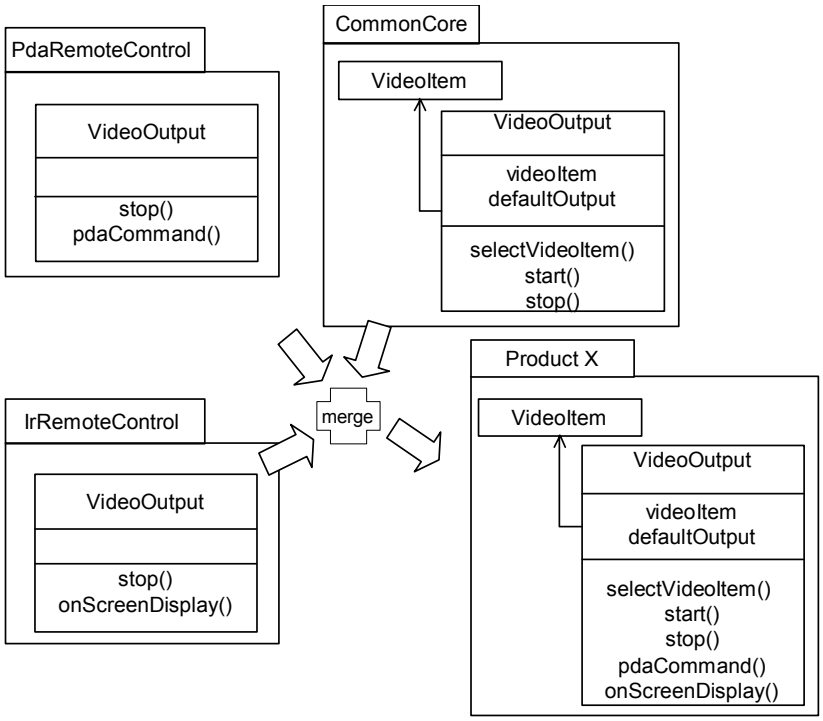



*Abb. 40: Beispiel für Verschmelzen von Klassen mit ihren Attributen und Operationen*

`ProductX` ist ein Hypermodule, das die Erstellung eines Produkts mit beiden Features PDA-Fernsteuerung und Infrarot-Fernsteuerung steuert. Dazu werden drei Hyperslices dadurch verschmolzen, dass die Klassen `VideoItem` und `VideoOutput` als ihre Elemente wiederum verschmolzen werden. In jedem der drei Hyperslices ist eine Klasse `VideoOutput` enthalten, alle drei Klassen werden also zu einer verschmolzen, dabei werden ihre Elemente wiederum verschmolzen. Die Operation `stop()` ist in mehr als einem Hyperslice enthalten, alle drei Methodenrümpfe werden verschmolzen. Die Reihenfolge der Implementierungen wird durch die Integrationsmethode Verschmelzen nicht definiert, hierfür wird die Integrationsmethode Ordnen (siehe 5.5) benötigt. Keins der Attribute und keine der anderen Operationen ist in mehr als einem Hyperslice enthalten, Verschmelzen führt nur zu einer einfachen Übernahme dieser Elemente der Klassen.

Die Entwickler der drei Hyperslices haben die Zerlegung der Klasse `VideoItem` und der Operation so vorgenommen, dass die erläuterte Integration zu funktionierenden Produkten führt. Dabei wurden sie bezüglich der Wahl der Integrationsmethoden, Signaturen der Operationen, Namensvergabe, Sichtbarkeit und Hierarchie durch eine übergeordnete Instanz (das sog. Hyperspace-Engineering, siehe 6.1.2) koordiniert.

### 5.4.4 Verschmelzen von Akteuren und Use-Cases

Für Akteure und Use-Cases muss die unter 5.4 für die Integrationsmethode Verschmelzen allgemein definierte Semantik vor allem für die Beziehungen zu Akteuren und Use-Cases sowie um include-, extend- und Vererbungsbeziehungen erweitert werden. Außerdem sind Use-Case-Beschreibungen zu berücksichtigen, die zwar nicht mit der UML [UML 2001] standardisiert wurden, jedoch breite Anwendung finden. In der Literatur werden verschiedene Schemata für strukturierte Texte vorgeschlagen, beispielweise in [Cockburn 1997], [Oestereich 1999] und [Hitz Kappel 2003]. Hier wird auf die gemeinsame Grundmenge dieser Beschreibungen Bezug genommen, die aus den Bestandteilen *Vorbedingungen*, *Durchführung*, *Nachbedingungen* und *Ausnahmen* besteht. Der Be-

standteil *Durchführung* beschreibt den normalen Ablauf als Folge nummerierter Schritte, der durch zusätzliche Schritte im Bestandteil *Ausnahmen* um Abweichungen vom normalen Ablauf erweitert wird. Der Bezug zwischen Schritten im Bestandteil *Ausnahmen* auf Schritte der *Durchführung* wird durch die Nummerierung hergestellt. In der Use-Case-Beschreibung des Beispiels in Abb. 41 links oben schließt sich der Schritt 2a1 aus *Ausnahmen* an den Schritt 2 von *Durchführung* an, wenn die bei 2a genannte Bedingung erfüllt ist.

**Vorbedingungen**
Jeweils zu verschmelzende Use-Cases sind alle abstrakt oder alle konkret.

**Nachbedingungen**
Ein Zielakteur kommuniziert zusätzlich mit den Use-Cases, zu denen die Quellakteure Kommunikationsbeziehungen besitzen. Umgekehrt kommunizieren Use-Cases statt mit dem Quellakteur nun mit dem Zielakteur. Redundante Kommunikationsbeziehungen können entfernt werden.

Ein Zielakteur erbt zusätzlich von den Akteuren, von denen die Quellakteure erben. Untergeordnete Akteure, die von einem Quellakteur erben, erben nun vom Zielakteur. Redundante Vererbungsbeziehungen können entfernt werden. Die gleiche Semantik gilt auch für Vererbungsbeziehungen zwischen Use-Cases.

Ein Ziel-Use-Case enthält zusätzlich die Use-Case-Beschreibungen der Quell-Use-Cases:

> *Vorbedingungen* von Quell-Use-Cases enthält der Ziel-Use-Case zusätzlich, was einer logischen UND-Verknüpfung entspricht.
>
> Ein Ziel-Use-Case enthält zusätzlich die *Durchführung* der Quell-Use-Cases. Deren Reihenfolge kann bei Bedarf durch Angabe zusätzlicher Ordnen-Integrationsmethoden im Hypermodule festgelegt werden, anderenfalls ist sie undefiniert. Die Reihenfolge der Schritte innerhalb einer *Durchführung* bleibt erhalten.
>
> Ein Ziel-Use-Case enthält zusätzlich alle *Ausnahmen* der Quell-Use-Cases. Beziehen sich diese *Ausnahmen* auf bestimmte Schritte einer *Durchführung*, so werden die Bezüge an die ggf. geänderte Nummerierung angepasst.

Ein Ziel-Use-Case enthält zusätzlich alle Erweiterungsstellen der Quell-Use-Cases. Da Erweiterungsstellen durch ihren Namen und den Bezug auf einen anderen Use Case beschrieben werden, und die Identität von Bezügen der Erweiterungsstellen aufgrund der vagen Definition in der Version von [UML 2001] nicht prüfbar ist, kann die Redundanz nur anhand der Namen geprüft werden. Redundante Erweiterungsstellen werden beim Verschmelzen entfernt, ebenso wie alle Bezüge. Bei der Migration auf die nächste Version der UML kann die Behandlung von Erweiterungsstellen voraussichtlich präzisiert werden.

Ein Ziel-Use-Case erweitert zusätzlich diejenigen Use-Cases, die durch Quell-Use-Cases mittels extend erweitert werden. Erweiterungen von Quell-Use-Cases durch extend beziehen sich nun auf den Ziel-Use-Case. Entstehen dabei identische extend-Beziehungen, können diese verschmolzen werden. Sind mit extend-Beziehungen Bedingungen verknüpft, werden diese dabei UND-verknüpft.

Ein Ziel-Use-Case erhält zusätzliche include-Beziehungen zu denjenigen Use-Cases, die von Quell-Use-Cases mittels include eingeschlossen werden. Use-Cases, die Quell-

Use-Cases einschließen, werden mit dem Ziel-Use-Case durch include verbunden. Redundante include-Beziehungen werden wieder entfernt.

Ein Ziel-Use-Case erhält zusätzlich die Aktivitätsgraphen der Quell-Use-Cases, die in den Aktivitätsdiagrammen enthalten sind. Übereinstimmende Aktivitätsgraphen werden entsprechend 5.4.7 verschmolzen.

**Ausnahmen**
Die Behandlung von Ausnahmefällen ist immer dann erforderlich, wenn die Regeln der Hyper/UML-Modelle oder UML verletzt werden. In den meisten Fällen ist nur ein Abbruch mit Fehlermeldung möglich.

Entstehen durch das Verschmelzen von Akteuren oder Use-Cases, die in Vererbungsbeziehungen stehen, Zyklen in der Vererbungshierarchie, erfolgt ein Abbruch. Die UML lässt keine Zyklen in Vererbungshierarchien zu.

Stehen Bedingungen von extend-Beziehungen einer verschmolzenen Use-Case-Beschreibung im Widerspruch zueinander, bricht die Integration mit einer Fehlermeldung ab. Die UML lässt hier keine Widersprüche zu. Die Widerspruchsfreiheit kann nur geprüft werden, wenn die Bedingungen in einer formalen Sprache wie OCL definiert sind.

Stehen *Vor*- oder *Nachbedingungen* einer verschmolzenen Use-Case-Beschreibung im Widerspruch zueinander, kann dies aufgrund des informalen Charakters der Beschreibung nicht ermittelt werden.

Enthält nach dem Verschmelzen ein Use-Case Erweiterungsstellen mit dem gleichen Namen wie übergeordnete Use-Cases, bricht die Integration mit einer Fehlermeldung ab, da dies zu einem nicht gültigen UML-Modell führen würde.

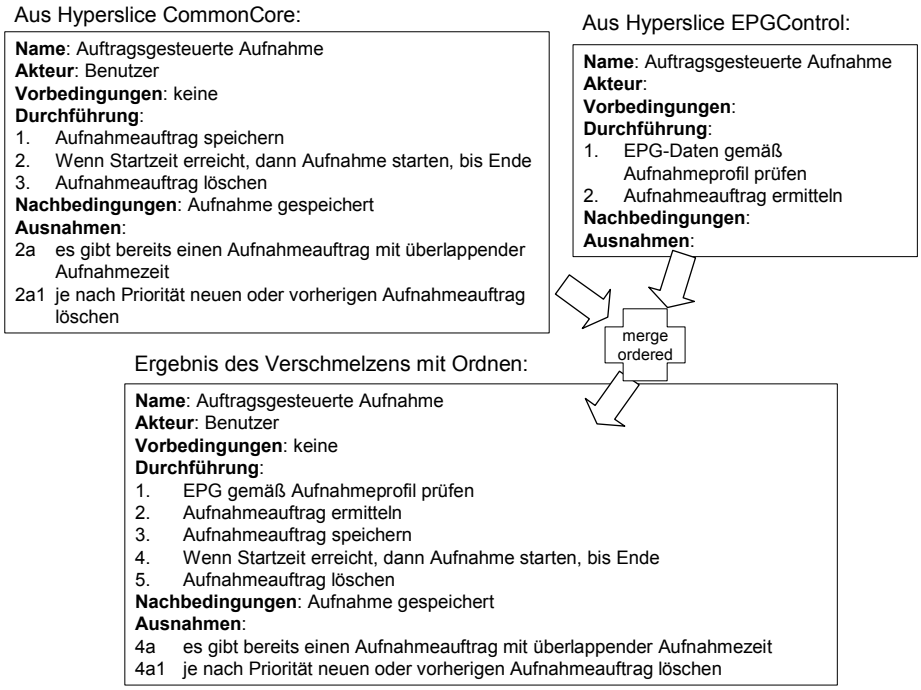



*Abb. 41: Beispiel für das Verschmelzen von Use-Case-Beschreibungen*

Das Beispiel Abb. 41 zeigt das Verschmelzen von zwei Use-Case-Beschreibungen. Das Beispiel bezieht sich auf das variable Feature *Aufnahme mit EPG-Steuerung* (siehe

Abb. 23, S. 70), bei dem Aufnahmeaufträge durch den gewünschten Inhalt vorgegeben werden anstatt durch Beginn, Ende und Kanal wie sonst üblich. Die Inhalte der Bestandteile *Durchführung* beider Quell-Use-Cases werden unter der durch die Integrationsmethode Ordnen (siehe 5.5) definierten Reihenfolge zusammengefügt; die Nummerierung der Schritte im Bestandteil *Ausnahmen* werden entsprechend angepasst. Bei *Vorbedingungen* und *Nachbedingungen* finden keine Verknüpfungen statt, weil jeweils einer der Quell-Bestandteile leer ist.

### 5.4.5 Verschmelzen von Zustandsautomaten und Zuständen

Die Integration dynamischer Eigenschaften von Hyperslices ist von besonderer Bedeutung für die Erstellung von Produkten einer Produktlinie, insbesondere für die Generierung von Quellcode und von Testfällen. Zustandsdiagramme sind eine der Diagrammformen der UML für die Modellierung des Verhaltens von Objekten, wobei Zustandsautomaten und (Objekt-) Zustände die wesentlichen Modellinhalte darstellen. Die Semantik der Integration von Zustandsautomaten und Zuständen muss gegenüber der allgemeinen Semantik des Verschmelzens (siehe 5.4) um die folgenden Punkte erweitert werden.

Zur Illustration wird auf das Beispiel in Abb. 49 (S. 125) verwiesen. Dort wird eine Verschmelzung von drei Hyperslices gezeigt, deren Zustandsgraphen sich auf ein Objekt der Klasse `VideoOutput` beziehen. Das Ergebnis ist im Hypermodule productY enthalten.

**Vorbedingungen**

Zustandsautomaten beziehen sich in der UML auf Zustände innerhalb eines Kontextes, der von Instanzen der betreffenden Klassen gebildet wird. Eine Integration von Zustandsautomaten über Kontexte hinweg ist nicht ohne weiteres möglich, sie wird von der Integrationsmethode Verschmelzen nicht unterstützt. Zu verschmelzende Zustandsautomaten müssen deshalb alle einem identischen Kontext oder alle keinem Kontext zuordnet sein. Diese Bedingung ist erfüllt, wenn eine beliebige Integrationsmethode die Identität herstellt, beispielweise durch ein Verschmelzen der Kontexte.

Zustände sind Teil einer Hierarchie von Zuständen. Eine Integration von Zuständen über verschiedene Hierarchieebenen hinweg ist nicht ohne weiteres möglich. Deshalb müssen zu verschmelzende Zustände in einer identischen Hierarchieebene stehen. Diese Bedingung wird ebenfalls unter Berücksichtigung aller Integrationsmethoden eines Hypermodules geprüft.

Allen zu verschmelzenden Zuständen sind die gleichen (oder keine) entry-Aktionen und exit-Aktionen zugeordnet.

**Nachbedingungen**

Ein Zielzustandsautomat erhält zusätzlich die in den Quellzustandsautomaten definierten Zustände. Übereinstimmende Zustände gemäß der Operation Operation.correspondsTo() (siehe S. 98) werden miteinander verschmolzen.

Einem Zielzustand ist die gleiche entry-Aktion, exit-Aktion und Aktivität zugeordnet wie den zu verschmelzenden Zuständen, falls sie dort definiert ist. Er enthält zusätzlich die Menge der aufschiebbaren Ereignisse der Quell-Zustände, wobei die Menge keine redundanten Ereignisse aufnimmt.

Ein Zielzustand enthält die „äußeren“ eingehenden oder ausgehenden Transitionen der Quellzustände als eingehende oder ausgehende Transitionen. Er enthält zusätzlich die

inneren Transitionen der Quellzustände. Ein Zielzustand enthält einen flachen History-Zustand, wenn mindestens ein Quellzustand einen solchen enthält. Zu diesem History-Zustand hin- oder wegführende Transitionen in Quellzuständen führen auch zu einem Zielzustand hin beziehungsweise weg vom entsprechenden History-Zustand. Das Gleiche gilt für tiefe History-Zustände.

Ein Zielzustand weist einen Startzustand auf, wenn mindestens ein Quellzustand einen Startzustand enthält. Von Startzuständen ausgehende Transitionen gehen auch vom Startzustand des Zielzustands aus. Für Endzustände gilt die gleiche Semantik, hier werden hinführende Transitionen in gleicher Weise auf den Zielzustand übertragen.

Für die Verfeinerungen von Zuständen enthält die UML UND- und ODER-Verfeinerungen. Beide Verfeinerungen werden verschieden verschmolzen. Ein Zielzustand ist ODER-verfeinert und enthält zusätzlich die Zustände, die in ODER-verfeinerten Quellzuständen enthalten sind, wenn mindestens ein zu verschmelzender Zustand ODER-verfeinert, aber keiner UND-verfeinert ist. Übereinstimmende ODER-Verfeinerungen verschmelzen ebenfalls in der hier beschriebenen Weise miteinander.

Ein Zielzustand ist UND-verfeinert und enthält zusätzlich die Zustände, die in UND-verfeinerten Quellzuständen enthalten sind, wenn mindestens ein zu verschmelzender Zustand UND-verfeinert ist. Sind ein oder mehrere Quellzustände ODER-verfeinert, enthält der Zielzustand deren Verfeinerungs-Zustände in einer weiteren UND-Verfeinerung. Übereinstimmende Verfeinerungen verschmelzen ebenfalls in der hier beschriebenen Weise miteinander.

Neben den bereits behandelten Zuständen können in Zustandsgraphen noch sog. Pseudozustände vorkommen, die nur Notationshilfen darstellen, denen aber keine Aktivitäten zugeordnet werden. Pseudozustände von Quellzuständen sind in den jeweiligen Zielzuständen ebenfalls enthalten, verschmelzen jedoch nicht miteinander. Dazu gehören Entscheidungsknoten, Verbindungsstellen, Gabelungen, Vereinigungen und Sync-Zustände.

Zustände sind durch Referenzen mit anderen UML-Modellelementen verbunden; wenn zum Beispiel Instanzen von Klassen auf einen Quellzustand der Verschmelzung verweisen, werden Referenzen auf den entsprechenden Zielzustand übernommen.

**Ausnahmen**
Ausnahmen von der oben beschriebenen Semantik sind für den Fall redundanter und ungültiger Modellelemente vorzusehen. Redundante Transitionen werden entfernt. Zwei Transitionen sind redundant, wenn sie dieselben Zustände verbinden, durch das gleiche Ereignis ausgelöst werden und beim Auslösen die gleiche Aktion ausführen. Dies trifft bei inneren Transitionen zu; äußere Transitionen verbinden möglicherweise unterschiedliche Zustände. Überwachungsbedingungen der entfernten Transition werden durch logisches UND mit der übriggebliebenen Transition verknüpft. Führt die Verknüpfung zu einer widersprüchlichen Überwachungsbedingung, muss die Integration abgebrochen werden. Eine Prüfung der Widerspruchsfreiheit setzt eine formale Beschreibung der Überwachungsbedingungen voraus, wie etwa mittels OCL.

### 5.4.6 Verschmelzen von Signalen
Die Semantik der Verschmelzung von Signalen ist lediglich in Bezug auf Vererbungsbeziehungen und auf enthaltene Parameter gegenüber der allgemeinen Semantik des Verschmelzens von 5.4 zu erweitern.

**Nachbedingungen**
Ein Signal, das Ergebnis der Verschmelzung ist (Zielsignal), erbt zusätzlich von den übergeordneten Signalen der Quellsignale. Untergeordnete Signale, die bisher von Quellsignalen erben, erben nun vom Zielsignal. Redundante Vererbungsbeziehungen werden wieder entfernt.

Ein Zielsignal enthält zusätzlich die in den Quellsignalen enthaltenen Parameter. Redundante Parameter werden (auch aus Untersignalen) entfernt, wobei die gesamte Vererbungshierarchie des Zielsignals berücksichtigt werden muss. Beim Auslösen von Quellsignalen schaltende Transitionen schalten nun beim Auslösen des Zielsignals.

**Ausnahmen**
Da Vererbungsbeziehungen in der UML keine Zyklen aufweisen dürfen, führt ein solcher Fall beim Verschmelzen von Signalen ebenfalls zu einem Fehler.

### 5.4.7 Verschmelzen von Aktivitätsgraphen und –zuständen

Aktivitätsgraphen beschreiben den Ablauf ihres Kontexts, meist von Use-Cases, als Folge von Aktivitätszuständen mit Zuordnungen in Form von Verantwortlichkeitsbereichen (engl.: swim lanes). Aktivitätsgraphen sind in der UML eine Sonderform der Zustandsautomaten. Aus diesen Besonderheiten resultieren Erweiterungen der Semantik gegenüber 5.4.

**Vorbedingungen**
Das Verschmelzen von Aktivitätsgraphen über verschiedene Kontexte hinweg ist nicht ohne weiteres möglich. Für die Integrationsmethode Verschmelzen müssen die Graphen deshalb alle einem identischen Kontext oder alle keinem Kontext zugeordnet sein. Diese Bedingung wird unter Berücksichtigung aller Integrationsmethoden geprüft, so dass sie durch eine beliebige Integration erfüllt werden kann, die eine Identität herstellt, beispielweise durch das Verschmelzen der jeweiligen Kontexte.

**Nachbedingungen**
Ein Zielaktivitätsgraph enthält die verschmolzenen Abläufe der Quellaktivitätsgraphen. Dazu gehören Aktivitätszustande, Transitionen, Entscheidungsknoten, Gabelungen, Verbindungsstellen, Vereinigungen, Sync-Zustände und Objektflüsse. Die Reihenfolge der Abläufe ist dabei nicht definiert, wenn nicht der Entwickler durch die Integrationsmethode Ordnen des Hypermodules eine Reihenfolge festgelegt hat. Die einzelnen Abläufe der Quellaktivitätsgraphen werden über ihre Start- und Endzustände verknüpft. Der Zielaktivitätsgraph beginnt mit dem Startzustand des ersten Ablaufs und endet mit dem Endzustand des Letzten. Transitionen, die zum Endzustand eines vorhergehenden Ablaufs führen, werden mit dem Startaktivitätszustand des jeweils nächsten Ablaufs verbunden.

Ein Zielaktivitätsgraph enthält zusätzlich die Verantwortlichkeitsbereiche der Quellaktivitätsgraphen. Redundante Bereiche werden entfernt. Die Verschmelzung führt zu keiner Änderung der Zuordnung zum Kontext, deshalb bleiben die Aktivitätszustände den gleichen Verantwortlichkeitsbereichen zugeordnet.

Zu verschmelzende Aktivitätszustände werden in undefinierter Reihenfolge durch Transitionen verbunden, wenn nicht durch die Integrationsmethode Ordnen eine Reihenfolge festgelegt wurde. Zu einer Gruppe zu verschmelzender Aktivitätszustände hinführende Transitionen führen statt dessen zum ersten Zielaktivitätszustand der Reihenfolge. Von dieser Gruppe ausgehende Transitionen gehen nun vom letzten Zielakti-

vitätszustand der Reihenfolge aus. Objektflüsse der zu verschmelzenden Aktivitätszustände bleiben unverändert.

Verweist ein Aktivitätszustand zwecks genauerer Beschreibung auf einen zu verschmelzenden Aktivitätsgraphen, verweist diese Referenz anschließend auf den Zielaktivitätsgraphen.

**Ausnahmen**
Enthält ein zu verschmelzender Aktivitätsgraph keinen Start- oder Endzustand, wird sein Ablauf zwar in den Zielaktivitätsgraphen übernommen, aber nicht mit den anderen Abläufen verknüpft. Es sind weitere Integrationsmethoden wie Ordnen erforderlich, damit ein gültiges Modell erzeugt werden kann.

Enthält ein Zielaktivitätsgraph nach dem Verschmelzen gleiche Aktivitätszustände, so werden die überzähligen entfernt (außer wenn durch die Integrationsmethode Ersetzen - siehe Abschnitt 5.7 - etwas anderes festgelegt wird). Gleiche Aktivitätszustände weisen neben gleichem Namen auch gleichen Verantwortlichkeitsbereich, gleiche entry-Aktion oder Referenz auf einen identischen Aktivitätsgraphen sowie Gleichheit bezüglich dynamicArguments, dynamicMultiplicity und isDynamic auf. Für nähere Informationen zu diesen Eigenschaften von Aktivitätszuständen sei auf [UML 2001] verwiesen. Bei fehlender Gleichheit muss die Integration wegen Fehlers abgebrochen werden.

Entstehen beim Verschmelzen redundante Transitionen, können sie auch hier entfernt werden. Redundanz bedeutet hier, dass Transitionen dieselben (Aktivitäts-)Zustände verbinden sowie beim Auslösen die gleichen Aktionen ausführen. Überwachungsbedingungen der Transitionen werden in gleicher Weise wie die von Transitionen von Zustandsautomaten geprüft und verknüpft (siehe Abschnitt 5.4.5).

## 5.5 Integrationsmethode Ordnen

Beim Verschmelzen von Elementen werden gleichartige Elemente bestimmter Typen aneinandergereiht. Soll dabei eine bestimmte Reihenfolge eingehalten werden, ist diese im Hypermodule zu definieren. Eine solche Reihenfolge ist für Operationen, für die Bestandteile Durchführung und Ausnahmen von Use-Case-Beschreibungen, für Aktivitätsgraphen sowie für Aktivitätszustände von Bedeutung.

Ein Beispiel für das Ordnen beim Verschmelzen von Operationen ist in Abb. 40 (S. 103) erwähnt, wo die Operation `stop()` in jeder der zu verschmelzenden 3 Klassen enthalten ist. Die Integrationsmethode Ordnen gibt dem Entwickler die Möglichkeit, mit before und after in einer ähnlichen Weise wie bei Hyper/J die Reihenfolge der Bestandteile in der entstehenden Operation zu steuern.

### 5.5.1 Syntax und Regeln

Die Integrationsmethode Ordnen gibt an, in welcher Reihenfolge Modellelemente verschmolzen werden. Dazu setzt sie die betreffenden Hyperslices in eine Reihenfolge, wie in der graphischen Syntaxbeschreibung in Abb. 42 durch die mehrfache, geordnete Assoziation `unit` gezeigt wird.

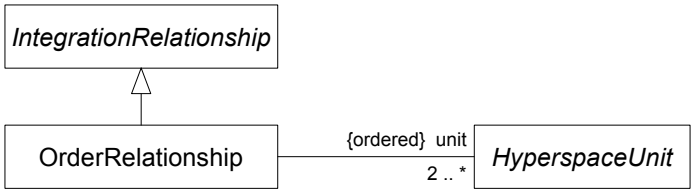



*Abb. 42: Integrationsmethode Ordnen im Metamodell von Hyper/UML*

Die Integrationsmethode Ordnen liefert genauere Informationen zur Ausführung der Integrationsmethode Verschmelzen. Widersprüche bezüglich der Reihenfolge von Elementen würden ein Hypermodule ungültig werden lassen. Die folgenden Regeln beschreiben die Integrationsmethode genauer. Eine Definition der Regeln mit OCL ist in [Böllert 2002] enthalten.

- Die zu ordnenden Elemente sind nicht mehrfach in der Integrationsmethode enthalten. Anderenfalls wäre die Reihenfolge der Elemente nicht eindeutig bestimmbar.
- Die zu ordnenden Elemente sind den Belangen des Hypermodules zugeordnet. Damit wird verhindert, dass unbeteiligte Elemente in die Integration einbezogen werden.
- Die zu ordnenden Elemente sind vom selben Typ.
- Bei den zu ordnenden Elementen handelt es sich um Operationen, Use-Cases, Aktivitätsgraphen oder Aktivitätszustände. Für andere Typen von Elementen hätte die Integrationsmethode Ordnen keine Bedeutung.

### 5.5.2 Semantik

Die Semantik der verschiedenen Typen von Elementen unterscheidet sich in Bezug auf die Bestandteile und Eigenschaften, für die eine Reihenfolge relevant ist. Sie wirkt sich nur auf Nachbedingungen aus. Vorbedingungen und Ausnahmen wurden im Rahmen der Integrationsmethode Verschmelzen dargestellt, hier sind diesbezüglich keine Präzisierungen erforderlich.

Beim Verschmelzen von Operationen führt die Zieloperation die Implementierungen der Quelloperationen in der angegebenen Reihenfolge aus.

Ein Ziel-Use-Case enthält die Durchführungen aus den Quell-Use-Cases in der angegebenen Reihenfolge. Diese Reihenfolge wird ebenfalls auf die Verschmelzung damit verbundener Aktivitätsgraphen angewendet.

Die Anwendung der Integrationsmethode Ordnen auf zu verschmelzende Aktivitätsgraphen führt zur angegebenen Reihenfolge der Abläufe im Zielaktivitätsgraphen. Verschmelzen zu ordnende Aktivitätszustände, werden sie in der angegebenen Reihenfolge verknüpft.

## 5.6 Integrationsmethode Summieren

Bei der Komposition von Operationen mit der zuerst vorgestellten Integrationsmethode Verschmelzen werden Rückgabewerte eines der zu verschmelzenden Teile als Eingabewert des nächsten verwendet. Sollen anstelle des letzten Rückgabewertes die Rückgabewerte aller verschmolzenen Quelloperationen zurückgegeben werden, ist deren Vereinigung zu definieren. Bei der Integrationsmethode Summieren gibt zu diesem

Zweck eine Summenfunktion an, wie ein solcher gemeinsamer Rückgabewert gebildet wird. Mit dieser Summenfunktion kann der Entwickler die Komposition steuern. Beispiele für solche Summenfunktionen sind die Addition oder Multiplikation zweier numerischer Rückgabewerte, die Verkettung zweier Listen oder Zeichenketten, die Vereinigung zweier Mengen oder Arrays, sowie das Zusammenfügen mehrerer Parameter auch unterschiedlicher Typen zu einem Array.

Abb. 43 zeigt als Beispiel die zwei zu verschmelzenden Operationen `getSetupMenu()`, die als Rückgabeparameter jeweils ein Array mit Menüeinträgen liefern. Die Menüeinträge gehören zu den Setup-Funktionen der gemeinsamen Features sowie des variablen Features Persönliche Profile (siehe Featuremodell in Abb. 23, S. 70), die jeweils von der Klasse `Customization` verwaltet werden. Die Rückgabeparameter beider Operationen sind zu einem Array zu vereinigen, so dass die verschmolzene Operation die zusammengefassten Menüeinträge liefert. Die in der Hypermodule-Spezifikation angegebene Summenfunktion `SummaryFunctions::listAddAll()` fasst die Rückgabewerte der beiden Operationen zu einem Array zusammen, das alle Elemente enthält.

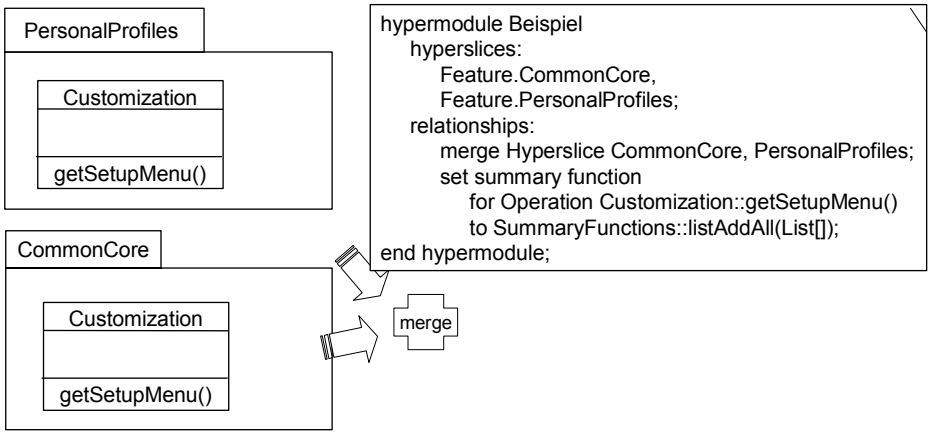



*Abb. 43: Verschmelzen mit Summieren von zwei Operationen am Beispiel, rechts die Hypermodule-Spezifikation*

### 5.6.1 Syntax und Regeln

Die graphische Syntax der Integrationsmethode Summieren (Abb. 44) enthält als summarizedUnit die Operation als Ergebnis der Verschmelzung und als summaryFunction die Summenfunktion, die den gemeinsamen Rückgabewert bildet.

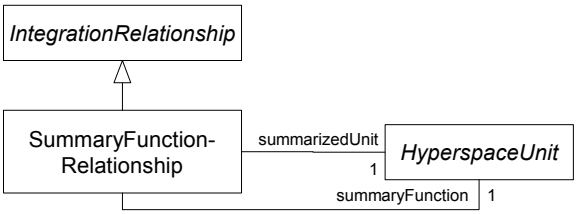



*Abb. 44: Integrationsmethode Summieren im Metamodell von Hyper/UML*

Die folgenden Regeln beschreiben die Integrationsmethode genauer:

- Das zu summierende Element und die Summenfunktion dürfen zur Vermeidung einer endlose Rekursion nicht identisch sein.
- Das zu summierende Element und die Summenfunktion sind Operationen. Die Integrationsmethode ist nur für Elemente dieses Typs vorgesehen.
- Das zu summierende Element und die Summenfunktion sind den im Hypermodule angegebenen Belangen zugeordnet. Anderenfalls würden unbeteiligte Elemente in die Integration einbezogen.

Für die Umsetzung der Regeln wird die folgende Operation benötigt:

SummaryFunctionRelationship.units()
: liefert das zu summierende Element und die Summenfunktion zurück

### 5.6.2 Semantik

Die Integrationsmethode Summieren steuert mittels einer Summenfunktion, wie Rückgabewerte von verschmolzenen Operationen zu einem gemeinsamen Wert zusammengefasst werden.

**Vorbedingungen**

Bei einer zu summierenden Operation und Summenfunktion handelt es sich um konkrete Operationen mit einem Rückgabeparameter von identischem Typ. Diese Identität kann unter Einbeziehung aller Integrationsmethoden eines Hypermodules hergestellt werden, beispielweise durch das Verschmelzen von Typen.

Die Summenfunktion ist eine Klassenfunktion, d.h. sie kann ohne Vorhandensein einer Instanz der Klasse benutzt werden.

Die Summenfunktion empfängt einen Parameter vom gleichen Typ wie die Rückgabeparameter, jedoch mit der Vielfachheit „*“. Damit wird ermöglicht, eine Liste mit Rückgabewerten der verschmolzenen Operationen zu verarbeiten.

**Nachbedingungen**

Während der Ausführung der Operation wurden die Ergebnisse der verschmolzenen Implementierungen (vgl. Abschnitt 5.4.3) in einer Liste gesammelt. Die Summenfunktion bildet daraus einen gemeinsamen Rückgabewert, den die zu summierende Funktion zurückgibt. Beispiele für solche Summenfunktionen sind die Vereinigung von Listen oder Arrays, die Addition oder Multiplikation von Zahlen, sowie die Verkettung von Zeichenketten.

## 5.7 Integrationsmethode Ersetzen

Die Integrationsmethode Ersetzen dient im Gegensatz zur Methode Verschmelzen zum vollständigen Ersatz eines Elements durch an anderes gleichen Typs. Diese Integrationsmethode wird beispielweise benötigt, wenn der in Verschmelzen enthaltene Automatismus zur Vereinigung gleichartiger Elemente deaktiviert werden soll oder wenn bei sog. negativen Anforderungen (siehe 6.3.1) ein Teil der Funktionalität deaktiviert werden soll. Dabei müssen enthaltene Elemente berücksichtigt und Referenzen entsprechend korrigiert werden. Ersetzende Elemente müssen zumindest die gleiche Schnittstelle aufweisen wie das ersetzte Element, damit alle anderen Modellelemente gültig bleiben, die diese Schnittstelle benutzen. Es wäre möglich, fehlende Teile einer Schnittstelle durch Integrationsmethoden zu ergänzen oder überzählige zu entfernen.

Im Vergleich zum Verschmelzen ist das Ersetzen eine Integrationsmethode mit stärkeren Wechselwirkungen zwischen den beteiligten Elementen. Aus Sicht des Entwicklers sind dabei umfangreiche Abhängigkeiten zwischen beteiligten Modellelementen und Hyperslices zu beachten. Durch Maßnahmen im Zuge der Integration erhöht sich die Komplexität der Hypermodules. Die Unabhängigkeit der Hypermodules, eine Forderung des Hyperspace-Ansatzes, wird stark eingeschränkt, was weitreichende Auswirkungen auf ihre semantische Gültigkeit hat. Wegen dieser Konsequenzen wurde entschieden, diese Integrationsmethode zunächst nur für Operationen und Aktivitätszustände zu definieren. Alle anderen Modellelemente sind mit anderen Integrationsmethoden zu behandeln, vorrangig mittels Verschmelzen.

Bezogen auf Operationen bedeutet diese Entscheidung, dass Operationen von Klassen die kleinsten in Hyper/UML (und Hyper/J) bearbeitbaren Einheiten einer Software sind: Operationen können nur im Ganzen ersetzt werden. Eine Manipulation kleinerer Einheiten wie einzelner Anweisungen des Quellcodes wäre nicht in einem UML-Modell darstellbar, die Auswirkungen einer Manipulation wären außerdem für den Entwickler nur schwer abschätzbar.

Bei Implementierung der in Abschnitt 10.2.2 vorgestellten Produktlinie „Siedler von Catan“ wurde festgestellt, dass einerseits die Integrationsmethode für die Elementtypen Operationen und Aktivitätszustände von einem Entwickler beherrscht werden kann und dass der dabei erforderliche Aufwand vertretbar bleibt, und dass andererseits damit bereits ausreichend praxiswirksame und mächtige Integrationsmethoden zur Verfügung stehen. Die Abwägung zwischen dem Aufwand zur Sicherstellung gültiger Hyper/UML-Modelle und dem praktischen Nutzen sollte jedoch durchaus Gegenstand zukünftiger Arbeiten sein. Ein hoher Grad der bei der Modellierung angewendeten Formalisierung, beispielweise durch Nutzung der OCL in Modellen wird hier zusätzliche Werkzeugunterstützung ermöglichen, die wiederum die Beherrschung mächtigerer Integrationsmechanismen erlauben wird.

### 5.7.1 Syntax und Regeln

Abb. 45 zeigt in der graphischen Syntax, dass die Integrationsmethode auf zwei Elemente verweist, von denen das eine (overriddenUnit) durch das andere (overridingUnit) ersetzt wird. Nach der Ausführung der Operation existiert das zu ersetzende Element im Hypermodule nicht mehr. Damit wird die Regel aus dem UML-Metamodell eingehalten, dass ein Element kein übereinstimmendes Element enthalten darf.

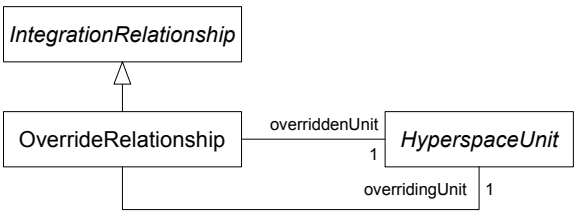



*Abb. 45: Integrationsmethode Ersetzen im Metamodell von Hyper/UML*

Die im folgenden genannten Regeln werden durch die in 5.7.2 und 5.7.3 folgenden Festlegungen in den Vorbedingungen für die beiden Typen von Elementen weiter präzisiert. Für die dazu gehörenden OCL-Ausdrücke sei wieder auf Anhang A von [Böllert 2002] verwiesen.

- Das zu ersetzende und das ersetzende Element sind nicht identisch.
- Das zu ersetzende und das ersetzende Element sind vom selben Typ.
- Das zu ersetzende und das ersetzende Element sind vom Typ Operation oder Aktivitätszustand.

Für die Umsetzung der Regeln wird die folgende Operation benötigt:

SummaryFunctionRelationship.units()
liefert das zu ersetzende und das ersetzende Element zurück

### 5.7.2 Semantik für Operationen

Operationen einer Klasse werden nur im Ganzen ersetzt. Bei Erfordernis des Ersetzens von Teilen einer Operation muss diese deshalb durch Refactoring in mehrere Operationen zerlegt werden. Damit die Auswirkungen für weitere Teile eines Objektmodells überschaubar bleiben, werden außerdem enge Forderungen an die Übereinstimmung von ersetzter und ersetzender Operation gestellt.

**Vorbedingungen**

Die ersetzende Operation und die zu ersetzende Operation müssen in einer identischen Klasse definiert sein. Diese Bedingung wird unter Berücksichtigung aller Integrationsmethoden eines Hypermodules überprüft. Sie kann beispielweise dadurch erfüllt werden, dass eine andere Integrationsmethode Klassen verschmilzt.

Die ersetzende Operation und die zu ersetzende Operation haben die gleiche Signatur, d.h. sie haben den gleichen Namen und die gleiche Parameterliste für Eingabe- und Rückgabeparameter. Falls für Parameter ein Default-Wert definiert ist, muss dieser ebenfalls übereinstimmen. Diese Bedingung stellt die Identität der Schnittstelle für den Aufrufer der Implementierung sicher.

Die ersetzende Operation und die zu ersetzende Operation haben gleiche Eigenschaften. Dazu gehören Klassen- oder Instanzoperationen, konkret oder abstrakt, abgeleitet oder nicht abgeleitet, sowie concurrency.

**Nachbedingungen**

Die Klasse der ersetzten Operation enthält statt dessen die ersetzende Operation einschließlich deren Implementierung. Aktionen beziehen sich jetzt auf die ersetzende Operation. Ereignisse, die von der ersetzten Operation ausgelöst würden, werden nun von der ersetzenden Operation ausgelöst.

### 5.7.3 Semantik für Aktivitätszustände

**Vorbedingungen**

Der ersetzende Aktivitätszustand und der zu ersetzende Aktivitätszustand gehören zu einem identischen Aktivitätsgraphen und sind dem gleichen (oder keinem) Verantwortungsbereich zugeordnet. Diese Bedingung wird unter Berücksichtigung aller Integrationsmethoden eines Hypermodules überprüft. Sie kann beispielweise dadurch erfüllt werden, dass eine andere Integrationsmethode Identität herstellt.

**Nachbedingungen**

Der Aktivitätsgraph enthält statt des zu ersetzenden Aktivitätszustands nun den ersetzenden Aktivitätszustand mit allen seinen Eigenschaften. Dazu gehören entry-Aktion,

Einschränkungen und Eigenschaftswerte sowie die Eigenschaften dynamicArguments, dynamicMultiplicitiy und isDynamic (siehe [UML 2001]).

Die Transitionen vom und zum Aktivitätszustand bleiben unverändert. Das Gleiche gilt für ein- und ausgehende Objektflüsse.

**Ausnahmen**
Beim Ersetzen von Aktivitätszuständen können redundante Transitionen entstehen; diese werden entfernt, ihre Überwachungsbedingungen werden verknüpft. Zur Redundanz von Transitionen und zur Verknüpfung der Überwachungsbedingungen gelten die gleichen Ausnahmen wie beim Verschmelzen von Transitionen von Zustandsautomaten (siehe 5.4.5).

## 5.8 Werkzeugunterstützung

Erfahrungen mit Modell-basierten Entwurfsmethoden haben gezeigt, dass sie nur mit Werkzeugunterstützung erfolgreich in der Praxis genutzt werden können. Wegen der Komplexität heutiger Softwaresysteme und der zusätzlichen Komplexität durch die Variabilität von Produktlinien ist die Werkzeugunterstützung für eine Modellierung mit Hyper/UML von besonderer Bedeutung. Die Entwicklung von Werkzeugunterstützung für graphische und textuelle Darstellungen stellt einen wichtigen Beitrag zur Verständlichkeit und zur Beherrschung der Komplexität dar.

Für die Modellierung mit UML sind bereits eine Reihe von Werkzeugen mit recht guter Unterstützung für methodisches Vorgehen am Markt [Cetus]. Da - bezogen auf die Funktionalität eines Werkzeugs - die normale Modellierung mit UML einen großen Teil der Modellierung mit Hyper/UML ausmacht, müssten die verfügbaren Werkzeuge nur geringfügig erweitert werden. Außerdem wäre die Entwicklung und Markteinführung eines neuen CASE-Werkzeugs mit so hohem Aufwand verbunden, dass dies nur von etablierten Unternehmen geleistet werden kann. Deshalb wird zugunsten einer Erweiterung bestehender Werkzeuge auf eine Eigenentwicklung verzichtet.

Im Folgenden werden, aufbauend auf den Ausführungen in [Böllert 2002] Anforderungen an Erweiterungen von Werkzeugen spezifiziert. Solche Erweiterungen betreffen zunächst die Erweiterung des den Modellen zugrundeliegenden Metamodells. Für die Speicherung der Modelle wird meist ein Repository verwendet, das die Anforderungen des Metamodells erfüllen muss. Eine Nutzerschnittstelle mit zusätzlichen Dialogen zur Verwaltung der Hyper/UML-Elemente, Programmteile für die Prüfung der Modelle und Werkzeuge für ihre zusätzlichen graphischen Darstellungen müssen ebenfalls zu den Erweiterungen gehören. Darüber hinaus ist die Bereitstellung zusätzlicher Schnittstellen für Programmiersprach-Implementierungen wie Hyper/J erforderlich. Für folgende zusätzliche Aktivitäten wird Unterstützung benötigt:

- Verwalten von Hyperspaces: Hyperspaces sind zu definieren, Elemente sind Punkten im Hyperspace zuzuordnen.
- Modellieren von Hyperslices: Hyperslices könnten als UML-Pakete mit dem Stereotyp <<hyperslice>> modelliert werden. Gegenüber einer Modellierung mit der UML sind dann kaum Erweiterungen notwendig. Für eine komfortable Unterstützung des Entwicklers bei Änderungen und Weiterentwicklung von Hyperslices sind Darstellungen der semantischen Abhängigkeiten sinnvoll, die bei einer Integration entstehen. Von großer Bedeutung ist dabei eine Vorschaufunktion, die dem Entwickler die Verschmelzung gerade bearbeiteter Hyperslices

und ihrer Elemente zeigt, damit er die Konsequenzen von Veränderungen bewerten und prüfen kann.

- Verwalten von Hypermodules: Zur Spezifikation der Hypermodules gehören die Angabe der relevanten Belange und der Integrationsmethoden. Für die Belange sind Beziehungen zu Features des Featuremodells auszudrücken. Die Erfahrungen in [Böllert 2002] haben gezeigt, dass für die Integrationsbeschreibung eine textuelle Darstellung entsprechend der angegebenen Syntax sinnvoller ist als eine graphische Darstellung.
- Durchführung der Integration: Die durch ein Hypermodule spezifizierte Integration muss unter Zugriff auf die Modellelemente der Hyperslices ausgeführt werden. Dazu sind die genannten Regeln, Operationen sowie Vor- und Nachbedingungen zu implementieren. Die graphische Darstellung des Ergebnisses erfordert Methoden zur sinnvollen Anordnung der Diagramme.

Am Markt existieren einige wenige Werkzeuge, die eine Erweiterung ohne Eingriff in die Werkzeugentwicklung zulassen. MetaEdit+ [Metacase] ist ein sogenanntes Meta-CASE-Tool, das eine Definition und Verwaltung beliebiger Elemente zulässt. Das public-domain-Produkt ArgoUML [Argo] und sein professionell hergestellter Nachfolger Poseidon [Poseidon 2002] bieten durch eine Plug-in-Schnittstelle flexible Möglichkeiten der Erweiterung und des Zugriffs auf Repository-Daten. Bei einer Integration der Erweiterungen in Werkzeug und Repository ist auf die Effizienz bezüglich des Ressourcenbedarfs besonderes Augenmerk zu legen.

Bei der Weiterentwicklung der UML zur Version 2.0 soll die Erweiterung mittel sogenannter UML-Profile und die Integration verschiedener Modelle besser unterstützt werden. Es ist zu erwarten, dass dafür bereitgestellte CASE-Werkzeuge auch die Darstellung und Zerlegung von Objektmodellen unterstützen.

# 6 Feature-getriebene Komposition mit HyperFeatureRSEB

*In diesem Kapitel wird die Methode HyperFeatuRSEB vorgestellt, die aus einer Verknüpfung der Produktlinien-Methode FeatuRSEB und dem Hyperspace-Ansatz entstanden ist. Die grundlegenden Entwicklungsprozesse werden untersucht, die auf der Modellierung von Variabilität mit Featuremodellen und auf modellbasierter Zerlegung und Komposition mit Hyper/UML beruhen.*

Die Methode HyperFeatureRSEB entstand, wie bereits im Überblick über die Methodik ALEXANDRIA (S. 51ff) dargestellt, durch Kombination die Methode FeatuRSEB (siehe Abschnitt 2.1.5) mit dem Hyperspace-Ansatz (siehe Abschnitt 2.3.4). Der Hyperspace-Ansatz erlaubt eine Zerlegung in feingranulare, überschneidungsfreie Komponenten. Mittels Hyper/UML können diese vorgefertigten Komponenten modellbasiert entwickelt werden. Die Zerlegung erfolgt entsprechend den Features einer Produktlinie. Damit kann erreicht werden, dass der Entwurf eines Produkts als featuregesteuerte Komposition von Modellen erfolgt, der ebenso wie die Implementierung durch Komposition teilweise automatisierbar ist. Da die gewünschte Flexibilität der Produktlinie durch Komposition erreicht wird, und entstehen schlanke Produkte, die keine unnötige Funktionalität enthalten, die erhöhten Ressourcenbedarf erfordert.

Wie in Abschnitt 3.3 bei der Vorstellung der Entwicklungsprozesse gezeigt wurde, gehören zur Methodik ALEXANDRIA mehrere Prozesse. Zerlegung und Komposition werden von den Prozessen Produktlinien-Engineering und Produkt-Engineering geleistet, die als Teile der Methode HyperFeatureRSEB im Rahmen der hier vorgestellten Arbeiten entwickelt wurden. Die Methode wurde gegenüber der Darstellung in [Böllert 2002] weiterentwickelt, damit die Handhabung von unvollständig gebundenen Variabilitäten ermöglicht wird, damit Präzisierungen der Featuremodelle wie in Kapitel 4 vorgestellt, unterstützt und Aktivitäten der evolutionären Weiterentwicklung eingebunden werden.

## 6.1 Produktlinien-Engineering

Nachdem bei der Vorstellung der Entwicklungsprozesse in Abschnitt 3.3.1 bereits ein Überblick über die Entwicklung und Pflege von Produktlinien gegeben wurde, stellt dieser Abschnitt die dazu gehörenden Aktivitäten und Prozesse im Zusammenwirken mit Modellen und Beschreibungsmitteln genauer vor.

Zum Prozess des Produktlinien-Engineering gehören drei Gruppen von Aktivitäten, (siehe Abb. 17 auf Seite 55) Analyse, Modellierung und Implementierung der gemeinsamen Teile einer Produktlinie. Der Prozess ist auf die Erweiterbarkeit und Wartbarkeit der Produktlinie ausgerichtet, damit Änderungen mit geringem Aufwand umgesetzt werden können und eine lange Lebensdauer der Produktlinie erreicht werden kann. Dadurch werden sowohl die Amortisation der Investitionen ermöglicht als auch die Anwendungsprozesse der Domäne auf lange Zeit unterstützt. Der Prozess trägt iterativ-inkrementellen Charakter: Bei jeder Abfolge der Aktivitäten werden weitere Komponenten bereitgestellt, Änderungen an Komponenten oder Architektur vorgenommen oder Erweiterungen umgesetzt.

### 6.1.1 Domänenanalyse

Die Aktivitäten der Domänenanalyse wurden in den Abschnitten 4.1 und 4.3 in Bezug auf die Modellierung mit Features detailliert beschrieben. Zur Vermeidung von Dopp-

lungen wird hier nur noch eine zusammenfassende Einordnung in den gesamten Entwicklungsprozess gegeben. Dabei werden der Bezug zur iterativen Zerlegung in Hyperslices sowie die Verknüpfungen mit den Aktivitäten der Hyperspace-Engineering- und Hyperspace-Engineering-Prozesse ergänzt.

Die Aktivitäten bei der Domänenanalyse folgen der Darstellung in Abb. 46. Sie werden als inkrementelle Folge solange wiederholt, bis das Anforderungsmodell vollständig vorliegt. Zunächst werden Anforderungen an die Produktlinie erfasst, beschrieben und strukturiert. Dabei kommen Techniken sowohl der konventionellen Anforderungsanalyse als auch der Domänenanalyse zum Einsatz. Es wird ein Anforderungsmodell erstellt, dass entsprechend den üblichen objektorientierten Methoden aus Use-Case-Modell, Klassenmodell und Schnittstellenmodell bestehen kann [Hitz Kappel 2003]. Gleichzeitig wird eine Unterscheidung nach gemeinsamen und variablen Anforderungen im Featuremodell abgelegt. Dazu werden Scoping-Entscheidungen gefällt, die sowohl die weitere Verfeinerung der Anforderungen als auch die weitere Entwicklungsrichtung der Produktlinie steuern. Die Entscheidungen werden vor allem unter wirtschaftlichen Gesichtspunkten getroffen [Riebisch et al. 2001a].

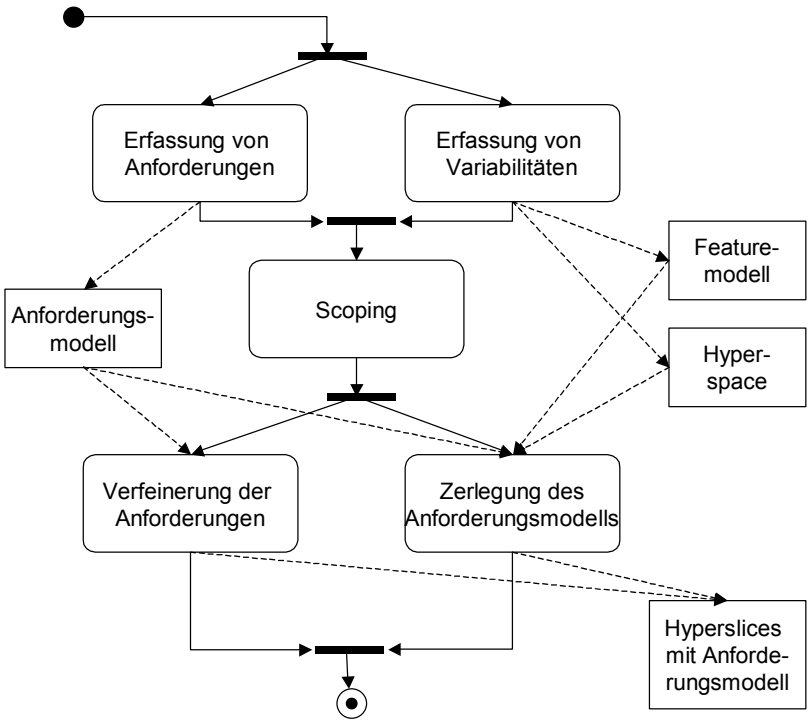



*Abb. 46: Aktivitäten einer Iteration der Domänenanalyse als UML-Aktivitätsdiagramm*

Die vorgesehene Variabilität der Produktlinie wird im Featuremodell mit Bezug auf die Anforderungen dargestellt. Parallel dazu wird ein Hyperspace definiert, der die Zerlegung von Anforderungen, Entwurf und Implementierung steuert. Für eine spätere featuregesteuerte Komposition von Produkten werden überschneidungsfreie Komponenten des Anforderungsmodells entsprechend den Features benötigt. Deshalb wird im Hyperspace eine Dimension für Features eingerichtet, in der alle variablen Features die Zerlegungskriterien liefern.

Die weitere Analyse und Verfeinerung der Anforderungen erfolgt gleichzeitig mit schrittweiser Zerlegung des Anforderungsmodells. Bei dieser Zerlegung werden Zuord-

nungen zwischen Anforderungen und Features dadurch vorgenommen, dass je Feature ein Hyperslice definiert wird, in dem alle dazugehörigen Anforderungen modelliert werden. Alle Anforderungen der gemeinsamen Features werden in *einem* weiteren Hyperslice gekapselt. In gleicher Weise wie das Anforderungsmodell werden auch Modelle für die Ableitung von Tests in Hyperslices zerlegt und für die Komposition von Tests bereitgestellt. Die Modelle werden statt mit der Modellierungssprache UML mit der in Kapitel 5 vorgestellten Hyper/UML ausgedrückt. Die Aktivitäten dazu werden in den Abschnitten 4.1 und 4.3 detailliert beschrieben.

### 6.1.2 Domänenmodellierung

Ziel der Domänenmodellierung ist die Entwicklung eines Objektmodells, das die funktionalen und nicht-funktionalen Anforderungen der erfüllt, die bei der Domänenanalyse ermittelt wurden. Während das Anforderungsmodell beschreibt, *was* für Eigenschaften die Produktlinie aufweisen soll, beschreiben Objektmodell und nachfolgende Implementierung *wie* diese Eigenschaften erreicht werden. Das Objektmodell sowie die nachfolgende Implementierung sollen eine Zerlegung aufweisen, die eine Komposition nach Features zulässt. Die Zerlegung muss deshalb mit der des Anforderungsmodells übereinstimmen; entsprechend dem Hyperspace-Ansatz entstehen dabei Hyperslices als überschneidungsfreie Komponenten.

Die Domänenmodellierung lässt sich in zwei Teile gliedern, das Hyperspace-Engineering und das Hyperslice-Engineering. Abb. 47 zeigt das Zusammenwirken beider Teile. Durch Vergleich mit Abb. 5 (S. 20) wird deutlich, wie die Ansätze FeatuRSEB und Hyperspace hier verknüpft wurden.

Das Hyperspace-Engineering ist für die Entwicklung der Architektur des Objektmodells in Abstimmung mit Featuremodell und Anforderungsmodell zuständig, während das Hyperslice-Engineering die aufgetrennten Teile des Objektmodells entwickelt und verwaltet.

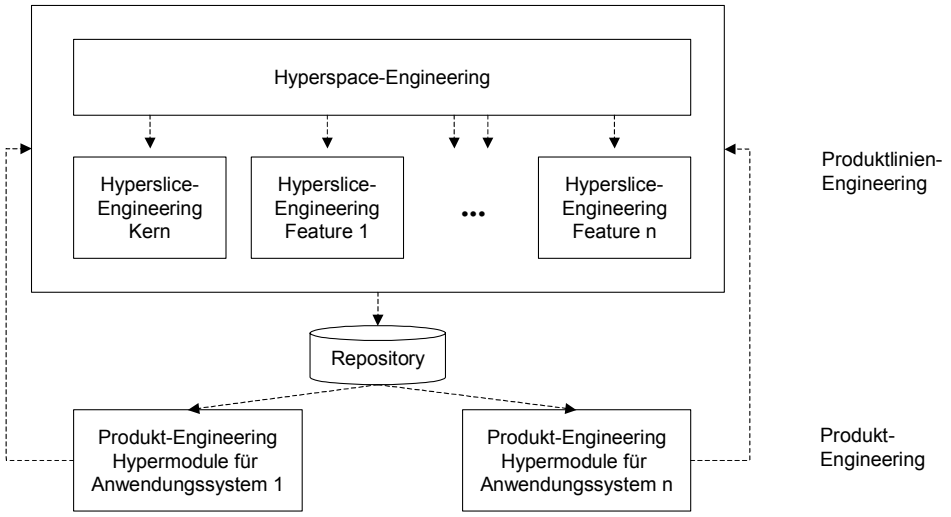



*Abb. 47: Teilprozesse der Domänenmodellierung im Zusammenspiel mit dem Produkt-Engineering*

#### Entwicklung einer Referenzarchitektur

Das Ergebnis der Modellierung der Produktlinie wird als Referenzarchitektur bezeichnet. Sie beschreibt die Architektur der Produktlinie - als Zusammenwirken von Kompo-

nenten, ihren Schnittstellen und den Abhängigkeiten zwischen ihnen - mit Darstellung der Variabilität. Es weist dazu eine Zerlegung auf, die den variablen Features des Featuremodells entspricht. Für die Ableitung von Produkten ist zusätzlich zu dieser Zerlegung eine Unterscheidung nach benötigtem Bindungszeitpunkt erforderlich. Werden mehrere Bindungszeitpunkte benötigt, existieren für einzelne Features mehrere verschiedene Komponenten mit unterschiedlichen Objektmodellen.

Im Hyperspace-Engineering wird zunächst eine Architektur entwickelt; die vorher modellierten Anforderungen werden wie üblicher unter Anwendung objektorientierter Entwurfsmethoden entwickelt. Als gut geeignet hat sich die Methode der Architekturentwicklung von [Bosch 2000] in Kombination mit den Mitteln von [Shaw et al. 1996] erwiesen. Bei der Architekturentwicklung werden zuerst funktionale Anforderungen mittels grundlegender Abstraktionen umgesetzt. Dann wird eine Dekomposition durchgeführt, und es werden Komponenten und Schnittstellen definiert. Bei der Schaffung unabhängiger Komponenten bietet die Anwendung von Architekturstilen wie dem der Plug-in-Komponenten-Schnittstellen von Eclipse [Eclipse 2003] besonders gute Möglichkeiten. Ein solcher Architekturstil ist deshalb mit der genannten Methode gut kombinierbar.

Bei der Dekomposition muss in Ergänzung zur genannten Methode die Zuordnung zu Features berücksichtigt werden. Anschließend wird die entstandene Architektur daraufhin überprüft, ob sie die funktionalen und nichtfunktionalen Anforderungen erfüllt. Insbesondere innere Qualitätsmerkmale wie Wartbarkeit und Portabilität sind für eine Produktlinie von Bedeutung, denn diese beeinflussen die spätere Evolution stark. Die Überprüfung wird in der Praxis meist unter Hinzunahme der maximal möglichen Menge von Features durchgeführt. Zur Verbesserung der Architektur bezüglich ungenügend erfüllter Qualitätsmerkmale werden im dritten Schritt entsprechende Veränderungen an der Architektur vorgenommen, die als Transformation bezeichnet werden. Zu den Mitteln zur Beeinflussung der Qualitätsmerkmale gehören

- die Aufteilung von Qualitätsanforderungen auf einzelne Bestandteile der Architektur,
- die Umsetzung nichtfunktionaler in funktionale Anforderungen [Bosch 2000],
- die Anwendung von Design Patterns für Objektmodell und Architektur wie in [Buschmann et al. 1996] und [Gamma et al. 1996] und
- die Einführung von weiteren Architekturstilen wie Filter, Schichten, Blackboard-Kommunikation, kommunizierenden Objekten und impliziten Aufruftechniken [Shaw et al. 1996].

Zerlegungen erfolgen wiederum mit Berücksichtigung der Zuordnung zu Features. Darüber hinaus wird bei der Entwicklung vermerkt, welcher Teil der Architektur welche Anforderungen realisiert. Die dabei getroffenen Entscheidungen werden als Traceability-Links (siehe 4.2) zwischen Features und Objektmodell-Elementen wie Klasse, Operation, Zustand und Zustandsübergang vermerkt. Im iterativen Prozess der Architekturentwicklung werden schrittweise Zerlegungen zur Trennung der Objektmodell-Elemente vorgenommen, die zu verschiedenen Features gehören. Dazu werden Hyperslices als Komponenten ins Objektmodell eingeführt, die getrennte Elemente aufnehmen. Dabei übernimmt das Hyperspace-Engineering die Koordination der Zerlegung. Die Koordination bezieht sich vor allem auf Definition und Aufteilung von Klassen als zentralen Elementen eines Objektmodells. Es entsteht eine Zerlegung entsprechend der des Anforderungsmodells, weil jeder Hyperslice des Objektmodells durch Traceability-

Links einem Hyperslice des Anforderungsmodells zugeordnet wird. Die weitere Entwicklung und Verfeinerung des Objektmodells wird als Hyperslice-Engineering durchgeführt.

**Koordination mittels Hyperspace-Engineering und Integrationsstrategie**
Der Entwicklungsprozess des Hyperspace-Engineering steuert die Zerlegung anhand der Dimensionen, die außerdem im Featuremodell abgebildet werden. Das Hyperspace-Engineering berücksichtigt dabei die spätere Komposition von Produkten. Da einzelne Hyperslices noch kein ausführbares System ergeben, wird später im Zuge des Produkt-Engineering eine Integration durchgeführt. Zur Vermeidung manueller Tätigkeiten und zur Automatisierung der Serienfertigung soll hier ein Generator zum Einsatz kommen, der die Komposition von Hyperslices vornimmt und dabei den nach Kundenwunsch ausgewählten Features folgt. Bei diesem Generator handelt es sich um das Hyper/UML- beziehungsweise das Hyper/J-Werkzeug. Dazu muss eine Spezifikation des Hypermodules erzeugt werden, die beschreibt, welche Features und Hyperslices zu integrieren sind. Mittels dieser Spezifikation erzeugt das Hyper/UML-Werkzeug ein Objektmodell. In ähnlicher Weise wird auch vom Hyper/J-Werkzeug die Implementierung eines Produkts durch Komposition erzeugt.

Die Elemente der zu integrierenden Hyperslices sind meist nicht orthogonal zueinander. So sind aufgrund der Forderung nach deklarativer Vollständigkeit zahlreiche Elemente auch in solchen Hyperslices als abstrakte oder leere Elemente enthalten, in denen sie referenziert werden. Beim Integrieren solcher Hyperslices treten somit semantisch gleiche Elemente auf, die durch Verschmelzen oder Ersetzen vereinigt werden müssen. Beispielsweise müssen einem Zustand mehrere Zustandsübergänge zugeordnet werden. Die Bedingungen für die Ausführung der Zustandsübergänge sind dazu zu verknüpfen. Der Generator benötigt Informationen zu Art und Weise der Integration, damit er eine Spezifikation erstellen kann. In HyperFeatuRSEB werden diese Informationen während des Hyperspace-Engineering erstellt und als Integrationsmethoden hinterlegt. Wählt ein Kunde Features aus, übernimmt der Generator die entsprechenden Integrationsmethoden in die Hyperspace-Spezifikation. Diese steuern dann die Integration der entsprechenden Hyperslices zu einem Produkt mit den gewünschten Features.

Wegen der hohen Anzahl möglicher Kombinationen zwischen Hyperslices können die Ziele des Ansatzes – vereinfachte Produkterstellung durch Komposition und Evolution durch bessere Wartbarkeit - nicht erreicht werden, wenn die Integrationsmethoden manuell gesteuert werden sollen. Deshalb wird in HyperFeatuRSEB eine sogenannte *Integrationsstrategie* festgelegt, die als verbindliche Vorgabe für alle zu integrierenden Elemente mit frühem Bindungszeitpunkt die Integrationsmethode Verschmelzen festlegt. Der Generator wird danach für alle übereinstimmenden Elemente diese Integration in Hyperspace-Spezifikationen festlegen, ohne dass ein Entwickler hierzu weitere Festlegungen treffen muss. Nur wenn der Entwickler mit dieser Integrationsmethode nicht die gewünschten Ergebnisse erzielen kann, oder wenn für einen späteren Bindungszeitpunkt Variabilität in einem Produkt erhalten bleiben muss, legt er Abweichungen von dieser Strategie fest. Durch die Vorgabe der Integrationsstrategie verringert sich nicht nur der Aufwand bei der Komposition, sondern auch die Komplexität der Produktlinie, was die Wartbarkeit verbessert. Im Abschnitt 10.3.2 werden quantitative Ergebnisse dazu angegeben.

Die Integrationsstrategie vereinfacht auch das Hyperslice-Engineering, weil die Entwickler bei der Zerlegung und Entwicklung fast immer die Integrationsmethode Verschmelzen berücksichtigen können und nur für einen äußerst geringen Teil der Kombi-

nationsmöglichkeiten konkret zu definierende Integrationsmethoden festlegen müssen. Diese Fälle konzentrieren sich auf das Verschmelzen von Operationen mit Rückgabewerten. Werden Operationen (als Konsequenz aus der Integrationsstrategie) durch Verschmelzen integriert, ist ein gemeinsamer Rückgabewert zu berechnen. Anstelle einer allgemeingültigen Vorgabe ist hier eine manuelle Definition der Integrationsmethode und der Summierungsfunktion durch den Entwickler erforderlich. Es gibt allerdings Möglichkeiten, durch geeignetes Refactoring der Operationen die Vereinigung von Rückgabewerten zu vermeiden [Böllert 2002][Fowler 1999].

**Entwurf von Objektmodellen im Hyperslice-Engineering**
Während der iterativ-inkrementellen Erstellung des Objektmodells entstehen Hyperslices, denen Entwurfselemente zugeordnet werden. Dabei wird eine Zerlegung vorgenommen, die eine Zuordnung von Entwurfselementen zu Features vornimmt. Bei der Verfeinerung und Zerlegung des Objektmodells führt der Entwickler ähnliche Tätigkeiten wie bei der Entwicklung einer Architektur oder beim Refactoring (siehe 8.3.4) durch. Abb. 48 zeigt den Ablauf als UML-Aktivitätsdiagramm.

Durch die Zerlegung in Hyperslices wird das Objektmodell etwas komplexer: Seine Komplexität steigt dadurch, dass ein Entwickler Kombinationsmöglichkeiten zwischen Hyperslices in die Betrachtung einbeziehen muss. Dadurch wird die Verständlichkeit verringert. Durch die Festlegung einer Integrationsstrategie wird die Verständlichkeit verbessert, da im Regelfall nur die Integrationsmethode Verschmelzen angewendet wird. Modellierungswerkzeuge können unter Nutzung dieser Integrationsstrategie die Auswirkungen von Änderungen im Zusammenhang mit anderen Hyperslices darstellen und so die Arbeit von Entwicklern unterstützen.

Dieser Vorteil der Integrationsstrategie wirkt sich um so stärker aus, je besser die Zerlegung des Objektmodells darauf abgestimmt ist. Basierend auf Erfahrungen der Fallstudie von [Halle 2001] und [Böllert 2002] wurden Regeln formuliert, die eine Anwendung der Integrationsmethode Verschmelzen unterstützen. Die Beachtung der Regeln führt dazu, dass manuell zu definierende Integrationsmethoden zusätzlich zur Integrationsstrategie vermieden werden können:

- Bei der Festlegung der hierarchischen Gliederung der Architektur müssen semantisch gleiche Elemente sich in allen Hyperslices in der für Hyper/UML definierten Elementhierarchie (siehe Abb. 31, S. 91) auf gleicher Ebene befinden. Die Hyperslices müssen der Referenzarchitektur folgen, wie sie durch den Hyperspace definiert ist.
- Semantisch übereinstimmende Elemente des Objektmodells müssen den gleichen Namen tragen. Bei Operationen muss zusätzlich die Signatur übereinstimmen.

Die Einhaltung der Regeln dient dazu, dass die Hyper/UML- und Hyper/J-Werkzeuge zusammengehörende Elemente mit den in den Abschnitten 2.3.5 und 5.4ff genannten Methoden erkennen und verschmelzen können, ohne dass zusätzliche Integrationsmethoden definiert werden müssen. Die zur Einhaltung der Regeln über getrennte Hyperslice-Engineering-Prozesse (und ggf. Entwicklerteams) notwendige Koordination benötigt eine übergeordnete Instanz. Der Prozess (und das Team) des Hyperspace-Engineering ist hierfür prädestiniert, denn es verfügt durch die Festlegung der Referenzarchitektur über die notwendigen Informationen zur Überprüfung der Architektur der Hyperslices. Weiterhin wurde vom Hyperspace-Engineering im vorherigen Schritt das Anforderungsmodell entwickelt und überprüft. Außerdem sammelt dieser Prozess

Meldungen über erstellte Klassen der einzelnen Hyperslices zwecks Definition von Integrationsmethoden; es kann dabei auch deren Einordnung in Architektur und Namensraum koordinieren.

Von besonderer Bedeutung ist die Koordinierung derjenigen Hyperslice-Engineering-Prozesse, die zueinander in Beziehung stehende Features und Hyperslices betreffen. Solche Beziehungen sind im Featuremodell beispielsweise als require- oder exclude-Beziehung dargestellt. Die Abhängigkeiten spiegeln sich in entsprechenden Abhängigkeiten zwischen Hyperslices wieder und erfordern eine Abstimmung der Arbeitsweise, falls die Entwicklung von unterschiedlichen Teams vorgenommen wird. Zwar verhindert die deklarative Vollständigkeit das Auftreten direkter Referenzen zwischen Elementen verschiedener Hyperslices, doch durch Bezug auf semantisch gleiche Klassen entstehen indirekte Abhängigkeiten, die sich bei der Integration auswirken.

Die Hyperslice-Engineering-Prozesse zu Umsetzung der gemeinsamen Features sind ebenfalls von besonderer Bedeutung für den Erfolg einer Produktlinie. Meist wird für diese Features *ein* Hyperslice gebildet, der als Kern der Produktlinie ihr grundlegendes Objektmodell enthält. Von diesem Objektmodell sind alle anderen Hyperslices für variable Features abhängig. Deshalb hat es sich als sinnvoll erwiesen, das Hyperslice-Engineering für gemeinsame Features personell mit dem Hyperspace-Engineering zusammenzulegen. Durch Verringerung des Koordinationsaufwands kann die Qualität von Zerlegung und Architektur gesteigert werden. Die Anzahl von Abhängigkeiten verringert sich; die Wartbarkeit der Produktlinie wird verbessert.

**Modellierung von Variabilität in Hyperslices**
Die Anwendung des Hyperspace-Ansatzes ermöglicht eine einfachere Art der Modellierung von Variabilität, als dies in anderen Ansätzen einschließlich FeatuRSEB möglich ist. Durch Verlagerung von Variabilität in die Integration von Hyperslices anstelle von Variationspunkten, extend-Beziehungen, Vererbung, Design Patterns und Transformationsregeln ist es möglich, Objektmodelle ohne künstliche Abstraktionen und damit ohne zusätzliche Komplexität zu erstellen. Eine Schaffung von Komponenten mit Plug-in-Schnittstellen (siehe 2.2.3) unterstützt die hier vorgestellten Methoden dadurch, dass sie für den dazu geeigneten Teil der Features ebenfalls Variabilität mit einem Minimum an künstlichen Abstraktionen verwirklicht.

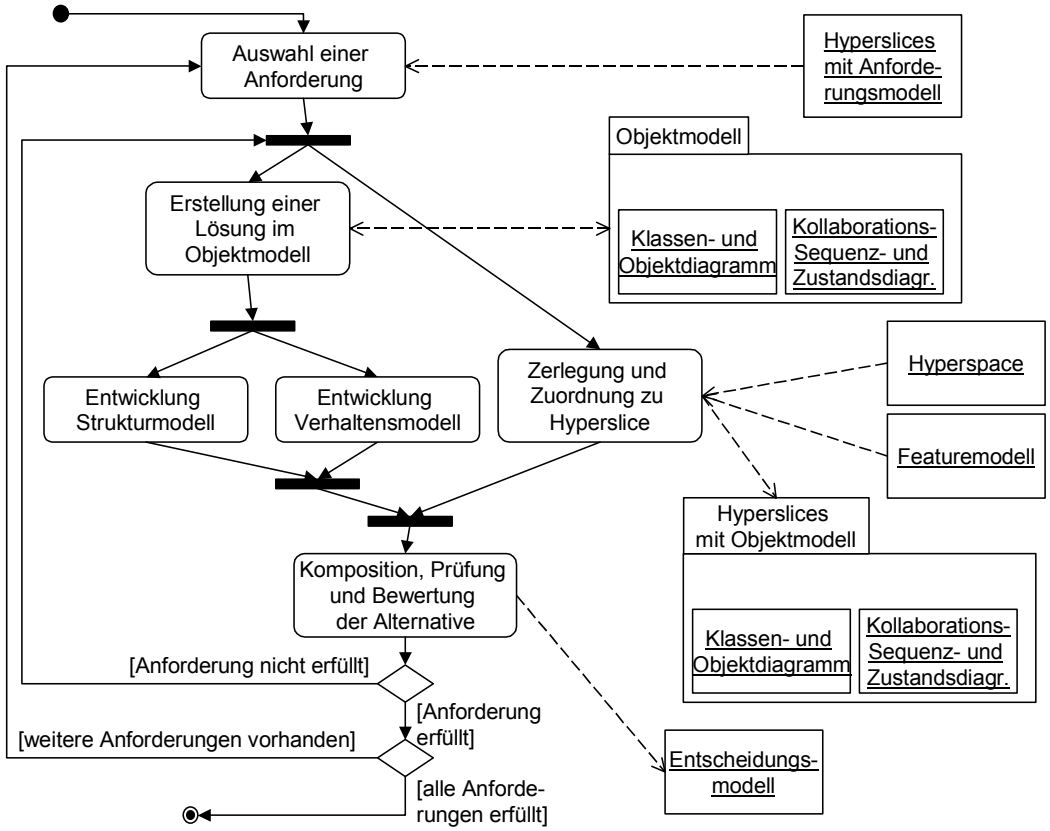



*Abb. 48: Aktivitäten der Domänenmodellierung*

Die Umsetzung von Variabilität in den Anforderungen im Objektmodell und in der späteren Implementierung hängt vom Bindungszeitpunkt ab. Bei frühem Bindungszeitpunkt wird Variabilität durch Kombinationsmöglichkeiten von Hyperslices erreicht. Durch die Art der Zerlegung wird erreicht, dass die gewünschten Kombinationen zu gültigen Produkten führen. Die Bindung von Varianten erfolgt während der anschließenden Komposition mittels Integrationsstrategie. Da die Variabilität in diesem Fall im lauffähigen Produkt nicht mehr enthalten ist, werden später keine Steuerungsmechanismen benötigt.

Abb. 49 zeigt eine Zerlegung, die sich auf das Featuremodell des Fallbeispiels (Abb. 23, S. 70) bezieht. Es zeigt Zustände eines Objekts der Klasse `VideoOutput`. Die Zustände `pdaCommandAcknowledge` und `onScreenDisplay` können nur in den zu den Features PDA-Fernbedienung oder Infrarot-Fernbedienung gehörenden Hyperslices auftreten. Wenn die betreffenden Features wie hier für `productY` ausgewählt sind, werden bei der Komposition die Zustände im Hyperslice `commonCore` um diese Zustände und die dazugehörenden Zustandsübergänge ergänzt.

Für späte Bindungszeitpunkte, beispielweise zur Setup- oder Laufzeit, muss Variabilität im Produkt erhalten bleiben. Variable Teile werden nebeneinander modelliert, wobei anstelle der Integrationsmethoden „Umschaltmechanismen“ mittels Kollaborationen modelliert werden. Auf diese Techniken wird in Abschnitt 7.1 eingegangen. Für die eigentliche Bindung der Variabilität werden Werkzeuge benötigt, damit beispielweise beim Setup Komponenten ausgewählt oder zur Laufzeit Parameter gesetzt werden können. Solche Werkzeuge sind Gegenstand von Abschnitt 7.2. Sie arbeiten nach Regeln, die die Abhängigkeiten zwischen den Features im Featuremodell repräsentieren und die während des Hyperspace-Engineering zu entwickeln sind.

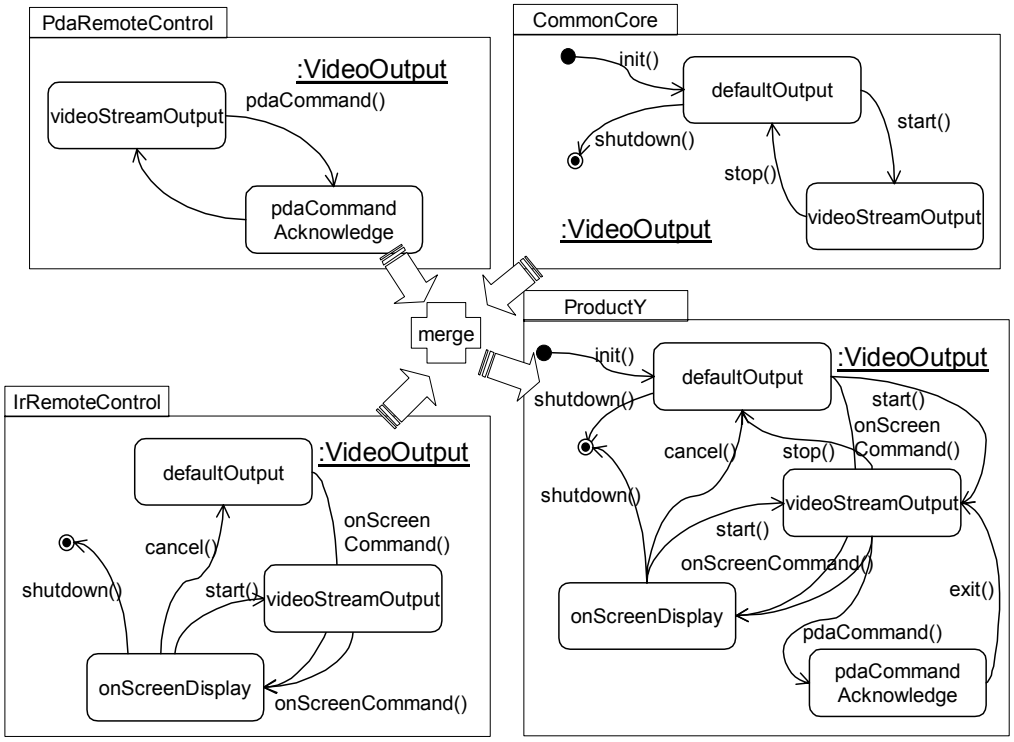



*Abb. 49: Featuregesteuerte Zerlegung mit dem Ergebnis der Verschmelzung am Beispiel eines Zustandsdiagramms*

**Dokumentation der Referenzarchitektur**

Für die Entwicklung und Evolution der Produktlinie ist die Verständlichkeit der Referenzarchitektur von großer Bedeutung. Die Zerlegung des Objektmodells in Hyperslices für jedes variable Feature und die Bereitstellung von Traceability Links zu Features und zur Implementierung erleichtern das Verständnis für die Auswirkung von Änderungen. Trotzdem hat die Bereitstellung einer aussagekräftigen Dokumentation mit umfangreichen Navigationsmöglichkeiten große Bedeutung für Pflege und Weiterentwicklung. Entsprechend der Ziele der vorgestellten Arbeiten muss eine automatisierte Erstellung der Dokumentation erreicht werden. Der Hyperspace-Ansatz kann hier, in Verbindung mit konventionellen Werkzeugen zur Automatisierung der Dokumentation wie javadoc oder Doxygen [Doxygen], zur automatisierten Komposition genutzt werden. Für die gewünschte Dokumentation kann nicht nur eine Zerlegung der Generierungsspezifikationen genutzt werden, es können darüber hinaus auch zusätzliche Features zur Steuerung von Umfang und Inhalt eingeführt werden.

### 6.1.3 Domänenimplementierung

Die Implementierung einer Produktlinie wird als eine wiederverwendbare Plattform bezeichnet, sie stellt die Realisierung der Referenzarchitektur dar. Ziel der Gestaltung der wiederverwendbaren Plattform ist es, im Prozess des Produkt-Engineering eine automatisierte Komposition von Produkten entsprechend ausgewählter Features zu ermöglichen. Aus dem Objektmodell der Referenzarchitektur wird nach den Methoden der objektorientierten Programmierung Quellcode entwickelt. Zur Implementierung wird eine Programmiersprache eingesetzt, die den Hyperspace-Ansatz unterstützt.

Bisher steht hierfür mit Hyper/J eine Abbildung der Programmiersprache Java auf den Ansatz zur Verfügung, eine Abbildung weiterer Programmiersprachen ist vorgesehen.

Die Domänenimplementierung erfolgt wie bereits die Domänenmodellierung in getrennten Prozessen, dem Hyperspace-Engineering für die gesamte wiederverwendbare Plattform und dem Hyperslice-Engineering mit einem Prozess je Hyperslice. Jeder Hyperslice bildet eine Komponente. Entsprechend den Komponenten des Objektmodells, die einzelnen Features zugeordnet sind, werden entsprechende Quellcode-Komponenten entwickelt. Die dabei angewendeten Technologien zur Implementierung von Variabilität unterscheiden sich nach dem geforderten Bindungszeitpunkt. Wie bereits für die Zerlegung des Objektmodells gilt auch für die Implementierung: Jedem variablen Feature wird für jeden Bindungszeitpunkt und jede entsprechende Implementierungstechnologie eine Quellcode-Komponente zugeordnet. Die Referenz von einer Komponente zum dazugehörigen Feature und zur Dokumentation wird mittels Traceability-Links in der in Abschnitt 4.2 vorgestellten Weise umgesetzt.

Für einen frühen Bindungszeitpunkt wird die Implementierung nach dem Hyperspace-Ansatz zerlegt. Das Hyperspace-Engineering ist für die Implementierung der Architektur verantwortlich und koordiniert dazu die Schnittstellen der Quellcode-Komponenten, die in Hyperslice-Engineering-Prozessen entwickelt werden. Außerdem koordiniert das Hyperspace-Engineering die Zerlegung und die Integration und bereitet dadurch die spätere Komposition von Produkten vor. Zusätzlich zur Zerlegung können zur Umsetzung der Variabilität auch Plug-in-Schnittstellen eingesetzt werden (siehe 2.2.3).

Für die inkrementelle Entwicklung ist zusätzlich zu den üblichen Mitteln der objektorientierten Programmierung, zu Sprach-spezifischen Patterns (sog. Idioms) und zu den Mitteln des Hyperspace-Ansatzes das Konzept der *Stubs* verwendbar (Abb. 50). Für Komponenten, deren Schnittstelle benötigt wird, die aber erst zu einem späteren Zeitpunkt voll implementiert werden, können zunächst Platzhalter, sog. Stubs, eingesetzt werden. Ein Stub wird genauso erstellt wie eine Komponente, weist jedoch eingeschränkte Leistungen auf, die von Methodenaufrufen ohne Funktionalität über eingeschränkte bis zu voller Funktionalität bei eingeschränkten Qualitätsmerkmalen reichen können. Im Zuge der inkrementellen Entwicklung lassen sich Komponenten schachteln, wobei unter Nutzung von Stubs in jedem Inkrement Vollständigkeit und Testbarkeit bis zum benötigten Grad erreichbar sind.

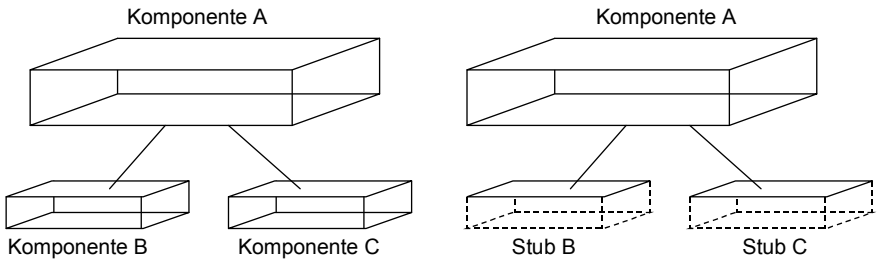



*Abb. 50: Anwendung von Stubs (rechts) als Platzhalter für benötigte Komponenten (links)*

**Implementierung mit Hyper/J**

Im Fall der Nutzung von Hyper/J ist bei der Zerlegung vor allem die Einhaltung der Regeln für die Sichtbarkeit der Klassen und für Namensräume von Bedeutung, die ähnlich wie bei der Domänenmodellierung die Anwendung der Integrationsstrategie Verschmelzen unterstützen. Durch die Vorgabe der Integrationsstrategie wird die Anzahl zusätzlich notwendiger Hyper/J-Integrationsmethoden stark reduziert. Während der

Produkt-Implementierung werden Integrationsmethoden und -strategie zur Ableitung einer Hypermodule-Spezifikation genutzt, anhand der das Hyper/J-Werkzeug den Java-Quellcode erzeugt.

Der Zerlegung liegen Entwurfsentscheidungen zugrunde, die in Referenzen der Klassen und Operationen zu Elementen des Objektmodells abgebildet werden. Jeder Hyperslice der Referenzarchitektur, der Hyper/UML-Modelle enthält, wird in einen Hyperslice mit Quellcode in Hyper/J umgesetzt. Die Zerlegung und Implementierung wird als iterative Folge von Aktivitäten durchgeführt. Dabei werden neben den Vorgaben des Objektmodells auch weitere, insbesondere nicht-funktionale Anforderungen des Anforderungsmodells umgesetzt. Bei der Zerlegung sind neben Refactoring-Techniken (siehe 8.3) die Design Patterns `Command` und `State` [Gamma et al. 1996] sehr vorteilhaft nutzbar, wie die Erfahrungen aus [Ulrich 2002] zeigen.

Das Hyperslice-Engineering erfolgt als Menge separater Prozesse für jeden Hyperslice, die jedoch bezüglich Schnittstellen und Namensvergabe vom Hyperspace-Engineering koordiniert werden. Die einzelnen Hyperslices werden so unabhängig wie möglich gestaltet. Direkte Referenzen zwischen Hyperslices werden dadurch vermieden, dass referenzierte Elemente im betreffenden Hyperslice definiert werden, und dass die Forderung nach deklarativer Vollständigkeit erfüllt wird (siehe 2.3.4). Im Fall der Benutzung von Hyper/J bedeutet dies die Definition abstrakter Klassen und abstrakter Operationen, oder die Verwendung einer Pseudo-Implementierung mit der Anweisung `throw new UnimplementedError()`. Diese Anweisung wird vom Hyper/J-Werkzeug erkannt, die betreffende Operation wird daraufhin als abstrakte Operation behandelt. Operationen, die als Referenz dienen, sind mit der gleichen Signatur wie die referenzierte Operation zu implementieren. Müssen Klassen eingefügt werden, so ist die deklarative Vollständigkeit ebenfalls zu beachten. Im Fall einer Vererbungshierarchie sind alle Oberklassen zu deklarieren. Instanzvariablen (sog. Felder) sind in identischer Weise ebenfalls zu deklarieren.

Die Koordination der Namensvergabe im Hyperspace-Engineering muss sicherstellen, dass semantisch übereinstimmende Klassen in verschiedenen Hyperspaces gleiche Namen tragen und semantisch gleiche Operationen zusätzlich die gleiche Signatur aufweisen. Die Vorgabe des Verschmelzens als Integrationsstrategie wirkt sich hier vereinfachend auf die Komplexität der Implementierung aus, da Entwickler im Regelfall nur diese Integration bei der Prüfung von Kombinationsmöglichkeiten zwischen Hyperslices berücksichtigen müssen. Für einzufügende Pakete hat sich die Namenvergabe nach dem Schema `produktlinie.hyperslice.originalerPaketname` als sinnvoll erwiesen.

Sind neben dem Quellcode weitere Produkte wie eine Nutzerdokumentation oder Inbetriebnahmehilfsmittel zu implementieren, so werden diese ebenfalls in Hyperslices zerlegt, die Features zugeordnet werden. Die Integrationsstrategie ist für die Komposition von Text- und Hypertextdokumenten meist ausreichend, da üblicherweise eine frühe Bindung der Variabilität erfolgt. Falls es erforderlich ist, wird im Hyperspace eine spezielle Integrationsmethode definiert. Die Hyperslices sind auf die verwendeten Werkzeuge hin anzupassen. Im Fall von Dokumentationen reichen die durch Makros und Steueranweisungen erreichbaren Kompositionsmöglichkeiten üblicher Dokumentationsgeneratoren wie Doxygen [Doxygen] aus, damit die Integrationsstrategie Verschmelzen umgesetzt werden kann.

Bei der Verwendung des Hyperspace-Ansatzes sind auch weitergehende Anpassungen solcher zusätzlichen Produkte möglich. Beispiele hierfür sind separate Dokumentatio-

nen für unterschiedlich erfahrene Nutzergruppen oder Inbetriebnahmeanweisungen für unterschiedliche Konfigurationen der Einsatzumgebung. Zum Generieren hierfür angepasster Produkte sind weitere Features zu definieren, die als zusätzliche Zerlegungen zu weiteren Hyperslices der Dokumentation führen.

**Bereitstellung von Testmitteln**
Variabilität und Zerlegung erhöhen die Komplexität einer Produktlinie gegenüber Einzel-Produkten. Die Bedeutung und der Aufwand für die Softwaretests steigt dadurch. Eine manuelle Durchführung von Tests würde wegen der Kombinationsmöglichkeiten der Hyperslices so hohen Aufwand erfordern, dass damit die Vorteile des Ansatzes wieder verloren gehen könnten. Deshalb müssen automatisierte Tests genutzt werden, beispielweise mit dem Werkzeug JUnit [JUnit], wie das schon in vielen Softwareentwicklungs-Projekten üblich ist. Testklassen für JUnit werden ebenfalls in Hyper/J implementiert. Sie können in gleicher Weise wie das Anforderungs- und das Objektmodell in Hyperslices zerlegt und im Zuge des Produkt-Engineering durch das Hyper/J-Werkzeug zu Java-Testklassen zusammengesetzt und mit JUnit ausgeführt werden. Die Spezifikation der Testfälle kann dadurch aus dem Anforderungsmodell gewonnen werden, dass zum Beispiel der Ansatz zur Testfallgenerierung aus Anforderungsmodellen unter Nutzung der UML [Riebisch et al. 2002b] verwendet wird. Der Aufwand für Tests kann auf diese Weise stark reduziert werden.

### 6.1.4 Änderungen und Wartung

Die Durchführung von Änderungen erfolgt im Zuge der bereits beschriebenen iterativ-inkrementellen Entwicklung, die Aktivitäten der Veränderung, Anpassung und Prüfung enthält. Für die Ermittlung der Auswirkungen einer Änderung auf andere Modell- und Implementierungselemente ist eine Analyse der Abhängigkeiten anhand der Traceability-Links (siehe 4.2) hilfreich, die Features mit Modellelementen verbinden. Für die Referenzen kann eine Vollständigkeitsprüfung den Entwickler dabei unterstützen, bei der Umsetzung und Integration von Änderungen Konsistenz und Korrektheit zu erreichen. Weitere Unterstützung erhält er durch die Navigationsmöglichkeiten der Entwicklungswerkzeuge, die Auswirkungen von Änderungen auf andere Hyperslices verständlich machen, sowie durch automatisierte Testfälle, die fehlerhafte Veränderungen aufdecken.

Zur Integration geänderter Elemente gehört die Aktualisierung der Abhängigkeiten. Von besonderer Bedeutung sind dabei wieder die Traceability-Links. Abhängigkeiten und zulässige Konfigurationen von Elementen werden gemäss Lokalitätsprinzip in jedem Artefakt beschrieben (siehe 9.2.5). Alle betroffenen Referenzen werden im Repository aktualisiert, wobei die Werkzeuge des Konfigurationsmanagements die Konsistenzsicherung der doppelten Referenzen in den einzelnen Artefakten übernehmen.

Änderungen mit weiter reichenden Auswirkungen, beispielweise auf die Architektur, erfordern die Migration betroffener Komponenten. Für Entscheidungen zur Architektur, für Architektur-Änderungen und für die Migration gelten die Aussagen der Abschnitte 8.2 und 8.3.

## 6.2 Produkt-Engineering

Der Prozess des Produkt-Engineering soll die automatisierte Bereitstellung von Produkten auf der Basis der Produktlinie ermöglichen. Er ist eng an den Prozess und die Ergebnisse des Produktlinien-Engineering gekoppelt (siehe auch Abb. 16 auf S. 54). Der Prozess existiert für jedes Produkt einmal; er verläuft iterativ-inkrementell.

### 6.2.1 Anforderungsanalyse

Die Erhebung der Anforderungen eines Kunden an ein Produkt erfolgt anhand der für die Produktlinie erfassten Anforderungen. Ziel der Anforderungsanalyse ist ein Anforderungsmodell, das eine möglichst vollständige Komposition aus den für die Produktlinie bereits implementierten Komponenten erlaubt. Das Featuremodell bietet zunächst einen Überblick über Anforderungen, die von der Produktlinie erfüllt werden. Außerdem zeigen Beziehungen und Abhängigkeiten zwischen Features Möglichkeiten und Grenzen der Kombination von variablen Features. Das Anforderungsmodell der Produktlinie enthält Detailinformationen zu Struktur und Verhalten in zerlegter und den Features zugeordneter Form.

Entsprechend dem Hyperspace-Ansatz wird für das zu erstellende Anforderungsmodell zunächst ein Hypermodule angelegt. Die vom Kunden gewünschten Features des Featuremodells werden ausgewählt und als Dimensionen der Zerlegung im Hypermodule hinterlegt. Im Zuge der Verfeinerung der Anforderungen werden die entsprechenden Hyperslices des Anforderungsmodells der Produktlinie ausgewählt.

Im Hypermodule wird während der Verfeinerung der Anforderungen schrittweise die Spezifikation zusammengestellt, die anschließend auf Korrektheit überprüft wird, bevor das Hyper/UML-Werkzeug die Hyperslices zu einem Anforderungsmodell des Produkts integriert. Für ausgewählte Features ist zu entscheiden, ob das Produkt Variabilitäten enthalten soll, die erst beim Setup oder zur Laufzeit instanziert werden sollen. Anschließend liegt ein Anforderungsmodell in UML vor. Testfälle für das Produkt können in gleicher Weise durch Komposition erzeugt werden wie das Anforderungsmodell. Anforderungsanalyse und Scoping sind im Detail in Kapitel 4 vorgestellt worden.

Lassen sich nicht alle Anforderungen des Kunden aus der Produktlinie abdecken, wird mittels Scoping entschieden, ob diese Anforderungen als Ergänzungen nur für dieses Produkt, als Erweiterung der Produktlinie oder überhaupt nicht in Zusammenhang mit der Produktlinie umgesetzt werden. Diese Aktivität ist in Abschnitt 4.6 sowie in [Riebisch et al. 2001a] beschrieben. Erweiterungen sind im Produktlinien-Engineering zu entwickeln; Ergänzungen werden als Komponenten der Anforderungsbeschreibung manuell erstellt. Beide Alternativen unterscheiden sich durch ihren Aufwand.

Erweiterungen der Produktlinie sind mit hohem Aufwand verbunden, müssen doch die neuen Teile in die Referenzarchitektur und die wiederverwendbare Plattform eingefügt und dazugehörige Integrationsmethoden angepasst werden. Nur wenn der Hersteller aus der Erweiterung für zukünftige Kunden Nutzen in Form eines erweiterten Produktspektrums ziehen kann, wird er einen Teil der Mehrkosten als Investition übernehmen.

Rechtfertigt die absehbare Entwicklung eine solche Investition nicht, bleibt noch die zweite Alternative der Ergänzung eines Produkts. Zusätzliche Anforderungen werden dadurch umgesetzt, dass das generierte Objektmodell sowie der generierte Quellcode manuell ergänzt werden. HyperFeatuRSEB ermöglicht, dass dieser Aufwand für nachfolgende Versionen wiederholt genutzt werden kann, wie in Abschnitt 6.3 dargestellt wird.

### 6.2.2 System-Entwurf

Beim Systementwurf wird das Objektmodell des gewünschten Produkts entwickelt. Für alle mit der Produktlinie realisierbaren Features wird dieser Schritt durch das Hyper/UML-Werkzeug durchgeführt, das entsprechend den ausgewählten Hyperslices des Anforderungsmodells die entsprechenden Hyperslices des Objektmodells im Hypermodule zusammensetzt. Dazu werden die Traceability-Links von Features zu Objektmo-

dell-Elementen ausgewertet. Anhand des Bindungszeitpunkts für die verschiedenen Features wird unterschieden, wie die Erstellung des Objektmodells erfolgt. Für eine frühe Bindung der Variabilität erfolgt die Komposition der Hyperslices des Objektmodells mittels Hyper/UML-Werkzeug, das die Integrationsstrategie oder eventuell Integrationsmethoden anwendet. Die Komposition von Objektmodellen mit Hyper/UML ist Gegenstand von Kapitel 5.

Für späte Bindungszeitpunkte werden die für die jeweilige Implementierungstechnologie vorgesehenen Hyperslices oder Komponenten integriert, in dem die entsprechenden Auswahlmechanismen ins Objektmodell eingefügt werden. Dabei werden mittels Kollaborationen, entsprechenden Plug-in-Schnittstellen der Komponenten oder weiteren Maßnahmen Variationspunkte in das Objektmodell aufgenommen, die eine Bindung bei Setup oder zur Laufzeit erlauben. Für Werkzeuge zur Durchführung dieser Bindung werden in der Entwurfsphase Regeln modelliert, die Abhängigkeiten zwischen Features im Featuremodell abbilden. Die Regeln werden im Hypermodule abgelegt und während der Implementierung ausgewertet. Aussagen zur Umsetzung später Bindungszeitpunkte werden in Kapitel 7 gegeben.

Weitere Produkte wie Dokumentationen für Benutzung, Betrieb und Weiterentwicklung werden in gleicher Weise durch Komposition von Hyperslices erstellt, das Verfahren entspricht dem für frühe Bindung der Variabilität.

Für Anforderungen außerhalb der Produktlinie enthält die Produktlinie keine geeigneten Objektmodelle und Integrationsmechanismen. Diese Anforderungen werden deshalb manuell als Ergänzungen des Objektmodells in separaten Hyperslices erstellt und integriert, wie in Abschnitt 6.3 dargestellt wird.

### 6.2.3 System-Implementierung

Für alle Anforderungen, die Bestandteil der Produktlinie sind, erfolgt die Generierung von Quellcode ohne Eingriff von Entwicklern aus dem Bestand der wiederverwendbaren Plattform der Produktlinie. Die benötigten Hyperslices werden anhand der Traceability-Links des Objektmodells und des Featuremodells ermittelt. Für alle Features mit frühem Bindungszeitpunkt wird der Quellcode der betreffenden Hyperslices vom Hyper/J-Werkzeug dadurch zusammengefügt, dass die Integrationsstrategie oder eventuell im Hyperspace definierte spezielle Integrationsmethoden angewendet werden. Es entsteht Quellcode in Java, der in ein lauffähiges System übersetzt und getestet werden kann.

Für Features mit spätem Bindungszeitpunkt werden benötigte Variabilitäten mittels Kollaborationen oder anderen Technologien implementiert. Der entsprechende Quellcode für diese Bindungsmechanismen wird entsprechend dem Objektmodell hinzugefügt. Für die spätere Bindung dieser Variabilitäten werden die während der Modellierung abgeleiteten Regeln genutzt, damit je nach Technologie beispielsweise Konfigurationsdateien für Konfigurator-Werkzeuge erzeugt oder Setup-Werkzeuge instanziert werden können. Dabei wird sichergestellt, dass die Bindung der im Produkt verbliebenen Variabilität den Beschränkungen des Featuremodells folgt. Weitere Aussagen zu Aktivitäten der Implementierung bei späten Bindungszeitpunkten sind in Kapitel 7 enthalten.

Die Implementierung von Ergänzungen des Objektmodells, die außerhalb der Produktlinie liegen, muss manuell vorgenommen werden. Der erzeugte Quellcode ist dazu zu ergänzen. Werden Ergänzungen in der in Abschnitt 6.3 dargestellten Weise als separate Hyperslices gekapselt, können sie in Nachfolgeversionen mit geringem Zusatzaufwand

wiederverwendet werden. Die Inbetriebnahme und Wartung dieser Ergänzungen erfordert Kenntnisse über Referenzarchitektur und Implementierung der wiederverwendbaren Plattform der Produktlinie sowie über verwendete Technologien zur Implementierung später Bindungszeitpunkte. Diese Tätigkeiten sind deshalb mit Unterstützung des Hyperspace-Engineering durchzuführen.

Weitere, zusätzlich zum Quellcode benötigte Ergebnisse wie Dokumentationen werden aus den als Hyperslice vorgefertigten und zerlegten „Implementierungen“ mit der Integrationsstrategie generiert.

Test und Inbetriebnahme haben große Bedeutung für den Erfolg der Produktentwicklung. Da während der Entwicklung der Produktlinie nicht alle Kombinationsmöglichkeiten der Features getestet werden können, muss bei der Produktentwicklung nachgewiesen werden, dass die Anforderungen des Kunden erfüllt werden. Da für Tests hohe Aufwendungen erforderlich sind, werden sie weitgehend automatisiert. Die erforderlichen Testfälle werden aus Anforderungen und Verhaltensmodell abgeleitet [Riebisch et al. 2002b], die benötigten Testprogramme werden in Hyperslices zerlegt, mit der Produktlinie verwaltet und parallel zur Komposition des Quellcodes ebenfalls generiert.

## 6.3 Erweiterung der Produktlinie

Die Erweiterung von Produktlinien hat eine große Bedeutung in der Praxis, weil Software-Produktlinien fast nie „auf der grünen Wiese“ entwickelt werden. Sie werden statt dessen typischerweise auf der Basis erfolgreicher Produkte erstellt und dann schrittweise um weitere Features erweitert. Grund dafür ist die erforderliche Menge an Investitionen für eine vollständige Produktlinie und die mit der Entwicklung verbundenen Risiken. Die Methode HyperFeatuRSEB unterstützt die Erweiterung von Produktlinien in zwei Formen, als Aufnahme separat entwickelter Software und als schrittweise Erweiterung um neue Anforderungen.

### 6.3.1 Umsetzung neuer Anforderungen

Während der Anforderungsanalyse für neue Produkte (siehe 6.2.1) werden oft Anforderungen mit hoher Priorität gestellt. Sie werden durch *Ergänzung eines Produktes* oder durch *Erweiterung der Produktlinie* realisiert. Die Entscheidung darüber wird beim Scoping getroffen (siehe 4.6). Eine solche Erweiterung kann nicht nur für zusätzliche Features erforderlich werden, sondern auch dann, wenn eine Anforderung darin besteht, dass zwei Features gleichzeitig vorhanden sein sollen, die sich aufgrund von Abhängigkeiten wie *exclude* bisher gegenseitig ausgeschlossen haben.

#### Ergänzung eines Produkts um neue Anforderungen

Die Ergänzung eines Produkts um Features außerhalb der Produktlinie erfolgt im Ansatz FeatuRSEB manuell. Das Anforderungsmodell, das generierte Objektmodell und der generierte Quellcode sind manuell zu ergänzen. Für nachfolgende Versionen des betreffenden Produkts müssen diese Tätigkeiten wiederholt werden, damit ist dann der bereits investierte Aufwand verloren. Da diese Ergänzungen nicht Bestandteil der Produktlinie sind, wird ihre Entwicklung außerdem nicht durch das Produktlinien-Engineering koordiniert; für die oft zu wiederholende Ergänzung sind genaue Kenntnisse über die Referenzarchitektur erforderlich, die sonst bei der Produktentwicklung nicht benötigt werden. Sie sind in der Regel nur bei wenigen Entwicklern des Produktlinien-Engineering vorhanden.

HyperFeatuRSEB ermöglicht durch Einbeziehung des Hyperspace-Ansatzes, dass Ergänzungen auch für nachfolgende Versionen und sogar für Produkte mit anderen

Konfigurationen genutzt werden können. Alle Ergänzungen von Anforderungs- und Objektmodell sowie der Implementierung werden dazu als jeweils separates Hyperslice gekapselt und können so durch Komposition in weitere Produkte integriert werden.

**Erweiterungen der Produktlinie um neue Anforderungen**
Erweiterungen der Produktlinie stellen die zweite Möglichkeit dar, neue Anforderungen für Produkte einer Produktlinie zu realisieren. Im Zusammenhang mit dem Scoping wurde erwähnt, dass eine Erweiterung einer Produktlinie sehr viel mehr Aufwand erfordert als eine Ergänzung eines Produkts. Der höhere Aufwand wird deutlich, wenn man zwei verschiedene Kategorien von Anforderungsänderungen gegenüberstellt: *Positive Anforderungen* fügen Funktionalität und andere Eigenschaften zu einem Produkt hinzu, wogegen *negative Anforderungen* das Entfernen von Funktionalität erfordern. Hohen Aufwand erfordert vor allem die zweite Kategorie, wenn sie sich als Erweiterung auf eine Produktlinie auswirken soll und nicht nur als Ergänzung auf ein Produkt.

Die Umsetzung von neuen Anforderungen erfolgt in den in Abb. 51 dargestellten und im Folgenden angeführten Schritten im Produkt- und Produktlinien-Engineering.

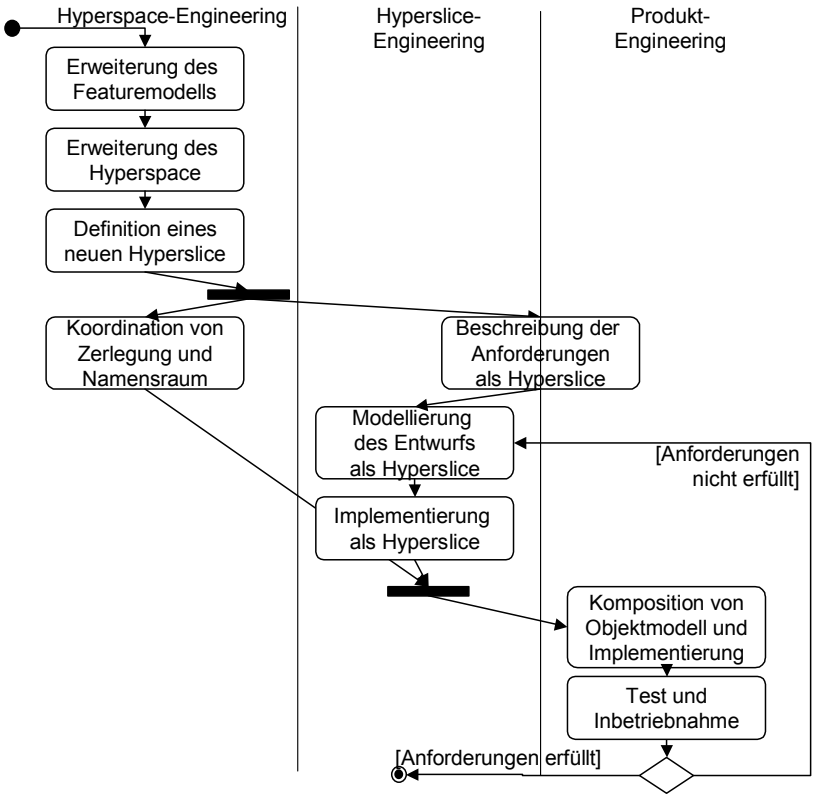



*Abb. 51: Erweiterung einer Produktlinie als UML-Aktivitätsdiagramm*

1. Erweiterung des Featuremodells. Eine neue Anforderung wird als variables Feature ins Featuremodell aufgenommen; Beziehungen zu bestehenden Features sind zu definieren. Dabei ist die Gesamtheit der variablen Anforderungen aller Produkte zu berücksichtigen:

    1.1 Minimieren der Abhängigkeiten zwischen Features (vgl. Abschnitt 4.4.1);

1.2 Verfeinerung und weitere Zerlegung der Features so dass alle vorgesehenen Produkte daraus zusammengestellt werden können;

1.3 Reduzieren der Variabilität, damit der Aufwand für das Objektmodell und die Implementierung begrenzt wird.

2. Erweiterung des Hyperspace der Produktlinie. Für ein neues Feature wird ein neuer Belang zur Feature-Dimension des Hyperspace hinzugefügt. Eine Traceability-Beziehung verknüpft den Belang mit dem dazugehörigen Feature.
3. Definition eines neuen Hyperslice für das neue Feature als Teil der Produktlinie.
4. Beschreibung der Anforderungen, Modellierung und Implementierung. Änderungen am Anforderungsmodell, am Objektmodell und im Quellcode werden jeweils im betreffenden Hyperslice gekapselt. Falls spezielle, von der Integrationsstrategie abweichende Integrationsmethoden erforderlich sind, werden diese definiert.
5. Generieren des Produkts durch Komposition. Das Produkt für den betreffenden Kunden wird mit Hilfe der Werkzeuge generiert, wozu die Feature-Konfiguration so zu verändern ist, dass die Ergänzungen einbezogen werden.

Die Schritte werden für jede Anforderung iterativ wiederholt, bis alle Anforderungen eines Kunden erfüllt sind. Dabei muss der Schwerpunkt methodischer Unterstützung auf Schritt 4 gelegt werden, weil Änderungen hohe Fehler-Risiken bergen.

Die Umsetzung von Änderungen in diesem Schritt ist für beide Kategorien individueller Anforderungen verschieden. Positive Anforderungen sind relativ einfach zu realisieren. Die Integrationsstrategie Verschmelzen reicht meist zum Hinzufügen zusätzlicher Modellelemente oder Klassenbestandteile aus. Gegebenenfalls müssen die Integrationsmethoden Ordnen und Summieren genutzt werden, damit beispielsweise bestimmte Abläufe und Erweiterungen von Rückgabewerten von Operationen erzielt werden können.

Die Umsetzung negativer Anforderungen erfordert die Zerlegung eines Features in Teile, das vorher in Form eines Hyperslices oder als Bestandteil des gemeinsamen Kerns realisiert war. Hier unterscheiden sich die zwei genannten Möglichkeiten Ergänzung und Erweiterung: Bei einer Ergänzung wird nur die Sichtbarkeit oder Erreichbarkeit des zu entfernenden Feature-Teils verhindert; Bei einer Erweiterung werden die Hyperslices mit Objektmodell und Implementierung so zerlegt, dass der zu entfernende Feature-Teil vom Rest des ursprünglichen Features abgetrennt wird. Die Ergänzung ist einfacher zu erreichen; sie führt jedoch dazu, dass nicht nutzbare Teile in der Implementierung verbleiben, die ihrerseits Ressourcen verbrauchen. Eine Erweiterung erfordert neben dem Aufwand der Zerlegung eine Überprüfung der Integrationsmethoden, die sich auf die betroffenen Hyperslices beziehen, sowie Aufwand für den Test und die Inbetriebnahme der geänderten Hyperslices.

Ein Beispiel soll die Erweiterung illustrieren. In die Produktlinie des Fallbeispiels (siehe 3.4) soll als neue Anforderung die Aufnahme, Speicherung und Wiedergabe anhand persönlicher Profile gesteuert werden. Ziel ist die Anpassung an den persönlichen Geschmack des jeweiligen Nutzers. Eine weitere neue Anforderung dient einer Zugangsbeschränkung für minderjährige Nutzer bezüglich von Inhalten und Umfang des Video-Konsums. Abb. 52 zeigt die Erweiterung des Featuremodells gegenüber Abb. 23 (S. 70) um drei neue Features Inhaltssteuerung, Persönliche Profile und Kinderschutz.

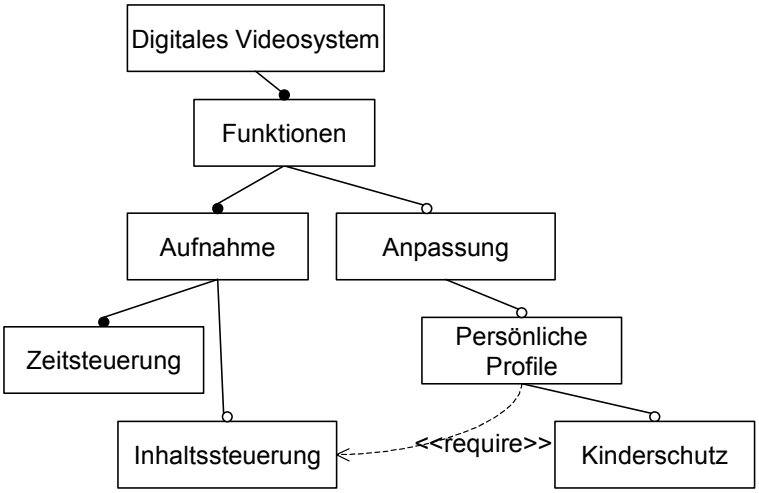



*Abb. 52: Erweiterung des Featuremodells des Fallsbeispiels*

Die entsprechenden Änderungen am Objektmodell zeigt Abb. 53. Im Hyperslice `PersonalProfiles` für das Feature Persönliche Profile gibt es eine zusätzliche Klasse `Profile`, die Informationen zum Wunschprofil enthält, die vorrangig zur inhaltsgesteuerten Aufnahme von Videos dienen. Bei Kinderschutz wird diese Klasse um ein neues Attribut `maxConsumption` erweitert, mit dem der maximale Videokonsum gesteuert werden kann. Die existierenden Operationen `start(), stop()` und `selectVideoItem()` im Hyperslice `CommonCore` werden durch die gleichnamigen Operationen von `ChildProtection` ersetzt, damit die Wiedergabe durch die Kriterien Mindestalter (im Vergleich zur Altersangabe für die Nutzer eines Videos) und Maximalkonsum gesteuert wird. Die Operationen `start()` und `stop()` enthalten außerdem die Erfassung der Dauer zur Überwachung des Maximalkonsums. Damit das gewünschte Verhalten eines Stopps der Wiedergabe bei Überschreibung der vorgegebenen Maximaldauer erreicht wird, ist eine zusätzliche Bedingung in die Wiedergabe des gemeinsamen Kerns der Produktlinie aufzunehmen, von der Auswirkung her also eine negative Anforderung.

Durch die Steuerung der Komposition kann der Entwickler das Ergebnis beeinflussen. Für die Operationen `start()`, `stop()` und `selectVideoItem()` der Klasse `VideoOutput` legt er die Integrationsmethode Ersetzen fest, damit die entsprechenden Operationen im Hyperslice `CommonCore` deaktiviert werden. Im Fall negativer Anforderungen ist häufig diese Integrationsmethode erforderlich, damit durch Entfernen oder Überschreiben ungewünschtes Verhalten deaktiviert wird.

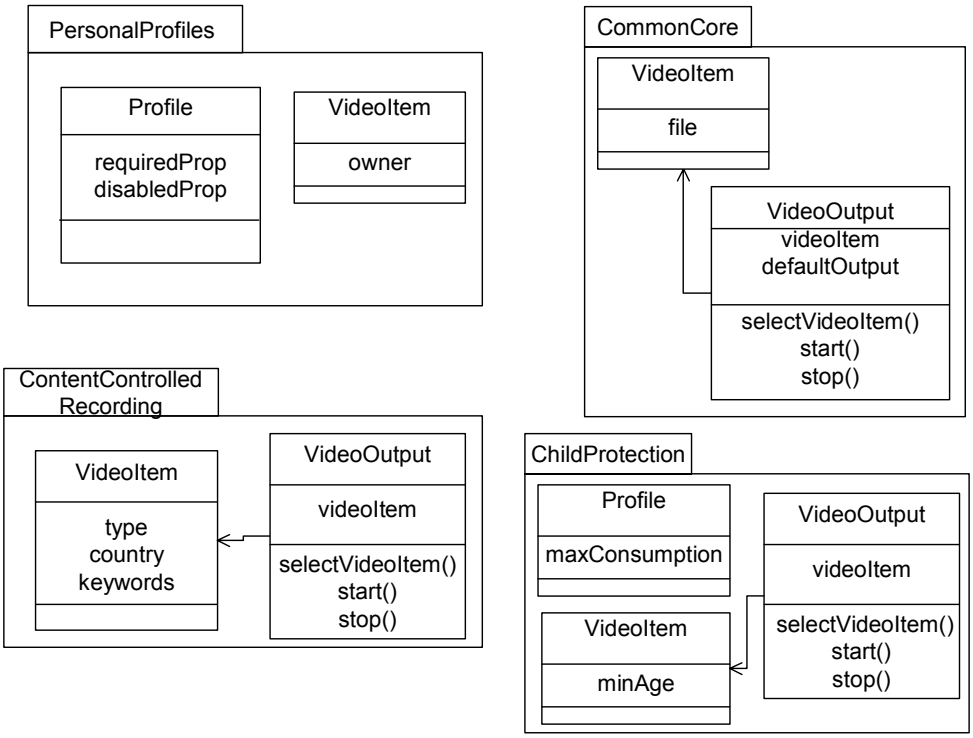



*Abb. 53: Zusätzliche Hyperslices zur Umsetzung der Erweiterung um Kinderschutz aus Abb. 52 als UML-Klassendiagramm*

Die Zerlegung von Hyperslices mit Objektmodell und Quellcode erfordert Entwurfs- und Refactoring-Aktivitäten, wie sie im Produktlinien-Engineering durchgeführt werden (siehe Abschnitte 6.1.2 und 6.1.3). Von besonderer Bedeutung für den Erfolg ist dabei die Berücksichtigung der Auswirkung von Änderungen. In der vorgestellten Methodik ALEXANDRIA wird dies durch die Traceability-Beziehungen erreicht. Außerdem wird während dieser Tätigkeiten im Produktlinien-Engineering gute Klarheit der Modelle angestrebt. Durch iteratives Vorgehen sollen Risiken minimiert werden. Entscheidungen für technische Konzepte sind an der wirtschaftlichen Strategie auszurichten, die vom Management mittels Produktlinie verfolgt und von den Kundenwünschen beeinflusst wird.

**Auflösung von Abhängigkeiten zwischen Features**

Bisher wurden positive und negative Anforderungen unterschieden, die zum Hinzufügen oder Entfernen von Features führen. Ein weiterer Fall individueller Anforderungen ist gegeben, wenn Features gewünscht werden, die zwar zum Bestand der Produktlinie gehören, aber in der gewünschten Kombination nicht zulässig sind. Bei der Anforderungsanalyse für das betreffende Produkt (siehe Abschnitt 6.2.1) wird dies daran sichtbar, dass die gleichzeitige Auswahl der gewünschten Features durch eine Abhängigkeitsbeziehung im Featuremodell ausgeschlossen wird. Dieser Ausschluss kann beispielsweise durch eine exclude-Beziehung oder durch eine Gruppierung als Alternative ausgedrückt sein. Derartige Abhängigkeiten, ihre Ursache und ihre Auswirkungen sind in den Abschnitten 4.1 und 4.5 erörtert worden. Wurde im Scoping zugunsten der Realisierung der betreffenden Kombination von Anforderungen entschieden, wird das weitere Vorgehen von der Ursache der Ausschlussbedingung bestimmt.

Wenn als Ergebnis der Anforderungsanalyse der wechselseitige Ausschluss von zwei Features festgelegt wurde, dann ist lediglich zu prüfen, ob sich die dazugehörigen Hy-

perslices zu semantisch korrekten Hypermodules integrieren lassen. Gegebenenfalls sind spezielle Integrationsmethoden festzulegen oder Veränderungen an den Hyperslices vorzunehmen.

Häufig bestehen Ausschlussbedingungen zwischen Features wegen fehlender Kombinierbarkeit der dazugehörigen Objektmodelle oder der Implementierungen. Solche Fälle entstehen beispielsweise dadurch, dass sich Klassen oder Operationen in Aufbau und Schnittstellen unterscheiden, oder dass begrenzte Ressourcen die gleichzeitige Implementierung verhindern. Damit eine Integration der betroffenen Hyperslices erreicht wird, müssen Veränderungen an Objektmodell und Implementierung vorgenommen werden, die auch die Architektur der Produktlinie betreffen können. Diese Aktivitäten werden im Produkt-Engineering vorgenommen. Dabei ist durch Tests sicherzustellen, dass andere Konfigurationen der betreffenden Hyperslices von den Änderungen nicht beeinträchtigt werden.

Wurde durch die Veränderungen die Abhängigkeit erfolgreich aufgelöst und ist dies durch Tests nachgewiesen, kann die betreffende Abhängigkeitsbeziehung im Featuremodell entfernt werden. Die nun zulässige Kombination der betreffenden Features steht damit für alle weiteren Produkte und Kunden zur Verfügung.

**Bewertung**
Die Ergänzung oder Erweiterung der Produktlinie um individuelle Anforderungen führt zu neuen Hyperslices, oder Hyperslices werden zerlegt und verändert. Aufgrund dieser Eigenschaft des Ansatzes HyperFeatuRSEB sind solche Änderungen nicht nur für eine Version oder ein Produkt nutzbar, sie stehen auch für weitere Produkte und andere Kunden zur Verfügung, weil sie dort durch Konfiguration der Features direkt eingebunden werden können. Der Ansatz unterstützt die Evolution von Produktlinien dadurch, dass einmal vorgenommene Änderungen in die zukünftige Entwicklung einbezogen werden. Einer Aufteilung einer Produktlinie in mehrere Entwicklungspfade, wie sie bei vielen manuell entwickelten Produktlinien zu beobachten ist, wird damit vorgebeugt.

### 6.3.2 Aufnahme externer Produkte und Komponenten in eine Produktlinie

Zur evolutionären Weiterentwicklung von Software – in der Stufe Evolution des Stufen-Modells, siehe 2.4.2 – gehört auch die Aufnahme existierender Software in eine bestehende Produktlinie. Dabei handelt es sich meist um konventionell entwickelte Softwaresysteme und Produkte oder um wiederverwendbare Komponenten. Für die Aufnahme solcher Software kann es verschiedene Anlässe geben:

- Bei der Entwicklung von Produktlinien ist es aus wirtschaftlichen Gründen meist sinnvoll, einzelne Produkte zunächst außerhalb der Produktlinie zu entwickeln, und sie später bei abschätzbarem Kundenkreis und Marktchancen zu integrieren. Dadurch werden die wirtschaftlichen Risiken und die Investitionshöhe verringert.
- Die Zusammenführung von Firmen und deren Produktportfolios ist häufig Anlass für eine Integration von Softwaresystemen in eine Produktlinie.
- Die Wiederverwendung externer Komponenten innerhalb einer Produktlinie erfordert ebenfalls die Integration der Komponenten.

Abb. 54 zeigt die Tätigkeiten bei der Aufnahme in Bezug zu Hyperslice- und Hyperspace-Engineering als UML-Aktivitätsdiagramm. Die aufzunehmende Software weist eine Menge von Features auf, die in manchen Fällen variabel sind. Bei der Aufnahme in eine Produktlinie muss die Variabilität dieser Software oft durch Refactoring erweitert werden. Auch die Architektur muss meist verändert werden. Dazu ist zunächst

eine Analyse dieser Software mittels Reverse Engineering notwendig, wenn Anforderungs- und Entwurfbeschreibung nicht alle für die Veränderung notwendigen Informationen liefern (siehe 8.1). Das Resultat sind eine Anforderungsbeschreibung, ein Featuremodell sowie eine Architekturbeschreibung dieser Software, die durch Traceability-Links miteinander verbunden sind.

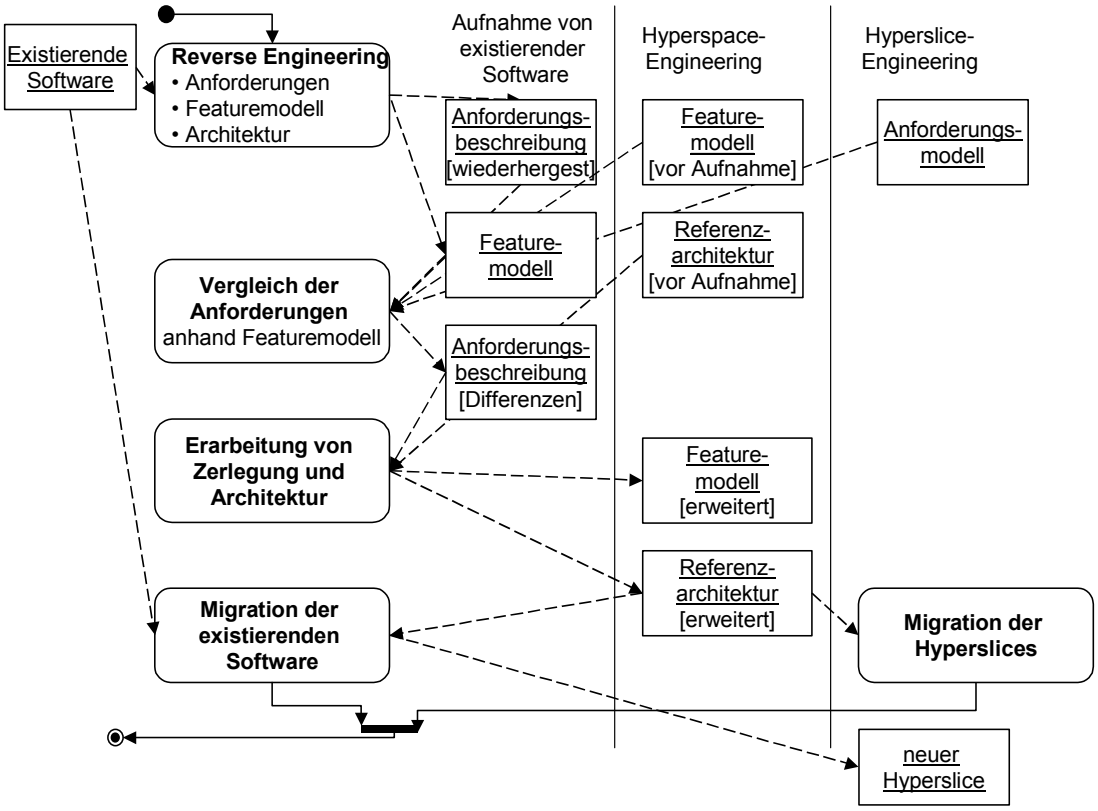



*Abb. 54: Aufnahme existierender Software in eine Produktlinie im Überblick*

Der Vergleich zwischen aufzunehmender Software und Produktlinie beginnt mit den Anforderungen und bezieht sich auf alle Merkmale von Variabilität bis hin zu inneren und nichtfunktionalen Qualitätsmerkmalen. Die dabei festgestellten Differenzen werden bei der Erarbeitung der neuen Architektur ausgewertet (siehe 8.2). Ergebnis sind eine Referenzarchitektur und eine Zerlegung, die der Variabilität und den Anforderungen nach der Aufnahme entspricht.

Für die aufzunehmende Software beschreiben die Ergebnisse die geforderten Schnittstellen und die geforderte Zerlegung, damit sie als Hyperslice Teil der Produktlinie werden kann. Die Veränderungen an der Software werden durch Migration vorgenommen (siehe 8.3). In manchen Fällen sind auch Veränderungen an bestehenden Hyperslices der Produktlinie erforderlich, insbesondere wenn die Zerlegung verfeinert werden muss – ähnlich wie bei der Erweiterung um negative Anforderungen (siehe 6.3.1). Die Migration umfaßt alle Bestandteile wie Anforderungen, Featuremodell, Architektur, Quellcode und Testmittel. Für die betroffenen Hyperslices sind die Integrationsmethoden zu definieren oder anzupassen.

## 6.4 Werkzeuge

Zur erfolgreichen praktischen Anwendung der Methode HyperFeatuRSEB werden Werkzeuge benötigt, die einzelne Aktivitäten unterstützen. Solche Aktivitäten und

Aufgaben sind in Tabelle 2 zusammengestellt. Für einen großen Teil der benötigten Unterstützung sind bereits Werkzeuge verfügbar.

Für Erstellen und Pflege von Quellcode sind das *Hyper/J-Werkzeug* sowie marktübliche Java-Entwicklungswerkzeuge verfügbar. Das Werkzeug Hyper/J [Tarr et al. 2002] erlaubt die Entwicklung und Zerlegung von Hyper/J-Quellcode und seine Komposition zu Java-Quellcode. Es befindet sich jedoch in einem recht frühen Entwicklungsstadium; bezüglich der Integration in Java-Entwicklungsumgebungen sowie bezüglich der Unterstützung bei Routinetätigkeiten wie Ergänzen leerer Operationen für deklarative Vollständigkeit lässt es noch einige Wünsche offen. Für die Unterstützung bei der Navigation und Suche in Hyperslices sind Hervorhebungen von Integrationsmethoden und von abstrakten Deklarationen sowie Suchfunktionen über Namensräume hinweg wünschenswert. Für Arbeiten der Zerlegung ist die Integration in eine Entwicklungsumgebung gemeinsam mit einem Refactoring-Browser [Fowler 1999] sinnvoll, damit Tätigkeiten der Verlagerung und der Aufteilung automatisiert werden können. Auch weist das Werkzeug Hyper/J noch einige Fehler auf. Das Entwicklerteam von IBM leistete jedoch im Verlauf der Fallstudien sehr gute Unterstützung, einige Fehler wurden hierbei behoben.

Für die Erstellung von Produkten aus den in der Produktlinie vorhandenen Hyperslices wurde ein *Produkt-Konfigurator* (Abb. 55) entwickelt. Er bildet Regeln des Featuremodells für Abhängigkeiten zwischen Features ab und stellt auf deren Basis eine Nutzerschnittstelle zur Auswahl der Features für neue Produkte bereit. Der Konfigurator prüft die Gültigkeit der Auswahl und generiert daraus ein lauffähiges Produkt. Er entnimmt dem Repository dazu die notwendigen Quelltext-Dateien und veranlasst ihre Übersetzung sowie ihre Komposition mit dem Hyper/J-Werkzeug. Dieser Konfigurator wurde für die Fallstudie „Siedler von Catan“ entwickelt, er ist jedoch unabhängig von einer konkreten Produktlinie [Halle 2001].

*Tabelle 2: Aktivitäten und mögliche Werkzeugunterstützung bei HyperFeaturRSEB*

| **Aktivitäten des Produktlinien-Engineering** | **Werkzeug-Unterstützung** |
|---|---|
| • Erstellung des Featuremodells | Featuremodell-Editor z.B. AmiEddi (siehe 4.7) |
| • Festlegung der Variabilität und Definition des Hyperspace | Hyper/UML-Werkzeug (siehe 5.8) |
| • Erstellung und Zerlegung von Anforderungsmodellen | |
| • Entwicklung der Architektur | Refactoring-Browser (siehe 8.4) |
| • Entwicklung und Zerlegung von Objektmodellen<br>o Koordination der Hyperslice-Entwicklung Objektmodell (Integrationsmethoden, Hierarchisierung, Namensvergabe)<br>o Überprüfung Objektmodell | |
| • Entwicklung und Zerlegung von Quellcode<br>o Koordination der Hyperslice-Entwicklung Quellcode (Integrationsmethoden, Namensvergabe)<br>o Test der Hyperslices | Hyper/J und marktübliche Java-Entwicklungswerkzeuge<br>Generator für Testfälle |
| • Verwaltung des Repositories<br>o Traceability-Links<br>o Konsistenzbedingungen | |
| **Aktivitäten des Produkt-Engineering für Produkte auf Basis der Produktlinie:** | |
| • Anforderungen analysieren | Marktübliche Werkzeuge zur Anforderungs-Spezifikation |
| • Anforderungsmodell und System konfigurieren | Produkt-Konfigurator |
| • Konfiguration überprüfen | Produkt-Konfigurator |
| • System zusammensetzen | Hyper/J und marktübliche Java-Entwicklungswerkzeuge |
| • Steuerungsmechanismen für späte Bindung bereitstellen | Generator für Werkzeuge zum späten Binden (siehe 7.2) |
| • Test und Inbetriebnahme | Generator für Testfälle |
| **Zusätzliche Aktivitäten für Produkte mit individuellen Anforderungen:** | |
| • Entscheidung über die Realisierungsart individueller Anforderungen | |
| • Veränderung des Objektmodells | Hyper/UML-Werkzeug (siehe 5.8)<br>Refactoring-Browser (siehe 8.4) |
| • Veränderung des Quellcode | Hyper/J und Java-Entwicklungswerkzeuge,<br>Refactoring-Browser (siehe 8.4) |
| • Test und Inbetriebnahme | Generator für Testfälle |

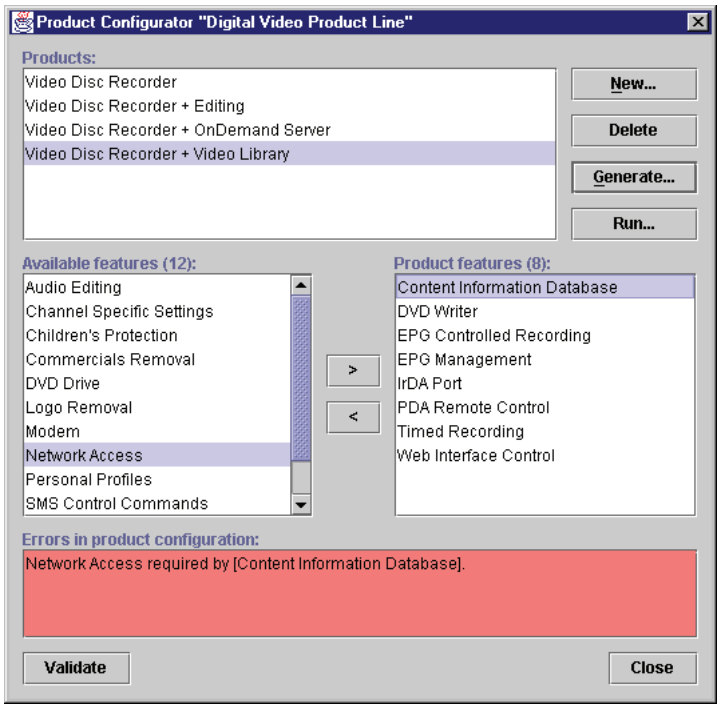



*Abb. 55: Produkt-Konfigurator für die Produktlinie Digitales Videosystem*

Für die Bereitstellung von Testmitteln werden Arbeiten zur modellbasierten *Testfallgenerierung* durchgeführt, die für die Unterstützung von Produktlinien-Methoden konzipiert sind. Auf diese Arbeiten wird in Abschnitt 9.4 eingegangen.

Darüber hinaus besteht die Notwendigkeit einer Werkzeugunterstützung für die Integration der Entwicklungsschritte, für die Koordination der verschiedenen Entwicklungsaktivitäten sowie für die Konsistenzsicherung bei Änderungen. Die *Integration der Entwicklungsschritte* muss durch eine Kopplung der verschiedenen Werkzeuge unterstützt werden. Die Verfügbarkeit eines Werkzeugs für alle Entwicklungsaktivitäten ist aufgrund der Marktsituation für CASE-Werkzeuge nicht zu erwarten. Eine Gliederung der Unterstützung in einzelne, unabhängige Werkzeuge ist u.a. aus Gründen der Flexibilität sinnvoll [Züllighoven et al. 1998]. Die Integration dieser Werkzeuge kann über die Nutzung eines gemeinsamen Repositories und über die Bereitstellung von Schnittstellen erfolgen. Eine solche Werkzeug-Integration ist Voraussetzung dafür, dass die Kopplung von Elementen aus Anforderungsmodellen, Objektmodellen und Quellcode mit Features durch Traceability-Links realisiert werden kann (siehe 4.7). Auch Prüfungen auf Konsistenz erfordern eine solche Integration.

Als geeignetes Mittel für eine Integration heterogener Werkzeuge können XML-Schnittstellen dienen. Auch eine Koordination unterschiedlicher Repositories einzelner Werkzeuge lässt sich unter Nutzung von XML mit verfügbaren Werkzeugen wie Jedi [JEDI 1999] erreichen. So wurde beispielweise AmiEddi [AmiEddi 2002] um eine XML-Schnittstelle erweitert, damit Featuremodelle in ein Repository eingebunden werden können.

Eine erfolgreiche Koordination von Entwicklungsaktivitäten beruht vor allem auf der Kooperation der Entwickler, die diese Arbeiten ausführen. Trotzdem ist hierfür Werkzeugunterstützung notwendig und sinnvoll. Werkzeuge können beispielsweise Informa-

tionen bereitstellen, Abläufe organisieren, Lösungsalternativen verifizieren sowie die Bewertung von Lösungsalternativen dokumentieren. Die Koordination zwischen Hyperspace- und Hyperslice-Engineering bei der Entwicklung und Zerlegung von Objektmodellen (siehe 6.1.2) sei hier als Beispiel vorgestellt. Bei den in Abb. 48 (S. 124) dargestellten Aktivitäten wären die in Tabelle 3 dargestellten Maßnahmen der Werkzeugunterstützung sinnvoll. Derartige Maßnahmen sollten als Bestandteil von CASE-Werkzeugen und Entwicklungsumgebungen realisiert werden.

*Tabelle 3: Werkzeugunterstützung bei Aktivitäten der Domänenmodellierung*

| **Aktivität** | **Maßnahme zur Unterstützung** |
|---|---|
| Zuordnung von Modellelementen zu bestimmten Features und Anforderungen (dabei Koordination mit Hyperspace) | • Auswahlmöglichkeit für Features und dazugehörige Anforderungen<br>Speicherung einer getroffenen Zuordnung als Traceability-Link im Repository |
| Zerlegung eines Hyperslices in zwei neue Hyperslices (dabei Koordination mit Hyperspace und anderen Hyperslices) | • Einblendung der dazugehörenden Features als Ausschnitt des Featuremodells<br>• Einblendung von Klassen und Operationen anderer Hyperslices aus dem Repository bei der Bezugnahme auf diese<br>• Einblenden des Quellcodes aus dem Repository<br>• automatisierte Anwendung eines Refactoring-Szenarios zur Verlagerung in Hyperslices<br>• Darstellung von Kompositions-Resultaten bei der Anwendung der Integrationsstrategie für Teile von Objektmodell und Quellcode<br>• Anpassung der Traceability-Links |
| Änderungen an einer Klasse oder Operation (dabei Koordination mit Hyperspace und anderen Hyperslices) | • Einblendung von Klassen und Operationen anderer Hyperslices aus dem Repository bei Bezugnahme auf diese<br>• Einblenden des Quellcodes aus dem Repository<br>• Darstellung von Kompositions-Resultaten bei der Anwendung der Integrationsstrategie für Teile von Objektmodell und Quellcode<br>• graphische Darstellung von Sichtbarkeitsbereichen und Hierarchien<br>• Einblendung bereits vergebener Bezeichner aus dem Repository mit Navigationsmöglichkeit zur Inspektion von Modell und Quellcode |

# 7 Variabilität von Produkten zur Laufzeit

*In diesem Kapitel werden Methoden vorgestellt, die die Methode HyperFeatuRSEB um Variabilität mit spätem Bindungszeitpunkt erweitern. Es werden sowohl die Möglichkeiten der Komposition derartiger Variabilität als auch Werkzeuge zur Konfiguration der damit erstellten Produkte vorgestellt.*

Die Komposition variabler Elemente nach dem Hyperspace-Ansatz ermöglicht schlanke Produkte durch frühe Bindung der Variabilität, wobei Modell und Quellcode nur die Features umsetzen, die der Kunde ausgewählt hat. Viele Anwendungsgebiete erfordern auch spätere Entscheidungen, beispielweise während der Installation oder zur Laufzeit. Dazu muss in Modell und Quellcode die gewünschte Variabilität der Produktlinie erhalten bleiben; über Werkzeuge oder Hilfsmittel ist die sogenannte späte Bindung von Variabilität zu steuern.

Zahlreiche bereits am Markt befindliche Produktlinien sind bei der Installation oder Inbetriebnahme konfigurierbar. Dabei wird meist ein Entwicklungsprozess ähnlich dem konventionellen angewendet, der auf bekannten Technologien wie Komponenten basiert und keinen Generator anwendet. Eine solche Entwicklung hat einige Nachteile wie:

- Deutlich höhere Komplexität von Architektur und Code gegenüber der vorgestellten Methode und gegenüber konventioneller Softwareentwicklung
- Weiterentwicklung erfolgt meist nur im Quellcode; Modelle veralten schnell
- Abhängigkeiten zwischen Varianten sind komplex und fehlerträchtig
- Geringe Wartbarkeit gefährdet Flexibilität und Langlebigkeit

Die in der Arbeit entwickelten Methoden führen zu Produktlinien, bei denen diese Nachteile vermieden werden und die eine frühe Bindung von Variabilität ermöglichen. Falls erforderlich, können mit diesen Methoden auch Produktlinien mit später Bindung von Variabilität entwickelt werden. Die entwickelten Methoden können für solche Produktlinien zwei Ansätze zur Unterstützung bieten:

1. Produktlinien mit HyperFeatuRSEB stellen auch eine späte Bindung von Variabilität bereit
2. durch Feature-gesteuerte Werkzeuge wird die Konfiguration unterstützt.

Beide Ansätze erweitern den Einsatzbereich der Methode HyperFeatuRSEB. Der erste Ansatz ist Gegenstand des Abschnitts 7.1 dieses Kapitels. Die Arbeiten zu diesem Themenbereich sind noch nicht abgeschlossen, insbesondere eine Bewertung unter industriellen Randbedingungen soll in laufenden Projekten noch erbracht werden. Es wird im Folgenden gezeigt, wie der Framework-Ansatz unter Nutzung von Kollaborationen und Design Patterns in die Komposition mittels HyperFeatuRSEB einbezogen werden kann. Auf diese Weise wird eine späte Bindung der Variabilität erreicht; einige Vorteile wie schlanke Systeme und geringer Overhead durch Variabilität fallen aber weg. Durch Design Patterns und Auswahlmechanismen werden Strukturen eingefügt, die die Komplexität der Lösung erhöhen, aber über die Variabilität hinaus keinen Beitrag zur Funktionalität leisten. Trotzdem bleiben durch die Komposition einige Vorteile erhalten, und zwar für alle diejenigen Features, die durch frühe Bindung festgelegt werden können.

Der zweite Ansatz wird in Abschnitt 7.2 behandelt. Er stellt eine Übergangslösung auf dem Weg zu besserer technologischer Unterstützung der Produktlinienentwicklung dar, wie sie vor allem bei stufenweisem Vorgehen in der Praxis benötigt werden. Dazu wird auf der Basis der Abhängigkeiten zwischen Features ein Konfigurationswerkzeug bereitgestellt, das die Einhaltung der Regeln für gültige Konfigurationen gewährleistet und prüft und daraus Konfigurationsfiles oder Parametersätze generiert. Bezogen auf die Prozesseinteilung nach Abb. 16 (S. 54) handelt es sich um ein separates Werkzeug, für das als Teil des Produktlinien-Engineering Regeln erstellt werden, die im Produkt-Engineering eingesetzt werden.

## 7.1 Laufzeit-Variabilität bei der Nutzung von Hyperslices

Für die Implementierung von Variabilität zur Laufzeit eines Softwaresystems existieren verschiedene Technologien, die Gegenstand der Analyse in Abschnitt 2.2.3 waren: Eine Reihe von Design Patterns bieten Lösungen zur Trennung von Varianten bei Erhaltung der Wartbarkeit, wie die Patterns *Abstract Factory*, *Visitor*, *Command*, *Decorator*, *Builder*, *Factory Method*, *Template Method* sowie *Strategy* [Gamma et al. 1996]. Die *Reflection*-Konzepte von objektorientierten Programmiersprachen bieten weitere Möglichkeiten für flexible Schnittstellen, wie auch Architekturkonzepte durch Plug-in-Komponenten oder andere (siehe 2.2.3).

Die Nutzung der Vorteile des Hyperspace-Ansatzes durch Komposition ist auf drei verschiedenen Wegen möglich: durch Realisierung als zusätzliches Feature, durch manuell gesteuertes oder durch automatisches Einfügen eines Design-Patterns als Verbindungsglied zwischen zwei Hyperslices. Diese Möglichkeiten werden in den folgenden Abschnitten kurz dargestellt, an einem Beispiel aus der Produktlinie Digitales Videosystem illustriert und dann bewertet.

Das Feature `Loeschautomatik` soll durch den Nutzer zur Laufzeit auswählbar sein (siehe Abb. 23, S. 70). Es ergänzt die Verwaltung der Bibliothek um eine Klasse `MemoryController` mit Funktionen zum „Vergessen“ solcher Videos, die länger nicht benutzt wurden. Das „Vergessen“ erfolgt in zwei Stufen, nach einer Phase ohne Zugriff erfolgt eine Kennzeichnung als „zu löschen“; nach einer weiteren Phase ohne Zugriff schließt sich dann die tatsächliche Löschung an, wenn für neue Aufnahmen Speicherplatz benötigt wird. Als Zugriff werden gewertet, wenn ein Video wiedergegeben wird und wenn es in einer Treffermenge bei Suchanfragen an die Bibliothek enthalten ist. Beide Fälle erhalten eine unterschiedliche Wertung. Optional erfolgt vor dem Löschen eine Rückfrage an den Nutzer.

Das Feature `Loeschautomatik` ergänzt das Feature `Bibliothek`, interagiert mit den (gemeinsamen) Features `Aufnahme` und `Wiedergabe` und beeinflusst die `Bedienung`. Ohne insgesamt auf Modell und Implementierung genauer einzugehen, sollen einige Unterschiede zwischen früher und später Bindung der Variabilität untersucht werden. Dieses Feature bietet verschiedene Bedienstrategien zur Steuerung des Löschens von Videosequenzen. Vorhandene Bedienfolgen werden bei Hinzufügen dieses Features um weitere Zustände ergänzt. Die Unterschiede sind in Tabelle 4 dargestellt. Die Auswirkungen auf Modell und Implementierung werden in den folgenden Abschnitten zur Darstellung der Unterschiede untersucht. Weiterhin gibt es eine Bedienmöglichkeit zum Ein- und Ausschalten des Features Löschautomatik, doch diese bezieht sich auf das Steuern der Variabilität des ganzen Features zur Laufzeit und wird durch ein Werkzeug wie in Abschnitt 7.2 ermöglicht.

*Tabelle 4: Auswirkungen von früher und später Bindung des Features Löschautomatik auf Modell und Implementierung*

| Gegenstand | Frühe Bindung | Bindung zur Laufzeit | Art der Auswirkung |
|---|---|---|---|
| Aktion bei Speicherplatzmangel vor Aufnahmestart | Ersetzen des Abbruchs zwecks Löschen (Kern) durch Aufruf der Löschfunktion | Prüfung, ob das Feature ausgewählt ist, dann entweder Löschfunktion oder Abbruch | Variationspunkt Zustandsübergang |
| Einstellung der Rückfrage vor dem Löschen | Zusätzlicher Menüpunkt bei „Einstellungen“ sowohl für PDA-, Web- als auch OnScreen-Dialoge | Prüfung, ob das Feature ausgewählt ist, danach entweder Menüpunkt einfügen oder übergehen | Variationspunkt Listenelement |
| Markierung der Videos | Ergänzung der Klasse `VideoItem` um Attribute `lastPlayed` und `lastSearched` für Zeitstempel und Zugriffsmethoden `setPlayed()`, `setSearched()` | Dto. | Zusätzliche Attribute ohne Variation, Variationspunkt Operation |
| Feststellen des Zugriffs durch Wiedergabe | Ergänzung der Operation `play()` um den Aufruf `VideoItem.setPlayed()` zum Setzen des Zeitstempels | Prüfung, ob das Feature ausgewählt ist, dann entweder Setzen des Zeitstempels oder nicht | Variationspunkt Operation |

### 7.1.1 Laufzeit-Variabilität als separates Feature

Die erste Möglichkeit zur Realisierung von Laufzeit-Variabilität im Kontext des Hyperspace-Ansatzes besteht in der Schaffung eines separaten Hyperslice, das die Variabilität zwischen den betroffenen Hyperslices kapselt. Es wird im Featuremodell durch ein separates Feature repräsentiert, in diesem Beispiel `LoeschautomatikRuntime`. Die Featureauswahl zur Definition eines Produkts umfasst im Gegensatz zur Darstellung in Abschnitt 4.5.2 keine Festlegung des Bindungszeitpunkts: die Entscheidung für dieses Feature erfolgt während der Anforderungsanalyse. Das Objektmodell umfasst die Prüfungen in der 3. Spalte von Tabelle 4 und benutzt sonst die Bestandteile des Hyperslice `Loeschautomatik`, das für frühe Bindung vorgesehen wurde (2. Spalte der Tabelle). Die Implementierung besteht ebenfalls aus einem Hyperslice für die Laufzeit-Entscheidungen mit Bezug auf den Hyperslice `Loeschautomatik`. Die Integration dieser beiden Hyperslices mit dem Kern erfolgt mittels Integrationsstrategie `merge` (siehe 6.1.2). Abb. 56 stellt den Sachverhalt der letzten Zeile der Tabelle im Klassendiagramm dar.

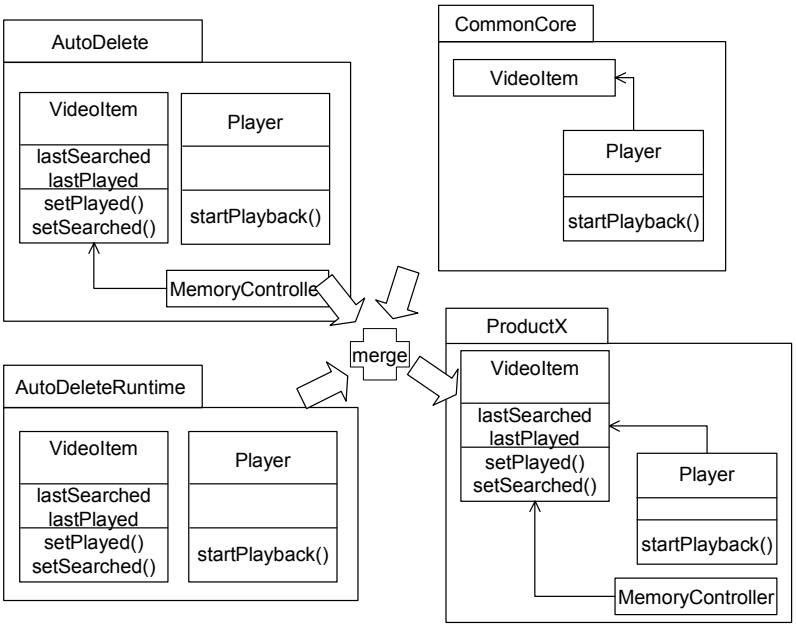



*Abb. 56: Komposition mit speziellem Hyperslice für Laufzeit-Variabilität*

Das spezielle Hyperslice verbindet zwei „echte“ Hyperslices, wodurch wieder eine frühe Bindung möglich wird. Damit können die Vorteile des Ansatzes HyperFeatuR-SEB genutzt werden. Diese Methode bietet folgende Vorteile:

- Es ist eine einfache Entwicklung möglich, auch als produktspezifische Ergänzung einer Produktlinie (siehe 6.3.1),
- Die Vorteile des Hyperspace-Ansatzes wie Schlankheit bleiben bei den übrigen Hyperslices nutzbar,
- Die Methode ist für jede Art von Variabilitätspunkten anwendbar.

Dem stehen als Nachteile gegenüber, dass eine separate, manuelle Modellierung und Implementierung für jede gewünschte Laufzeit-Variabilität erforderlich ist. Der Hyperslice für frühe Bindung muss unbedingt soweit wie möglich benutzt werden, damit Code-Duplikate vermieden werden, die die Wartbarkeit stark beeinträchtigen würden.

### 7.1.2 Laufzeit-Variabilität durch Einfügen einer Kollaboration

Diese Möglichkeit erreicht Laufzeit-Variabilität durch Vorgabe einer Kollaboration als Verbindungsglied zwischen zwei Hyperslices. Sie vermeidet ein spezielles Hyperslice, es bindet den Hyperslice für einen frühen Bindungszeitpunkt unter Nutzung spezieller Integrationsmechanismen ein, die anstelle der üblichen Integrationsmethode vom Hyperslice- oder Hypermodule-Entwickler vorgegeben wurde. Diese Integrationsmechanismen basieren auf Design-Pattern, deren Integration durch Kollaborationen gesteuert vom Hyper/UML- und Hyper/J-Werkzeug umgesetzt wird. Kollaborationen beschreiben das Zusammenwirken von Elementen bei der Problemslösung; im Fall von Pattern veranschaulichen sie die Rolle von einzelnen Klassen und Objekten in Bezug auf die Prinziplösung des Patterns. Zur Beschreibung der Kollaborationen wird die Lösung der sogenannten Erweiterten Kollaborationen von Ivanov [Ivanov 1999] genutzt, die auf Konzepten zur Framework- und Pattern-Instanzierung wie Active Cookbooks [Pree1999] aufbaut.

Die Erweiterten Kollaborationen [Ivanov 1999] basieren auf einer Erweiterung der Kollaborationen der UML um eine Kapselung ähnlich einem UML-Paket und um Erweiterungspunkte, sogenannte Hot Spots. Diese Erweiterungspunkte werden dazu genutzt, die abstrakte Lösung eines Patterns auf eine konkrete Lösung mit Klassen oder Objekten abzubilden und damit zu instanzieren. Eine Erweiterte Kollaboration ist in der Lage, alle Abhängigkeiten und Bedingungen zu beschreiben, die ein Pattern kennzeichnen. Je nach Art der Elemente, die durch die Erweiterte Kollaboration zusammengefasst werden, können verschiedene Arten von Patterns dargestellt werden, wie Tabelle 5 zeigt.

*Tabelle 5: Typen von Elementen bei verschiedenen Arten von Patterns*

| **Art des Pattern** | **Typen der Elemente der Erweiterten Kollaborationen** |
|---|---|
| Architectural Patterns | Pakete und Klassen |
| Design Patterns | Klassen, Attribute und Operationen |
| Analyse-Muster | Use Cases, Pakete und Klassen |
| Idiom | Klassen, Attribute und Operationen |

Die in einer Erweiterten Kollaboration beschriebenen Abhängigkeiten erlauben die werkzeuggestützte Instanzierung des betreffenden Patterns, wie sie beispielsweise im UML-Werkzeug OTW 2.4 [OTW24] implementiert wurden. In der Methodik ALEXANDRIA werden die Erweiterten Kollaborationen zur Komposition von Hyperslices unter Einfügen eines Patterns als „Umschalter“ genutzt.

Eine Kollaboration stellt den Bezug zwischen den zu integrierenden Hyperslices und dem Pattern dadurch her, dass es das Pattern instanziert und Klassen des Hyperslice in das Pattern einbindet. Als Beispiel soll das Design Pattern *Strategy* [Gamma et al. 1996] benutzt werden, das entsprechend Abb. 57 von der Kollaboration instanziert wird.

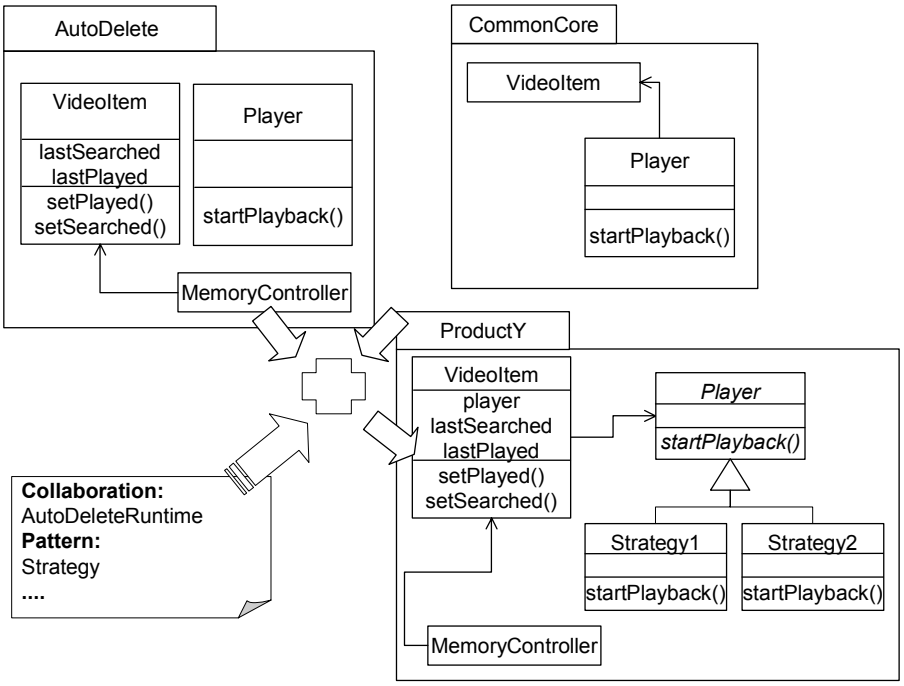



*Abb. 57: Laufzeit-Variabilität durch Komposition unter Einfügen des Design Patterns Strategy*

Das Design Pattern *Strategy* wird in Abb. 58 als UML-Klassendiagramm gezeigt. Es wird hier eingesetzt, weil das Verhalten eines Objekts der Klasse `XY` zur Laufzeit verändert werden soll. Bei einer Lösung ohne solche Variabilität würde die Klasse `XY` eine Operation `method()` enthalten, die das Verhalten implementiert. Gemäß *Strategy* enthält die Klasse einen Verweis `param` auf eine andere Klasse, die diese Operation enthält. Werden mehrere solche Klassen (hier `Strategy1` und `Strategy2`) mit unterschiedlichen Implementierungen von `method()` bereitgestellt, kann zur Laufzeit durch Ändern des Verweises `param` im betreffenden Objekt von `XY` das Verhalten „umgeschaltet“ werden. Dazu muss für die verschiedenen Klassen `Strategy1` und `Strategy2` eine gemeinsame (abstrakte) Oberklasse `Strategy` existieren, die als Typ des Verweises `param` definiert wird.

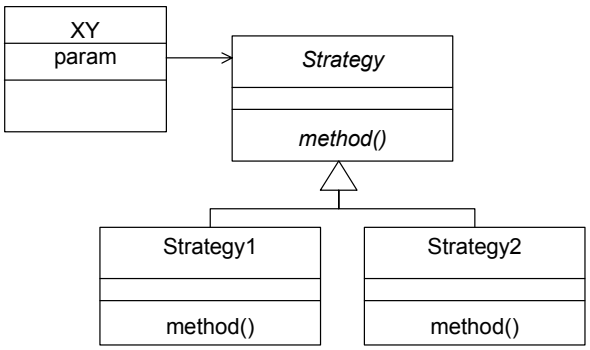



*Abb. 58: Design Pattern Strategy mit Ansatzstellen für die Instanzierung in Abb. 57*

Die Kollaboration verbindet die Hyperslices `AutoDelete` und `CommonCore` so miteinander, dass die gewünschte Funktionalität zur Laufzeit durch den Parameter `selectLoeschautomatik` aktiviert wird. Das Setzen des Parameters wird hier nicht betrachtet, dies wird durch ein Werkzeug gemäß 7.2 vorgenommen. Die Kollabo-

ration wird als Erweiterung der Integrationsmethoden in den Abschnitt `relationships` des Hypermodules aufgenommen:

```
hypermodule ProductY
    hyperslices:
        Feature.CommonCore
        Feature.Loeschautomatik,
        Feature.Bibliothek;
        Feature.PDAremoteControl;
    relationships:
        // integration relationship via collaboration
        collaboration AutoDeleteRuntime hyperslice CommonCore, Auto-
        Delete;
        merge hyperslice CommonCore, Library, PdaRemoteControl;
end hypermodule;
```

Die dazugehörige Kollaboration beschreibt, welche Elemente der betroffenen Hyperslices wie mittels Pattern zu verbinden sind. Hier ist die Steuerung durch den Parameter `selectAutoDelete` definiert, der die Auswahl des Features steuert. Falls der Parameter gesetzt ist, wird die Operation `Player.startPlayback()` aktiviert, die durch eine Integrationsbeziehung Verschmelzen aus den entsprechenden Operationen der Hyperslices `AutoDelete` und `CommonCore` erzeugt wurde. Falls der Parameter nicht gesetzt ist, wird die Operation des Hyperslices `CommonCore` ohne Verschmelzen aktiviert. Abb. 57 zeigt das Ergebnis der Komposition rechts unten im Klassendiagramm. Die dazugehörige Kollaboration wird innerhalb des Hypermodules definiert und benutzt eine ähnliche Syntax wie bei der Definition der Integrationsmethoden:

```
collaboration AutoDeleteRuntime
    Pattern:   Strategy
    XY.Param:  boolean selectAutoDelete
    Strategy1.method():
        merge AutoDelete::Player.startPlayback()
              CommonCore::Player.startPlayback()
    Strategy2.method():
        CommonCore::Player.startPlayback()
```

Die Vorteile dieser Möglichkeit beruhen darauf, dass während des Produkt-Engineering für die einzelnen Features nur deren Bindungszeitpunkte festgelegt werden. Anschließend wird während der Hypermodule-Definition für alle Features mit spätem Bindungszeitpunkt eine Kollaboration festgelegt, ohne dass für diese Bindungszeitpunkte spezielle Hyperslices entwickelt werden müssen. Dadurch ist deutlich weniger Aufwand für Modellierung und Implementierung eines Produkts erforderlich, für deren gewünschte Features bereits Hyperslices existieren.

Voraussetzung für die Anwendung ist die Fähigkeit des Hyper/UML- und des Hyper/J-Werkzeugs, die entsprechenden Kollaborationen ausführen zu können. Spezielle Hyperslices wie in Abschnitt 7.1.1 sind immer dann nicht notwendig, wenn eine geeignete Design-Pattern-Kollaboration verfügbar ist. Jede Kollaboration muss mit Regeln für ihre Anwendbarkeit versehen sein, damit sie einfach vom Hypermodule-Entwickler ausgewählt werden kann.

### 7.1.3 Laufzeit-Variabilität durch automatisches Einfügen von Design-Patterns

Diese Lösung basiert auf der in Abschnitt 7.1.2 beschriebenen Verfahrensweise, die durch umfassende Automatisierung weiterentwickelt wird. Durch automatisches Einfügen eines Design-Patterns als Verbindungsglied zwischen zwei Hyperslices wird Laufzeit-Variabilität erreicht. Der prinzipielle Ablauf der Komposition ist in Abb. 59 dargestellt, wobei die Aufgaben des Entwicklers links und die Tätigkeiten des Werkzeugs rechts dargestellt sind. Das Werkzeug wählt anhand der konkreten semantischen Übereinstimmung zwischen den zu integrierenden Hyperslices eine Kollaboration aus und implementiert damit die Laufzeit-Variabilität. Die Kollaboration ist Teil einer Menge von Kollaborationen für verschiedene Situationen der Komposition. Jede Kollaboration ist durch Vor- und Nachbedingungen beschrieben, so dass eine automatisierte Auswahl ohne Beteiligung eines Hyperslice- oder Hypermodule-Entwicklers möglich ist.

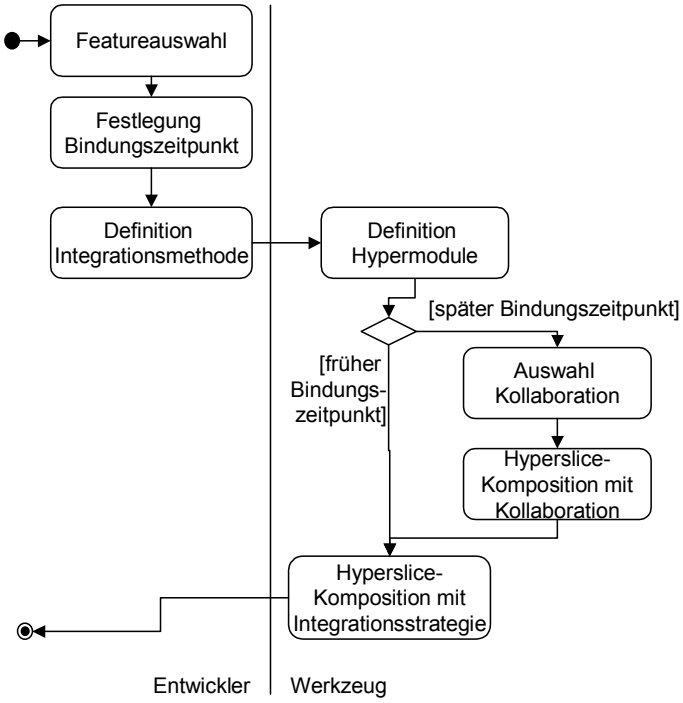



*Abb. 59: Ablauf der Komposition als UML-Aktivitätsdiagramm*

Die praktische Anwendbarkeit dieser Möglichkeit setzt die Verfügbarkeit einer ausreichenden Menge von Kollaborationen voraus, die neben den funktionalen Anforderungen auch nichtfunktionale Merkmale wie Effizienz und Testbarkeit erfüllen. Die Formulierung der Vor- und Nachbedingungen der Kollaborationen, die derzeit mittels OCL vorgenommen werden, müssen für den Entwickler von Hyperslice und Hypermodule ausreichend verständlich, handhabbar und leistungsstark sein. Diese dritte Möglichkeit ist Gegenstand von Arbeiten, die gegenwärtig durchgeführt werden.

## 7.2 Featuregesteuerte Erzeugung von Konfigurationswerkzeugen

Für Produkte mit später Bindung der Variabilität ist eine Steuerung dieser – zur Laufzeit im Produkt vorhandenen – Variabilität erforderlich, die zeitlich und personell getrennt von der Definition und Entwicklung des Produkts stattfindet. In einfachen Fällen erfolgt dies durch Konfigurationsdateien, die bei Einhaltung einer vorgegebenen Syntax vom betreffenden Produkt ausgewertet werden. Zwecks Fehlervermeidung und besserer Benutzbarkeit werden zur Auswahl variabler Features meist Werkzeuge mit graphischer

Nutzerschnittstelle angewendet, die Gegenstand dieses Abschnitts sind. Diese Werkzeuge sind zusammen mit den betreffenden Produkten im Rahmen der Produktlinie zu entwickeln und bereitzustellen. Hier besteht das Ziel, diese Werkzeuge so zu vorzufertigen, dass sie mit wenig oder keinen manuellen Entwicklungstätigkeiten ihre Aufgabe anzupassen sind. Die Anpassung wird im Folgenden als Generierung bezeichnet, sie gehört zum Produkt-Engineering-Prozess (siehe Abb. 16, S. 54). Die Implementierung von verallgemeinerten Werkzeugen, die eine solche Aufgabe übernehmen, ist derzeit Inhalt zweier Industrie-Kooperationen im Bereich Bilderkennung (siehe 10.2.3) und eingebettete Systeme (siehe 10.2.5).

Für die Bindungszeitpunkte Installation oder Inbetriebnahme werden Werkzeuge zum Binden der Variabilität häufig separat oder als Teil von Setup-Werkzeugen bereitgestellt, wie bei vielen Betriebssystemen. Für die Bindung von Variabilität zur Laufzeit sind die Werkzeuge meist in die Produkte eingebunden oder an sie gekoppelt. Viele konventionell entwickelte Textverarbeitungssysteme weisen beispielweise einen Optionen- oder Eigenschaften-Dialog auf, mit dem variable Bestandteile ausgewählt und aktiviert oder variable Komponenten wie Plug-ins angekoppelt werden. Wegen dieser Auswahl variabler Bestandteile hat sich in der Praxis die im folgenden verwendete Bezeichnung Konfiguration für diese Aktivität etabliert.

Unabhängig von der unterschiedlichen Ankopplung und den verschiedenen Mechanismen der Implementierung der Variabilität unterliegen diese Werkzeuge gemeinsamen Anforderungen:

- Arbeiten nach Regeln, die die Abhängigkeiten zwischen (den Laufzeit-variablen) Features abbilden
- Bereitstellung des Ergebnisses (wie Steuerungsvariablen und Parameter) in der benötigten Form und mit korrekter Semantik
- Gestaltung der graphischen Nutzerschnittstelle GUI aus vorgefertigten Elementen wie Checkbox und Radio Button, entsprechend den (Laufzeit-variablen) Features des Produkts
- Bereitstellung des Ergebnisses in der benötigten Form (wie Steuerungsvariablen und Parameter) und mit korrekter Semantik

Abb. 60 zeigt die grundsätzliche Struktur eines solchen Werkzeugs, dessen Nutzerschnittstelle (GUI) in die des zu konfigurierenden Produkts eingebunden sein kann. Es speichert ein Modell der Abhängigkeiten der Features als Ausschnitt des ursprünglichen Featuremodells der Produktlinie, das allerdings nur diejenigen Features und Abhängigkeiten umfasst, die mit Hilfe des Werkzeugs zu konfigurieren sind. Solche Modelle werden oft in verschlüsselter Form gespeichert, weil Informationen über Möglichkeiten und Grenzen einer Produktlinie vor Konkurrenten verborgen bleiben sollen. Dieses Modell der Abhängigkeiten steuert die graphische Nutzerschnittstelle. Die vom Nutzer getroffene Auswahl wird ebenfalls im Modell gespeichert und dazu genutzt, eine Ausgabe in der benötigten Form zu erzeugen. Dabei handelt es sich häufig um Parametersätze oder Textdateien.

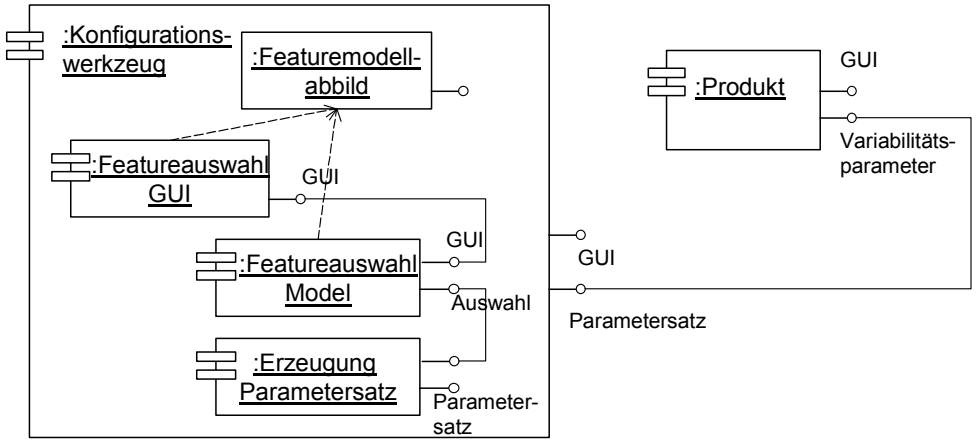



*Abb. 60: Struktur eines Werkzeugs zur Steuerung der Variabilität als Komponentendiagramm*

Die Generierung eines solchen Werkzeugs im Zuge des Produkt-Engineering umfasst die folgenden Aufgaben:

1. Generierung des Featuremodell-Ausschnitts mit Darstellung der Abhängigkeiten
2. Generierung der Nutzerschnittstelle
3. Anpassung der Mechanismen zur Erzeugung der benötigten Ausgabe wie Textdatei oder Parametersatz

Die erste Aufgabe nutzt zur Implementierung dieser Abhängigkeiten die Architekturmittel der Produktlinie. Im Rahmen der vorgestellten Arbeiten wurde wegen seiner Eignung für eine plattformübergreifende Integration die Extensible Markup Language XML zur Darstellung von Anforderungen und Featuremodell verwendet. Während der System-Implementierung (siehe 6.2.3) wird auf der Basis der OCL-Regeln des Featuremodells eine XML-Datei erzeugt, die Informationen über Abhängigkeiten enthält. Eine entsprechende Umsetzung von Datenmodell und Abhängigkeiten in XML wird von [Streitferdt 2003] entwickelt. Die Datei muss vom Konfigurationswerkzeug verarbeitet und in eine interne Repräsentation umgesetzt werden.

Die Nutzerschnittstelle des Werkzeugs als zweite Aufgabe stellt die Auswahlmöglichkeiten dar, die der Variabilität entsprechen. Eine automatisierte Generierung der Nutzerschnittstelle würde die Möglichkeit bieten, manuellen Programmieraufwand zu verringern. Nach den bisherigen Erfahrungen mit Featuremodellen in der industriellen Praxis überwiegen für die Auswahl von Features einfache Abhängigkeiten der Typen Alternative, `require` und `exclude` (siehe 4.1). Lediglich für Wertebereiche von Parameter-Features sind häufig kompliziertere Abhängigkeiten anzutreffen. Für solche Fälle kann die Nutzerschnittstelle aus Grundelementen gebildet werden, wie in Abschnitt 7.2 dargestellt wird, denn die Wertebereiche von Parametern beeinflussen die graphischen Elemente nicht unmittelbar. Kompliziertere Abhängigkeiten werden dort durch eine Darstellung von Regeln veranschaulicht.

Damit die dritte Aufgabe ohne individuelle Implementierung für jedes einzelne Produkt einer Produktlinie gelöst werden kann, werden entweder Kompositionstechniken des Hyperspace-Ansatzes zur Schaffung eines Generators verwendet oder es wird mit Compilertechniken unter Anwendung des Schemas Scanner-Parser-Evaluator die XML-Repräsentation des Featuremodells vom Scanner und Parser verarbeitet; vom

Evaluator wird unter Berücksichtigung der getroffenen Featureauswahl das Resultat beispielweise als Konfigurationsdatei erzeugt.

**Abbildung einfacher Feature-Abhängigkeiten in graphischen Nutzerschnittstellen**
Für die Entwicklung einfacherer Auswahlmöglichkeiten genügt in vielen Fällen die graphische Darstellung durch übliche Bedienelemente graphischer Nutzerschnittstellen, wie Check-Button, Radio-Button und Mehrfachseiten [JavaDesign 1999]. Sie werden hier für Microsoft Windows dargestellt, existieren aber in ähnlicher Form auch für andere Systeme wie MacOS und X-Windows.

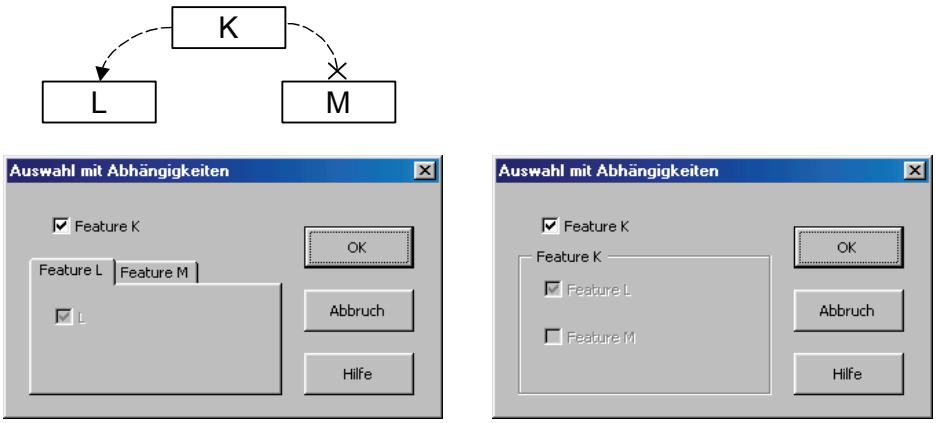



*Abb. 61: Features mit require- oder exclude-Abhängigkeiten und graphische Nutzerschnittstelle mit und ohne Mehrfachseiten*

Soll dem Nutzer ein variables Feature ohne Abhängigkeiten oder eine Gruppierung von Features mit mehr als einer Auswahlmöglichkeit angeboten werden, bietet sich die graphische Darstellung durch das Element *Check-Button* an. Ist aus einer Gruppierung von Features nur eine alternative Auswahl möglich (alternative Features), wird die graphische Darstellung durch das Element *Radio-Button* angewendet. Wird eine neue Auswahl getroffen, wird die vorherige zurückgenommen. Sind Features durch eine *require*- oder *exclude*-Abhängigkeit von der Auswahl eines anderen Features abhängig, ist die graphische Darstellung durch Kennzeichnen der Ungültigkeit der abhängigen Features (mit *Disable* und *Hide*) oder durch Kennzeichnen der Ungültigkeit in zugeordneten *Mehrfachseiten* möglich. Bei Nutzung von *Mehrfachseiten* könnten dann auch Mengen abhängiger Features dargestellt werden (Abb. 61).

Zur Verbesserung der Verständlichkeit von Abhängigkeiten sollte für den Bediener bei allen Veränderungen einer solchen Darstellung ein Hinweis angezeigt werden, wie zum Beispiel „Feature M wurde damit deaktiviert“. Besonders kritisch für die Verständlichkeit sind Fälle, in denen Features beeinflusst werden, die gerade nicht angezeigt werden. Bei mangelnder Verständlichkeit, Benutzbarkeit und Handhabbarkeit müssen andere Formen der Darstellung der Zusammenhänge und Regeln gesucht werden.

**Graphische Darstellung von Feature-Abhängigkeiten mittels Regeln**
Für eine große Menge an variablen Features oder für kompliziertere Abhängigkeiten zwischen Features würde die Verwendung der einfachen GUI-Elemente zu mangelnder Benutzbarkeit und Handhabbarkeit führen. In Anlehnung an übliche Bedienmöglichkeiten können dem Nutzer für solche Fälle Auswahlmöglichkeiten aus Listen angeboten werden. Bei einer großen Anzahl von Features ist es nach Erfahrung des Autors günstig, dem Anwender Vorschläge für geeignete Gruppen von Features anzubieten. Zur Ver-

besserung der Handhabbarkeit von Auswahlvorgängen mittels Regeln hat sich die Anzeige von Statusmeldungen über Verletzungen der Abhängigkeiten als geeignet erwiesen.

Eine solche Darstellung wurde für das Konfigurator-Werkzeug verwendet, das als Teil der Produktlinie „Siedler von Catan“ entwickelt wurde (siehe 10.2.2). Abb. 55 (S. 140) zeigt das Beispiel eines solchen Werkzeugs für die Produktlinie Digitales Videosystem; im unteren Teil wird die Verletzung einer Regel durch einen Hinweis angezeigt. Auch dieses Werkzeug arbeitet auf der Basis von Abhängigkeiten des Featuremodells, wie sie in Abschnitt 4.4 vorgestellt wurden.

# 8 Evolution durch Reengineering und Refactoring

*In diesem Kapitel wird die Methode HyperFeatuRSEB um Aspekte des Reengineering ergänzt, damit existierende Software in eine Produktlinien-Entwicklung einbezogen und dann evolutionär weiterentwickelt werden kann. Dazu wird eine Methode des Reverse Engineering vorgestellt, die auf der Nutzung von Featuremodellen beruht. Für das Refactoring und die Migration wird gezeigt, wie vorhandene Methoden an die Besonderheiten der Produktlinien-Entwicklung angepasst werden.*

Die evolutionäre Weiterentwicklung existierender Software erfordert deren Anpassung und Überarbeitung. Tätigkeiten der Überarbeitung werden als Reengineering bezeichnet; für Veränderungen der Struktur – oft ohne Änderung der Funktionalität – hat sich die Bezeichnung Refactoring etabliert (siehe 2.6). Im Zusammenhang mit der Produktlinien-Entwicklung gibt es verschiedene Aufgaben, die mit Reengineering und Refactoring gelöst werden müssen werden:

- *Etablieren einer Produktlinie aus Einzelsystemen.* Die Entscheidung zur Entwicklung einer Produktlinie fällt häufig, wenn bereits erfolgreiche Produkte als konventionell entwickelte Softwaresysteme existieren. Das Zusammenführen dieser Produkte auf der Basis einer gemeinsamen Architektur erfordert das Reengineering und Refactoring der Produkte, damit eine Produktlinie entsteht.
- *Einbindung existierender Komponenten.* Oft gibt es wiederverwendbare Komponenten, die Lösungen anbieten, die innerhalb einer Produktlinie benötigt werden, sog. Components of the Shelf COTS. Die Erweiterung der Produktlinie um solche neuen Bestandteile soll den Entwicklungsaufwand für die Produktlinie reduzieren. Die Integration solcher Komponenten erfordert Architektur-Änderungen und Refactoring. Häufig sind nicht nur die Architekturen der zu einer Produktlinie hinzukommenden Komponenten zu ändern, sondern auch die Architektur der Produktlinie selbst.
- *Einbindung existierender (Legacy-) Systeme zur Weiterentwicklung.* In vielen Unternehmen existieren Legacy-Softwaresysteme (siehe 2.6), die wichtige oder sogar geschäftskritische Aufgaben ausführen, und die weiterentwickelt werden müssen. Haben Sie mangels Wartbarkeit die Stufe Evolution des Stufenmodells (Abb. 14, S. 44) bereits überschritten, sind Analyse und Weiterentwicklung notwendig, bevor Veränderungen und Einbindung vorgenommen werden können. Für eine Weiterentwicklung in Zusammenhang mit einer Produktlinie können neben Architekturänderungen auch Zerlegungen bezüglich Features und Variabilität erforderlich sein.
- *Änderungen der Variabilität.* Sind neue variable Features erforderlich, oder sollen bisher gemeinsame Features jetzt variable Features werden, ist eine Zerlegung von Kern und Komponenten einer Produktlinie gemäß Separation of Concerns erforderlich. Die Zerlegung von Modellen und Klassen in Hyperslices wird durch das Refactoring erreicht. Solche Zerlegungen sind bei der Erweiterung einer Produktlinie während der Domänenmodellierung und -implementierung notwendig, vor allem bei der Umsetzung negativer Anforderungen (siehe 6.3.1) und bei Änderungen für späte Bindungszeitpunkte (siehe 7.1).
- *Vereinfachen der Architektur.* Die Aufrechterhaltung von Wartbarkeit und Flexibilität erfordern oft Strukturänderungen und -vereinfachungen, die ebenfalls durch Reengineering und Refactoring erreichbar sind.

- *Änderung nichtfunktionaler Anforderungen.* Die Umsetzung neuer oder geänderter nichtfunktionaler Anforderungen und Qualitätsmerkmale wie Zeitverhalten, Datendurchsatz, Robustheit und Effizienz erfordert neben funktionalen Erweiterungen Änderungen an der Architektur, die durch Refactoring erreicht werden können.

Insgesamt umfasst Reengineering drei unterschiedliche Arten von Tätigkeiten: Analyse und Verstehen der bisherigen Lösung sowie der ihr zugrundeliegenden Anforderungen (bezeichnet als Reverse Engineering), die Entwicklung einer neuen Architektur als Umsetzung bisheriger und neuer Anforderungen, sowie die Überführung von Modellen und Implementierung in die neue Architektur (bezeichnet als Migration). Bei den Anpassungen, die zur Überführung notwendig sind, handelt es sich nur selten um Änderungen der Funktionalität, sondern meist um Veränderungen der Struktur (sogenanntes Refactoring). Diese Tätigkeiten werden in den drei Abschnitten dieses Kapitels dargestellt.

Für den Fall der Legacy-Systeme bildet das Reverse Engineering zur Rekonstruktion der fehlenden Informationen und Analyse einen Schwerpunkt. Für die Zusammenführung von Einzelsystemen und die Einbindung von Komponenten sind Analyse und Reverse Engineering ebenfalls meist notwendig, bevor eine Weiterentwicklung erfolgen kann. In diesen drei Fällen gehört die Entwicklung einer neuen, gemeinsamen Architektur zu den zu lösenden Aufgaben. Anschließend müssen alle Teile durch Migration und Refactoring angepasst werden, die von der Entwicklung der Architektur betroffen sind.

Bei den letzten drei der oben genannten Aufgaben sind ebenfalls Veränderungen an der Architektur vorzunehmen. Der größte Anteil des Aufwands entfällt dabei meist auf die Anpassung der Teile, die von den Veränderungen betroffen sind. In allen sechs Fällen stellt der überwiegende Teil der Anpassungen Refactoring-Tätigkeiten dar.

Es existiert bereits eine Reihe von Techniken, die in Forschung und Praxis anerkannt und etabliert sind. Dabei handelt es sich um Methoden für die Entwicklung von Software-Architekturen (siehe Abschnitt 2.1) sowie um Techniken für Refactoring und Migration (siehe Abschnitt 2.6). Diese Techniken können auch für die Produktlinien-Entwicklung angewendet werden. Dazu werden in den Abschnitten 8.2 und 8.3 einige Erweiterungen und Ergänzungen gegenüber dem Stand der Technik vorgestellt, die wegen der Besonderheiten der Produktlinienentwicklung notwendig oder möglich sind.

Die vorhandenen Techniken zur Analyse vorhandener Software durch Reverse Engineering erfüllen nicht alle Anforderungen für die Weiterentwicklung von Produktlinien (siehe Abschnitt 2.5). Deshalb wird in Folgenden eine Methode der statischen Analyse unter Verwendung von Featuremodellen zur Strukturierung von Domänenwissen vorgestellt. Sie wird durch vorhandene dynamische Analysetechniken ergänzt.

## 8.1 Reverse Engineering einer existierenden Architektur

Reverse Engineering als Wiedergewinnen von Informationen aus vorangegangenen Entwicklungsschritten bezieht sich beispielweise auf die Ermittlung von Entwurfsstrukturen und Architekturwissen von früheren Anforderungen und Entwurfsentscheidungen. Im Fall von Legacy-Systemen existiert oft nur der Quellcode als zuverlässige Informationsquelle, weil Dokumente über Architektur und Anforderungen nicht aktualisiert wurden oder überhaupt nicht existieren. Im Fall von Komponenten oder konventionellen Softwaresystemen fehlt häufig ein Teil der benötigten Informationen. Damit eine Evolution als Teil einer Software-Produktlinie möglich wird, hat das Reverse Engineering neben der Gewinnung dieser Informationen weitere Aufgaben:

- Die Ermittlung von Traceability-Links zur Verknüpfung von Implementierung, Entwurf und Anforderungen sowie Features
- Das Erkennen bereits enthaltener, aber nicht dokumentierter Variabilität als Anhaltspunkt für eine Feature-gesteuerte Zerlegung
- Das Auffinden von Variationspunkten in früheren Entwurfsentscheidungen

Gefundene Informationen sind dabei zu sammeln und in die Modelle aufzunehmen, damit sie konsolidiert werden können. Die als Program Comprehension bezeichneten existierenden Methoden des Reverse Engineering (siehe Abschnitt 2.5) liefern wertvolle Informationen, die vom Entwickler zum Verständnis des Quelltexts einer Software benötigt werden. Abstraktere Informationen, wie sie für das Verständnis der Architektur oder der Anforderungen benötigt werden, werden von ihnen jedoch nicht ausreichend zur Verfügung gestellt. Die in Abschnitt 2.2.1 genannten Methoden, die Domänenwissen auswerten, benutzen strukturierte Beschreibungen mit natürlichsprachlichen Elementen oder Graphen mit Begriffsknoten, die einer Verarbeitung mit Werkzeugen nur eingeschränkt zugänglich sind. Es existieren Methoden, die Features zur Strukturierung von Software-Eigenschaften nutzen, diesen fehlt jedoch der Bezug zur Domänenanalyse. Alle diese Methoden weisen jedoch Merkmale auf, die hier benötigt werden. Durch eine Verknüpfung zwischen diesen Ansätzen wurde deshalb eine Methode zum Featuremodell-basierten Reverse Engineering entwickelt [Pashov Riebisch 2003]. Sie setzt gegenüber den existierenden Ansätzen veränderte Schwerpunkte, die eine Einbindung in die Methodik ALEXANDRIA ermöglichen. Durch die enge Verknüpfung zwischen den verschiedenen Modellen, insbesondere mit dem Featuremodell, ergibt sich die Möglichkeit, dieses während des Reverse Engineering zu nutzen. Die Methode ist dadurch gekennzeichnet, dass sie Featuremodelle zur Strukturierung von Informationen aus der Domänenanalyse nutzt und dass sie mittels Traceability-Links Verknüpfungen zwischen Anforderungen und Domäneninformationen, Features, Architekturelementen und Quellcode herstellt. Ermittelte Informationen werden durch ein inkrementelles Vorgehen verifiziert und in Modelle übernommen. Die Methode wurde in enger Kooperation mit industriellen Projektpartnern entwickelt und evaluiert.

Diese Methode weist vier wesentliche Schritte auf:

- Ermitteln früherer Anforderungen, die schrittweise bis zu Testfall-Szenarien ausgebaut werden
- Anwendung von Featuremodellen zum Sammeln und Strukturieren gewonnener Informationen
- Inkrementelle Hypothese-Aufstellung, Verifikation und Vervollständigung anhand von statischer und dynamischer Analyse (Synthese-Analyse-Zyklus)
- Schrittweises Zusammentragen von Architekturinformationen, Korrigieren der Anforderungsbeschreibung und ermitteln potentieller Variabilitätspunkte

Die Tätigkeiten des Reverse Engineering werden dabei als inkrementeller Prozess durchgeführt, der aus den Abschnitten Aufstellung eines Featuremodells, Statische Analyse und Verifikation sowie Dynamische Analyse besteht. Abb. 62 zeigt den Ablauf als UML-Aktivitätsdiagramm, wobei drei Informationsbereiche Anforderungen, Entwurfsmodell und Implementierung als Verantwortungsbereiche dargestellt sind. Die Aktivitäten werden in den folgenden Abschnitten kurz vorgestellt. Die Methode liefert neben den eigentlichen Reverse-Engineering-Ergebnissen wie verifizierten früheren Anforderungen und Beschreibung der gegenwärtigen Architektur des betrachteten

Systems außerdem Szenarien und Testfälle, die für derzeitig gültige Anforderungen relevant sind. Auf der Basis dieser Testfälle kann die anschließende Architekturentwicklung und Migration (siehe 8.2 und 8.3) effektiver und mit geringeren Risiken durchgeführt werden.

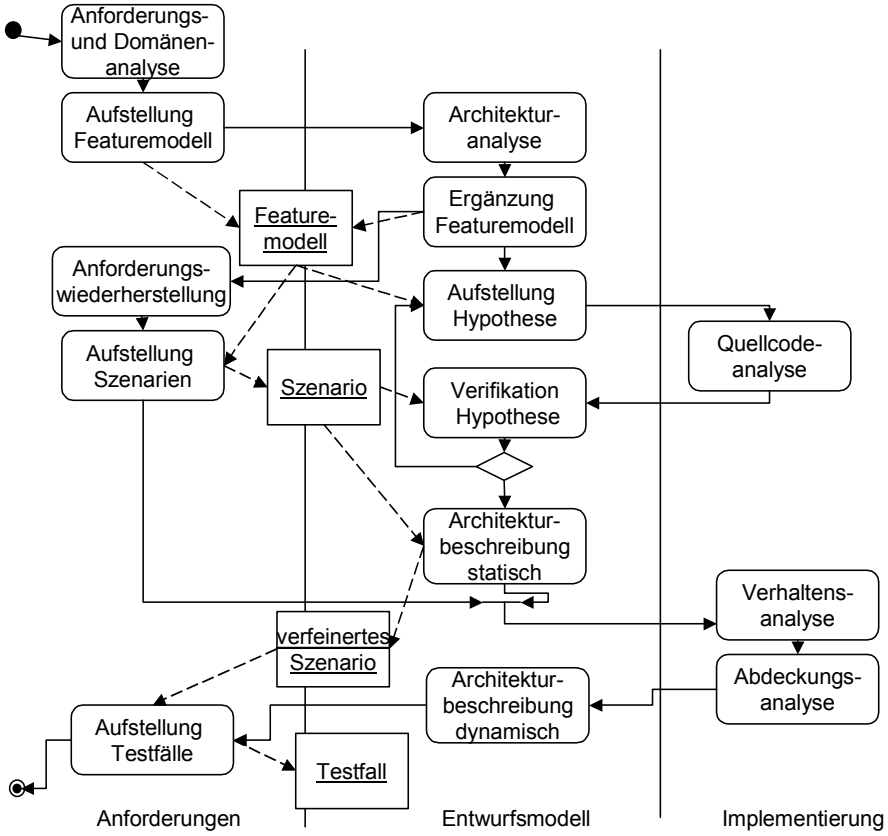



*Abb. 62: Ablauf bei der Aufstellung von Architektur, Szenarien und Testfällen*

Die Aktivitäten und Schritte werden in den folgenden Abschnitten vorgestellt. Zur Illustration werden Beispiele aus dem Digitalen Video-Projekt genutzt, hier zum Reverse Engineering der vdr-Komponente [Schmidinger 2002]. Weiterführende Informationen dazu sind in der Dokumentation in [Preiß 2003] beschrieben.

### 8.1.1 Aufstellung eines Featuremodells für existierende Systeme

Beim ersten Schritt werden die Anforderungen an das existierende System zusammengetragen und zur Aufstellung einer Anforderungsbeschreibung und eines Featuremodells benutzt. Dabei werden alle verfügbaren Informationsquellen genutzt, wie vorhandene Anforderungsdokumente, Testprotokolle, Nutzerhandbücher sowie Experten aus der Entwicklung und aus der Kundenbetreuung. Konventionelle Techniken der Anforderungsanalyse werden zum Strukturieren der Informationen verwendet (siehe 2.2.1, 2.5). Die gewonnenen Anforderungen werden zunächst verallgemeinert und zu Features eines Featuremodells zusammengefasst. Das Featuremodell vereinfacht die Navigation in den gesammelten Anforderungen und gibt ggf. erste Informationen über die Variabilität des untersuchten Systems. Außerdem ermöglicht es die Herstellung von Bezügen zwischen Domänenwissen in Form von Begriffen und ihren Erläuterungen und Anforderungen.

Als Beispiel wird hier das Featuremodell der vdr-Komponente gezeigt (Abb. 63). Als Informationsquellen standen für die Analyse Produktbeschreibungen verschiedener im Handel erhältlicher Videorecorder und Digitaler Satellitenempfänger zur Verfügung. Neben dem Featuremodell wurden die vorgefundenen Informationen zum Aufbau eines Glossars sowie für Beschreibungen von Use Cases benutzt.

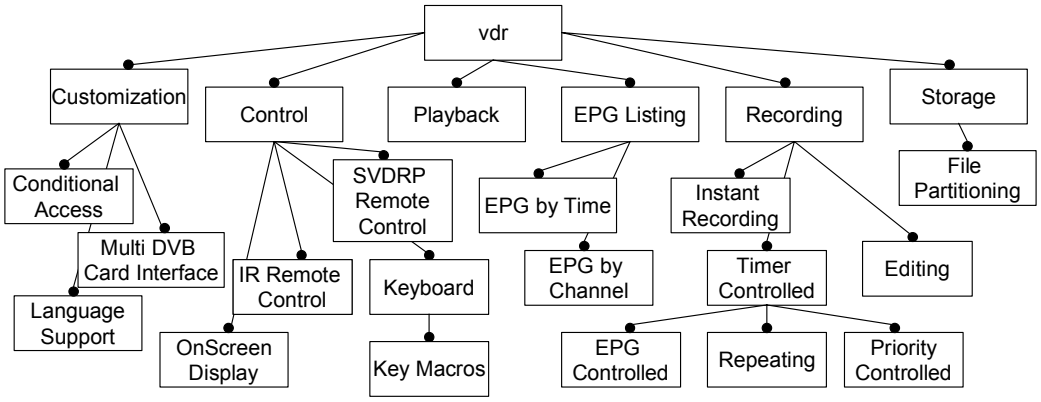



*Abb. 63: Beispiel des vdr-Featuremodells*

Anschließend werden alle verfügbaren Informationen über die existierende Architektur zusammengetragen. Dabei werden Wissen von Experten, Architekturinformationen aus Quellcode, Konfigurations- und Installationsdokumenten sowie vorhanden Entwurfsmodellen genutzt. Die gewonnenen Informationen über Entwurfsentscheidungen, Architekturelemente und deren Verhalten werden, soweit sie nichtfunktionale Produktmerkmale betreffen, ebenfalls im Featuremodell abgelegt.

Als wesentliches Architekturelement des Beispiels wird die Plug-in-Schnittstelle der vdr-Komponente vorgestellt. Die Analyse ergab, dass diese Schnittstelle die Variabilität der Komponente implementiert, indem allgemeine Funktionen eine Erweiterung durch andere Plug-ins zulassen. Die Dokumentation ergab die in Abb. 64 dargestellten Informationen.

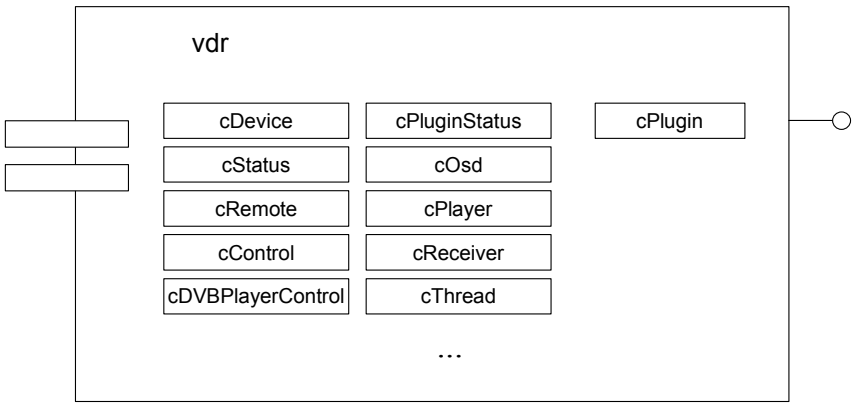



*Abb. 64: Die vdr-Komponente in UML-Symbolik mit der durch cPlugin gebildeten Schnittstelle*

Jedes identifizierte Architekturelement wird daraufhin analysiert, welchem Teil der Anforderungsbeschreibung es entspricht. Identifizierte Bezüge zu Anforderungen werden als Traceability-Links zwischen Modellelementen ähnlich den Beziehungen in Abb.

25 (S. 73) vermerkt und gespeichert. In Abhängigkeit von der Menge verfügbarer Architekturinformationen können zahlreiche Anforderungen dabei bereits dadurch verifiziert werden, dass Architekturelemente gefunden werden, die zu ihrer Realisierung gehören. Werden dabei auch Informationen zum Verhalten ermittelt, können diese zum Formulieren von Szenarien als Teil von Use-Case-Beschreibungen genutzt werden, die Details der Anforderungen beschreiben und bei der statischen und dynamischen Analyse angewendet werden. Die Szenarien dienen auch als Ausgangspunkt bei der späteren Formulierung von Testfällen.

Weitere Informationen über die Architektur werden dadurch gewonnen, dass der existierende Quelltext auf Design Patterns (siehe 2.2.3) analysiert wird. Design Patterns weisen auf Variabilität einer Architektur hin, oft geben Sie Anhaltspunkte für frühere Entwurfsentscheidungen. Zur Suche nach Design Patterns wurde ein Ansatz des Mustervergleichs entwickelt, der syntaktische Informationen aus dem Quelltext gewinnt, diese anhand von Positiv- und Negativ-Merkmalen mit den Design Patterns nach [Gamma et al. 1996] vergleicht und ebenfalls für den nächsten Schritt zur Überprüfung bereitstellt [Naumann 2001]. Dieses Verfahren weist gegenüber existierenden Verfahren eine höhere Qualität der Suchergebnisse auf und wird gegenwärtig in Zusammenarbeit mit industriellen Partnern weiterentwickelt [Philippow et al. 2003].

### 8.1.2 Statische Analyse und Verifikation

Bei den ersten beiden Schritten wurden Informationen über Anforderungen und Architektur ausgewertet, die aus Quellen außerhalb der Software stammen und deren Korrektheit im allgemeinen nicht geklärt ist. Ähnlich zu vielen weiteren Methoden wird die vorliegende Software nun durch ein iteratives Vorgehen aus Aufstellen und Verifizieren von Hypothesen untersucht (siehe 2.5). Da die Informationen des Featuremodells zum Aufstellen der Hypothesen genutzt werden, können auch unsichere, widersprüchliche oder veraltete Informationen ausgewertet werden. Dabei werden auch die gefundenen Design Patterns verifiziert.

Das Zusammenführen von Anforderung, Feature und Architekturelement als Hypothese mit Quellcode-Elementen erfolgt in Cross-Referenz-Tabellen. Hypothesen beschreiben darin vermutete Beziehungen zwischen Features und Architekturelementen, wie Komponenten, Klassen, Operationen, Interfaces, Design Patterns oder Kommunikationselementen. Tabelle 6 zeigt einen Ausschnitt des Beispiels zur Illustration; die Verwaltung der Tabellen erfolgt normalerweise werkzeuggestützt, denn eine Darstellung wie in Tabelle 6 wäre wegen der Größe der Tabellen nicht praktikabel. Jede gefüllte Zelle der Tabelle enthält einen vermuteten Zusammenhang zwischen einem Feature (Zeile) und einem Architektur- oder Quellcode-Element (Spalte). Im Beispiel wird eine Beziehung zwischen der Klasse `cSVDRPhosts` und dem Feature `SVDRP Remote Control` vermutet. Eine andere Hypothese unterstellt, dass die Klasse `cKeys` das Feature `Key Macros` unterstützt. Zur Verifikation einer solchen Hypothese wird eine Analyse des Quellcodes vorgenommen. Dabei werden konventionelle Analysetechniken eingesetzt, es werden funktionale und Datenstrukturen und der Steuerfluss ausgewertet. Wurde eine Hypothese verifiziert, wird dies in der betreffenden Zelle der Tabelle markiert (v). Auch falsifizierte Hypothesen werden vermerkt, damit gleiche Hypothesen nicht mehrfach bearbeitet werden (f). Die Identifikation von Design Patterns hilft, neben Bezügen zu Features und nichtfunktionalen Anforderungen bereits potentielle Variationspunkte für die späteren Schritte der Architekturdefinition (siehe 8.2) zu finden.

*Tabelle 6: Cross-Referenz-Tabelle des Beispiels (Ausschnitt)*

| Features | Klasse cDiseqc | Klasse cSVDRPhosts | Klasse cKeys | Datenstruktur tChannelParame-terMap |
|---|---|---|---|---|
| Multi DVB Card Interface | h | | | f |
| IR Remote Control | | | | |
| Key Macros | | | h | |
| Language Support | | | v | |
| SVDRP Remote Control | | v | | |
| EPG by Time | | | | |
| Instant Recording | | | | |
| EPG Controlled | | | | h |
| Priority Controlled | | | | |
| File Partitioning | | | | |

Die Verifikation einer Hypothese hat entweder deren Bestätigung zur Folge, oder es werden Informationen für die Aufstellung einer modifizierten oder neuen Hypothese gesammelt. Außerdem werden die Informationen schrittweise zu einer Architekturbeschreibung zusammengestellt. Auf diese Weise entsteht zum Beispiel eine Beschreibung des Protokolls SVDRP. Während des iterativen Vorgehens von Hypothesen-Aufstellung und -Verifikation wird so eine Architekturbeschreibung des existierenden Systems erstellt. Informationen über das Verhalten werden zum Verfeinern der Szenarien der Use Cases genutzt; die Anforderungsbeschreibung wird verfeinert und ggf. korrigiert. Die üblicherweise große Menge von Informationen wird durch (werkzeuggestützte) Verwaltung in Cross-Referenzen und durch Bezüge zum Featuremodell besser überschaubar und beherrschbar. Der Kern des Analyse- und Verifikationsprozesses muss zwar vom Bearbeiter durchgeführt werden, Browser-Werkzeuge unterstützen ihn jedoch bei Suche und Vergleich sowie beim Ablegen gefundener Informationen.

Als Ergebnis der statischen Analyse liegt eine weitgehend verifizierte Anforderungs- und Architekturbeschreibung vor, wobei die Verifikation der Hypothesen dazu beigetragen hat, falsche oder unvollständige Teile zu korrigieren. Für die vdr-Komponente wurde zusätzlich eine teilweise Quellcode-Dokumentation angefertigt [Preiß 2003], in die alle ermittelten Informationen aufgenommen wurden und die spätere Änderungen vereinfachen soll.

### 8.1.3 Dynamische Analyse

Bei der dynamischen Analyse werden noch fehlende Informationen über Interaktionen der Bestandteile, über die Benutzung von Schnittstellen und über das Verhalten ergänzt. Eine Entwicklung neuer Methoden zur dynamischen Analyse ist nicht Gegenstand der Methodik ALEXANDRIA, weil sich die etablierten Techniken wie die in Abschnitt 2.5 genannten Profiling- und Visualisierungsverfahren ausreichend gut eignen. Zur Analyse werden beispielweise Szenarien ausgeführt und die anhand der Instrumentierung gewonnenen Daten ausgewertet. Während dieser Untersuchung können weitere Cross-

Referenzen ermittelt und Verhaltensmodelle für einzelne Szenarien verfeinert werden. Aspekte der Nebenläufigkeit können analysiert werden. Die Verhaltensmodelle für einzelne Szenarien werden zusammengeführt und in Zustandsmodelle für Objekte, Komponenten, oder Subsysteme überführt. Die Verhaltensbeschreibung der verfeinerten Szenarien wird zu Testfällen ergänzt, die später bei Migration und Refactoring benötigt werden. Szenarien und Testfälle ergänzen Featuremodell und Anforderungsbeschreibung, die sich auf das existierende System beziehen und für die folgenden Schritte als Referenz dienen.

## 8.2 Überführung in eine Referenzarchitektur

Für die ersten drei der am Anfang des Kapitels genannten Fälle – die Integration von Legacy-Systemen, von Komponenten und von konventionell entwickelten Softwaresystemen in eine Produktlinie – ist jeweils eine gemeinsame Architektur für die Produktlinie zu entwickeln. Die zu entwickelnde Architektur definiert dabei die Schnittstellen zwischen den zusammenzuführenden Teilen. Diese Schnittstellen werden vor allem durch die funktionalen Anforderungen der Teile bestimmt.

Die Architektur muss außerdem die nichtfunktionalen Anforderungen der zusammenzuführenden Teile erfüllen. Abhängig von den konkreten Anforderungen im Einzelfall erfordert dies nicht nur Änderungen der existierenden Software, sondern auch Änderungen der Architektur der Produktlinie. Bei solchen Entscheidungen über Architekturänderungen spielen wirtschaftliche Gesichtspunkte und die Risiken einer Migration eine große Rolle. Maßnahmen zur Risiko-Verringerung wie die Bereitstellung von Testfällen (siehe 8.1 und 9.3) und ein iterativ-inkrementelles Vorgehen sind deshalb Bestandteil der Methodik.

Für die Entwicklung einer neuen Architektur sind die bewährten Methoden der Architekturentwicklung gut geeignet, die schon in Zusammenhang mit der Entwicklung der Referenzarchitektur in Abschnitt 6.1.2 angewendet wurden, wie die Methoden von Bosch [Bosch 2000], Shaw und Garlan [Shaw et al. 1996]. Die folgenden Abschnitte zeigen, welche Besonderheiten bei Anwendung dieser Methoden bei der Verknüpfung von existierender und Zielarchitektur zu einer Produktlinienarchitektur zu beachten sind.

Als Beispiel wird die Integration der bereits im vorigen Abschnitt dargestellten vdr-Komponente [Schmidinger 2002] in die Produktlinie des Digitalen Videosystems gezeigt. Dafür ist eine Architektur zu entwickeln.

### 8.2.1 Vergleich von Anforderungen und Erweiterung der Architektur

Entsprechend dem Vorgehen nach Bosch [Bosch 2000] sind zunächst die Anforderungen und Qualitätsmerkmale der zu verbindenden Teile zu vergleichen. Zuerst werden funktionale Anforderungen berücksichtigt, bevor nichtfunktionale Anforderungen betrachtet werden. Für beide Arten von Anforderungen wird der Vergleich durch die verallgemeinerte Darstellung im Featuremodell vereinfacht. Bei diesem Vergleich sind außerdem die Variabilität und die Bindungszeitpunkte zu untersuchen.

**Behandlung funktionaler Anforderungen**

Die funktionalen Anforderungen an die Produktlinie und die im Ergebnis des Reverse Engineering ermittelten Anforderungen der existierenden Software werden untersucht. Durch Zuordnung sowohl funktionaler Elemente als auch funktionaler Anforderungen zu beiden Featuremodellen und deren Vergleich kann festgestellt werden, welche gemeinsamen Anforderungen bestehen und welche Elemente zum Kern der Produktlinie

gehören müssen. Variable Features werden ebenfalls verglichen, wobei deren Bindungszeitpunkte zu beachten sind.

Für das Beispiel sind die Featuremodelle der existierenden vdr-Komponente (Abb. 63, S. 160) und der Produktlinie (Abb. 23, S. 70) zu vergleichen. Dabei wird festgestellt, dass einige Features der vdr-Komponente in der Produktlinie als variable Features gefordert werden, wie z.B. `EPG-Steuerung`. Es müssen deshalb Möglichkeiten geschaffen werden, die dazu gehörigen Elemente der Implementierung bei Bedarf zu deaktivieren oder zu entfernen. Variabilität ist bei der vdr-Komponente durch eine Plug-in-Schnittstelle vorgesehen, die wegen der verwendeten Konfigurationsfiles einen Bindungszeitpunkt zum Start des Systems ermöglicht. Die Produktlinie fordert für einige variable Features wie `Werbung Entfernen` eine Bindung zur Laufzeit, weshalb zusätzliche Veränderungen der betreffenden Komponenten erforderlich sind.

Alles, was nicht wegen gemeinsamer Features für den Kern der Produktlinie erforderlich ist, wird aus der Architektur entfernt oder ggf. in produktspezifische Erweiterungen ausgelagert. In einem solchen Fall nicht übereinstimmender Anforderungen kann anhand der Featuremodelle entschieden werden,

- überflüssige Features zu entfernen,
- teilweise überlappende Features in gemeinsame und variable Teile zu trennen und
- Widersprüche zwischen Features aufzulösen.

Können Widersprüche funktionaler Anforderungen nicht aufgelöst werden (beispielweise wegen Forderungen nach Erfüllung konkurrierender, unvereinbarer Schnittstellen), ist zu prüfen, ob die Erfüllung beider Anforderungen innerhalb einer Produktlinie sinnvoll ist. Häufig resultieren Widersprüche aus nichtfunktionalen Anforderungen. Solche Fälle treten auf, wenn ein System nach mehreren Kriterien optimiert wird, wie nach minimaler Laufzeit und minimalem Speicherplatzbedarf. Lassen sich solche Anforderungen durch Hinzunahme funktionaler Architekturelemente erfüllen, kann der Konflikt dadurch aufgelöst werden, dass die Architekturelemente zu unterschiedlichen Hyperslices zugeordnet werden. Diese Auflösung erfolgt dann in den nächsten Schritten. Sind Teile einer Produktlinie mit einer Framework- oder Komponentenarchitektur realisiert, können die Empfehlungen von [Bosch 2000] (Kap. 12) zur Konfliktlösung angewendet werden.

Bei der sich anschließenden Dekomposition und Schnittstellendefinition wird versucht, die als relevant erkannten Schnittstellen des existierenden Systems wenig zu verändern, damit der Aufwand für die anschließende Migration gering bleibt. Das ist allerdings nur dann möglich, wenn auch die nichtfunktionalen Anforderungen erfüllt werden. Auch solche Anforderungen sind deshalb anhand des Featuremodells auf Gemeinsamkeiten und Unterschiede zu untersuchen. Für nicht gemeinsame Anforderungen ist mittels Scoping zu entscheiden, ob solche Anforderungen trotzdem bei der Architekturentwicklung berücksichtigt werden sollen, damit bevorstehende Erweiterungen ermöglicht werden. Das Scoping nach Abschnitt 4.6 sieht hierfür den Fall 2 vor.

Bei der im vdr-Beispiel enthaltenen Variabilität durch die Plug-in-Schnittstelle (siehe Abb. 64, S. 160) wird zunächst geprüft, welche variablen Features sich mit diesem Architekturmechanismus implementieren lassen. Das positive Ergebnis dieser Prüfung führt dazu, dass die Architektur der Produktlinie so verändert wird, dass dieser Architekturmechanismus für die Implementierung aller variabler Features der Videobearbei-

tung übernommen wird. Diese Entscheidung wird gefällt, weil der Aufwand für die erforderlichen Änderungen an der Produktlinie als gering eingeschätzt wird gegenüber dem Nutzen durch die Einführung einer erprobten Schnittstelle und durch die unveränderte Wiederverwendbarkeit der bereits implementierten Features.

Alle Architektur-Veränderungen wie Einfügen oder Ändern von Schnittstellen, Zerlegung und Namensvergabe sind durch das Hyperspace-Engineering zu koordinieren (siehe 6.1.2), wobei die spätere Komposition berücksichtigt werden muss.

Bei der Zerlegung der zu variablen Features zugeordneten Elemente werden Variationspunkte des existierenden Systems beachtet, soweit sie während des Reverse Engineering ermittelt wurden. Benötigte Variabilität wird zur Zerlegung genutzt, nicht benötigte Variabilität wird markiert. Sie wird im folgenden Migrationsschritt durch Refactoring entfernt, damit die Komplexität der Lösung geringer wird.

Aus ökonomische Gründen ist bei der Erweiterung einer Architektur ein möglichst kleiner Aufwand der nachfolgenden Migrations- und Refactoringschritte anzustreben. Es muss versucht werden, die existierende Architektur möglichst wenig zu verändern. Während der Architekturänderungen kann die Migrationsplanung bereits vorbereitet werden.

**Behandlung nichtfunktionaler Anforderungen**
Der Vergleich nichtfunktionaler Anforderungen zwischen existierenden Teilen und Produktlinie erfolgt ebenfalls unter Nutzung des Featuremodells. Einige solcher Anforderungen sind häufig durch funktionale Bestandteile implementiert [Bosch 2000]. Ein Beispiel für einen solchen Bestandteil ist ein lokaler Datenspeicher in einem verteilten System, der zur Erfüllung der Anforderungen zum Zeitverhalten beiträgt. In solchen Fällen erfolgt ein ähnliches Vorgehen wie es oben für funktionale Anforderungen gezeigt ist.

Die Mehrheit nichtfunktionaler Anforderungen wird jedoch meist durch Anpassungen implementiert, die über ein ganzes Softwaresystem verteilt notwendig sind. Unterscheiden sich die diesbezüglichen Eigenschaften der einzubindenden existierenden Teile von den Anforderungen der Produktlinie, ist ein Refactoring der Teile notwendig, das einen sehr hohen Aufwand erfordern kann. Gehören solche Anforderungen zu variablen Features, sind beim Refactoring Hyperslices zu bilden. Führt der dafür notwendige Aufwand zur Entscheidung für eine Neuentwicklung dieser Teile, ist dies mit geringerem Risiko als bei konventionellen Methoden möglich, weil durch die im Abschnitt 8.1 vorgestellte Methode beim Reverse Engineering eine Anforderungsbeschreibung und Testfälle erstellt wurden.

Im Einzelfall kann bei hohen Anforderungen an nichtfunktionale Merkmale wie Flexibilität und Offenheit auch eine völlige Abkehr von der existierenden Architektur und der Wechsel beispielweise zu einer Object-Request-Broker-Architektur (ORB) [CORBA] notwendig werden, wenn die dabei höheren Migrationsaufwendungen durch den erreichten Nutzen gerechtfertigt werden. Andererseits können Konflikte zwischen den Anforderungen auch zu der Entscheidung führen, dass die Entwicklung eines Produkts innerhalb der Produktlinie wegen des Entwicklungsaufwands nicht sinnvoll ist, auch wenn Domänen- und Marktanalyse zu einem anderen Schluss führten.

### 8.2.2 Bewertung der Architektur

Bevor weitere Veränderungen vorgenommen werden, ist die Architektur auf die Erfüllung aller funktionalen und nichtfunktionalen Anforderungen zu prüfen. Wegen der

Bedeutung der Architektur für die Langlebigkeit und Flexibilität einer Produktlinie sind hierfür formale Reviews und Inspektionen geeignet (siehe 9.3.2), da die inneren Qualitätsmerkmale zu bewerten sind. Wartbarkeit und Klarheit sind wichtige Kriterien für Langlebigkeit. Für die Flexibilität ist besonders die Art der Zerlegung in Kern und variable Hyperslices daraufhin zu bewerten, ob die Architektur des Kerns alle gemeinsamen Features erfüllt und alle Produkte ermöglicht, sowie in welchem Umfang sie zusätzliche Elemente für variable Features enthält.

Beim oben dargestellten Beispiel sind das Zeitverhalten bei der Verarbeitung der Videoströme und die Flexibilität durch den Architekturmechanismus Plug-in-Schnittstelle wichtige Untersuchungsziele der Inspektion. Als Ergebnis der Inspektion entsteht eine Mängelliste, die weitere notwendige Refactoring- und Migrationsaufgaben enthält. Sie ist beim folgenden Schritt der Planung der Migration ebenfalls verwendbar.

## 8.3 Migration und Integration von Objektmodell und Quellcode

Beim vorangegangenen Schritt wurde eine Architektur entwickelt, die aus den Anforderungen an zwei Softwareteile abgeleitet wurde. Diese Softwareteile sind nun während der Migration so zu verändern, dass sie der Architektur entsprechen und integriert werden können. Im Beispiel sind u.a. Komponenten zur Video-Bearbeitung von Werbeblöcken und zur EPG-Verarbeitung von der Migration entsprechend der Plug-in-Schnittstelle betroffen.

Für die letzten drei der am Anfang des Kapitels genannten Fälle – die Änderung der Variabilität, die Vereinfachung der Architektur und die Veränderung nichtfunktionaler Anforderungen – wurde keine neue Architektur entwickelt, sie wurde lediglich verändert. Auch danach sind die von der Änderung betroffenen Teile der Produktlinie zu migrieren, dass sie der Architektur entsprechen.

Die Architektur beschreibt insbesondere die Interaktionen der Elemente bezüglich ihrer Schnittstellen, Protokolle und Abhängigkeiten, wie im oben angeführten Beispiel der Plug-in-Schnittstelle. Die einzelnen betroffenen Elemente (beispielweise Komponenten und Hyperslices) sind so zu verändern, dass sie die geforderten Schnittstellen aufweisen. Die Migration erfolgt für Objektmodelle und Implementierung.

Eingangsdaten für die Migration sind Architekturbeschreibung, Komponentendefinition und Schnittstellenbeschreibung (aus 8.2), Testfälle und Szenarien (in 8.1 entwickelt) sowie die vorher existierenden Elemente mit ihren Modellen und ihrem Quellcode. Als Ergebnis werden Komponenten geschaffen, die entsprechend der Architektur zusammenwirken, die nach Vorgabe des Hyperspace in Hyperslices zerlegt wurden und die jeweiligen funktionalen und nichtfunktionalen Anforderungen erfüllen.

Die Schritte bei der Migration der Elemente und der jeweiligen Modelle und Implementierungen folgen im Wesentlichen den nach dem Stand der Technik üblichen Vorgehensweisen (siehe 2.6). Dazu gehören die Reengineering-Methoden des SEI [Reeng-SEI], die von Bosch beschriebenen Migrationstechniken [Bosch 2000] sowie die Refactoring-Techniken [Fowler 1999]. Wegen der Besonderheiten der Methodik ALEXANDRIA sind zusätzliche Tätigkeiten im Zusammenhang mit der Nutzung von Featuremodellen, der Zerlegung in Hyperslices sowie der Komposition erforderlich. In Abb. 65 werden die Schritte im Überblick dargestellt, sie werden in den folgenden Abschnitten kurz erläutert.

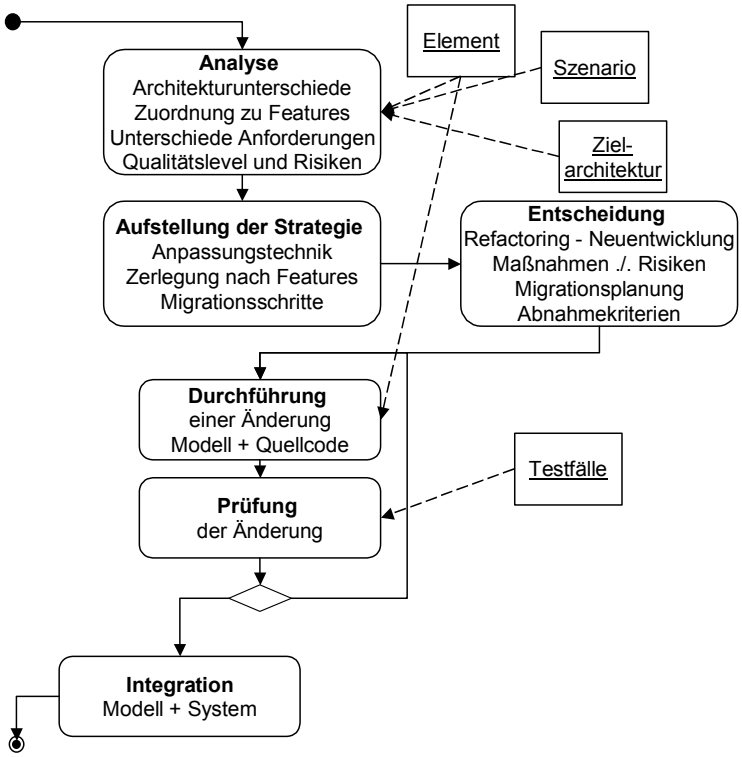



*Abb. 65: Schrittfolge bei der Migration als Aktivitätsdiagramm*

### 8.3.1 Analyse der zu migrierenden Elemente

Als erster Schritt der Migration erfolgt eine *Analyse.* Für die einzelnen Elemente sind die Unterschiede zur Ziel-Architektur, Unterschiede der nichtfunktionalen Eigenschaften und Qualitätsmerkmale sowie Risiken für ein Refactoring festzustellen. Aus diesen Informationen werden Aussagen über die Risiken und den Aufwand der Migration abgeleitet, die als Basis für die folgenden Entscheidungen über eine Migrationsstrategie verwendet werden.

Im Beispiel der Einbindung der vdr-Komponente bezieht sich der Vergleich auf die Variationspunkte und die Steuerung der Variabilität zwischen der Zielarchitektur Plug-in-Schnittstelle und den vorhandenen Komponenten, z.B. EPGmanagement. Die dabei festgestellten Unterschiede betreffen vor allem die Prinzipien des Aufrufs und der Parameterübergabe. Die Plug-in-Schnittstelle definiert Funktionsaufrufe, Referenzen auf Objekte werden nur in wenigen Fällen übergeben. Die bei der Migration anzupassende Komponente EPGmanagement verwaltet jedoch EPG-Informationen durch Referenzen auf Objekte. Diese Unterschiede legt eine Zerlegung von EPGmanagement oder vdr in mehrere Komponenten sowie ein teilweises Wrapping nahe.

Gegenüber dem Vorgehen nach [Reeng-SEI] haben die Ziele Langlebigkeit und Flexibilität für eine Produktlinie besondere Bedeutung. Deshalb sind diejenigen Qualitätsmerkmale der Elemente besonders wichtig, die Indikatoren für das Erreichen dieser Ziele darstellen, wie Konformität mit Codierstandards, Wartbarkeit und Grad der Dokumentation. Da es sich hier um innere Qualitätsmerkmale handelt, sind Reviews und Inspektionen (siehe 9.3.2) zur Ermittlung und Bewertung geeignet. Diese Bewertung verfolgt die gleichen Ziele wie die Bewertung der Architektur (siehe 8.2.2), hat aber statt den Schnittstellen und der Interaktion von Komponenten deren Rumpf zum Gegenstand.

Für die als Beispiel angeführten Komponenten vdr und EPGmanagement wird eine solche Inspektion durchgeführt. Dabei werden neben den Komponenten selbst die beim Reengineering der vdr-Komponente gewonnenen Informationen in Form von Architekturbeschreibung und Quellcode-Dokumentation verwendet (siehe Beispiel aus 8.1). Die Inspektion ergibt eine Liste mit Unterschieden bezüglich des Programmierstils und mit Hinweisen auf potentielle Risiken eines Refactorings, wie z.B. eine enge Kopplung zwischen Klassen `cPlugin` und `cPluginManager` und dem DLL-Konzept (Klassen `cDll` und `cDlls`). Diese Angaben fließen in die Aufwandsbewertung ein, die bei Erstellung der Migrationsstrategie vorgenommen wird.

Für eine vereinfachte Bewertung, beispielweise zum Gewinnen eines Überblicks bei sehr vielen Komponenten, ist die Nutzung einfacher quantitativer Kenngrößen (sog. Metriken) sinnvoll, wie sie in ausreichender Zahl zur Verfügung stehen (siehe 9.3.1). Beispiele für solche Kenngrößen sind die Menge der Verweise (bei objektorientierten Programmiersprachen), Menge der Sprünge (bei prozeduralen Programmiersprachen) jeweils bezogen auf den Umfang des betrachteten Quellcode-Abschnitts. Eine derartige Bewertung mittels Metrik-Werkzeugen wurde beispielsweise in dem in Abschnitt 10.2.3 genannten Projekt durchgeführt.

Reviews und Inspektionen liefern genauere Aussagen über Klarheit und Wartbarkeit des Quellcodes, erfordern aber einen deutlich höheren Aufwand als die Bewertung mittels Kenngrößen.

### 8.3.2 Aufstellung der Migrationsstrategie

Bei der Aufstellung der *Strategie* für die Migration steht die Wahl von Technik und Mechanismen sowie eine Zerlegung des Refactoring in eine Folge kleiner Schritte mit kurzen Iterationen im Mittelpunkt. Dabei ist zusätzlich die Zerlegung in Hyperslices und Komponenten vorzusehen, wobei die Forderungen des Featuremodells nach Variabilität und die Eigenschaften der späteren Komposition - insbesondere der Integrationsstrategie – zu beachten sind.

Im Beispiel wird die Zerlegung von EPGmanagement in separate EPG-Komponenten vorgesehen, so dass je variables Feature eine Plug-in-Komponente entsteht (Abb. 66). Eine Komponente EPGstorage wird dem Feature EPG-Verwaltung (siehe Featuremodell aus Abb. 23, S. 70) zugeordnet, die Komponente EPGcontrol dem Feature EPG-Steuerung. Eine zusätzliche Zerlegung in Hyperslices ist dadurch nicht notwendig. Traceability-Links <<ImplementedBy>> verweisen von den Features auf die resultierenden Komponenten in Entwurf beziehungsweise Implementierung.

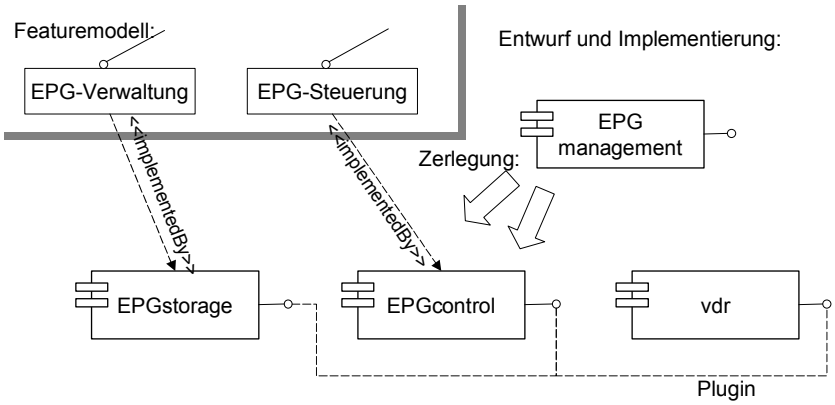



*Abb. 66: Zerlegung der Komponente EPGmanagement entsprechend der Plug-in-Schnittstelle und der variablen Features*

Bei einer Zerlegung sind hierfür Erfahrungen wie die Best Practices des SEI [Reeng-SEI] verwendbar. Die Vorgehensweise ist jedoch um Schritte zum Separation of Concerns zu erweitern. Die Koordination durch das Hyperspace-Engineering (siehe 6.1.2) dient der Einbindung in die Produktlinie. Dabei liefert das Featuremodell wichtige Informationen für Zerlegung, Variabilität und Zuordnung zu Anforderungen. Im Unterschied zum Hufeisen-Verfahren [Reeng-SEI] ist wegen der Nutzung des Featuremodells beim Reverse Engineering ein Top-Down-Vorgehen möglich, was zu Vorteilen bei großen Systemen mit vielen Elementen führt.

### 8.3.3 Entscheidung über Refactoring oder Neuentwicklung

Die *Entscheidung*, ob und wie Refactoring durchgeführt wird, ist unter wirtschaftlichen und Entwicklungs-strategischen Gesichtspunkten zu treffen. Es ist abzuwägen, ob Risiko, Aufwand und Nutzen in einem angemessenen Verhältnis zueinander stehen. Der Aufwand wird neben dem eigentlichen Arbeitsaufwand auch vom Bedarf an kompetenten Entwicklern und von den Risiken durch unvorhergesehene Probleme bestimmt. Der erwartete Nutzen besteht in geringerem Programmieraufwand sowie im Gewinn an Wartbarkeit und Robustheit gegenüber einer Neuentwicklung und in der Vermeidung der Risiken einer Neuentwicklung. Die vorher aufgestellten Testfälle tragen ebenfalls dazu bei, durch frühzeitige und umfassende Tests die Risiken des Refactoring zu verringern.

Im oben dargestellten Beispiel werden die Risiken der einzelnen Aktivitäten in Form einer Liste zusammengetragen. Eine quantifizierte Bewertung erfolgt jeweils mittels geschätztem Minimal- und Maximalaufwand, weil diese Daten mit geringem Aufwand und ausreichender Genauigkeit durch Entwickler zu ermitteln sind. In die Schätzung fließen die durch Metriken gewonnenen Daten, die für eine Aktivität notwendigen Aufgaben entsprechend der Migrationsstrategie sowie Erfahrungswerte ein. Tabelle 7 zeigt einen Ausschnitt aus einer solchen Liste. Je größer das Risiko einer Migrationsaktivität, desto größer ist die Differenz zwischen Minimal- und Maximalaufwand.

Bei der Festlegung der Reihenfolge der Durchführung erfolgt eine Aufteilung in kleine Schritte. Dabei werden die kritischsten Fälle sowie diejenigen, bei denen die Änderungen am dringendsten erforderlich sind, zuerst untersucht und bearbeitet. Für die Festlegung einer Migrationsplanung sind für Einzelfälle noch genauere Analysen durchzuführen, beispielsweise durch prototypische Realisierungen und Tests. Bei Elementen mit mangelhafter Stabilität ist häufig kein effektives Refactoring mehr möglich. In solchen

Fällen erfolgt eine Neuentwicklung, deren Risiken durch Vorhandensein einer Schnittstellenbeschreibung und der Testfälle der betreffenden Elemente aus dem Reverse Engineering geringer sind als bei einem konventionellen Vorgehen.

*Tabelle 7: Risikobewertung der Migrationsaktivitäten des Beispiels (Ausschnitt)*

| Risiko / Aktivität | Paket / Klasse / Operation | minAuf-wand | maxAuf-wand |
|---|---|---|---|
| Wrapping der EPG-Objekte | EPGstorage | 0,5 | 1,5 |
| Aufteilung der Zugriffsoperationen | EPGstorage, EPGcontrol | 1 | 3 |
| Umstellung Aufrufreihenfolge | cPlugin, EPGcontrol | 1 | 10 |

### 8.3.4 Anpassung oder Refactoring der Elemente

Die eigentliche Durchführung der Veränderungen erfolgt je nach Entscheidung für White-Box- oder Black-Box-Verfahren unter Nutzung der verschiedenen existierenden Techniken (siehe 2.4.4).

**White-Box-Verfahren: Refactoring**

Die White-Box-Verfahren zielen auf eine Anpassung durch Veränderung der internen Struktur. Der Begriff des Refactoring hat hier besondere Bedeutung erlangt, er bezieht sich auf „Neu-Schneiden“ des Quellcode ohne (wesentliche) Änderung der Funktion. Für Refactoring liegen bewährte Vorgehensbeschreibungen für kleine Einzelschritte vor, die mit geringen Risiken angewendet werden können [Fowler 1999][Kerievsky 2002][Roberts 1999]. Zunehmend werden sie durch Werkzeuge unterstützt. Zusätzlich zu den Schritten nach diesen Methoden müssen beim Refactoring die Traceability-Links angepasst werden, damit deren Konsistenz erhalten bleibt. Diese Aufgabe wird von den Entwicklern ausgeführt.

Das Refactoring in einer Folge kleiner Schritte ermöglicht nicht nur den Erhalt des Überblicks für den Entwickler und die Vermeidung von Fehlern, es ermöglicht gleichzeitig eine schrittweise Prüfung der Architektur. Diese Prüfung zeigt, ob die objektorientierte Zerlegung klar und verständlich ist und ob die Zerlegung gemäß Separation of Concerns den Features entsprechend durchgeführt und erfolgreich bezüglich größtmöglicher Unabhängigkeit der Hyperslices ist. Durch eine Nutzung der Testfälle kann unmittelbar nach den Veränderungen geprüft werden, ob Fehler aufgetreten sind. Testläufe können auch zur Prüfung von nichtfunktionalen Anforderungen wie der Effizienz genutzt werden, die als Anlass für sofortige Verbesserungen dienen.

**Black-Box-Verfahren: Wrapping**

Das wichtigste Black-Box-Verfahren ist das Wrapping, bei dem die gesamte Interaktion einer Komponente mit der Umgebung durch einen sogenannten Wrapper geleitet wird, damit sowohl die Funktionalität als auch das Verhalten beeinflusst werden kann. Die Nutzung des Wrapping für eine Legacy-Komponente, die zu mehreren Features gehört, führt zur Zuordnung zu mehreren Hyperslices, wenn die betreffende Legacy-Komponente nicht zerlegt wird. Dies verringert die Schlankheit der daraus erstellten Produkte. Damit die Wartbarkeit und Klarheit der Lösung nicht beeinträchtigt werden,

sind die Bezüge zwischen den beteiligten Hyperslices durch Traceability-Links zu dokumentieren.

**Mapping auf Komponentenstandard**
Für die Einbindung von Fremd-Komponenten (COTS), die häufig als binäre oder anderweitig unveränderliche Komponenten vorliegen, kann auch die Entscheidung für einen existierenden Komponentenstandard notwendig und sinnvoll sein. In diesem Fall muss die Architektur die notwendigen Charakteristika des betreffenden Komponentenstandards aufweisen. Für COM+-Komponenten muss beispielsweise die `IUnknown`-Schnittstelle zur Prüfung des Interfaces genutzt werden, bei Enterprise Java Beans EJB sind Home- und Remote-Interface über Methodenaufrufe zu nutzen, im Fall der Einbindung in CORBA-Architekturen ist beispielweise eine Beschreibung mittels Interface-Definition-Language IDL zu definieren und es sind Basisdienste wie die der Registrierung beim Broker zu bedienen.

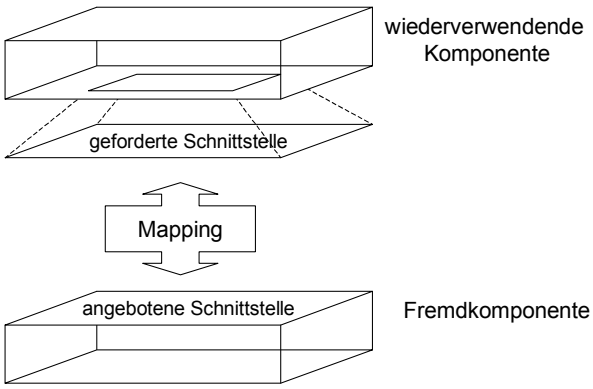



*Abb. 67: Anpassung von Fremd-Komponenten durch Mapping*

Falls durch Architektur oder Komponenten geforderte Schnittstellen nicht mit den von Fremdkomponenten angebotenen Schnittstellen übereinstimmen, muss die Abstimmung durch ein Mapping herbeigeführt werden. Beispielsweise kann die Wrapping-Technik zum Ergänzen semantisch oder syntaktisch differierender Teile der Schnittstelle genutzt werden. Diese Technik ist außer bei Fremd-Komponenten auch für Komponenten aus anderen Produkten oder aus früheren Entwicklungen anwendbar, sowie auch für Legacy-Systeme, die durch Wrapping als Komponente behandelbar sind.

Als Beispiel wird für die Video-Produktlinie ein Wrapper gezeigt, der eine Komponente zum Entfernen eines Logos über die Plug-in-Schnittstelle einbindet. Abb. 68 zeigt den Wrapper `cLogoWasherWrapper` als Klasse, der die gesamte Kommunikation zwischen dem durch `cPlugin` gebildeten Interface und der Komponente `cLogoWasher` abwickelt. Zwischen Interface und Komponente übereinstimmende Operationen könnten im einfachsten Fall direkt weitergeleitet werden. Alle Operationen, die nicht von der Komponente bedient werden, müssen vom Wrapper behandelt werden. Dazu gehört in diesem Fall auch die Anmeldung der Komponente, die Bereitstellung der Menüinhalte (hier `MainMenuEntry()`, `MainMenuAction()`) und die Speicherung der vom Bediener gewählten Einstellungen, die der Komponente als Parameter übergeben werden (hier `SetupMenu()`, `SetupParse()`, `SetupStore()`, `ConfigDirectory()`).

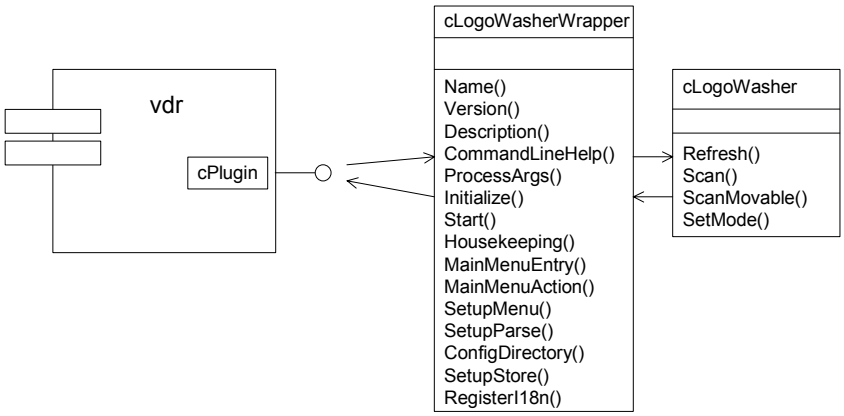



*Abb. 68: Beispiel eines Wrappers für die Integration einer Komponente über die Plug-in-Schnittstelle*

### 8.3.5 Integration

Die Integration als letzter Schritt des Reengineering umfasst gegenüber konventionellen Aktivitäten zusätzlich die Aktualisierung der in Repository und Modellen abgelegten Abhängigkeiten. Dazu gehören unter anderem Traceability-Links und Informationen über zulässige Konfigurationen, die gemäss Lokalitätsprinzip in jedem Artefakt zu beschreiben sind. Die Aktualisierung der betroffene Referenzen im Repository können von einem Konfigurationsmanagement-Werkzeug übernommen werden, damit dieser Teil der Konsistenzsicherung automatisiert wird.

Falls produktspezifische Hyperslices als Erweiterungen der Produktlinie (siehe 6.3.1) von Architekturänderungen betroffen sind, sind diese ebenfalls zu migrieren, so dass die letzte Version dieser Hyperslices mit dem geänderten Objektmodell der Produktlinie übereinstimmt.

Bei der Integration zeigt sich ein wichtiger Vorteil der Methode HyperFeatuRSEB gegenüber framework- oder komponentenorientierten Ansätzen ohne generative Techniken wie der aus [Atkinson et al. 2002]: die äußerst aufwendige und riskante Migration und Reintegration der letzten Version der Produkte ist nicht notwendig, denn diese werden für die nächste Version neu generiert. Die Migration der betroffenen Komponenten in Form von Hyperslices reicht für eine Migration aus, der Aufwand ist hier nicht doppelt zu leisten (vergleiche [Atkinson et al. 2002] Kap. 17). Der zusätzliche Testaufwand kann gering gehalten werden, wenn Testfälle zur automatisierten Prüfung vorliegen oder generiert werden können.

## 8.4 Werkzeuge

Die Aufgaben von Werkzeugen für die Bearbeitung existierender Software liegen im wesentlichen in den beiden Bereichen

- der Unterstützung des Bearbeiters beim Erfassen und Verwalten der Menge an Informationen sowie
- der Übernahme von Abfolgen formaler Schritte für Änderungen und Tests zur Vermeidung neuer Fehler.

Die *Menge an Informationen* über existierende Softwaresysteme üblichen Umfangs überschreitet die vom menschlichen Kurzzeitgedächtnis verwaltbare Menge deutlich. Es handelt sich bei den Informationen um große Mengen von Fakten, Alternativen, Hypothesen und Beziehungen. Besondere Schwierigkeiten entstehen durch große Unter-

schiede im Abstraktionsgrad der Informationen, beispielweise zwischen solchen aus Anforderungen und aus Quellcode. Anforderungen an Werkzeuge zur Speicherung, Verwaltung und Darstellung von Informationen bestehen deshalb in

- Speicherung und Auflistung von Zwischenergebnissen mit teilweise geringem Grad von Zuverlässigkeit (wie Entscheidungsalternativen, Hypothesen, Vorstellungen und Ideen),
- Speicherung von erkannten Verknüpfungen und Beziehungen zwischen Elementen, sowie Navigation anhand solcher Beziehungen,
- Analyse von Abhängigkeiten zwischen verschiedenen Teilen von Dokumenten, Modellen und Quellcode (Aufruf einer Schnittstelle, Zugriff auf Variable, Bezug zu Anforderungsdokumenten) und
- Verwaltung von Verknüpfungen zwischen Informationen mit unterschiedlichem Abstraktionsgrad, von der Anforderungsbeschreibung bis zum Quellcode

Der zweite genannte Bereich der *Übernahme von Abfolgen von Schritten* dient der Unterstützung bei Veränderungen, wie sie außer bei Migration und Refactoring auch bei der Zerlegung und inkrementellen Überarbeitung notwendig sind. Hierbei können Folgen einfacher Tätigkeiten zusammengefasst und durch Automatisierung wiederholt ausgeführt werden, wie

- Umbenennen,
- Teile zwischen Strukturelementen verschieben,
- einfache Strukturänderungen,
- Auflösung oder Erstellung von Bezügen,
- Aktualisierung aller Referenzen bei Veränderungen, einschließlich der Traceability Links und
- Automatisierter Test zum Nachweis des Erhalts der Funktionalität.

Derartige Operationen können durch Werkzeuge dadurch automatisiert werden, dass beispielweise Regeln abgearbeitet werden, die einen Zugriff auf Bezeichnung und Semantik über die Einbeziehung syntaktischer Information erlauben. So ist eine Unterscheidung zwischen einer Operation `next()` und einer Variablen `next` sowie zwischen gleichnamigen Klassen in verschiedenen Paketen oder Hyperslices erforderlich. Dazu ist die Bereitstellung von Informationen über Modelle, Anforderungen, Variabilität und Implementierung in integrierter Form notwendig, damit Abhängigkeiten dargestellt werden können. Die Formulierung von Regeln ist mit Vor- und Nachbedingungen mit OCL ähnlich den Kompositionsregeln von Hyper/UML möglich (siehe 5.4 und [Böllert 2002]). Neben der Vermeidung von Risiken durch unvollständige Veränderungen und Folgefehler trägt eine derartige Automatisierung auch zur Steigerung der Produktivität bei.

Weitere wichtige Aspekte für die Steigerung der Produktivität durch Werkzeugunterstützung sind

- Integration mit anderen Werkzeugen,
- Schnellere Abarbeitung,
- Möglichkeit des Rückgängig-Machens und

- Graphische Darstellungen und Navigationsunterstützung.

Erste Werkzeuge zur Unterstützung solcher Aufgaben waren integrierte Entwicklungsumgebungen wie für Smalltalk von ParcPlace, die auch von neueren Werkzeugen für andere Sprachen nachempfunden wurden, wie unter anderem das Werkzeug IBM VisualAge. Sie verwalten den Code in einem Repository und ermöglichen dadurch beispielweise den direkten Zugriff auf alle Aufrufer einer Operation oder alle Zugriffe auf eine Instanzvariable. Einige Namensänderungen sind durch Aktivieren eines einzelnen Menüpunkts durchführbar. Manche Änderungen werden dadurch sehr vereinfacht.

Inzwischen sind einige Werkzeuge am Markt verfügbar, die Teile der oben geforderten Unterstützung leisten. Dazu gehören Refactoring-Browser aus dem Extreme-Programming-Umfeld [Fowler 1999], obgleich diese Werkzeuge vor allem auf der Ebene des Quellcodes operieren. Einige Werkzeuge erlauben auch die Bearbeitung von Modellen, wie ein Refactoring Browser für UML [Boger et al. 2002] mit Integration in ein CASE-Werkzeug, wobei die Plug-in-Möglichkeiten des ArgoUML-Nachfolgers Poseidon genutzt werden.

# 9 Einbindung in Prozessgestaltung, Projekt- und Qualitätsmanagement

*Methodische Produktlinien-Entwicklung muss auch die organisatorische Gestaltung der Entwicklungsprozesse, das Projekt- und das Qualitätsmanagement unterstützen. Die Entwicklung von Beiträgen zu diesen Aspekten war nicht Schwerpunkt dieser Arbeit. Ausgehend von den Erfahrungen des Autors wird auf Besonderheiten beim Einsatz existierender Methoden bei der Entwicklung von Produktlinien eingegangen.*

Dieses Kapitel befasst sich mit nicht-technischen Aspekten der Entwicklung von Produktlinien, die für einen erfolgreichen Einsatz der ALEXANDRIA-Methodik in der Industrie unabdingbar sind. In Ergänzung zu den softwaretechnischen Methoden in den übrigen Teilen dieser Arbeit werden hier Methoden, Regelungen und Techniken der Gestaltung des Entwicklungsprozesses, des Projektmanagements sowie des Qualitätsmanagements betrachtet. Meist stehen dafür bewährte Elemente zur Verfügung, die einzeln bereits in der Vergangenheit zum Erfolg verschiedener Projekte und Unternehmen beigetragen haben. Für die erfolgreiche Anwendung dieser Elemente bei einer Produktlinienentwicklung mit generativen Techniken müssen Anpassungen daran vorgenommen und veränderte Schwerpunkte gesetzt werden. Dies erfolgt hier aufgrund der in Forschung und Industrie gesammelten Erfahrungen des Autors.

Eine umfassende Behandlung der Methoden kann eine wissenschaftlichen Arbeit auf dem Gebiet der Informatik aus zwei Gründen nicht leiste: Einerseits werden diese Methoden und Regelungen sehr stark vom Charakter des jeweiligen Unternehmens und seiner Produkte geprägt und müssen auf diese abgestimmt werden, so dass hier nur verallgemeinerte Aussagen möglich wären. Andererseits existieren sowohl in der Literatur als auch in den Unternehmen umfangreiche Darstellungen zu den Grundlagen, Methoden und Techniken für die Entwicklung individueller und Standard-Software, auf die weitgehend zurückgegriffen werden kann. Auf diese wird im Folgenden verwiesen. Sind Unterscheidungen und Veränderungen vorzunehmen, werden sie durch Gegenüberstellung von Regelungen für Produktlinienentwicklung zu Regelungen für "konventionelle" Softwareentwicklung ohne Produktlinien kenntlich gemacht.

Die Organisation von Entwicklungsprozessen betrifft die Arbeitsteilung und die Verantwortlichkeit der Mitarbeiter. Sie hat damit großen Einfluss auf die Kommunikation, die Motivation der Entwickler und auf deren Einstellung zur Qualität und zum langfristigen Erfolg. Zum organisatorischen Umfeld der Softwareentwicklung gehören Regelungen und Praktiken im Entwicklungsprozess sowie im Projekt- und Qualitätsmanagement. Solche Regelungen werden in Unternehmen häufig in Form eines Vorgehensmodells zusammengefasst. Wirksamkeit und Erfolg der Methoden von ALEXANDRIA sind in einem industriellen Umfeld nur erreichbar, wenn die Methoden von diesen organisatorischen Regelungen unterstützt werden. Dazu werden vor allem Prinzipien und Kriterien dargestellt, die wegen der Besonderheiten der Produktlinienentwicklung für deren Erfolg wichtig sind und die Beiträge zum Erreichen der Ziele entsprechend Kapitel 1 leisten.

## 9.1 Gestaltung des Entwicklungsprozesses

Der Software-Entwicklungsprozess wiederspiegelt die wirtschaftlichen Ziele. Bei der konventionellen Entwicklung stehen die Produktivität, die Termin- und Budgeteinhaltung, die Kundenzufriedenheit sowie die funktionalen und nicht-funktionalen Produktmerkmale im Vordergrund. Die in dieser Arbeit verfolgten Ziele der Flexibilität, der

Langlebigkeit und der kurzen Time-to-Market stellen dazu keinen Widerspruch dar, sondern verlagern die Wichtung. Für die Langlebigkeit haben das Steuern von Veränderungen sowie die Erhaltung der Wartbarkeit und der Konsistenz besondere Bedeutung. Flexibilität und kurze Time-to-Market erfordern eine zielgerichtete und koordinierte Zerlegung, einen geringen Aufwand bei der Komposition und eine weitgehende Automatisierung von Teilschritten.

Der Produktlinien-Entwicklungsprozess besteht aus mehreren Teilprozessen, die sich durch ihre Größenordnung und ihren Zeithorizont unterscheiden. Sie sind alle durch ein iterativ-inkrementelles Vorgehen gekennzeichnet und ordnen sich in die Darstellung in den Kapiteln 3 und 1 ein. Fünf Teilprozesse können unterschieden werden:

1. Entwicklung einer Version einer Komponente
2. Entwicklung eines Features und dazugehöriger Komponenten
3. Entwicklung der Architektur der Produktlinie
4. Entwicklung eines Produkts von den Anforderungen bis zur Auslieferung
5. Entwicklung der Produktlinie

Während der erste Teilprozess wenige Unterschiede zu konventionellen Softwareentwicklungsprozessen aufweist, sind – bedingt durch die höhere Komplexität gegenüber einer konventionellen Entwicklung – alle weiteren Teilprozesse durch sehr viel höhere Anforderungen an Kooperation und Kommunikation gekennzeichnet. Dies ist durch Vorkehrungen im Vorgehensmodell zu berücksichtigen. Von besonders großer Bedeutung sind Verknüpfungen von Entwurfs- und Codierungstätigkeiten in Produktlinien- und Produkt-Engineering mit:

- *Anforderungsanalyse*-Tätigkeiten, wie zum Beispiel iterativer Präzisierung von Anforderungs- und Featuremodellen, wenn im Zuge des Entwurfs weitere Informationen benötigt werden oder wenn bei Separation of Concerns von Objektmodell-Hyperslices Interaktionen zwischen Features aufzulösen sind (siehe auch Abschnitt 4.1).
- *Konstruktiven Qualitätssicherungsmaßnahmen*, wie der Definition von Richtlinien für Architekturbeschreibung und Separation of Concerns zum Erreichen großer Klarheit und Wartbarkeit, wie sie in Abschnitt 9.3.3 untersucht werden.
- *Analytischen Qualitätssicherungsmaßnahmen*, wie zum Beispiel bei der Prüfung der Vollständigkeit, Korrektheit und Verständlichkeit einer Architekturbeschreibung oder eines Objektmodell-Hyperslices bezüglich Klarheit und Wartbarkeit, wie sie in Abschnitt 9.3.2 untersucht werden.
- *Konfigurationsmanagement*, beispielweise mit Versionsmanagement von Objektmodell-Hyperslices mit Aufrechterhaltung von Beziehungen zum Featuremodell, Anforderungsmodell, zur Implementierung und zur Dokumentation (siehe 9.2.5).
- *Abnahme* mit Feedback der Konsumenten an die jeweiligen Entwickler. Als besonders kritisch erweist sich nach bisherigen Erfahrungen die Abnahme wieder zu verwendender Teile durch andere Softwareentwickler, bei der besonders strenge Prüfungen bezüglich aller Qualitätsmerkmale üblich sind. Im Vorgehensmodell sind hierfür Kriterien und formelle Abnahmen zu definieren, sowie formelle Kommunikationspfade und Supportregelungen für Problemfälle. Bei unzureichenden Regelungen besteht vor allem bei Projektgrößen von mehr als

10 Mitarbeitern die Gefahr von Kommunikationsdefiziten, die sich in Schuldzuweisungen und Misstrauen bei auftretenden Mängeln äußern und durch fehlende Problemlöse-Fähigkeiten den langfristigen Erfolg der Produktlinienentwicklung gefährden. Auch das *Projektmanagement* muss durch Führung und Motivation mitwirken, diesen Gefahren zu begegnen.

Die Verknüpfung zwischen den Entwicklungsprozessen des Produktlinien- und Produkt-Engineering (siehe 6.1 und 6.2) stellt eine prinzipiell neue Aufgabe dar. Insbesondere die *Koordination* zwischen beiden Prozessen kann zwar durch Werkzeuge unterstützt werden, damit aus der Menge der Informationen resultierende Fehler vermieden werden; von entscheidender Bedeutung für das Gelingen von Evolution und Wiederverwendung sind jedoch die kognitiven, sozialen und Problemlösungs-Fähigkeiten der beteiligten Mitarbeiter. Hierfür steht die Unterstützung der Kommunikation, die Vermittlung bei Konflikten, die Unterstützung bei Kompromiss-Entscheidungen sowie die Motivation bezüglich des Ziels im Vordergrund. In den Abschnitt 9.2.3 und 9.2.4 wird darauf weiter eingegangen.

Erfahrungen bei der Anwendung der Produktlinien-Methodik sowie anderer Techniken der Wiederverwendung haben gezeigt, dass wegen der Komplexität von Produktlinien und wegen der langfristigen Entwicklungsziele vor allem die folgenden beiden, sehr unterschiedlichen Bereiche der Entwicklungsprozesse besonderer Einflussnahme und Koordination bedürfen, damit Entwicklung erfolgreich verläuft:

- Veränderung und Refactoring, insbesondere in Verbindung mit Separation of Concerns
- Weitergabe von Wissen über Problemlösungen zwischen Entwicklern und zwischen Entwicklungsteams.

Bei *Veränderungen und Refactoring* von Software-Systemen ist es wichtig, Zusammenhänge zu begreifen und Abhängigkeiten zu erkennen. Wenn diese Veränderungen die Separation of Concerns zum Ziel haben wie beispielsweise bei der Entwicklung von Hyperslices, sind zusätzlich Gemeinsamkeiten zu analysieren und die spätere Komposition zu berücksichtigen. Bei diesen Tätigkeiten werden besondere Anforderungen an die Fähigkeiten der Entwickler gestellt, abstrahieren und Zusammenhänge herstellen zu können. Diese Fähigkeiten müssen mittels Werkzeugen durch graphische Veranschaulichung von Zusammenhängen in Modellen und Werkzeugen sowie durch Hilfsmittel zur Analyse unterstützt werden. Berücksichtigung finden diese Forderungen im hier entwickelten Ansatz beispielsweise in der Verknüpfung der Modelle durch Traceability-Links (siehe 4.2 und 6.1.2), in der automatisierten Erstellung von aktuellen Dokumentationen [Sametinger Riebisch 2002] sowie in Navigationsmöglichkeiten der Werkzeuge, die während der Entwicklung die Auswirkungen späterer Kompositionen veranschaulichen (siehe 6.4). Auch verfügbare Werkzeuge wie Refactoring-Browser [Boger et al. 2002] können dies unterstützen. Bei der Erstellung von Modellen sind diese Forderungen zu berücksichtigen und zu prüfen, wie in Abschnitt 9.3 dargestellt wird.

Die *Weitergabe von Wissen zwischen Entwicklern und Entwicklungsteams* ist für den Erfolg einer Produktlinie besonders wichtig, weil Fehler in einzelnen Komponenten Auswirkungen auf viele Produkte oder die gesamte Architektur haben können. Es wurden zwar zahlreiche Ansätze zum Wissens- und Erfahrungsmanagement in Wissenschaft und Industrie entwickelt, die auf Organisationsformen und Informationssystemen aufbauen, doch sind die erreichten Ergebnisse bisher eher unbefriedigend. Statt auf Mittel des Informationsmanagements zu setzen, sind deshalb vor allem die Kommunikation zwischen den Mitarbeitern zu fördern sowie gut verständliche Beschreibungsmit-

tel anzuwenden, wie bereits beim oberen Punkt genannt. Organisierte Formen der Kommunikation wie regelmäßige Architekturzirkel, interne Informationsforen zu Komponenten oder ein interner Helpdesk zu softwaretechnischen Problemen können zur Weitergabe von Wissen beitragen. Außerdem ist der Einfluss des Managements durch die unmittelbaren Vorgesetzten auf das Arbeitsklima und das Ausräumen von Konflikten und damit für die Motivation wesentlich.

Durch die hohe Veränderungsgeschwindigkeit der Anforderungen und durch deren mangelnde Vorhersagbarkeit bestehen in den oben genannten Entwicklungsprozessen solche Bedingungen, die Vorteile für *Agile Prozesse* bieten, bis hin zur Anwendbarkeit von Prinzipien des Extreme Programming XP [Beck 1999]. Solche Prinzipien sind in vielen Unternehmen gegenwärtig Gegenstand intensiver Diskussionen und Untersuchungen. Ihre Anwendung führt oft zu den erwarteten Erfolgen. Es zeigt sich, dass ihr Erfolg vor allem dann gefährdet ist, wenn die notwendigen Voraussetzungen fehlen oder wenn nur einzelne Prinzipien herausgegriffen werden. Als wichtigste Voraussetzungen zeigten sich Qualifikation und Verantwortungsbewusstsein der Mitarbeiter. Nach den Erfahrungen des Autors sind folgende Prinzipien von XP mit gutem Erfolg anwendbar, weil sie die Ziele Wartbarkeit, Flexibilität und Anwendungsorientierung unterstützen können:

- Gemeinsame Verantwortung für den Quellcode
- Codier-Standards
- Umfassendes und kontinuierliches Refactoring
- Einbeziehung des Kunden
- Einfachheit der Strukturen
- Entwicklung von Tests vor der Codierung
- Kontinuierliche Integration
- Kurze Release-Zyklen
- Ständige Peer Reviews oder paarweises Programmieren

Andere Prinzipien wie eingeschränkte Dokumentation der Architektur, der Anforderungsbeschreibung und der Entwicklungsstandards tragen nicht zum Erreichen der Ziele bei, weil sie dem Umfang der Aufgaben und der erforderlichen Arbeitsteilung nicht entsprechen. Statt dieser Prinzipien sind Unterstützungsmittel wie Online-Dokumentation und graphische Werkzeuge mit gemeinsamem Repository hilfreich. Zum Erhalt der Flexibilität sollten regelmäßige Überarbeitungen solcher Vorgaben wie Entwicklungsstandards unter Beteiligung der Entwickler durchgeführt werden.

## 9.2 Projektmanagement

Das Projektmanagement für die Produktlinienentwicklung unterscheidet sich von dem bei konventioneller Softwareentwicklung durch den Zeithorizont der Zielstellungen. Der Erfolg einer Produktlinie ergibt sich erst nach einer Reihe erfolgreicher Produkte. Auch nach einer mehrjährigen Entwicklung sollen Änderungen noch mit gleichbleibend geringem Aufwand möglich sein. Erfahrungen des Autors mit Software-Wiederverwendung in industriellen Projekten zeigen, dass der Erfolg der Wiederverwendung durch fehlende Langfristigkeit der Ziele des Projektmanagements gefährdet und verhindert wird.

Aus der Forderung nach Langlebigkeit und Evolution einer Produktlinie ergeben sich im Bereich des Projektmanagements folgende Aufgaben-Schwerpunkte, die hohe Anforderungen an Fähigkeit und Kompetenz der Führungskräfte auf allen Ebenen stellen:

- Schaffen der Voraussetzungen durch Sichern der Unterstützung durch das obere Management, durch Auswahl geeigneter Mitarbeiter, durch deren Qualifikation und Motivation.
- Vorgeben von Zielen für die Produktlinien-Entwicklung und deren Abstimmung mit langfristigen Unternehmenszielen.
- Optimale Nutzung von Metriken zur Ermittlung wichtiger Ziel-Kenngrößen und zur frühzeitigen Feststellung von Abweichungen.
- Optimaler Regelungsgrad durch Schaffen ausreichend präziser und detaillierter Regelungen zum Minimieren von Fehlern, Mängeln und Abweichungen. Es darf jedoch nur so weit regulierend eingewirkt werden, dass die Flexibilität der Entwicklung sowie Motivation und Engagement der Mitarbeiter noch erhalten bleiben (Abb. 69). Dies kann vor allem durch die Mitwirkung der Mitarbeiter an der Festlegung und Überarbeitung von Regelungen erreicht werden (siehe 9.2.3).
- Know-how-Transfer zwischen Bearbeitern und organisatorischen Einheiten

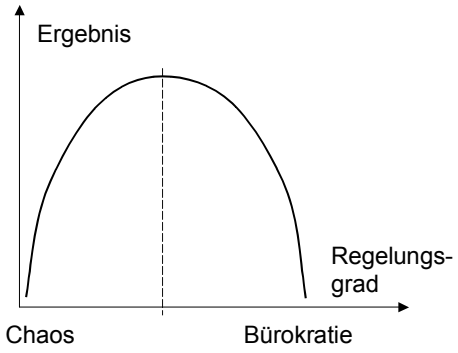



*Abb. 69: Auswirkung des Regelungsgrads auf das Ergebnis*

Außerdem sind die Regelungen für das Konfigurationsmanagement und Change Management (siehe 9.2.5) an die langfristigen Ziele und die große Änderungshäufigkeit anzupassen.

Das Projektmanagement umfasst die Aufgaben der Planung, der Organisation, der Führung (Kommunikation, Motivation, Kooperation und Koordination), sowie des Konfigurationsmanagements. Sie werden in den folgenden Abschnitten behandelt. Das Qualitätsmanagement trägt auch zur Steuerung der Entwicklung bei, wegen seiner abweichenden Aufgabenstruktur wird es jedoch separat im Abschnitt 9.3 dargestellt.

### 9.2.1 Planung

Eine Produktlinienentwicklung erfordert eine langfristige Bereitstellung von Investitionen, wofür die Planung wichtige Voraussetzungen schafft. Die Planung stellt gleichzeitig dem Management eines Unternehmens die Steuerungsmittel zur Verfügung, mit denen die betriebswirtschaftlichen Ziele erreicht werden können. Die Kostenkalkulation des Produktlinien-Engineering liefert Informationen darüber, welche Kosten auf einzelne Produkte umzulegen sind. Dabei sind Aufwendungen für kontinuierliches Refactoring von Produktlinie und Produkten vorzusehen, damit dem Effekt des Architectural Decay (siehe 1.2) vorgebeugt wird. Über die Kostenschätzung für einzelne Features kann auf die Angebots- und Preisbildung für Produkte Einfluss genommen werden.

Enges Zusammenwirken von softwaretechnischen und betriebswirtschaftlichen Entscheidungen trägt hier zum langfristigen Erfolg bei.

Die Planung der Entwicklung von Produktlinie und Produkten wird einerseits am Featuremodell ausgerichtet , andererseits werden Entscheidungen beim Scoping durch betriebswirtschaftliche Kriterien und strategische Ziele bestimmt, wie in Kapitel 4 dargestellt wurde. Für die Aufwandsschätzung können nach geringer Anpassung konventionelle Verfahren wie COCOMO II genutzt werden [Kluge 2003].

Als Modell für Entscheidungen über Investitionen, beispielweise über die Erweiterung einer Produktlinie um neue Features, hat sich die Bewertung einer zukünftigen Produkt-Möglichkeit ähnlich einer Call-Option etabliert [Geppert Roessler 2001].

### 9.2.2 Organisation

Die Projektorganisation unterstützt die Ziele des Entwicklungsprozesses wie Produktivität und langfristigen Erhalt der Wartbarkeit dadurch, dass sie mittels Spezialisierung und Arbeitsteilung Teams als organisatorische Einheiten bildet, die Kommunikation zwischen diesen Teams organisiert und ihre Steuerbarkeit gewährleistet. Die Gliederung der organisatorischen Einheiten muss im Einklang mit der Unternehmensorganisation und mit den Entwicklungsprozessen stehen. Die Entwicklungsprozesse sind in Produkt- und Produktlinien-Engineering sowie Reengineering gegliedert. Das Produktlinien-Engineering besteht aus getrennten Prozessen für das Hyperspace-Engineering und das Hyperslice-Engineering (Abb. 70). Die organisatorischen Einheiten werden entsprechend dieser Prozesse gegliedert und erhalten die dazugehörenden Aufgaben.

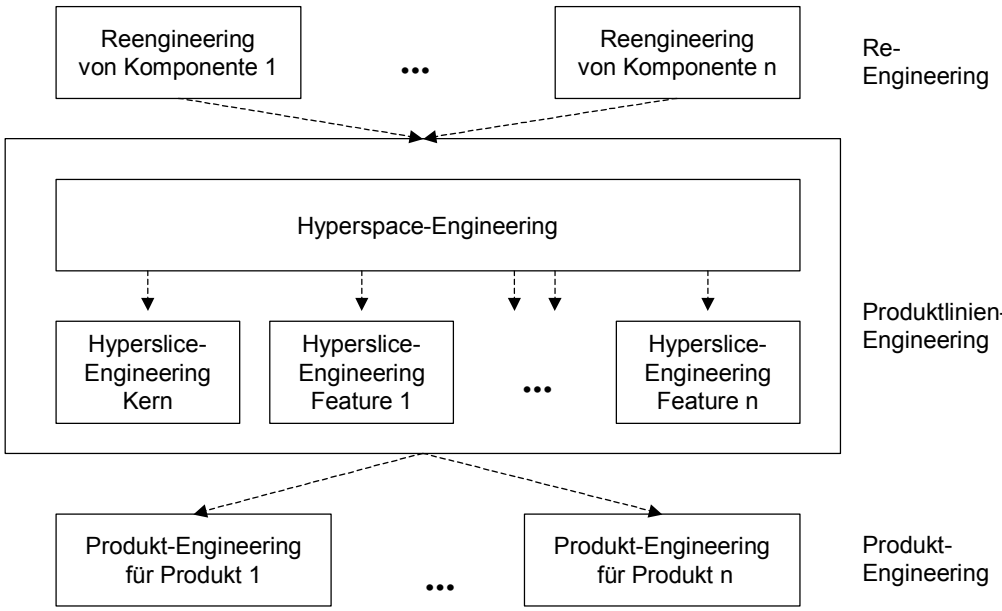



*Abb. 70: Prozesse des Produktlinien-Engineering*

Das Team des *Hyperspace-Engineering* hat eine zentrale Koordinationsaufgabe innerhalb der Produktlinienentwicklung. Es hat nicht nur die Entwicklung von Produkten zu koordinieren, sondern vor allem die zielgerichtete Weiterentwicklung der Produktlinie und alle Arbeiten des Hyperslice-Engineering. Damit diese Aufgabe von einem Team wahrgenommen werden kann, müssen dessen Mitglieder hohe Anforderungen an Abstraktionsvermögen, Kreativität, Methodenkompetenz und Koordinationsfähigkeit erfüllen. Das Team muss neben den Koordinationsaufgaben mit den Teams des Hyperslice-

Engineering vor allem für eine breite Anwendung der Produktlinie beim Produkt-Engineering werben. Neben einer solchen Marketing-Tätigkeit muss dieses Team insbesondere am Prozess der Kompromiss-Findung zwischen Kunden-Anforderungen und Produktlinien-Angeboten (siehe 4.5) mitwirken, damit ein möglichst hoher Anteil neuer Produkte aus vorgefertigten Hyperslices zusammengesetzt werden kann und damit an der Produktlinie möglichst geringe Veränderungen vorgenommen werden müssen. Durch Argumentation mit Vorteilen wie kürzeren Entwicklungszeiten und günstigerer Kostengestaltung kann auf die gewünschte Kompromissbildung und den Verzicht auf zusätzliche Anforderungen beim Kunden hingewirkt werden, ohne die Kundenzufriedenheit zu beeinträchtigen. Die Mitglieder des Hyperspace-Engineering-Teams müssen zur Marketing-Tätigkeit und zur Argumentation mit den genannten Vorteilen motiviert und stimuliert werden; eine Bewertung der Ergebnisse und Anerkennungen können dabei unterstützend wirken.

Die Weiterentwicklung der Produktlinie bezüglich der Einführung neuer Variabilität, neuer Bindungszeitpunkte oder zusätzlicher Features muss ebenfalls vom Hyperspace-Engineering-Team koordiniert werden. Entscheidungen werden dabei in Auswertung von Marktanalyse und Kundenwünschen sowie unter betriebswirtschaftlichen Gesichtspunkten beim Scoping getroffen (siehe 4.6). Damit das Team alle diese Aufgaben erfüllen kann, muss es aus Mitarbeitern mit Domänenwissen und mit Kompetenzen in den Bereichen Softwarearchitektur, Softwareentwicklung und Kostenkalkulation zusammengesetzt werden. Temporäre Entsendung und Job-Rotation, zum Beispiel zwischen den Teams der Produktentwicklung, des Hyperslice-Engineering oder der Kundenbetreuung haben sich dabei für die Kompetenzentwicklung und Kommunikation als nützlich erwiesen. Außerdem ist eine gewisse Mischung von Mitarbeitern unterschiedlicher Charaktertypen wichtig, damit einerseits durch Innovationsfreude und Risikobereitschaft Veränderung gefördert und andererseits durch Genauigkeit und Beständigkeit Erreichtes bewahrt wird; weiterführende Hinweise hierzu enthält [Dueck 2002].

Die Teams des *Hyperslice-Engineering* beliefern die Teams des Produkt-Engineering; sie stellen Hyperslices bereit, die für eine Komposition zu Produkten geeignet sind. Da die Entwicklung einzelner Hyperslices abgeschlossene und zeitlich begrenzte Aufgaben darstellt, bietet sich eine Organisation in Form von Sub-Projekten des Produktlinien-Engineering an. Stilistische Einheitlichkeit von Architektur und Quellcode im Interesse guter Wartbarkeit erfordert eine hohe Selbstdisziplin der Entwickler. Neben regelmäßigen Prüfungen kann die Einhaltung von Architektur- und Codierstil vor allem durch eine gemeinsame Einflussnahme der Entwickler auf Regelungen für solche Stilvorgaben, sowie für Dokumentation und Problemlösungs-Methoden erreicht werden. Der wesentliche Teil von Refactoring und Separation of Concerns wird von den Teams des Hyperslice-Engineering geleistet, trotzdem muss ein Teil der Entscheidungskompetenz über die Architektur und Namensvergabe beim Hyperspace-Engineering-Team liegen, damit es die Koordinationsfunktion erfüllen kann. Damit diese Verteilung der Aufgaben und Kompetenzen zwischen beiden Teams auf Dauer aufrechterhalten werden kann, ist ein temporärer Austausch der Mitarbeiter hilfreich. Ein solcher temporärer Tausch kann durch einen Mitarbeiter-Pool organisiert werden, aus dem die verschiedenen Hyperslice-Engineering-Teams zusammengesetzt werden und aus dem einige Mitarbeiter zeitweise im Hyperspace-Engineering mitarbeiten. Auch beim Hyperslice-Engineering ist der Einsatz von Mitarbeitern mit verschiedenen Charaktertypen hilfreich. Der Schwerpunkt der Aufgaben liegt jedoch im kreativen Bereich, wo gleichzeitig ein hohes Maß an technischen Kenntnissen über Methoden, typische Problemlösungen, Programmiersprachen und Werkzeuge erforderlich ist. Abb. 71 zeigt als Beispiel eine Organisation,

bei der das Produktlinien-Engineering unter der Verantwortung eines Produktlinien-Managers mit Job-Rotation in diesem Bereich durchgeführt wird. Die Teams des Produkt-Engineering sind in diesem Beispiel separat als Bestandteile anderer Bereiche der betreffenden Unternehmensstruktur organisiert.

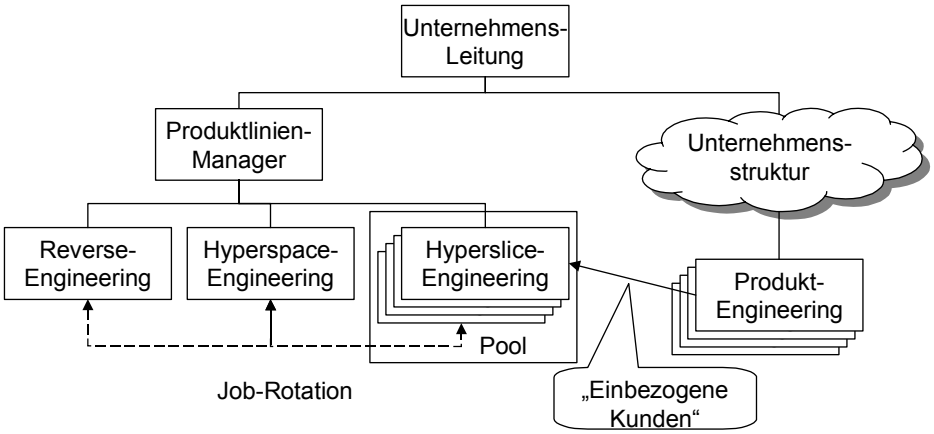



*Abb. 71: Beispiel für ein Organigramm*

Die Teams des *Produkt-Engineering* haben eine Mittler-Funktion, sie sind Konsumenten des Produktlinien-Engineering und treten gleichzeitig dem Kunden gegenüber als Produzenten auf. Solche Teams sollten aufgrund es temporären Charakters der Aufgaben ebenfalls in Projekten organisiert sein und in die Unternehmensorganisation eingebunden werden. Für jedes Produkt gibt es einen Verantwortlichen (häufig als Produktmanager bezeichnet) oder ein Team, das gemeinsam mit dem Kunden Anforderungen in der bereits geschilderten Weise als Kompromiss definiert (siehe 4.5 und 6.2.1), dabei Kosten- und Terminschätzungen vornimmt und Angebote erarbeitet. Diese Aktivitäten müssen im Zusammenwirken mit den Produktlinien-Engineering-Teams erfolgen. Gegenüber dem Produktlinien-Engineering vertritt das Produkt-Engineering die Anforderungen und Termine, verwaltet die Kosten und führt die Abnahme des Produkts durch. Vertreter wirken als „einbezogene Kunden“ in Hyperslice-Engineering-Teams mit, damit ihr unmittelbares Feedback über Anforderungen, Qualität und Probleme in den Prozess einfließt, damit sie Abnahme und Integration vornehmen und damit Konflikte vermieden werden. Entwicklungsleistungen außerhalb der Produktlinie sind ebenfalls durch das Produkt-Engineering durchzuführen oder zu veranlassen (siehe 6.2.3). Der Aufgabenbereich der Mitarbeiter deckt sämtliche Entwicklungstätigkeiten von der Anforderungsanalyse bis zur Inbetriebnahme und Kundenbetreuung ab, entsprechend vielfältige Kompetenzen sind erforderlich.

Ein Team des *Reverse Engineering* wird im Auftrag des Produktlinien-Engineering tätig, damit existierende Systeme oder Komponenten, wie etwa von Drittanbietern, in die Produktlinie integriert werden können. Da es sich ebenfalls um temporäre Aufgaben handelt, bietet sich dafür auch eine Projektorganisation an. Die Mitarbeiter dieses Teams müssen neben Methoden- und Problemlösungswissen über genaue Kenntnisse von Architektur und Vorgaben von Hyperslice- und Hyperspace-Engineering verfügen, zusätzlich zu Kenntnissen über Reverse-Engineering-Methoden. Dazu ist es sinnvoll, sie beispielweise über Job-Rotation zeitweise in Aufgaben des Produktlinien-Engineering einzubeziehen.

Die Beziehungen zwischen Produktlinien- und Produkt-Engineering müssen auf die *Unternehmensorganisation* des betreffenden Unternehmens abgestimmt sein. Je nach

Unternehmensorganisation können die Teams des Produkt-Engineering in vorhandene Sparten, Linien oder unabhängige Einheiten integriert sein. Wegen der langfristigen Ziele sollte das Produktlinien-Engineering der Unternehmensleitung direkt unterstellt sein (Abb. 71). Konflikte zwischen Produktlinien-Engineering und den verschiedenen Einheiten mit Produkt-Engineering ergeben sich durch die unterschiedlichen Ziele zwangsläufig. Die Unternehmensleitung sollte hier durch strategische Vorgaben vermittelnd einwirken. Bewertungssysteme und Metriken unter anderem für Wiederverwendungsanteil, Nutzungshäufigkeit und Anteil am Deckungsbeitrag können dazu beitragen, ausgeglichene Entscheidungen zwischen strategischen und kurzfristigen Zielen zu erreichen.

### 9.2.3 Kommunikation und Motivation

Kommunikation und Motivation haben hohe Bedeutung für den Erfolg von Softwareentwicklungs-Tätigkeiten. Führungskräfte in den oben genannten Engineering-Teams haben wegen der Bedeutung psychologischer Faktoren einen großen Einfluss darauf, dass langfristige Ziele einer Produktlinienentwicklung erreicht werden. Die Auswahl von Führungskräften mit ausreichenden psychologischen und sozialen Fähigkeiten ist vom oberen Management vorzunehmen. Die Unterstützung der Ziele durch das obere Management stellt eine Grundvoraussetzung für einen langfristigen Erfolg dar; sie muss durch regelmäßige Berichterstattung und Abrechnung gesichert werden.

In den Teams muss ein Klima geschaffen werden, das eine offene Kommunikation über Verbesserungen, Probleme und Fehler ermöglicht. So kann ein Wissenstransfer erreicht werden, der Erfahrungsberichte und offene Fehlerauswertung einschließt. Richtlinien und Standards können gemeinsam entwickelt, optimiert und erfolgreich angewendet werden. Führungskräfte müssen sowohl Verantwortung übernehmen als auch an Mitarbeiter weitergeben. Standards für Architektur und Codierung können dann trotz häufiger Änderungen eine wichtige Rolle bei der langfristigen Aufrechterhaltung von Wartbarkeit und Klarheit der Software übernehmen.

Konflikte, Vorbehalte und Misstrauen können durch unterschiedliche Interessen und Charaktere der beteiligten Mitarbeiter kaum von vornherein vermieden werden. Sie sind durch Kommunikation und Vermittlung vor allem durch die Führungskräfte auszuräumen. Organisatorische Maßnahmen können einbezogen werden, damit eine derartige Kommunikation aufgenommen und institutionalisiert wird. Regelmäßiges Training von Konfliktlösungsmethoden kann die Aufrechterhaltung der Kommunikation unterstützen. Etablierte Führungsmethoden wie Total Quality Management TQM können genutzt werden. Ständige Qualitätszirkel tragen zur Suche nach Verbesserungsmöglichkeiten und zur Nutzung von Optimierungspotential bei [Mellis et al. 1996].

Eine hohe Motivation ist eine wesentliche Voraussetzung dafür, dass Mitarbeiter Verantwortung für das Refactoring übernehmen und die Selbstdisziplin aufbringen, wie sie zu konsequenter Vereinfachung von Strukturen (beispielweise anstelle zwar technisch reizvoller, aber nicht optimaler Lösungen) notwendig ist. Diese Selbstdisziplin ist die Voraussetzung dafür, dass die in dieser Arbeit vorgestellten Methoden nach ihrer Einführung langfristig aufrechterhalten bleiben, und dass damit ein langfristiger Evolutions- und Reifungsprozess der Produktlinie erreicht wird.

### 9.2.4 Kooperation und Koordination

Wegen der hohen Komplexität einer Produktlinie und der Aufgaben während ihrer Entwicklung kommt der Koordination eine große Bedeutung zu. Organisatorische Mittel der Führung - wie beispielsweise eine rein hierarchische Organisation - können

diese Aufgabe nicht mit der erforderlichen Flexibilität erfüllen. Mitarbeiter müssen mit Aufgaben der Koordination beauftragt werden und die entsprechende Verantwortung übernehmen.

Wegen der Vielzahl unterschiedlicher Aufgaben und der Arbeitsteilung in verschiedene Entwicklungsprozesse muss eine gute Kooperation zwischen Mitarbeitern und Teams erreicht und aufrechterhalten werden. Werkzeuge können die Kooperation dadurch unterstützen, dass sie zum Beispiel den Namensraum für das Hyperspace- und das Hyperslice-Engineering verwalten oder Anforderungen anhand des Featuremodells zwischen Produktlinien- und Produkt-Engineering organisieren. Engagierte Mitarbeiter und Führungskräfte müssen bei Konflikten und gestörter Kooperation Einfluss nehmen. Führungsmethoden wie TQM können helfen, Erfahrungen beispielweise in Best-Practice-Reports zu sammeln und bewährte Prinzipien über Teamgrenzen hinweg zu etablieren.

### 9.2.5 Konfigurationsmanagement

Das Konfigurationsmanagement muss die Koordination von Veränderungen an Produkten gewährleisten. Die Arbeitsprinzipien unterscheiden sich dabei nicht grundsätzlich von denen bei konventioneller Softwareentwicklung, wie sie beispielweise durch das V-Modell definiert werden [V-Modell] oder für das Capability-Maturity-Reifegradmodell vorgeschlagen werden [Thaller 1993]. Durch die stärkeren Abhängigkeiten zwischen den Elementen und durch den iterativen Charakter der Aktivitäten ist eine stärkere Verzahnung des Konfigurationsmanagements mit den in Modellen beschriebenen Abhängigkeiten erforderlich. So werden beispielsweise Änderungszustände gegenüber dem V-Modell erweitert, weil durch Scoping-Entscheidungen (siehe 4.6) mehrere Möglichkeiten der Realisierung existieren, wie Abb. 72 zeigt.

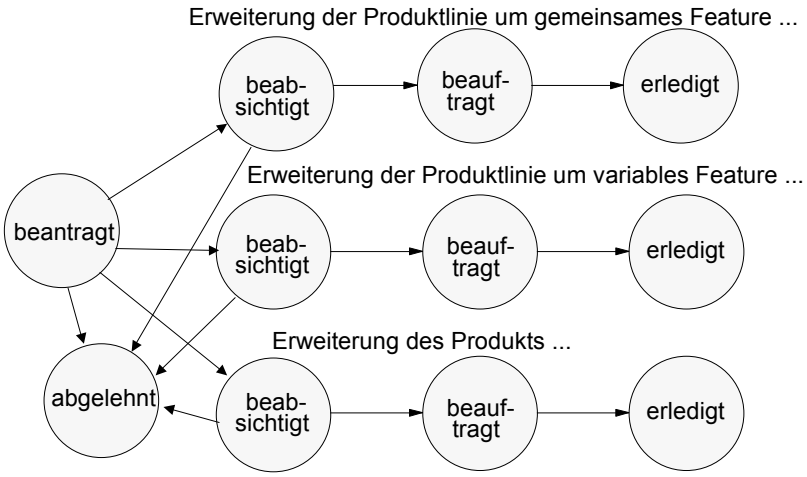



*Abb. 72: Zustände einer Änderungsanforderung an die Produktlinie (vereinfacht)*

Gegenstand des Konfigurationsmanagements sind Anforderungsbeschreibungen, Featuremodelle, Architekturbeschreibungen, Objektmodelle und Quellcode sowie Testmittel, Notizen, Beschreibungen von Entwurfsentscheidungen, Dokumentationen und Informationen über Konfigurationen bei Kunden. Wegen der komplexen Verknüpfungen zwischen den Elementen und der Aufteilung durch Separation of Concerns ist das *Prinzip der Lokalität* wichtig, damit Änderungen und das Konfigurationsmanagement selbst effektiv durchgeführt werden können. Es basiert auf den Prinzipien des Self-

Containment und der Entkopplung, wie sie für die Methodik KobrA in Abschnitt 17.1.3 von [Atkinson et al. 2002] beschrieben sind.

Das Prinzip der Lokalität fordert, dass alle Informationen über Abhängigkeiten im davon betroffenen Artefakt enthalten sind, und dass Artefakte wie Spezifikation, Modell und Implementierung unabhängig voneinander in Versionen zu gliedern sind, damit Abhängigkeiten minimiert werden. Daraus resultieren vor allem zwei Konsequenzen: zum Ersten ist die Versionierung selbst nicht Gegenstand des Konfigurationsmanagements, weil sie in jedem Artefakt enthalten ist; zum Zweiten wird jede Abhängigkeitsbeziehung zwischen zwei Artefakten durch ein Paar aus Vorwärts- und Rückwärts-Referenzen abgebildet. Bei allen Veränderungen müssen immer beide Referenzen aktualisiert werden, damit ihre Konsistenz erhalten bleibt.

Die Einführung dieses Prinzips erleichtert gegenüber der zentralen Speicherung von Abhängigkeiten und Versionierung sowohl die Navigation durch einfacheres Verfolgen von Beziehungen als auch die Organisation der Entwicklungsteams durch verringerte Kopplungen zwischen ihnen. Zur lokalen Speicherung von Abhängigkeiten gehört die lokale Angabe zulässiger Versionen von Artefakten, zu denen Beziehungen bestehen, damit die Gültigkeit von Konfigurationen geprüft werden kann. Diese lokale Speicherung stellt einen Unterschied zu den meisten Implementierungen des V-Modells dar, bei dem ein sog. Konfigurations-Identifikationsdokument gültige Konfigurationen beschreibt. Die durch die Lokalität entstehende Redundanz der Daten stellt einen Nachteil des Prinzips dar, der sich jedoch weniger stark auswirkt als die genannten Vorteile.

Änderungen unterscheiden sich nach dem Auslöser der Anforderungen in

- Behebung von Fehlern und Mängeln (sowohl funktionaler als auch nichtfunktionaler Merkmale),
- Anpassung an Änderungen der Umgebung (wie an technische Schnittstellen von Systembestandteilen),
- Verbesserungen aufgrund erhöhter Anforderungen und
- Realisierung zusätzlicher Anforderungen (sowohl funktionaler als auch nichtfunktionaler Merkmale).

Eine Änderungsanforderung wird unabhängig vom Auslöser als Entscheidung gemäß 4.6.1 vorbereitet (siehe Abb. 72); dazu werden Referenzen zu Anforderungs- oder Objektmodell oder zur Implementierung hergestellt. Die eigentliche Ausführung der Änderung durch Entwicklungstätigkeiten erfolgt meist im Zuge normaler Iterationen (siehe 6.1.4), zusätzliche Anforderungen werden jedoch meist als Erweiterungen von Produktlinien realisiert, wie in Abschnitt 6.3 behandelt. Änderungen werden zwecks langfristiger Konsistenzsicherung der Produktlinie vom Konfigurationsmanagement koodiniert. Unter Einbeziehung der in Abschnitt 6.1.4 angeführten Aktivitäten umfasst dies die folgenden Tätigkeiten:

- *Aktualisierung und Abgleich der betroffenen Elemente*. Entsprechend der bei der Betrachtung des Konfigurationsmanagements beschriebenen Abhängigkeiten sind den Änderungen entsprechende Abgleiche auszuführen. Bei Änderungen von Formulierungen müssen keine weiteren Elemente abgeglichen werden; bei Änderungen der Semantik wie beispielweise von Struktur, Verhalten oder Funktionalität muss dagegen ein Abgleich vorgenommen werden, wodurch sich im allgemeinen Folge-Änderungen ergeben. Methodische Unterstützung kann dabei das Verfahren zum Verfolgen von Abhängigkeiten nach [Meyer 2000] bieten.

- *Aktualisierung der Konfiguration.* Die Änderungen von Abhängigkeiten der Elemente untereinander ist lokal in den betroffenen Elementen abzugleichen. Zwecks Konsistenzsicherung doppelt verketteter Referenzen sollte diese Aufgabe ein Werkzeug übernehmen.
- *Anpassung der Build-Prozesse.* Für die Erstellung zukünftiger Produkte unter Einbeziehung der vorgenommenen Änderungen sind im Build-Prozess die betroffenen Elemente zu verknüpfen. Dabei sind Build-Prozesse für begleitende Ergebnisse wie Testmittel und Dokumentationen ebenfalls zu aktualisieren.
- *Aktualisierung von Verteilung und Installation, sog. Deployment.* Zum Deployment gehören Maßnahmen der Verteilung, Installation und Inbetriebnahme, gegebenenfalls einschließlich notwendiger Migrationsmaßnahmen.

Weiterführende Betrachtungen zur Einbindung des Konfigurationsmanagements in betriebliche Strukturen sind in den Projektergebnissen und Publikationen der Projekte ESAPS und CAFÉ enthalten [ESAPS][CAFE].

## 9.3 Qualitätsmanagement

Das Qualitätsmanagement sichert durch geeignete Maßnahmen, dass die Entwicklungsziele erreicht werden und dass dies gesteuert und kontrolliert werden kann. Bereits bei Wiederverwendungs- und Komponentenansätzen haben die Qualitätsmanagement-Maßnahmen große Bedeutung, weil durch einzelne mangelhafte Elemente *eine Reihe von Produkten* beeinträchtigt werden. Für den langfristigen Erfolg einer Produktlinien-Entwicklung entsteht eine zusätzliche Bedeutung des Qualitätsmanagements dadurch, dass alle Aktivitäten durch die hohe Komplexität der Produktlinien erschwert werden. Maßnahmen des Qualitätsmanagements dienen zur Steuerung und Optimierung der Aktivitäten. Werden Abweichungen von den Zielen frühzeitig erkannt, kann eine Korrektur mit geringerem Aufwand und mit geringerer Gefährdung des Erfolgs durchgeführt werden, als bei einer Erkennung von Fehlern und Mängeln an einem fertiggestellten Produkt.

### 9.3.1 Qualitätsmodell

Ein Qualitätsmodell spiegelt die Ziele und Kriterien des Qualitätsmanagements in Form von Produkt- und Prozessmerkmalen wieder; es stellt Kriterien für faktenbasierte Entscheidungen bereit, wie sie beispielweise von TQM vorgesehen werden. Eine Reihe von Zielen deckt sich aus Sicht der Kunden mit denen konventioneller Softwareentwicklung, wie die sog. äußeren Qualitätsmerkmale Korrektheit, Anwendbarkeit, und Effizienz [DIN 66272], die funktionale und nichtfunktionale Merkmale beschreiben. Für den Erfolg des entwickelnden Unternehmens sind Ziele wie Langlebigkeit und Evolution einer Produktlinie von großer Bedeutung, die durch die inneren Produktmerkmale Wartbarkeit und Portabilität als Aufwand für Veränderungen beschrieben werden. Außerdem können Prozessmerkmale wie Prozess-Reifegrad, Güte des Erfahrungsmanagements und der Kommunikation ins Qualitätsmodell einbezogen werden, mit denen Entwicklungsprozesse bewertet und beeinflusst werden können [Riebisch 2003].

Die Qualitätsmerkmale, ihre Wichtung und ihre quantitativen Kenngrößen sind in Abstimmung mit den Rahmenbedingungen des Unternehmens und der Produktlinie zu spezifizieren. Dabei sind die besonderen Erfolgskriterien von Produktlinien zu berücksichtigen, wie langfristige Erhaltung der Änderbarkeit, Güte der Separation of Concerns, Verständlichkeit der Modelle und Wissenstransfer zwischen Mitarbeitern. Da Fehler in Featuremodell und Hyperspace großen Einfluss auf die Gesamtheit der Pro-

dukte der Produktlinie haben, sind Kriterien für frühzeitige Prüfungen wichtig. Zur Aufstellung der quantitativen Kenngrößen eignet sich die Methode Goal-Question-Metric GQM [Basili et al. 1994]. Ein Beispiel für eine quantitative Kenngröße für Änderungsaufwand ist der

> *Durchschnittliche Änderungsaufwand:* Durchschnittlicher Aufwand (in Arbeitsstunden) für die Anforderungsbeschreibung, Modellierung, Implementierung und Integration einer Änderung, bezogen auf die Summe der Anzahl der geänderten Quellcode-Zeilen und der geänderten Modellelemente.

Diese Kenngröße erfüllt bei Vorliegen von Arbeitszeitdaten die Forderungen nach einfacher Erhebbarkeit, guter Korrelation, Objektivität und Vergleichbarkeit [Riebisch 2003]. Ihre Erhebung zu verschiedenen Zeitpunkten erlaubt Aussagen zur Flexibilität der Produktlinie, zur Qualität ihrer Architektur und zum entstandenen Architectural Decay.

Weitere Kenngrößen mit Einfluss auf Wartbarkeit und Portabilität können die Komplexität von Strukturen bewerten. Einfach erhebbare Kenngrößen können beispielweise die Anzahl von Kopplungen zwischen Komponenten oder innerhalb von diesen zählen. Daraus können Aussagen über Stärke der Abhängigkeiten und innere Festigkeit von Komponenten abgeleitet werden. Solche Kenngrößen sind jedoch lediglich Indikatoren für die interessierenden Qualitätsmerkmale, die Zuverlässigkeit ihrer Aussagen ist begrenzt.

Die im Qualitätsmodell enthaltenen Qualitätsmerkmale werden zum Auswählen und Steuern von Maßnahmen des Qualitätsmanagements genutzt. Besondere Bedeutung haben sie für konstruktive Maßnahmen, die zum Beispiel in Form von Vorgaben für Architekturstil und Codierstil und Richtlinien für Dokumente den Entwicklungsprozess beeinflussen und für frühzeitige Prüfungen. Steigt beispielweise der mit der Kenngröße des Beispiels erhobene Änderungsaufwand, so kann dies ein Indiz für Mängel bei der Verständlichkeit der Software-Architektur sein. Es kann als Entscheidungskriterium für höhere Aufwendungen für Architekturqualität verwendet werden, von Schulungen zu Architekturstilen über gesteigertes Gewicht der Änderungsfreundlichkeit in Architektur-Reviews bis hin zu Refactoring-Maßnahmen.

Zur langfristigen Auswertung quantitativer Kenngrößen sind Metrikprogramme ein Mittel, das bei Beschränkung auf einfach zu erhebende und wichtige Daten eine gute Entscheidungsunterstützung liefern kann, beispielweise auch zur Risikobewertung und für Refactoring-Entscheidungen. Für weitergehende Informationen zur Aufstellung, Einführung und Aufrechterhaltung von Metrikprogrammen sei auf [Dumke 1992] verwiesen. Es existiert eine Vielzahl von Metriken für objektorientierte Entwürfe und Software wie beispielweise [Erni Lewerentz 1996]. Eine Einordnung bietet [Mens et al. 2003].

### 9.3.2 Analytische Maßnahmen

Analytische Qualitätssicherungsmaßnahmen dienen der Bewertung der Erfüllung von Qualitätsmerkmalen. Frühzeitig im Entwicklungsprozess angewandt, ermöglichen sie die Einflussnahme auf die und die Optimierung der Prozesse, und tragen sowohl zur Verbesserung der Qualitätsmerkmale als auch der Produktivität bei. Einen Überblick und weiterführende Informationen über solche Maßnahmen bieten [Wallmüller 2001], [Balzert 2001] und [Riebisch 2003]. Wie auch von TQM gefordert, sollten sie fest in den Entwicklungsprozess eingebunden werden. Die phasenbezogene und Merkmalbezogene Einbindung wird üblicherweise in einem Vorgehensmodell beschrieben.

Durch ein Vorgehensmodell wird beispielsweise festgelegt, dass kein Entwicklungsergebnis ohne Prüfung oder Abnahme zur Nutzung freigegeben wird, und dass Testfälle und Abnahmekriterien vor Beginn eines Entwicklungsschritts definiert werden [V-Modell]. Einzelne analytische Maßnahmen eignen sich unterschiedlich gut für die Bewertung einzelner Qualitätsmerkmale, Tabelle 8 zeigt eine solche Zuordnung [Riebisch 2003].

*Tabelle 8: Analytische Maßnahmen für die Bewertung verschiedener Qualitätsmerkmale*

| Qualitäts-merkmal | Analytische Maßnahmen (nach Eignung sortiert in fallender Reihenfolge) |
|---|---|
| Funktionalität | Tests, Inspektionen (bezüglich Vollständigkeit, Wiederspruchsfreiheit, Korrektheit),<br>Bewertung von Verteilungs-Szenarien (Interoperabilität) und Angriffs-Szenarien (Sicherheit) |
| Zuverlässig-keit | Inspektionen, Tests, Messung struktureller Merkmale, Fehlermodellierung, Zuverlässigkeitsmodelle |
| Benutzbarkeit | Bewertung von Prototypen, Versuchsreichen zur Benutzbarkeit, Interviews, Bewertung mit Kognitiven Modellen |
| Effizienz | Bewertung mit Belastungs-Szenarien, Ermittlung von Maximalbelastungen, Bewertung der Komplexität von Algorithmen |
| Wartbarkeit | Inspektionen und Reviews (bezüglich Verständlichkeit, Einfachheit, Fehlervermeidung), Messung struktureller Eigenschaften, Bewertung von Änderungsszenarien, Auswertung von Projektdaten, Bewertung der Variabilität (anhand des Featuremodells, der Architektur und Implementierung) |
| Portierbarkeit | Inspektionen und Reviews (bezüglich Plattform- und Werkzeugabhängigkeit) und Bewertung der Variabilität (anhand Architektur und Implementierung), Bewertung von Portierungsszenarien, |

Eine Bewertung innerer Qualitätsmerkmale wie Wartbarkeit und Portabilität ist vor allem durch *Reviews und Inspektionen* möglich. Diese Maßnahmen können gleichzeitig sowohl zur Aufrechterhaltung von Vorgaben und Richtlinien als auch zum Wissenstransfer und zur Motivation der Mitarbeiter eingesetzt werden. Mit ihrer Durchführung kann außerdem eine Überprüfung der Vorgaben und Richtlinien der Softwareentwicklung verbunden werden, ob sie für die Ziele ausreichend und angemessen sind.

*Tests* sind immer mit hohem Aufwand verbunden. Der Aufwand für Tests von Produktlinien ist sehr viel höher als bei konventionellen Produkten, weil er durch die Vielfalt möglicher Kombinationen eine viel größere Anzahl von Testfällen erfordert. Die Testfall-Erstellung und Testdurchführung sollte deshalb weitgehend automatisiert werden. In der ALEXANDRIA-Methodik werden Anforderungs- und Verhaltensmodelle zur Generierung von Testfällen verwendet, eine entsprechende Methode wird gegenwärtig im Rahmen eines DFG-Projekts entwickelt. Testprogramme zur automatisierten Testdurchführung werden wie Modelle und Quellcode durch Sepration of Concerns aufgetrennt und mit dem Hyperspace-Ansatz zusammengefügt, damit geeignete Testmittel für jedes Produkt bereitgestellt werden können (siehe Abschnitte 6.1 und 6.2).

### 9.3.3 Konstruktive Maßnahmen

Konstruktive Qualitätssicherungsmaßnahmen sollen die Entwicklung von Software ermöglichen, die *von vornherein* die geforderten Qualitätsmerkmale aufweist. Kon-

struktive Maßnahmen haben deshalb eine höhere Bedeutung als analytische, weil sie durch die Vermeidung von Nacharbeit die Produktivität steigern und durch Vermeiden von Änderungen als Folge solcher Nacharbeit zur Erhaltung der Wartbarkeit beitragen. Zahlreiche konstruktiven Maßnahmen wurden in diesem Kapitel bereits in einem anderem Zusammenhang erwähnt: Verbesserung von Kommunikation und Wissenstransfer, Maßnahmen zur Erhaltung der Motivation und des Qualitätsbewusstseins der Entwickler sowie Organisation von Entwicklungsprozessen und Vorgehensmodelle. Zur Gruppe der konstruktiven Maßnahmen gehören außerdem softwaretechnische Entwicklungs- und Modellierungsmethoden, wie sie mit den Methoden HyperFeatuRSEB und Hyper/UML Hauptgegenstand der Methodik ALEXANDRIA sind. Ein Überblick und weiterführende Informationen zu konstruktiven Maßnahmen sind in [Wallmüller 2001], [Thaller 1993], [Balzert 2001] und [Riebisch 2003] enthalten.

Für die langfristige Aufrechterhaltung der Evolution einer Produktlinie haben sich *Richtlinien und Vorgaben* als besonders geeignet dafür erwiesen, dass die Wirksamkeit von Entwicklungsmethoden in der Praxis gewährleistet wird. Sie beeinflussen die Arbeit der Entwickler bei Modellierung und Implementierung. Ihre Wirksamkeit steigt, wenn sie in Kombination mit Inspektionen und Reviews eingesetzt werden (Abb. 73). Die Richtlinien können dabei regelmäßig bewertet und in organisierter Weise an aktuelle Gegebenheiten angepasst werden. Für jeden einzelnen Mitarbeiter bieten sich damit Möglichkeiten der Einflussnahmeauf die Richtlinien, wodurch deren Akzeptanz, die Motivation und der Teamgeist steigen und der Entwicklungsprozess stabilisiert wird.


*Abb. 73: Verstärkung des Einflusses von Richtlinien durch Mitwirkung der Mitarbeiter*

Für die Ziele der Produktlinienentwicklung spielen Richtlinien für die Durchführung der Entwicklungsaktivitäten eine wichtige Rolle. Außerdem sind Richtlinien und Vorgaben zu Stil, Inhalt und Aufbau von Anforderungsmodell, Featuremodell, Architekturbeschreibung und –stil, Codierstil und Dokumentation von Bedeutung. Zusammenstellungen typischer Fehler können eine ähnliche Funktion wie Richtlinien und Vorgaben übernehmen und zur Fehlervermeidung beitragen. Auch sie können in Kombination mit Reviews und Inspektionen angewendet und gleichzeitig aktualisiert werden.

## 9.4 Anforderungen an Werkzeuge

Werkzeuge können methodisches Arbeiten nicht erzwingen oder ersetzen; die in diesem Kapitel genannten Methoden und Vorgehensweisen müssen von den Entwicklern und Bearbeitern akzeptiert und erlernt werden, bevor Werkzeuge erfolgreich benutzt werden können. Dies gilt besonders für Bereiche, in denen soziale Beziehungen eine wichtige Rolle spielen, wie Kommunikation, Koordination und Arbeitsteilung. Werkzeuge erleichtern vor allem Routinetätigkeiten und die Handhabung großer Informationsmengen

Für die organisatorische Unterstützung von Entwicklungsprozessen existieren eine Reihe von Werkzeugen, die sich in langfristigen und umfangreichen Projekten bewährt haben. Bei der Auswahl von Werkzeugen ist vor allem auf Integrierbarkeit in eine (ggf. existierende) heterogene Werkzeuglandschaft, auf Passfähigkeit zur Organisation, auf Automatisierbarkeit von Vorgängen sowie auf Benutzbarkeit und Funktionalität zu achten. Wegen der Vielgestaltigkeit dieser Kriterien können keine pauschale Empfehlungen für einzelne Werkzeuge gegeben werden, sondern nur Hinweise auf Kriterien zu ihrer Auswahl. Häufig müssen Kompromiss-Entscheidungen getroffen werden, weil kein Werkzeug alle Anforderungen vollständig erfüllt. Aufgrund der Erfahrungen des Autors ist der Schwerpunkt bei der Auswahl von Werkzeugen auf Einfachheit und Wirksamkeit anstelle von Perfektion zu legen.

Für das Konfigurationsmanagement als Teil der Entwicklungsprozess-Unterstützung existieren recht ausgereifte Werkzeuge. Bei ihrer Auswahl ist besonders auf Passfähigkeit zur Entwicklungsumgebung und zu Werkzeugen des Projektmanagements zu achten. So sollten die Konfigurationsmanagement-Werkzeuge in der Lage sein, Versionierungs- und Abhängigkeits-Informationen in den verschiedenen Ergebnissen bei Änderungen zu aktualisieren, damit die Konsistenz doppelter Referenzen erhalten bleibt und das Prinzip der Lokalität unterstützt wird (siehe 9.2.5). Workflow-Unterstützung kann zur Festlegung von Teilen der Entwicklungsprozesse genutzt werden, beispielsweise für die Abfolge „Abnahme vor Freigabe“. Automatisierungsfähigkeiten der Werkzeuge (zum Beispiel mittels Skriptsprachen) können dazu genutzt werden, dass Build- und Testvorgänge für jede Produktversion sowie die Erstellung von dazugehörigen Dokumentationen automatisiert werden. Eine weitgehende oder gar vollständige Festlegung von Entwicklungsprozessen und Rollen in Workflow-Mechanismen haben sich oft als hinderlich für Flexibilität und Effektivität der Entwicklung erwiesen; die Festschreibung von Prozessen mittels Werkzeugen sollte minimal gehalten werden.

Am Markt verfügbare Werkzeuge für das Projektmanagement unterstützen meist recht einfache Modelle von Entwicklungsprozessen wie Wasserfall-Modelle. Die meisten Werkzeuge können iterativ-inkrementelle Entwicklungsprozesse nur mangelhaft abbilden und unterstützen. Eine Unterstützung bei Koordinationsaufgaben zwischen Prozessen wie Produkt- und Produktlinien-Engineering sollte nicht erwartet werden. Hierfür können einfache eigene Werkzeuge geschaffen werden, die neben dem Zugriff auf Daten anderer Werkzeuge auch zu deren Integration beitragen und einfache Auswertungen bieten. Der Autor hat in größeren Projekten mit eigenen Werkzeugen auf der Basis von Tabellenkalkulationen, mit einfachen relationalen Datenbanken sowie durch Kopplung mit Build-, Konfigurationsmanagement- und Testwerkzeugen mittels Skripten gute Ergebnisse bei geringem Aufwand erreichen können [Riebisch 1997].

Werkzeugunterstützung für Kommunikation und Koordination kann durch Werkzeuge für verteilte Teams, die Entwicklungsprozesse durch Modellierung von Rollen und Abläufen unterstützen, geleistet werden. Im Interesse von geringem Aufwand, von Einfachheit und Flexibilität sollten auch hier minimale Lösungen genutzt werden. Verschiedene Werkzeuge bieten Kopplungen zu E-Mail und ermöglichen bei der Nutzung von Script-Programmierung eine automatisierte Benachrichtigung über Änderungen, Aufträge, das Erreichen von Meilensteinen oder ähnliches. Abhängigkeiten von Elementen über Modell-Grenzen hinweg wie Traceability-Links sollten dabei automatisiert gepflegt werden. Zunehmend bieten die Werkzeuge XML-Schnittstellen, die für die Pflege solcher Abhängigkeiten genutzt werden können. Außerdem erlauben die XML-Schnittstellen einen Zugriff auf interne Daten der Werkzeuge, die für die Erstellung von Auswertungen für das Management verwendet werden können.

Für die Automatisierung von Tests stehen sehr leistungsfähige Werkzeuge zur Verfügung, die neben der Prüfung von Funktionalität auch Effizienzaussagen liefern. Unterstützung für Prüfungen an Nutzerschnittstellen, technischen Schnittstellen und im Zusammenwirken mit Datenbanken ist dabei verfügbar. Teilweise Unterstützung für Reviews und Inspektionen kann durch Prüfung von Vorgaben erfolgen, soweit sie formal bewertbar sind, wie beispielweise Namenskonventionen, Schnittstellen- und Methodenbeschreibungen oder Komplexitätskenngrößen. Soweit Dokumente formal auswertbar sind, können Vollständigkeit und Konsistenz durch Werkzeuge geprüft werden, so beispielsweise auf vollständige Umsetzung aller Anforderungen durch ein Produkt, auf Einhaltung von Konsistenzregeln einer Konfiguration oder auf Konflikte in Namensräumen eines Hyperspace. Bewertungen weiterer sehr wichtiger Merkmale wie Verständlichkeit und Einfachheit einer Architektur müssen mittels Reviews und Inspektionen erfolgen, weil dafür die kognitiven Fähigkeiten der Mitarbeiter erforderlich sind.

# 10 Ergebnisdiskussion und Evaluierung

*Die Methodik ALEXANDRIA wurde in verschiedenen Projekten und Fallstudien zur Entwicklung und Evolution von Produktlinien angewendet. Die dabei festgestellten Eigenschaften der Methodik werden in diesem Kapitel dazu genutzt, eine Bewertung von Anwendungserfolgen in Bezug auf die angestrebten Ziele vorzunehmen. Die Eigenschaften der als Bewertungsobjekte verwendeten Projekte und Fallstudien werden zwecks Einordnung vorgestellt. Die zur Bewertung verwendeten Kriterien und die dabei ermittelten Ergebnissen werden zusammengestellt.*

## 10.1 Vorgehen bei der Bewertung

Die Methodik ALEXANDRIA umfasst Methoden und Techniken, die Beiträge zur Komposition von Software leisten und die Flexibilität der Softwareentwicklung im Sinne der schnellen Durchführung von Änderungen verbessern sollen. Dabei soll die Evolution von Softwaresystemen unterstützt werden, damit deren mögliche Nutzungsdauer verlängert wird. Flexibilität und Evolution hängen dabei eng mit der Wartbarkeit der Software zusammen.

Konkrete Beiträge zu diesen Zielen leisten die entwickelten Methoden durch die Zerlegung und automatisierte Komposition von Komponenten, durch die Beschreibung von deren Abhängigkeiten, durch Reverse Engineering der Architektur existierender Systeme und durch deren Refactoring.

Für die Eignung der Methoden ist die Skalierbarkeit eine wichtige Voraussetzung. Für das Erreichen von Flexibilität und Wartbarkeit ist ein geringer Aufwand für Änderungen wichtig. Bei früher Bindung der Variabilität ist Schlankheit der Produkte Ausdruck der Verringerung von Ressourcenverbrauch und Komplexität. Für die Evolution und Langlebigkeit ist der Aufwand zum Hinzufügen zusätzlicher Features ein wesentliches Kriterium.

Zu einer Bewertung von Entwicklungsmethoden würde ein idealer Ansatz darin bestehen, für die interessierenden Merkmale quantitative Kenngrößen (Metrics) festzulegen und diese Kenngrößen dann in Messreihen von einzelnen, vergleichbaren Software-Entwicklungsaufgaben zu ermitteln. Damit ließen sich mehrere Kurven von Werten erreichen, die Vergleiche für verschiedene Methoden, Domänen und Größen von Softwareprodukten zulassen. Ein Beispiel für eine solche quantitative Kenngröße für das Merkmal Zerlegungsaufwand wäre „Durchschnittlicher Aufwand zur Zerlegung eines Systems nach 10 Features, bezogen auf eine Größe von 5000 Lines of Code (LOC)“.

Eine derartige Bewertung würde einen beträchtlichen Aufwand erfordern. Sie ist deshalb nicht praktikabel. Eine Bewertung anhand von kleinen Entwicklungsaufgaben liefert keine vergleichbaren Aussagen, weil genaue Modelle fehlen, die eine Extrapolation von Ergebnissen bei Aufgaben kleinen Umfangs auf solche größeren Umfangs erlauben. Die Untersuchungen müssen deshalb mit Aufgaben großen Umfangs durchgeführt werden, wie er typisch für die Entwicklung der meisten Produktlinien ist. Die Durchführung von Mess*reihen* mit einer Vielzahl „echter“ Entwicklungsaufgaben ist nicht möglich, weil der dafür notwendige Aufwand weder in universitären Forschungsvorhaben noch in Unternehmen der Industrie geleistet werden kann, wenn nicht außergewöhnliche Bedingungen bestehen. Deshalb sind keine kontinuierlichen Messreihen mit Entwicklungsaufgaben unterschiedlicher Größe und für verschiedene Methoden,

sondern lediglich einzelne Untersuchungen und ein qualitativer Vergleich der Ergebnisse möglich.

## 10.2 Bewertungsobjekte

Die Eigenschaften der Methodik wurden bei den hier und im Abschnitt 3.4 angeführten Projekten und Fallstudien ermittelt und zur Bewertung genutzt. Damit die Bewertungsergebnisse eingeordnet und nachvollzogen werden können, werden die Charakteristika und Kennwerte der Projekte und Fallstudien im folgenden kurz vorgestellt.

### 10.2.1 Bibliotheksverwaltungssystem

Zur Bewertung der Eignung von vorhandenen sowie eigenen Produktlinien-Methoden für Featuremodellierung und Architektur wurde eine Fallstudie durchgeführt, deren Gegenstand eine Produktlinie für Verwaltungssysteme für verschiedene Typen von Bibliotheken wie Universitätsbibliotheken, Stadtbüchereien und Fahrbüchereien ist [Böllert Philippow 2000][Philippow et al. 2002]. Bei den Verwaltungssystemen handelt sich um dialogorientierte Informationssysteme. Der Umfang beträgt 50 Klassen und ca. 4000 LOC bei 9 variablen Features. Anforderungen, Variabilität und Architektur wurden in Anlehnung an die Produktlinie LIBERO [Libero] entwickelt.

### 10.2.2 Software-Implementierung des Spiels „Siedler von Catan“

Es wurde eine Software-Implementierung des Gesellschaftsspiels „Siedler von Catan“ als Computerspiel für PC entwickelt, das die Regeln und Varianten des Spiels verwirklicht. Die Entwicklung erfolgte als Produktlinie mit den im Handel erhältlichen Varianten des Gesellschaftsspiels „Basisspiel“, „Erweiterung für 5 und 6 Spieler“, „Seefahrer-Erweiterung“, „Seefahrer-Erweiterung für 5 und 6 Spieler“ und „Städte- und Ritter-Erweiterung“. Darüber hinaus wurden einzelne Merkmale als variable Features weiter verfeinert, damit weitere Produkte erstellt werden können, wie die Features „Ereigniskarten“, „Räuber“ und „erweiterte Bauphase“. Die Software realisiert die Regeln des Spiels und ermöglicht die Spielzüge für mehrere Mitspieler über eine graphische Nutzerschnittstelle (Abb. 74, aus [Halle 2001]). Diese Produktlinie wurde im Rahmen einer Reihe von studentischen Arbeiten [Halle 2001], [Ulrich 2002], [Kleinschmidt 2002] und anderen sowie einer Dissertation [Böllert 2002] entwickelt und untersucht.

Ziele der Fallstudie waren die Entwicklung der Produktlinien-Methoden HyperFeatuRSEB und Hyper/UML sowie der Vergleich und die Bewertung von FeatuRSEB und HyperFeatuRSEB bezüglich Schlankheit, Komplexitätszuwachs durch Variabilität und Skalierbarkeit. Zu den Ergebnissen gehört darüber hinaus der (produkt-unabhängige) Produktkonfigurator (Abb. 55, S. 140). Bei Entwicklung dieser Produktlinie wurde außerdem ein Vergleich der Eignung der generativen Ansätze AspectJ und Hyper/J vorgenommen.

Bei den Produkten der Produktlinie handelt es sich um Dialogsysteme ohne Echtzeitverhalten. Die Produktlinie umfasst 42 variable Features, die unter Beachtung der Abhängigkeiten etwa 38 Millionen verschiedene Produkte erlauben. Die Implementierung mit Hyper/J umfasst 630 Klassen und ca. 32000 Zeilen Quellcode LOC; sie weist eine mittlere Komplexität und mittleren Umfang auf, verglichen mit industriellen Softwareentwicklungs-Aufgaben.

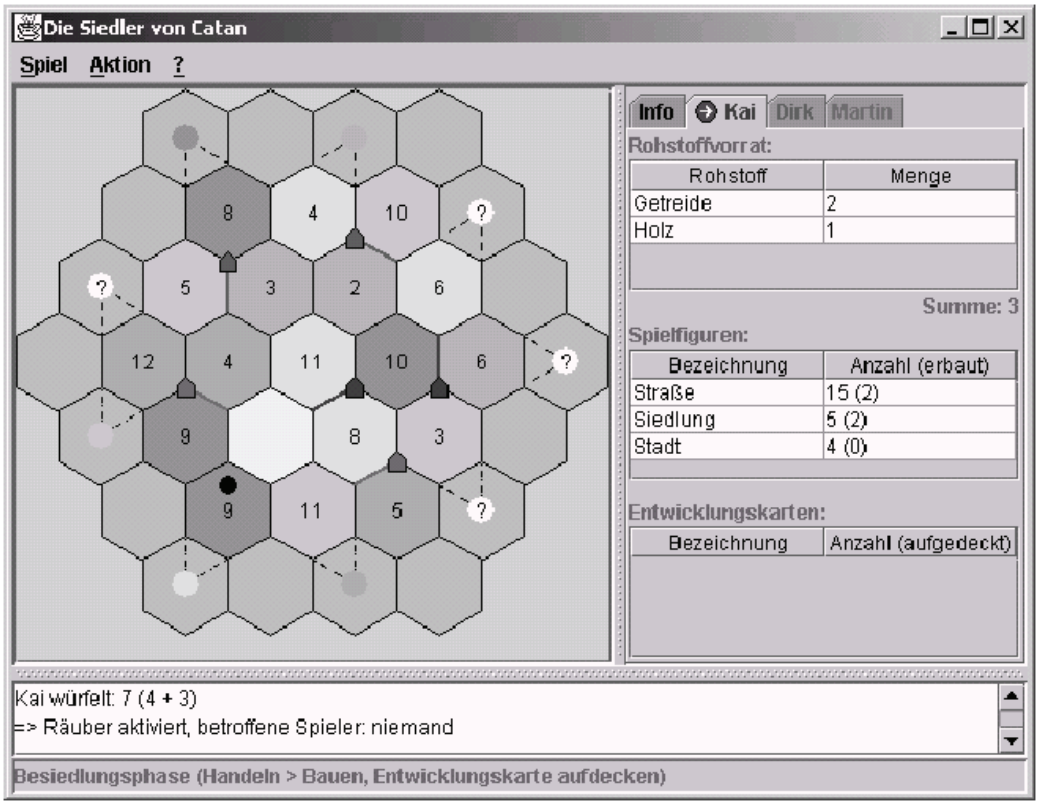



*Abb. 74: Graphische Oberfläche eines Spiel der Produktlinie "Siedler von Catan"*

### 10.2.3 Industrielles Bilderkennungssystem

Die Entwicklung der Reengineering-Methoden (Kapitel 1) wurden im Rahmen eines industriellen Projektes durchgeführt. Dabei sollten Methoden für Feature-Modellierung, Reverse Engineering und Refactoring, für die Beschreibung von Abhängigkeiten und Traceability sowie für Skalierbarkeits- und Methodenbewertung entwickelt werden. Gegenstand dieses Projekts ist die Entwicklung einer Produktlinie zur industriellen Bilderkennung im Bereich der Postautomatisierung. Diese Arbeiten wurden in Kooperation mit der Siemens Dematic GmbH Konstanz durchgeführt. Die dabei erzielten wissenschaftlichen Ergebnisse konnten veröffentlicht werden [Pashov Riebisch 2003], weitere Publikationen sowie eine Dissertation [Pashov 2003] sind in Vorbereitung.

Eine Komponentensystem mit Legacy-Charakter, dessen Variabilität durch Konfiguration steuerbar ist, ist in eine neue Produktlinie mit Komponenten umzusetzen. Sein Umfang beträgt deutlich über 200 kLOC. Die neue Produktlinie soll weitergehende Variabilität als das Vorgängersystem aufweisen. Abhängigkeiten der Features und der Komponenten werden in Modellen explizit beschrieben. Die Entwicklungsaufgabe bezieht sich auf verteilte Systeme mit teilweisen Echtzeitaufgaben, ein Schwerpunkt liegt im effektiven Zugriff auf große Datenmengen.

### 10.2.4 Digitales Video-System

Im Fachgebiet Prozessinformatik der TU Ilmenau wird im Rahmen mehrerer studentischen Arbeiten und einer Dissertation [Streitferdt 2003] eine Produktlinie Digitales Video-System [DVP] entwickelt. Da aus diesem Projekt die zur Illustration der Methodik verwendeten Beispiele stammen, wurde es bereits im Abschnitt 3.4 vorgestellt. Ziel dieser Produktlinie ist die Entwicklung und Untersuchung von softwaretechnischer Methoden in den Bereichen Anforderungs- und Domänenanalyse, Featuremodellierung,

Entwicklung von Softwarearchitekturen, Konfiguration von Laufzeit-Variabilität sowie iterativer Entwicklungsprozesse.

Die Aufgaben bei der Softwareentwicklung umfassen mehrere Domänen wie die der Informationssysteme, der Multimediadaten-Verarbeitung, der Hardware-Schnittstellen, der verteilten Datenverarbeitung und der Web-Services. Zur Produktlinie gehören Nutzerschnittstellen verschiedener Typen, von einfachen Anzeigen an Geräten über Menüdarstellungen am Fernsehgerät, Web-Schnittstellen bis hin zu graphischen Nutzerschnittstellen am PC- und PDA-Bildschirm. Die Online-Dokumentation der Produktlinie wird mit generativen Techniken erzeugt. Dazu werden Informationen aus Quelltexten, Repository und CASE-Werkzeugen miteinander verknüpft und unter Nutzung des Werkzeugs Doxygen [Doxygen] als Webseiten bereitgestellt [Preiß 2003].

Die Produktlinie umfasst Lösungen auf zwei verschiedenen Hardware-Plattformen, die unterschiedliche Charakteristika bezüglich Entwicklungsumgebung und Umfang der Ressourcen aufweisen. Videorekorder und onDemand-Server basieren auf handelsüblichen PC-Komponenten, die interaktive Fernbedienung wird von einen Palm-PDA mit eingeschränkten Ressourcen und separater Entwicklungsplattform realisiert [Dietzel 2003]. Die Produktlinien-Entwicklungsmethoden unterliegen deshalb ähnlichen Bedingungen wie bei eingebetteten Systemen.

Die Produktlinie weist einen mittleren Umfang auf. Der Quellcode für die PC-Plattform umfasst etwa 5000 LOC, wobei das Betriebssystem Linux und die Open-Source-Komponenten vdr [Schmidinger 2002] und linuxTV.org [Metzler Metzler 2002] sowie die dvb-Treiber [Metzler Metzler 2002] nicht eingerechnet sind. Dieser Teil der Produktlinie umfasst derzeit etwa 40 variable Features, die durch etwa 45 Komponenten realisiert werden. Zusätzlich zur Software für die PC-Plattform existieren für Palm-PDAs etwa 50 Klassen, die 30 Features realisieren.

### 10.2.5 Parkschein-Automaten

Eine Industriekooperation hat die Weiterentwicklung einer Produktlinie für Parkschein-Automaten-Software zum Ziel, wobei die Flexibilität erhöht und der notwendige Wartungsaufwand verringert werden sollen. Schwerpunkt des Methodeneinsatzes ist die Featuremodellierung, die Entwicklung von Software-Architekturen und die Konfiguration von Systemen mit frühem Bindungszeitpunkt der Variabilität.

Bei den Produkten der Produktlinie handelt es sich um eingebettete Systeme ohne Echtzeitforderungen. Ein Teil der Variabilität der Produkte hängt von der jeweiligen Konfiguration der Hardware ab. Aufgrund der eingeschränkten Ressourcen der Laufzeit-Umgebung bestehen Forderungen nach Schlankheit der Systeme.

Gegenstand der Untersuchung ist derzeit ein Teil der Produktlinie, der etwa 90 Features umfasst. Aufgrund der unternehmens-strategischen Bedeutung der Produktlinie wurde die Vertraulichkeit bezüglich Produktinformationen vereinbart. Die Forschungsergebnisse fließen jedoch in wissenschaftliche Publikationen ein, wie beispielweise in [Streitferdt et al. 2003].

## 10.3 Kriterien und Ergebnisse

Als Kriterien für die Eignung und Anwendbarkeit der entwickelten Methoden werden die Skalierbarkeit, der zusätzliche Aufwand für Flexibilität, die Schlankheit der Produkte als Minimierung von nicht benötigten Teilen sowie die Möglichkeit und der Aufwand bei Weiterentwicklung bewertet.

### 10.3.1 Skalierbarkeit von HyperFeatureRSEB

Skalierbarkeit ist ein wichtiges Kriterium, weil mit der Methode HyperFeatuRSEB die Grenzen der Vorgänger-Methoden FeatuRSEB und KobrA (siehe 2.1.5) in Bezug auf die maximale Größe der damit zu entwickelnden Produktlinien überwunden werden sollen.

Die Skalierbarkeit von HyperFeatuRSEB wird mit der der Methode FeatuRSEB verglichen. Bei FeatuRSEB wird die Variabilität mittels Frameworks und Design Patterns implementiert. Die Ergebnisse dieses Vergleichs sind auf einen Vergleich mit der Methode KobrA übertragbar, denn diese benutzt für die Implementierung der Variabilität ebenfalls diese Mittel.

Die Bewertung wurde bei der Entwicklung der Produktlinie „Siedler von Catan“ durchgeführt. Zunächst wurde das Basisspiel implementiert, das dann nach der Methode FeatuRSEB zu einer Produktlinie ausgebaut wurde. Dazu wurde zunächst die Spielvariante „Erweiterung für 5 und 6 Spieler“ implementiert. Dabei zeigte sich, dass bei dieser Methode das Objektmodell bei hinzukommenden Features durch zusätzliche Klassen, Operationen und Beziehungen zunehmend unübersichtlicher wird, auch wenn Vereinfachungen vorgenommen und Modelle und Architekturen genutzt werden. Die Klarheit der Lösung und damit ihre Wartbarkeit nimmt rasch ab. Als Konsequenz konnte die Entwicklung mit dieser Methode nach der Implementierung von 9 Features nicht fortgesetzt werden; eine Erweiterung um weitere Features war nicht durchführbar (siehe Tabelle 9).

Die Entwicklung des gleichen Spiels mit der hier entwickelten Methode HyperFeatuRSEB erfolgte in der gleichen Reihenfolge von Schritten und unter vergleichbaren Bedingungen bezüglich der Qualifikation der Entwickler und der Werkzeugunterstützung. Zunächst wurde das Basisspiel implementiert, danach die Spielvariante „Erweiterung für 5 und 6 Spieler“ sowie die Features der „Seefahrer-Erweiterung“, „Seefahrer-Erweiterung für 5 und 6 Spieler“ und „Städte- und Ritter-Erweiterung“. Dabei zeigt sich, dass der Zuwachs des Objektmodell-Umfangs bei einer Erweiterung um neue Features deutlich langsamer verläuft als bei FeatuRSEB (siehe Tabelle 9, aus [Böllert 2002]). Auch bei der Implementierung aller Features ergaben sich keine Probleme bei Klarheit und Wartbarkeit. Die Ursache für diesen Unterschied zwischen beiden Methoden resultiert daraus, dass HyperFeatuRSEB bei der Zerlegung überschneidungsfreie Komponenten ohne Einführung einer großen Anzahl zusätzlicher Elemente erlaubt.

Die Methode HyperFeatuRSEB liefert eine *deutlich bessere* Skalierbarkeit als die Methode FeatuRSEB. Für eine quantitative Bewertung der Skalierbarkeit müsste die Maximalgröße der Produktlinie ermittelt werden, bei der ein weiterer Ausbau nach der Methode HyperFeatuRSEB wegen gesunkener Wartbarkeit nicht mehr möglich wäre. Bei den gegebenen Vorgaben in Form von Spielregeln des Gesellschaftsspiels war eine solche Größe nicht erreichbar.

*Tabelle 9: Vergleich der mit beiden Methoden entwickelten Produktlinien*

| **Kenngröße** | **Anwendung von FeatuR-SEB** | **Anwendung von Hyper-FeatuRSEB** |
|---|---|---|
| Umfang Entwicklungsaufwand (in Personen-Monaten) | 2 | 8 |
| Anzahl Features | 9 | 42 |
| Anzahl verschiedener Produkte | 8 | 38 Millionen |
| Anzahl Use Cases | 24 | 84 |
| Anzahl UML-Diagramme | 20 | 70 |
| Anzahl Klassen | 122 | 300 |
| Anzahl Quelltext-Zeilen (ELOC) | 9100 | 18600 |

### 10.3.2 Aufwand für das Erreichen von Flexibilität

Es ist zu untersuchen, wie hoch der Aufwand für die Flexibilität der Software ist, der zusätzlich zur Implementierung der eigentlichen Anforderungen geleistet werden muss. Gegenstand der Untersuchung ist hier wieder die Produktlinie „Siedler von Catan“. Es wird ein Vergleich zwischen einer Implementierung des Basisspiels ohne Variabilität und der Implementierung als Produktlinie mit Variabilität vorgenommen.

Aufwand für die Flexibilität ist bei der ALEXANDRIA-Methodik bei der Zerlegung und bei der Komposition zu leisten. Dazu gehören die Schaffung von zusätzlichen Elementen und zusätzliche Refactoring-Aktivitäten. Die Anzahl von Elementen lässt sich gut erfassen (siehe unten), wogegen eine Trennung der Arbeitsschritte des Refactoring zwischen solchen zur Zerlegung und solchen zur Vereinfachung und Strukturierung der Lösung kaum möglich ist.

#### Zusätzlicher Aufwand für die Zerlegung

Für eine überschneidungsfreie Zerlegung muss bei der Implementierung die Funktionalität über verschiedene Klassen hinweg aufgeteilt werden. Bei Anwendung der Vorläufer-Methode FeatuRSEB erfolgt eine Aufteilung von Funktionalität durch künstliche Abstraktionen, die mittels Design Patterns und Vererbung realisiert werden. Sie lassen weder von den Bezeichnern noch von der Struktur her Rückschlüsse von Modell und Quellcode auf die zugrundeliegende (funktionale) Anforderung zu und erhöhen deshalb den Aufwand für das Verstehen durch den Entwickler drastisch.

Bei HyperFeatuRSEB erfolgt die Zerlegung von Funktionalität über Hyperslices hinweg. Damit der Entwickler eine Lösung versteht, muss er Klassen in mehreren Hyperslices analysieren und zusammenführen. Dadurch ist ein erhöhter Aufwand erforderlich. Da zusammengehörende Klassen und Elemente gleiche Bezeichner aufweisen und durch Festlegung der Integrationsstrategie in den meisten Fällen eine Komposition automatisiert möglich ist, können CASE-Werkzeuge dem Entwickler die betroffenen

Klassen mit partieller Komposition darstellen, so dass er die Lösung im unzerlegten Zustand prüfen kann. Dadurch bleibt die Erhöhung des Verständnis-Aufwands gering.

Betrachtet man den Aufwand über mehrere Entwicklungsphasen hinweg, wird eine Verlagerung des Aufwands gegenüber konventioneller Entwicklung deutlich. Zum einen erfordert die Zerlegung eine wesentlich detailliertere Anforderungsbeschreibung und der größte Teil des Aufwands fällt bei Planung, Konzeption und Entwurf der Architektur und der Hyperslices an. Zum anderen führt diese Detaillierung jedoch zu einer massiven Verringerung des Aufwands für Codierung und Test, verglichen mit konventioneller Entwicklung ohne Variabilität. Insgesamt wurde eine Erhöhung des Entwicklungsaufwands mit Zerlegung von *etwa einem Drittel* gegenüber der Entwicklung ohne Variabilität und ohne Zerlegung geschätzt [Halle 2001].

**Anzahl zusätzlicher Klassen und Abhängigkeiten**
Das Basisspiel ohne Variabilität umfasste unter Anwendung der Methode HyperFeatuRSEB 130 Klassen, die Erweiterung zur Produktlinie mit Variabilität umfasst 300 „echte“ Klassen. Zusätzlich mussten 370 Klassen in verschiedene Hyperslices eingefügt werden, damit die Forderung nach deren deklarativer Vollständigkeit erfüllt werden konnte. Dabei wurden 800 abstrakte oder leere Operationen eingefügt [Halle 2001]. Diese Ergänzungen könnten jedoch mittels Werkzeugen weitgehend automatisiert werden, so dass der dafür erforderliche Aufwand gering bleibt.

Zwischen den Ergebnissen der verschiedenen Phasen sind außerdem Verweise in Form von Traceability-Links erforderlich, damit die Komposition anhand einer Feature-Auswahl erfolgen kann. Die Anzahl dieser Links ergibt sich aus der Anzahl der Anforderungen, der Anzahl der Features sowie aus der Anzahl der Komponenten in Objektmodell und Quelltext. Bei der Produktlinie „Siedler von Catan“ sind weniger als 1000 Links erforderlich. Der erforderliche Aufwand für Erstellung und Pflege dieser Links kann durch Verwaltung in einem Repository und durch Werkzeugintegration (siehe 6.4) gering gehalten werden. Da dem Aufwand ein hoher Nutzen beim Prüfen der Auswirkungen und beim Durchführen von Änderungen gegenübersteht, würde er sich auch bei einer konventionellen Entwicklung amortisieren.

**Zusätzliche Vorkehrungen für die Komposition**
Ein zusätzlicher Aufwand für die Komposition entsteht dadurch, dass für die Komposition durch den Generator die Angabe einer Integrationsmethode für jeden Hyperslice und jedes darin enthaltene, nicht orthogonale Element erforderlich ist. Durch Vorgabe der Integrationsstrategie (siehe 6.1.2) soll dieser zusätzliche Aufwand vermieden werden. Es ist zu prüfen, inwieweit dies erfolgreich ist.

Die Wirksamkeit der Integrationsstrategie zur Vereinfachung wurde im Rahmen der Fallstudie „Siedler von Catan“ untersucht. Dabei konnte festgestellt werden, dass nur eine geringe Anzahl konkret zu definierender Integrationsmethoden notwendig war. Tabelle 10 zeigt die Ergebnisse, die sich auf eine Produktlinie mit einer Anzahl möglicher Integrationen zwischen Hyperslices größer $10^7$ bezieht [Böllert 2002]. Dabei wird deutlich, dass nur ein äußerst geringer Teil der Kombinationsmöglichkeiten zusätzlich zu definierende Integrationsmethoden erfordert. Diese Fälle konzentrieren sich auf die Verschmelzung von Operationen mit Rückgabeparametern, bei denen die Integrationsmethode Summieren statt einer automatisierten Behandlung angewendet werden muss. Durch geeignetes Refactoring wäre es allerdings möglich, für diese Fälle auch ohne Vereinigung von Operationen auszukommen [Böllert 2002][Fowler 1999].

*Tabelle 10: Anzahl einzeln definierter Integrationsmethoden zusätzlich zur Integrationsstrategie*

| Integrationsmethode | Anforderungsmodell | Objektmodell |
|---|---|---|
| Verschmelzen | 3 | - |
| Ordnen | 3 | 5 |
| Summieren | - | 30 |
| Ersetzen | 8 | 7 |
| **Summe** | **14** | **42** |

Zur Bewertung des zusätzlichen Aufwands wäre ein weiterer Vergleich der Methode HyperFeatuRSEB mit der Methode FeatuRSEB sinnvoll, damit der Zuwachs an zusätzlichen Elementen bei gleichem Umfang von Variabilität verglichen werden kann. Wie in Abschnitt 10.3.1 dargestellt wurde, war musste der Ausbau der Produktlinie „Siedler von Catan“ nach FeatuRSEB jedoch vorzeitig abgebrochen werden, so dass keine vergleichbaren Werte für die vollständige Implementierung mit FeatuRSEB vorliegen. Auch ohne einen solchen Vergleich ist jedoch eine Einschätzung anhand der Zerlegungstechniken möglich, wenn die Erfahrungen mit FeatuRSEB sowie die Bewertung des Aufwands durch zusätzliche Elemente aus den Abschnitten 2.1.5 und 2.2.3 einbezogen werden. Danach ist bei FeatuRSEB gegenüber HyperFeatuRSEB ein vielfacher Aufwand der Implementierung bei vergleichbarer Variabilität zu erwarten, der bei zunehmender Variabilität auch deutlich stärker zunimmt.

Wie zu erwarten war, erfordert die Flexibilität und Variabilität von Produktlinien einen Mehraufwand gegenüber konventionellen Softwaresystemen. Dieser Mehraufwand bewegt sich jedoch bei Anwendung der Methode HyperFeatuRSEB etwa im gleichen Rahmen wie bei der Wiederverwendung von Komponenten beim gegenwärtigen Stand der Technik. Es ist zu erwarten, dass ein Mehraufwand in dieser Höhe bei einer industriellen Anwendung der Methode akzeptiert werden kann.

### 10.3.3 Schlankheit der Produkte

Für Produktlinien, deren Produkte in einem Umfeld mit beschränkten Ressourcen eingesetzt werden, ist eine Beschränkung auf notwendige Teile erforderlich. Dieses als Schlankheit bezeichnete Merkmal ist vor allem bei Produktlinien mit früher Bindung der Variabilität anzustreben.

Mit der Methode FeatuRSEB entwickelte Produktlinien sind für Variabilität mit spätem Bindungszeitpunkt vorgesehen. Die Implementierung enthält alle variablen Bestandteile, deshalb sind keine schlanken Produkte erreichbar. Werden variable Teile in Komponenten ausgelagert, ist Schlankheit um so besser erreichbar, je besser eine überschneidungsfreie Zerlegung in feingranulare Komponenten gelingt. Erfahrungen mit der Wiederverwendung zeigen, dass feingranulare Komponenten nur für einen geringen Teil einer Lösung erreichbar sind (siehe 2.1.1).

Die Methode HyperFeatuRSEB ermöglicht die überschneidungsfreie Zerlegung und frühe Bindung von Variabilität. Bei der Entwicklung der Produktlinie „Siedler von Catan“ wurde eine Gegenüberstellung verschiedener Produkte und ein Vergleich der Anzahl enthaltener Features mit dem Umfang der Produkte vorgenommen. Tabelle 11 zeigt diese Gegenüberstellung für 8 verschiedene Produkte in Bezug auf die gesamte Produktlinie [Böllert 2002]. Beim Vergleich einzelner Produkte wird deutlich, dass sie unterschiedliche Anteile disjunkter Teilmengen der Features der Produktlinie umfassen. Das Produkt „Städte- und Ritter-Erweiterung“ weist beispielsweise bei nur 31% der

Features 79% der Klassen der Produktlinie auf, während die „Seefahrer-Erweiterung“ bei 36% der Features nur 58% der Klassen umfasst.

Diese Beispiele zeigen, dass die Methode HyperFeatuRSEB die Erstellung schlanker Produkte ermöglicht. Voraussetzung dafür ist jedoch, dass eine feingranulare Zerlegung von Anforderungen, Objektmodell und Implementierung entsprechend der Features erfolgt und dass nur wenige require-Abhängigkeiten zwischen den Features bestehen, die eine Hinzunahme zusätzlicher Features und Hyperslices in ein Produkt erfordern. Führt eine Forderung nach Schlankheit eines Produkts zu Forderungen an die Auflösung solcher Abhängigkeiten oder an die weitere Zerlegung von Features, können zu diesem Zweck Refactoring-Maßnahmen durchgeführt werden. Hinweise auf dazu geeignete Schritte sind in [Halle 2001] dargestellt.

*Tabelle 11: Gegenüberstellung verschiedener Produkte bezüglich Anzahl der Features und Umfang*

| Produkt | Anzahl der Features | Anzahl der Klassen | Umfang des Java-Bytecodes |
|---|---|---|---|
| Bezug: gesamte Produktlinie | 100% | 100% | 100% |
| Basisspiel | 13% | 52% | 44% |
| Basisspiel + Erweiterung für 5 und 6 Spieler | 21% | 53% | 46% |
| Seefahrer-Erweiterung | 36% | 58% | 52% |
| Seefahrer-Erweiterung + Erweiterung für 5 und 6 Spieler | 51% | 60% | 55% |
| Städte- und Ritter-Erweiterung | 31% | 79% | 75% |
| Städte- und Ritter-Erweiterung + Erweiterung für 5 und 6 Spieler | 46% | 80% | 77% |
| Städte- und Ritter-Erweiterung + Seefahrer-Erweiterung | 44% | 82% | 80% |
| Städte- und Ritter-Erweiterung + Seefahrer-Erweiterung + Erweiterung für 5 und 6 Spieler | 64% | 84% | 83% |

### 10.3.4 Möglichkeit der evolutionären Weiterentwicklung

Die Möglichkeit und der Aufwand für eine Weiterentwicklung wird neben der Skalierbarkeit der Methode vor allem von der Wartbarkeit der Produktlinie bestimmt. Das Merkmal Wartbarkeit bezieht sich auf die Ergebnisse des Entwicklungsprozesses [DIN 66272]. Hier sind vor allem die Untermerkmale Klarheit und Durchführbarkeit von Interesse, die den Aufwand für das Verstehen beziehungsweise für das Umsetzen einer Änderung bezeichnen.

Die entwickelte Methodik ALEXANDRIA weist verschiedene Merkmale auf, durch die der Aufwand für Änderungen und Erweiterungen verringert werden kann. Neben der Nutzung von Modellen, der Vorfertigung und flexibler, automatisierter Komposition sind dies Aspekte wie die Traceability-Links, die Integrationsstrategie, die Kapselung von Erweiterungen in separaten Hyperslices und die Erweiterung des Entwicklungsprozesses um Reengineering. Eine Architektur mit Plug-in-Schnittstellen (siehe 2.2.3)

bietet ebenfalls solche Merkmale, wurde aber lediglich als Ergänzung zur Methodik ALEXANDRIA erwähnt.

Der bei Erweiterungen von Produktlinien am häufigsten auftretende Fall ist das Einbringen zusätzlicher Features als zusätzliche Zerlegungen von existierenden Hyperslices. Die Aktivitäten der Zerlegung haben einen bedeutenden Einfluss auf eine erfolgreiche Weiterentwicklung der Produktlinie, weil sie die Voraussetzung für die Klarheit und für die Überschneidungsfreiheit der Hyperslices schaffen müssen. Von Entwicklern und Projektleitern wurde für diese Aktivitäten ein hoher Aufwand erwartet, der die Evolution behindern würde.

Der Aufwand für das Einbringen zusätzlicher Features wurde im Rahmen der Produktlinie „Siedler von Catan“ untersucht [Ulrich 2002]. Die Bewertung erfolgte anhand des Aufwands für die Erweiterung der Produktlinie um die „Seefahrer-Erweiterung“ im Vergleich zu einer Neuentwicklung ähnlichen Umfangs. Die Untersuchung ergab, dass zusätzlicher Aufwand vor allem bei den Aktivitäten

- Verfeinerung der Anforderungsbeschreibung,
- Erarbeitung einer Struktur von Hyperslices,
- Entwicklung neuer Abstraktionen, damit gemeinsame Teile unter Verwendung von *State*- und *Command*-Pattern abgetrennt werden können,
- Einfügen von abstrakten Elementen zur Herstellung deklarativer Vollständigkeit

erforderlich ist. Dieser Aufwand wurde mit dem Aufwand für die Implementierung der „eigentlichen“ Funktionalität der Erweiterung verglichen. Dabei ergab sich ein Verhältnis 1:2, der zusätzliche Aufwand umfasste nur etwa die Hälfte des geschätzten Aufwands für die (angenommene) Implementierung als eigenständige Software. Diese Bewertung deckt sich in etwa mit einer anderen Bewertung der Erweiterung der gleichen Produktlinie um die „Städte und Ritter“-Features, die in [Halle 2001] beschrieben wurde. In beiden Fällen wurde festgestellt, dass sich der zusätzliche Aufwand durch verbesserte Werkzeugunterstützung deutlich senken lässt. Der größte Einfluss der Werkzeuge wurde beim Einfügen von abstrakten Elementen für die deklarative Vollständigkeit und bei partiellen Kompositionen für die bessere Verständlichkeit festgestellt.

Die beobachtete Erhöhung des Entwicklungsaufwands um die Hälfte bezieht sich auf einen Umfang der Software von etwa 200 Klassen. Er liegt in etwa gleicher Größenordnung wie bei vergleichbaren Wartungsmaßnahmen bei konventionell entwickelter Software in der Industrie, obwohl von allen Beteiligten eine größere Erhöhung erwartet wurde. Die beobachteten Werte decken sich mit Erfahrungen des Autors aus eigenen Projekten in der Industrie und mit allgemeinen Schätzungen aus der Literatur.

Von besonderem Interesse für eine Bewertung ist die Prognose der Wartbarkeit bei größeren Systemen nach einer längeren Historie von Veränderungen an der Software. Es ist ein deutlicher Unterschied zu konventioneller Software-Entwicklung in durchschnittlichen industriellen Projekten festzustellen: bei der hier entwickelten Methode sind zahlreiche Refactoring-Maßnahmen in der Entwicklung enthalten, die zum Erhalt von Klarheit und damit von Wartbarkeit der Software führen. Der langfristige Erhalt von Wartbarkeit und die Vermeidung oder Verminderung des Effekts des Architectural Decay sind Bestandteil des entwickelten Vorgehens, Aufwendungen dafür sind auch in der Aufwandsbewertung enthalten.

Es wird eingeschätzt, dass die entwickelten Methoden zwar einen zusätzlichen Aufwand gegenüber der einmaligen Entwicklung erfordern, dass dieser Aufwand jedoch - im Vergleich zum erreichten Nutzen in Form von Flexibilität und Wartbarkeit - gering ist. Die entwickelten Methoden tragen dazu bei, eine langfristige und gleichzeitig flexible Entwicklung zu ermöglichen. Die Berücksichtigung weiterer, in Kapitel 9 genannter Aspekte verbessert das Verhältnis von Aufwand und Nutzen zusätzlich.

# 11 Ausblick

Die in dieser Arbeit vorgestellte Produktlinien-Methodik ALEXANDRIA enthält vorwiegend softwaretechnische Methoden. Für einen erfolgreichen Einsatz dieser Methodik in der industriellen Softwareentwicklung sind eine Reihe von Rahmenbedingungen erforderlich, die im Kapitel 9 behandelt wurden. Es gibt weitere Rahmenbedingungen, die hier nicht detailliert betrachtet werden konnten, wie

- Möglichkeiten der Beeinflussung psychologischer Faktoren bei der Einführung und Aufrechterhaltung der Produktlinien-Methoden,
- Methoden für das betriebliche Projektmanagement wie Abstimmung der Projektorganisation mit einer konkreten Unternehmensorganisation, Budgetbereitstellung sowie Einbindung in Planung und Risikomanagement,
- Vorgaben für die projektbezogene Organisation des Qualitätsmanagements mit Richtlinien und Prüfungen sowie deren detaillierte Integration in ein konkretes Vorgehensmodell eines Unternehmens mit seinen jeweiligen Rollen und Verantwortlichkeiten,
- die Erarbeitung erfahrungsbasierter Entscheidungskriterien für das Scoping bei der Produktentwicklung, bei der Produktlinien-Weiterentwicklung sowie für Refactoring-Entscheidungen.

Diese müssen in empirischen Studien und realen Projekten untersucht werden. Für die breite Anwendung von Produktlinien-Methoden werden Empfehlungen für erfolgreiche Lösungen auf organisatorischem und Management-Gebiet benötigt, die ähnlich den Best Practices von Qualitätsmanagement-Ansätzen wie CMM [Thaller 1993] strukturiert sein können.

Mit den vorgestellten Produktlinien-Methoden werden bei der Umsetzung in die Praxis wesentliche Fortschritte bei der Verbesserung der Wiederverwendbarkeit von Software erreicht. Weitere technologische Fortschritte stehen noch aus, wie die Beschreibung der Semantik von Komponenten - des „Was“ der Funktionalität - mit formalen Mitteln, wie etwa durch ausführbare Modelle. Solche Beschreibungen sind Ziel von Arbeiten auf dem Gebiet der formalen Spezifikationen [van Lamsweerde 2000]. Derartige Beschreibungen ermöglichten Prüfungen der Korrektheit der Komposition und der Erfüllung von Anforderungen durch eine bestimmte Auswahl von Features. Sie wären bei der Zerlegung durch Separation of Concerns sowie bei der Auflösung von Wechselwirkungen zwischen Features zur Automatisierung von Entwicklungsschritten anwendbar.

Die Weiterentwicklung der UML in der Version 2.0 läßt ebenfalls eine weitergehende Unterstützung für die Zerlegung erwarten, weil die Definition einer übergreifenden Funktionalität über Architekturelemente hinweg durch die verbesserte Integration der Modelle unterstützt wird. Für diese Modelle sind Anwendungsempfehlungen als Best Practices zu entwickeln, mit deren Hilfe dann neue Zerlegungen und die Migration vereinfacht werden können.

Gegenwärtig werden im Umfeld des Vorhabens Model-Driven Architecture MDA [MDA] wichtige Ansätze entwickelt, die dazu führen können, dass ausführbare Softwaresysteme in Zukunft aus Modellen anstatt durch manuelle Codierung erstellt werden können. Derartige Ansätze lassen ganz neue und viel weitergehende Fortschritte in

Bezug auf Flexibilität und Langlebigkeit von Softwaresystemen zu, als sie allein durch modellbasierte Komposition erreichbar sind. Die Ansätze sind mit dem Hyperspace-Ansatz so kombinierbar, dass mit ihren Mitteln Hyperslices mit Anforderungs- und Objektmodellen entwickelt werden können, aus denen die entsprechenden Hyperslices mit Quellcode generierbar sind.

Weitere Arbeiten des Autors zur modellbasierten Generierung von Testfällen haben ebenfalls eine Anreicherung von Modellen mit Informationen über die Semantik von Funktionalität und Verhalten zum Ziel. Der Nutzen für Produktlinien soll darin bestehen, dass Produkte einer Produktlinie durch generierte Testfälle mit geringem Aufwand geprüft werden können.

Die geplante und teilweise begonnene Weiterführung der Arbeiten umfasst außerdem

- die Erweiterung von Hyper/UML für Modelle des Deployment, was besonders für die Unterstützung von Inbetriebnahme und Asset-Management großer verteilter Produkte aus Produktlinien wichtig ist,
- die Bewertung der Methoden zum Erreichen von Laufzeit-Variabilität (Abschnitt 7.1) durch industrielle Fallstudien, sowie die Entwicklung von vorgefertigten Komponenten in Form von Erweiterten Kollaborationen für alle relevanten Design Patterns
- die Entwicklung von Best-Practice-Ansätzen für grob-granulares Refactoring auf Architektur-Niveau als Untersetzung der bekannten Refactoring-Empfehlungen auf Quellcode-Niveau (siehe 8.3.4), wobei auch die Anpassung der Traceability-Links unterstützt wird.

Auf dem Gebiet der Werkzeugunterstützung sind eine Reihe von Entwicklungen noch in Arbeit oder sind geplant. Dazu gehört die Erweiterung von graphischen UML-Entwicklungswerkzeugen für Hyper/UML (siehe 5.8), die Verknüpfung von Werkzeugen für verschiedene Entwicklungsphasen mittels Repository und die Verknüpfung der enthaltenen Elemente durch Traceability-Links (siehe 6.4) sowie die Integration eines Refactoring-Browsers zur Vereinfachung von Zerlegung und Migration (siehe 8.4). Dabei kann teilweise auf Werkzeuge zurückgegriffen werden, die als Ergebnisse anderer Projekte wie CAFÉ [CAFE] entwickelt worden sind. Die Bereitstellung geeigneter Werkzeuge ist für die erfolgreiche industrielle Anwendung der Methoden wichtig, weil ohne sie keine ausreichende Effektivität und Geschwindigkeit der Produktlinien-Entwicklung erreichbar ist.

## 12 Literatur


[Aksit et al. 1999] Aksit, M.; Tekinerdogan, B. ; Marcelloni, F. : Deriving Frameworks from Domain Knowledge. In [Fayad et al. 1999], S. 169-198.

[Ambler 2002] Ambler, S. W.: Agile Modeling - Effective Practices for Extreme Programming and the Unified Process. Wiley, 2002.

[AmiEddi 2002] AmiEddi 1.3 – Werkzeug zur Featuremodellierung. Als Download verfügbar unter http://www.generative-programming.org

[Argo] Aktuelle Informationen zu ArgoUML. http://argouml.tigris.org

[AspectJ] Aktuelle Informationen zu AspectJ. http://aspectj.org

[Atkinson et al. 2002] Atkinson, C., et al.: Component-based product line engineering with UML. Addison Wesley, 2002.

[Balzert 2001] Balzert, H.: Lehrbuch der Software-Technik. Spektrum, 2. Aufl. 2001.

[Basili et al. 1994] Basili, V.; Caldiera, G.; Rombach, D.: Goal/question/metric paradigm. In: J. C. Marciniak [Hrsg.]: Encyclopedia of Software Engineering. Wiley, 1994. Vol. 1, S. 528-532

[Bass et al. 1998] Bass, L.; Clements, P.; Kazman, R.: Software Architecture in Practice. Addison Wesley, 1998.

[Batory et al.1997] Batory, D.; Geraci, B.J.: Composition Validation and Subjectivity in GenVoca Generators. IEEE Transactions on Software Engineering 23 (2) February 1997, S. 67 – 82.

[Baxter 2002] Baxter, I.D.: DMS: Practical Code Generation and Enhancement by Program Transformation. Workshop on Generative Programming 2002 (GP2002) Austin (Texas), April 15, 2002. 7th International Conference on Software Reuse (ICSR7). Verfügbar unter http://www.cwi.nl/GP2002

[Beck 1999] Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley Longman, Reading/Massachusetts, 1999.

[Bednasch 2002] Bednasch, T.: Konzept und Implementierung eines konfigurierbaren Metamodells für die Merkmalmodellierung. Diplomarbeit. FH Kaiserslautern, Standort Zweibrücken, 2002.

[Boger et al. 2002] Boger, M.; Sturm, T.; Fragemann, P.: Refactoring Browser for UML. In: In: Aksit, M. et al.: Objects, Components, Architectures, Services and Applications for a Networked World. LNCS 2591, Springer 2003, S. 366-377.

[Boehm 1988] Boehm, B. W.: A Spiral Model of Software Development and Enhancement. Computer, May 1988, pp. 61-72.

[Böllert Philippow 2000] Böllert, K.; Philippow, I.: Erfahrungen bei der objektorientierten Modellierung von Produktlinien mit FeatuRSEB. In: Tagungsband 1. Deutscher Software-Produktlinien-Workshop (DSPL-1). Fraunhofer IESE, Kaiserslautern, 2000, S. 29-33.

[Böllert 2002] Böllert, K.: Objektorientierte Entwicklung von Software-Produktlinien zur Serienfertigung von Software-Systemen. Dissertation, TU Ilmenau, 2002.

[Booch 1991] Booch, G.: Object Oriented Design with Applications. Benjamin/-Cummings, 1991.

[Bosch 2000] Bosch, J.: Design and use of software architectures – Adopting and evolving a product-line approach. Addison Wesley, 2000.

[BOOTSTRAP] BOOTSTRAP Methodology and Tools for Software Process Assessment & Improvement. Bootstrap Institute, Brussels. http://www.bootstrap-institute.com/

[Brown et al. 1998] Brown, W.J. et al. : AntiPatterns – Refactoring Software, Architectures, and Projects in Crisis. Wiley, 1998.

[Buschmann et al. 1996] Buschmann, F.; Meunier R.; Rohnert, H.; Sommerlad, P.; Stahl, M., Pattern-Oriented Software Architecture: A System of Patterns. Wiley, 1996.

[CAFE] From Concepts to Application in System-Family Engineering (CAFÉ). Project ITEA 00004 of the Eureka Sigma!2023 Program. Project Homepage http://www.esi.es/cafe/

[Calder et al. 2002] Calder, M.; Kolberg, M.; Magill, M.H.; Reiff-Marganiec, S.: Feature Interaction – A Critical Review and Considered Forecast. Elsevier: Computer Networks, Volume 41/1, 2003. S. 115-141

[Cetus] Cetus Links on Links on OOA, OOD, UML – CASE Tools. http://www.cetus-links.org/

[Challa 1998] Challa, S.P.: Improving Polymorphism and Concurrency in Common Object Models. PhD. Virginia Tech, Blacksburg, USA, 1998. Online verfügbar unter http://csgrad.cs.vt.edu/~siva/diss.ps

[Clarke et al. 1999] Clarke, S.; Harrison, W.; Ossher, H.; Tarr, P.: Subject Oriented Design – Towards Improved Alignment of Requirements, Design and Code. Proc. of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA'99 , Denver, Colorado, USA. ACM, 1999. S. 325 – 339

[Clauß 2001] Clauß, M.: A proposal for uniform abstract modeling of feature interactions in UML. Proceedings FICSworkshop, 15th European Conference on Object-Oriented Programming (ECOOP'01), April 2001. S.. 21-25. Online verfügbar unter http://www.info.uni-karlsruhe.de/~pulvermu/workshops/ecoop2001/proceedings/FICS2001.pdf

[Clements et al. 1999] Clements, P.; Northrop, L.M.: A Framework for Product Line Practice. Technical Report, Software Engineering Institute, 1999.

[Coad Yourdon 1992] Coad, P.; Yourdon, E.: Object-Oriented Analysis. 2nd Ed. Yourdon Press, 1992

[Cockburn 1997] Cockburn, A.: Using Goal-Based Use-Cases. Journal of Object-Oriented Programming 10 (6) S. 56 – 62 (Nov. 1997).

[COM] Microsoft COM Technologies - Information and Resources for the Component Object Model-based technologies. http://www.microsoft.com/com/

[CORBA] OMG Common Object Request Broker Architecture. http://www.corba.org/

[Czarnecki et al. 2000] Czarnecki, K., Eisenecker, U.W.: Generative Programming. Addison Wesley, 2000.

[Dijkstra76] Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, 1976.

[DIN 66272] Informationstechnik - Bewerten von Softwareprodukten - Qualitätsmerkmale und Leitfaden zu ihrer Verwendung. DIN 66272. DIN Deutsches Institut für Normung e.V., Beuth, 1994.

[Dick 2000] Dick, J.: Rich Traceability. Telelogic AB, 2000. Online verfügbar unter http://www.telelogic.com/resources/show_article.cfm?ID=2044

[Dietzel 2003] Dietzel, R.: Konzeption und Entwicklung einer Systemfamilie für eine Universal-Fernbedienung auf Basis eines Palm-Handhelds. Diplomarbeit. TU Ilmenau, 2003. Online verfügbar unter [DVP]

[Dikel et al. 1997] Dikel, D.; Kane, D.; Ornburn, S.; Loftus, W.; Wilson, J.: Applying Software Product Line Architecture. IEEE Computer, August 1997, 49-55.

[Doxygen] Doxygen – Public Domain Software Documentation System. http://www.stack.nl/~dimitri/doxygen/

[Duden 2000] Duden – Die neue deutsche Rechtschreibung. 22. Auflage, Bibliografisches Institut, 2000.

[Dueck 2002] Dueck, G.: E-Man - Die neuen virtuellen Herrscher. Springer, 2. Auflage, 2002.

[Dumke 1992] Dumke, R.: Softwareentwicklung nach Maß. Vieweg 1992.

[DVP] Digital Video Projekt: Entwicklung eines Digitalen Videosystems als Software-Produktlinie. http://www.theoinf.tu-ilmenau.de/DVP

[Eclipse 2003] Eclipse Platform Technical Overview. White Paper. Object Technology International Inc., 2003. Online verfügbar unter http://www.eclipse.org

[EFQM] EFQM Excellence Model. Verfügbar unter http://www.efqm.org

[Eisenecker Schilling 2002] Eisenecker, U. W.; Schilling, R.: Generative Programmierung mit einem Frameprozessor. In: iX 10/2002, S. 114-121. Verfügbar unter http://www.delta-software-technology.com/neu/media/pdf/pv12007.pdf

[Erni Lewerentz 1996] Erni, K.; Lewerentz, C.: Applying Design-Metrics to Object-Oriented Frameworks. In: Proceedings of the 3rd International Software Metrics Symposium (March 25 - 26, 1996, Berlin, Germany), IEEE Computer Society Press, Los Alamitos, California, 1996. S. 64 – 74. Verfügbar unter http://www-sst.informatik.tu-cottbus.de/~wwwsst/LS-

SST/Publications/1996/1996-Applying_Design-Metrics_to_Object-Oriented_Frameworks.pdf

[ESAPS] Engineering Software Architectures, Processes and Platforms for System Families (ESAPS). ITEA project 99005 of the Eureka Sigma!2023 Program. Project Website http://www.esi.es/esaps/

[Fayad et al. 1999] Fayad, M.; Schmidt, D.C.; Johnson, R. [Eds.]: Building Application Frameworks – object-oriented foundations of framework design. Wiley, 1999.

[Fensel 1995] Fensel, D.: The Knowledge Acquisition and Representation Language KARL. Kluwer, 1995.

[Froehlich et al. 1999] Froehlich, G.; Hoover, H.H.; Liu, L.; Sorenson, P.: Reusing Hooks. In: [Fayad et al. 1999], S. 219-236.

[Fowler 1999] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.

[Gamma et al. 1996] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

[Generative] Generative Programmierung - Aktuelle Informationen. http://www.generative-programming.org

[Geppert Roessler 2001] Geppert, B.; Roessler, F.: Combining Product Line Engineering with Options Thinking. Proceedings of PLEES'01 - Product Line Engineering - The Early Steps, Erfurt, Germany 2001. Verfügbar unter http://www.research.avayalabs.com/techreport/ALR-2001-011-paper.pdf

[Goos 1994] Goos, G.: Programmiertechnik zwischen Wissenschaft und industrieller Praxis. Informatik-Spektrum 17 (1994), Heft 1, S. 11-20

[Griss et al. 1998] Griss, M.; Favaro, J.; d'Allesandro, M.: Integrating Feature Modeling with RSEB. Hewlett-Packard Comp., 1998.

[Halle 2001] Halle, M.: Fallstudie zur Entwicklung wiederverwendbarer Komponenten im Rahmen von Software-Produktlinien. Diplomarbeit. TU Ilmenau, 2001.

[HCI] Human Computer Interaction Archive - Principles, Guidelines and Standards. http://www.dcs.qmul.ac.uk/SEL-HPC/Articles/GeneratedHtml/hci.standards.html

[Hesse 1998] Hesse, W.: Vorgehensmodelle für objektorientierte Software-Entwicklung. in: R. Kneuper et al. (Hrsg.): Vorgehensmodelle für die betriebliche Anwendungsentwicklung; Teubner 1998, S. 110-135. Verfügbar unter http://www.mathematik.uni-marburg.de/~hesse/papers.html

[Harrison Ossher 1993] Harrison, W.; Ossher, H.: Subject-Oriented Programming – A Critique of Pure Objects. OOPSLA'93. ACM 1993, S. 411-428

[Hitz Kappel 2003] Hitz, M.; Kappel, G. : UML@Work - Von der Analyse zur Realisierung. dpunkt, 2. Auflage, 2003.

[Hull et al. 2002] Hull, M.E.C.; Jackson, K.; Dick, A.J.J.: Requirements Engineering. Springer, 2002.

[Hyperspace] Multidimensional Separation of Concerns: Software Engineering using Hyperspaces. http://www.research.ibm.com/hyperspace

[Ivanov 1999] Ivanov, E.: Eine Methodik für die Entwicklung und Anwendung von objektorientierten Frameworks. Dissertation, Technische Universität Ilmenau, Verlag ISLE, 1999.

[Jacobson et al. 1997] Jacobson, I; Griss, M.; Jonsson, P.: Software Reuse Architecture, Process and Organization for Business Success, acm Press, 1997

[Jakarta] Jakarta Tool Suite. http://www.cs.utexas.edu/users/schwarz

[JavaBeans] JavaBeans Specification. Sun. Verfügbar unter http://java.sun.com/products/javabeans/

[JavaDesign 1999] Java Look and Feel Design Guidelines. Version 1.0.2, Sun Microsystems, 1999. Verfügbar unter http://java.sun.com/jlf/dg/higtitle.alt.htm

[Javadoc] Sun Microsystems: Javadoc Tool Home Page, http://java.sun.com/j2se/javadoc/

[JEDI 1999] JEDI - Java Extraktion und Dissemination von Information. GMD Darmstadt, 1999. Verfügbar unter http://www.darmstadt.gmd.de/oasys

[Johnson Foote 1998] Johnson, R.E.; Foote, B.: Designing Reusable Classes. In: Journal of Object-Oriented Programming 1(5), June/July 1988, S. 39-42.

[JUnit] JUnit. http://www.junit.org

[Kang et al.1990] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A., Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[Kang et al. 1998] Kang, K., Kim, S., Lee, J., Kim, K., Shin E. and Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures, Annals of Software Engineering, 5, 1998, pp. 143-168.

[Kerievsky 2002] Kerievsky, J.: Refactoring To Patterns. Draft Version 0.16. Industrial Logic, 2002. http://industriallogic.com

[Kiczales et al. 1997] Kiczales, G., et al.: Aspect-Oriented Programming. In: Proc. 1997 European Conf. on Object-Oriented Programming (ECOOP'97). Springer, 1997. S. 220 – 242.

[Kiczales et al. 2001] Getting Started with AspectJ. Communications of the ACM (44) 10 (Oktober 2001), S. 59-65.

[Kleinschmidt 2002] Kleinschmidt, J.: Untersuchungen zu GenVoca und der Jakarta Tool Suite. Studienarbeit, Technische Universität Ilmenau, 2002.

[Kluge 2003] Kluge, R.: Integration der Aufwandsschätzung in die Requirements Engineering Phase der Systemfamilienentwicklung. Diplomarbeit, Technische Universität Ilmenau, 2003.

[Kotler Bliemel 1999] Kotler, P.; Bliemel, F.: Marketing.Management – Analyse, Planung, Umsetzung und Steuerung. Schäffer-Poeschel, 9. Auflage, 1999.

[Kubali 2003] Kubali, A.: Requirements Engineering für das DV-Projekt. Studienarbeit, Technische Universität Ilmenau, 2003.

[Libero] LIBERO - das integrierte Bibliothekssystem. LIB-IT GmbH. http://www.libit.de/html/clibero.html

[Lorentsen et al. 2002] Lorentsen, L.; Tuovinen, A.-P.; Xu, J.: Modelling Feature Interaction Patterns in Nokia Mobile Phones using Coulored Petri Nets and Design/CPN. Talk on Distributed Systems, University of Aarhus, Denmark, Februar 2002. Verfügbar unter http://www.daimi.au.dk/CPnets/workshop01/cpnpapers/Paper01.pdf

[Luckham et al. 1995] David C. Luckham, James Vera and Sigurd Meldal, Three Concepts of System Architecture, Rapide Technical Report CSL-TR-95-674, July 1995, http://pavg.stanford.edu/rapide/

[Macke 2001] Macke, S.: Generative Programmierung mit AspectJ. Diplomarbeit. FH Kaiserslautern, 2001. Online verfügbar unter [Generative].

[Mayrhauser Wang 1999] von Mayrhauser, A.; Wang, J.: Fault Architecture as an Indicator of Architectural Problems. Proc. Int. Conference on Software Engineering (ICSE ’99) - Workshop on Software Change and Evolution, Los Angeles May 17, 1999. Techn. Report No. 05.99.061; Università degli Studi del Sannio, Benevento, Italien; http://www.dur.ac.uk/~dcs1elb/csm/sce99/sce99.pdf

[Mens et al. 2003] Mens, T.; Demeyer, S.; Du Bois, B.; Stenten, H.; Van Gorp, P.: Refactoring - Current Research and Future Trends. In: Electronic Notes in Theoretical Computer Science, 2003. Verfügbar unter http://www.cis.uab.edu/info/faculty/bryant/ldta2003/3.pdf

[Meyer 2000] Meyer, T.: Dynamic Semantics Negotiation in Distributed and Evolving Software Systems: Towards Automated Semantic-Directed System Configuration. Dissertation, Universität Essen. Logos Verlag, Berlin, 2000.

[Naumann 2001] Naumann, S.: Reverse Engineering von Entwurfsmustern. Diplomarbeit. Technische Universität Ilmenau, 2001.

[Neighbors 1980] Neighbors, J. N.: Software Construction Using Components. Dissertation, University of California at Irvine, 1980.

[MDA] OMG Model Driven Architecture. http://www.omg.org/mda/

[Mellis et al. 1996] Mellis, W.; Herzwurm, G.; Stelzer, D.: TQM der Softwareentwicklung - Mit Prozeßverbesserung, Kundenorientierung und Change Management zu erfolgreicher Software. Vieweg 1996.

[Meta 2002] Meta Group: Vergleich zwischen Änderungen des Business-Modells und der Bereitstellung entsprechender IT-Anwendungen. Zitiert in: Computer Zeitung (26) 2002, 24.06.2002, S. 1

[Metacase] Metaedit+. http://www.metacase.com

[Metzler Metzler 2002] Metzler, R.; Metzler, M.: The linuxTV.org Project. 2002. Verfügbar unter http://www. linuxTV.org

[Mölders et al. 1998] Mölders. A.; Fengler, W.; Wolf, M.; Philippow, I.: Object-Oriented Modelling of Business Processes with an Object Process Net based on the Unified Modelling Language (UML). in: Proceedings of the Workshop „Workflow Management: Net-Based Concepts, Techniques and Tools“ within the XIX. International Conference on Application and Theory of Petri-Nets, Lissabon, 1998, pp. 22-39.

[Moriconi et al] Moriconi, M.: SADL ; SRI. http://www.csl.sri.com/sadl

[Müller et al. 2000] Müller, H.A.; Jahnke, J.H.; Smith, D.B.; Storey, M.-A.; Tilley, S.R.; Wong, K.: Reverse Engineering - A Roadmap. In: Finkelstein, A. (Ed.): The Future of Software Engineering. ACM Press 2000. Online verfügbar unter http://www.cs.ucl.ac.uk/staff/A.Finkelstein/fose/finalmuller.pdf

[Nuseibeh et al. 2000] Nuseibeh, B.; Easterbrook, S.: Requirements Engineering: A Roadmap. 2000. Online verfügbar unter http://www-dse.doc.ic.ac.uk/~ban/pubs/sotar.re.pdf

[Oestereich 1999] Oestereich, B.: Objektorientierte Softwareentwicklung – Analyse und Design mit der Unified Modeling Language. Oldenbourgh, 4. Auflage, 1999.

[Ossher et al. 2001] Ossher, T.; Tarr, P.: Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In: Software Architectures and Component Technology. Kluwer Academic Publishers, 2001. Kap. 10.

[OTW24] Object Technology Workbench OTW 2.4, http://www.otwsoftware.com

[Parnas 1972] Parnas, D.L: On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM, Vol. 15, No. 12, December 1972, pp. 1053 – 1058

[Pashov 2003] Paschov, I.: Migration Within Long-Life Software Systems. Manuskript zur Dissertation, TU Ilmenau, 2003.

[Pashov Riebisch 2003] Paschov, I., Riebisch, M.: Using Feature Modeling for Program Comprehension and Software Architecture Recovery. In: Amendment to Proceedings 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS'03), Huntsville Alabama, USA, April 7-11, 2003. IEEE Computer Society, 2003.

[Philippow et al. 2002] Philippow, I.; Böllert, K.; Streitferdt, D.; Riebisch, M.: Methodical Aspects for the Development of Software Product Lines. In: Proc. 2nd WSEAS Int. Conf. On Information Science and Applications (ISA’02) Cancun, Mexico, May 12-16, 2002. WSEAS Press, 2002, S. 1391-1396.

[Philippow et al. 2003] Philippow, I.; Streitferdt, D.; Riebisch, M.; Naumann, S.: An Approach for Reverse Engineering of Design Patterns (eingereicht zur Veröffentlichung)

[Poseidon 2002] Poseidon UML Tool, http://www.gentleware.com

[Pree1999] Pree, W.: Komponentenbasierte Softwareentwicklung mit Frameworks. dpunkt, 1999.

[Preiß 2003] Preiß, A.: Systemfamilienbasierte Dokumentation des Digital Video Projektes. Studienarbeit. TU Ilmenau, 2003.

[PLP-SEI] The Product Line Practice (PLP) Initiative. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 2003. http://www.sei.cmu.edu/plp/

[Rajlich Bennet 2000] Rajlich, V.T.; Bennet, K.H.: A Staged Model for the Software Life Cycle. Computer, July 2000 S. 66 – 71.

[Reeng-SEI] Engineering Practices: Reengineering. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 2003. http://www.sei.cmu.edu/reengineering/

[Riebisch 1993] Riebisch, M.: Wiederverwendung von prozeßnaher Software auf der Basis halbformaler Beschreibungen. Dissertation. TU Ilmenau, Fakultät Informatik und Automatisierung, Dezember 1993.

[Riebisch 1997] Riebisch, M.: Ganzheitliches Qualitäts- und Projektmanagement mit objektorientierten Technologien. Eingeladener Vortrag zur Vollversammlung der SAQ-Fachgruppe Qualitätssicherung von Software, 24.02.1997, Olten, Schweiz. SAQ, 1997.

[Riebisch et al. 1999] Riebisch, M.; Franczyk, B.: Evolutionary Development of Frameworks – from Projects to System Families IDPT 1999, Kusadasi, Turkey, June 27th – July 2nd, 1999. in: M.M.Tanik, A. Ertas [Eds.]: IDPT 1999. Society for Design and Process Science, 2000, S. 13. ISSN 1090-9389

[Riebisch et al. 2000] Riebisch, M.; Böllert, K.; Streitferdt, D.; Franczyk, B.: Extending the UML to Model System Families. IDPT 2000, Dallas, Texas, USA, 5.-8- Juni 2000. in: M.M.Tanik, A. Ertas [Eds.]: IDPT 2000. Society for Design and Process Science, 2000, S. 13. ISSN 1090-9389

[Riebisch et al. 2001a] Riebisch, M.; Streitferdt, D.; Philippow, I.: Feature Scoping for Product Lines. In: Proceedings PLEES 2001 - Intl. Workshop on Product Line Engineering – The Early Steps. Erfurt, 13. September 2001. Fraunhofer IESE, 2001. Online-Publikation unter http://www.iese.fhg.de/pdf_files/iese-050_01.pdf

[Riebisch et al. 2001b] Riebisch, M.; Philippow, I.: Evolution of Product Lines Using Traceability. ACM Conference on on Object-Oriented Systems, Languages and Applications (OOPSLA) 2001, Workshop on Engineering of Complex OO Systems for Evolution. October 15th, 2001, Tampa Bay, Florida. Online-Publikation unter http://www.dsg.cs.tcd.ie/ecoose/oopsla2001/papers.shtml

[Riebisch et al. 2002a] Riebisch, M.; Böllert, K.; Streitferdt, D., Philippow, I.: Extending Feature Diagrams with UML Multiplicities. 6th World Conference on Integrated Design & Process Technology (IDPT2002), Pasadena, CA, USA; June 23 - 27, 2002.

[Riebisch et al. 2002b] Riebisch, M.; Philippow, I.; Götze, M.: UML-Based Statistical Test Case Generation. In: Aksit, M. et al.: Objects, Components, Architectures, Services and Applications for a Networked World. LNCS 2591, Springer 2003, S. 394-411.

[Riebisch 2003] Riebisch, M.: Materialien zur Vorlesung Software-Qualitätssicherung für Studiengänge Wirtschaftsinformatik, Informatik, Ingenieurinformatik. TU Ilmenau, 2003.

[Riebisch Böllert 2003] Riebisch, M.; Böllert, K.: Feature-driven Composition of Software Systems
Using UML. Informatica, 2003 (eingereicht)

[Roberts 1999] Roberts, D.B.: Practical Analysis For Refactoring. PhD Thesis, University of Illinois, Urbana, IL, USA, 1999.

[Royce 1970] Royce, W.: Managing the development of large software systems. In Proceedings WESTCON, San Francisco, CA, USA, 1970.

[Royce 1998] Royce, W.: Software Project Management – a Unified Framework. Addison-Wesley, Reading, MA, 1998.

[Sametinger Riebisch 2002] Sametinger, J.; Riebisch, M.: Evolution Support by Homogeneously Documenting Patterns, Aspects and Traces. 6th European Conference on Software Maintenance and Reengineering. Budapest, Hungary, March 11-13, 2002 (CSMR 2002) . Computer Society Press, 2002. S. 134-140.

[Sametinger 97] Sametinger, J.: Software Engineering with Reusable Components Springer, 1997.

[Sametinger 2002] Sametinger, J.; Keller, R.: Compositional Design Reuse. Proc. CACIC 2002, VIII Argentinean Conference on Computer Science, Universidad de Buenos Aires, Argentina, October 15-18, 2002. Verfügbar unter http://www.swe.uni-linz.ac.at/publications/pdf/TR-SE-02.08.pdf

[Schmidinger 2002] Schmidinger, K.: vdr Project Homepage. 2002. Verfügbar unter http://www.cadsoft.de/people/kls/vdr

[Shaw et al. 1996] Shaw, M.; Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, April 1996.

[Simonyi 1995] Simonyi, C.: The death of computer languages, the birth of Intentional Programming. Tech. Rep. MSR-TR-95-52, Microsoft Research, 1995.

[Simos et al. 1996] Simos, M. et al.: Organization Domain Modeling (ODM). Guidebook, Version 2.0. Report STARS-VC-A025/001/00, Lockheed Martin, 1996.

[Streitferdt et al. 2000] Streitferdt, D.; Böllert, K.; Riebisch, M.: Featuremodell und Architektur einer Systemfamilie. In: Tagungsband 45. Internat. Wissenschaftl. Kolloquium IWK'2000, TU Ilmenau, 2000. S. 621 – 626.

[Streitferdt et al. 2003] Streitferdt, D., Riebisch, M., Philippow, I.: Formal Details of Relations in Feature Models. In: Proceedings 10th IEEE Symposium and Workshops on Engineering of Computer-Based Sys-

tems (ECBS'03), Huntsville Alabama, USA, April 7-11, 2003. IEEE Computer Society Press, 2003. S. 297-304

[Streitferdt 2003] Streitferdt, Detlef: Family Oriented Requirements Engineering. Manuskript zur Dissertation, TU Ilmenau, 2003.

[Systä 1999] Systä, T. : On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software. In Proc. of the 6th Working Conference on Reverse Engineering (WCRE99), Atlanta, Georgia, USA, October 1999, pp.304-313. Verfügbar unter http://www.cs.tut.fi/~tsysta/papers/systa.pdf

[Tarr et al. 1999] Tarr, Peri; Ossher, Harold; Harrison, William; Stanley, Sutton M. Jr.: N Degrees of Separation – Multi-Dimensional Separation of Concerns. ICSE'99 Los Angeles CA, ACM, 1999. S. 107-119

[Tarr et al. 2002] Tarr, P.; Ossher, H.: Hyper/J User and Installation Manual. 2002. Online verfügbar unter [Hyperspace]

[Thaller 1993] Thaller, G.E.: Qualitätsoptimierung der Software-Entwicklung : das Capability Maturity Model (CMM). Braunschweig [u.a.]: Vieweg, 1993.

[Tracz 1987] Tracz, W. (Ed.): Software Reuse - emerging technology. Computer Society Press, 1987.

[Ulrich 2002] Ulrich, D.: Fallbeispiel zur Anwendung des Hyperspace-Ansatzes und Hyper/J. Studienarbeit. TU Ilmenau, 2002.

[UML 2001] Object Management Group: Unified Modeling Language Specification, Version 1.4. http://www.omg.org, 2001

[V-Modell] Das V-Modell - Planung und Durchführung von IT-Vorhaben - Entwicklungsstandard für IT-Systeme des Bundes. Allgemeiner Umdruck Nr. 250: Vorgehensmodell. Verfügbar unter http://www.v-modell.iabg.de/

[van Lamsweerde 2000] van Lamsweerde, A.: Formal specification - a roadmap. Proceedings of the conference on The future of Software engineering, 2000 , Limerick, Ireland. ACM Press, 2000, S. 147 - 159

[Voget et al. 2000] Voget, S.; Angilletta, I.; Herbst, I., Lutz, P.: Behandlung von Variabilitäten in Produktlinien mit Schwerpunkt Architektur. Proceedings 1. Deutscher Produktlinien-Workshop (DSPL-1), Kaiserslautern, November 2000. IESE-Report No. 076.00/E, Fraunhofer IESE, Kaiserslautern; S. 23-28.

[Walker et al. 1998] Walker, R.J.; Murphy, G.C.; Freeman-Benson, B.; Wright, D.; Swanson, D.; Isaak, J.: Visualizing Dynamic Software System Information through High-level Models. Proceedings of the Conference on Object-Oriented Programing, Systems, Languages, and Applications (Vancouver, British Columbia, Canada; 18–22 October 1998), ACM SIGPLAN, pp. 271–283, 1998. Published as ACM SIGPLAN Notices, 33(10), October 1998.

[Wallmüller 2001] Wallmüller, E.: Software-Qualitätssicherung in der Praxis. Hanser, 2. Auflage, 2001.

[Weide et al. 1995] Weide, B.W.; Heym, W.D.; Hollingsworth, J.E.: Reverse Engineering of Legacy Code Exposed. Proceedings ICSE ’95, ACM, 1995, S. 327-331. Verfügbar unter http://www.cis.ohio-state.edu/rsrg/documents/95WHH.pdf

[WeissLai 1999] Weiss, D.M.; Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison Wesley, 1999.

[Wielinga et al. 1992] Wielinga, B.J.; Schreiber, A.T.; Breuker, J.A.: KADS – A modelling Apporach to knowledge engineering. Knowledge Acquisition 4(1), March 1992, S. 5-53.

[Wirfs-Brook Johnson 1990] Wirfs-Brook, R.J.; Johnson, R.E.: Surveying current research in object-oriented design. In: Communications of the ACM 33(9): S. 104-124, 1990.

[Zave 1999] Zave, P.: FAQ Sheet on Feature Interaction. AT&T, 1999. http://www.research.att.com/~pamela/faq.html

[Züllighoven et al. 1998] Züllighoven, H. et al., Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz, dpunkt, 1998.