

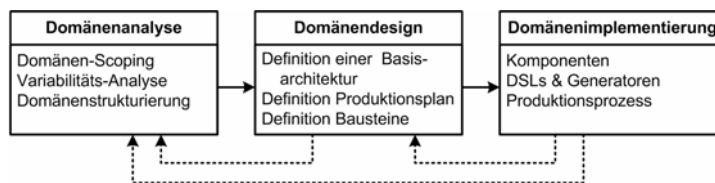
## 8.2 Anwendungsszenario: Product Line Engineering

Dieses Anwendungsszenario beschäftigt sich mit der Konzeption und Realisierung von Produktlinien. Es konkretisiert die allgemeinen architektonischen Tätigkeiten aus Abschnitt 8.1 hinsichtlich Product Line Engineering. Bevor jedoch diese Tätigkeiten näher diskutiert werden, werden zunächst einige Begriffe und Konzepte im Zusammenhang mit Product Line Engineering eingeführt und definiert. Ferner ist es wichtig festzuhalten, dass eine Produktlinie nur dann sinnvoll erstellt werden kann, wenn Erfahrungen mit mindestens zwei Produkten in einer Domäne vorhanden sind. Eine Organisation kann also kein Product Line Engineering in einer für sie unbekanntenen Domäne betreiben.

### Inhalt und Aufbau

Abbildung 8.2-1 zeigt die wichtigsten Elemente des Product Line Engineering. Die gestrichelten Pfeile sollen zum Ausdruck bringen, dass die einzelnen Tätigkeiten Inkremente darstellen und iterativ wiederholt werden, um die Produktlinie inkrementell zu verfeinern.

### PLE-Tätigkeiten



**Abb. 8.2-1:** Tätigkeiten des Product Line Engineering.

Die PLE-spezifischen Tätigkeiten können gemäß Tabelle 8.2-1 den allgemeinen, architektonischen Tätigkeiten aus Abschnitt 8.1 zugeordnet werden.

**Tab. 8.2-1:** Zuordnung von PLE-Tätigkeiten zu allgemeinen, architektonischen Tätigkeiten.

PLE-Tätigkeit	Allgemeine Tätigkeit
Domänenanalyse	Verstehen der Anforderungen
Domänenendesign	Entwerfen der Architektur
Domänenimplementierung	Umsetzen der Architektur

Beispiele dieses Vorgehens finden sich übrigens in der Fallstudie „Eingebettete Komponenteninfrastrukturen“ in Kapitel 11.

### 8.2.1 Produktlinien und Software-Systemfamilien

#### Was ist eine Produktlinie?

Eine Produktlinie besteht aus einer Menge von fachlich aufeinander abgestimmten Einzelprodukten mit einem gemeinsamen Zielmarkt – d.h., die Zusammenstellung ist kundengruppenspezifisch. Die einzelnen Produkte einer Produktlinie können zueinander in verschiedenen Beziehungen stehen:

- > Die Produkte können sich gegenseitig ergänzen (Beispiel: Microsoft Word, Excel, Access).
- > Sie können ein ähnliches oder gleiches Problem für verschiedene Umgebungen lösen (Beispiel: Opera für PC, Mac, PocketPC [OPER, 05]).
- > Oder sie können auch aus primär marketingtechnischen Gründen miteinander in Beziehung stehen (Beispiel: IBM Websphere Familie [IBMWS, 05]).

#### Produktlinien und Software-Systemfamilien

Es ist zunächst einmal essenziell festzuhalten, dass eine Produktlinie zunächst aus *Sicht der Anwender* Sinn machen muss. Die einzelnen Produkte müssen nicht zwangsläufig technisch Dinge gemeinsam haben. Natürlich ist Letzteres aus verschiedenen Gründen wünschenswert (weswegen man dieses bei Product Line Engineering auch meist implizit voraussetzt). Ist dies der Fall, so spricht man von einer Software-Systemfamilie.

Software-Systemfamilien stellen also eine technische Basis zur effizienten Umsetzung von Produktlinien dar. Deshalb wird im Kontext von Product Line Engineering (PLE) auch oft implizit davon ausgegangen, dass die Produkte einer Produktlinie eine Software-Systemfamilie darstellen. Auch wir werden dies im Rest dieses Anwendungsszenarios tun.

#### Weiterführende Literatur

Aus Platzgründen kann dieses Anwendungsszenario nur einen Überblick über PLE darstellen. Weitere Details finden sich in [Clements und Northrop 2001], [Weiss und Lai 1999], [Bosch 2000] und [Eisenecker und Czarnecki 2000].

#### Definition: Software-Systemfamilie

Der Begriff Software-Systemfamilie ist folgendermaßen definiert:

We consider a set of programs to constitute a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members [Parnas 1976].

Kern des Vorgehens ist also, den Fokus der Software-Entwicklung nicht auf einzelne Anwendungen bzw. Systeme zu legen, sondern Familien von ähnlichen Systemen zu betrachten.

Solche Ähnlichkeiten betreffen meist entweder die Architektur des Systems oder eine gemeinsame Fachlichkeit (bzw. beides: die Erfahrung zeigt, dass bestimmte Fachlichkeiten meist auch bestimmte Architekturen nahe legen). Das Fachgebiet, für welches eine solche Software-Systemfamilie entwickelt wird, wird als Domäne bezeichnet.

**Fachlich vs. technisch**

Natürlich sind die betreffenden Mitglieder der Familie nicht identisch – sonst müsste man ja bildlich gesprochen nur „die CD kopieren“. Stattdessen unterscheiden sich die Systeme an wohl definierten Stellen in wohl definierter Hinsicht. Man nennt diese auch Variabilitäten – die Analyse dieser Variabilitäten im Rahmen einer Domäne ist ein wichtiger Bestandteil von Product Line Engineering und wird weiter unten näher erläutert. Ein weiterer wichtiger Aspekt ist die systematische Wiederverwendung. Gemeinsam genutzte Artefakte sollen möglichst unverändert zwischen den verschiedenen Produkten wieder verwendet werden können.

**Variabilitäten**

**Architektur-Aspekte**

Produktlinien und Software-Systemfamilien sind besonders aus architektonischer Sicht interessant:

- > Wenn eine gemeinsame Architektur eine ganze Produktlinie für eine bestimmte Domäne tragen können soll, muss die Architektur natürlich eine gewisse Reife aufweisen.
- > Weiterhin muss die Architektur auch die betreffenden Variabilitäten unterstützen können, muss also die betreffenden Hooks (Stellen, an denen die produktspezifische Funktionalität eingehängt werden kann) bzw. Konfigurationsoptionen zur Verfügung stellen.
- > Auf der anderen Seite reift eine Architektur auch durch die Erfahrung, die man im Rahmen der Entwicklung der

**Reife**

**Flexibilität**

**Erfahrung**

verschiedenen Produkte der Produktlinie im Lauf der Zeit sammelt.

### **Beispielhafte Variabilitäten von Software-Systemfamilien**

Software-Systemfamilien (SSF) lassen sich bzgl. verschiedenster Aspekte eines Systems definieren. Bevor wir einige Beispiele dafür betrachten, möchten wir noch erwähnen, dass man nicht nur eine Reihe ganzer Systeme als SSF sehen kann, sondern natürlich auch Subsysteme oder Komponenten. SSF bzw. ein Produktlinienansatz lassen sich auf den verschiedensten Granularitätsstufen sinnvoll einsetzen (siehe Kapitel 4).

In welcher Hinsicht können sich nun die einzelnen Mitglieder einer SSF unterscheiden? Hier einige Beispiele:

#### **Umgebung**

> Umgebung: Systeme (mit identischer Funktionalität) können für verschiedene Umgebungen entwickelt werden. Solche „Umgebungen“ umfassen bspw. das Betriebssystem, die Datenbank, den Applikationsserver oder das Widget-Kit. Verschiedene Programmiersprachen sind eher unüblich (dies wird durch MDSD möglicherweise etwas realistischer, siehe unten).

#### **Skalierung**

> Skalierung: Skalierbarkeit ist eine interessante Variabilität. Dabei geht es nicht darum, dass ein *fest gefügtes* System mit steigendem Nutzer- bzw. Datenaufkommen skalieren soll. Die Variabilität besteht viel mehr darin, dass man das System für verschiedene Skalierungen unterschiedlich realisiert (bspw. mit/ohne Applikationsserver in der J2EE-Welt).

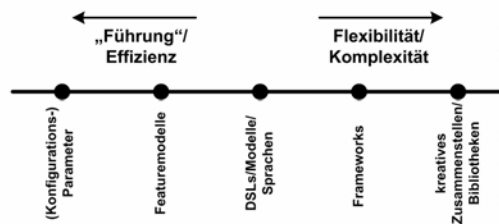
#### **Fachlichkeit**

> Fachlichkeit: Die „klassische“ Variabilität besteht in ähnlichen, aber nicht identischen Fachlichkeiten. Bankanwendungen sind beispielsweise immer mehr oder weniger gleich aufgebaut; sie unterscheiden sich fachlich vor allem in den Prozessen bzw. Kontenarten, die unterstützt werden sollen. Andere Beispiele sind z. B. Zündsteuergeräte für verschiedene Motoren.

### **8.2.2 Realisierungstechnologien und Zusammenhang zu MDS**

Offensichtlich muss man im Rahmen von Produktlinien eine Software-Architektur vorsehen, die die verschiedenen Variabilitäten

gebührend unterstützt. Krzysztof Czarnecki hat die verschiedenen Möglichkeiten anhand Abbildung 8.2-2 illustriert.



**Abb. 8.2-2:** Flexibilität vs. Effizienz.

Am ganz rechten Ende, das bedeutet maximal flexibel, aber auch maximal komplex, finden sich Systemfamilien, die schlicht aus einer Sammlung von Bausteinen bestehen (Komponenten, Klassen, Libraries etc.). Es ist Aufgabe des Produktarchitekten, aus diesen Bausteinen ein Produkt zu erstellen. Natürlich werden die Bausteine technisch kompatibel sein (Klassen alle in derselben Sprache, Komponenten basierend auf derselben Komponenteninfrastruktur), aber es existiert keine weitere Führung, wie die Bausteine zu einem Produkt zusammengesetzt werden sollen.

### **Kreatives Zusammenstellen**

Weiter links auf der Skala finden sich daher Frameworks. Frameworks definieren die prinzipielle Architektur eines Produktes der Produktlinie und definieren wohl definierte Hooks, an denen der Produktentwickler Produktspezifika „einhängen“ kann. Aufgrund der Tatsache, dass die Architektur bereits durch das Framework vorgegeben ist, ist das Zusammensetzen eines Produktes bereits etwas einfacher.

### **Frameworks**

Frameworks haben aber immer noch das Problem, dass sie (zumindest wenn sie eine bestimmte Größe überschreiten) recht schwer benutzbar sind – letztendlich stellen sie eine Form von White-Box Wiederverwendung dar. Um diese Situation zu entschärfen, kann man nun domänenspezifische Modellierungssprachen (DSLs – Domain-Specific Languages) auf das Framework aufsetzen (siehe Abschnitt 6.6.3). Anstatt dass der Entwickler nun manuell gegen das Framework implementiert, werden ihm zum Framework passende Modellierungssprachen zur Verfügung gestellt. Dadurch wird der Entwickler besser geführt, die

### **Domänenspezifische Sprachen**

Verwendung des Frameworks wird erheblich vereinfacht. Natürlich wird auch die Flexibilität etwas eingeschränkt.

**Zusammenhang zu MDS**

An dieser Stelle wird auch der Zusammenhang zu modellgetriebener Entwicklung deutlich: Die Verwendung von DSLs, um – spezifisch für eine Plattform – Anwendungen zu definieren, ist genau das Hauptmerkmal von MDS. Damit ist MDS *ein* Mittel zur Implementierung von Produktlinien.

**Feature-Modellierung**

Noch weiter links auf dem „Flexibilitätsstrahl“ finden sich Feature-Modelle – dazu später mehr. Kurz zusammengefasst stellen Feature-Modelle eine effiziente Möglichkeit dar, aus einer Menge von Konfigurationsoptionen eine erlaubte, sinnvolle Untermenge zu selektieren und damit ein Produkt der PL zu spezifizieren. Feature-Modelle eignen sich schlecht zur Definition von strukturellen Modellen, weswegen sie weiter links liegen als DSLs.

**Konfigurationsparameter**

Schlussendlich finden sich am ganz linken Ende die klassischen Konfigurationsparameter. Wenn die Konfigurationsmöglichkeiten der Produkte ausschließlich aus einer vordefinierten Menge von Konfigurationsparametern bestehen, so hat man natürlich nur noch sehr begrenzte Variabilitäten.

**Wann verwendet man was?**

**Evolution „von rechts nach links“**

Es stellt sich die Frage, wann man was verwenden soll. Unserer Erfahrung nach wird man auf dem Strahl ganz rechts beginnen und mit zunehmender Erfahrung bzgl. der Produkte der SSF weiter nach links wandern: Je reifer die Domäne, desto weiter links auf dem Strahl finden sich die Konfigurationsmöglichkeiten. Zu dieser Daumenregel gibt es aber durchaus Ausnahmen: Die Ausdrucksfähigkeit von Konfigurationsparametern ist in der Praxis oft nicht groß genug, weswegen man für strukturelle Beschreibungen bei der DSL und für strukturfreie bei Feature-Modellen hängen bleibt.

Man kann also sagen, dass die Produkte in guten Software-Systemfamilien mittels DSLs und Feature-Modellen beschreibbar sind.

### 8.2.3 Erstellen des Business Case

In diesem Abschnitt möchten wir kurz auf die Vorteile und Herausforderungen im Zusammenhang mit Produktlinien und deren Erstellung eingehen, die bei der Erstellung eines Business Case berücksichtigt werden können. In diesem Fall geht es wohlgerne nicht um den Business Case für ein einzelnes System, sondern um eine ganze Reihe von Systemen innerhalb einer Software-Systemfamilie.

Produktlinien machen immer dann Sinn, wenn eine Organisation mehr als ein „Produkt“ in einer Domäne entwickelt. Die Verwendung von Softwaresystemfamilien zur Implementierung bringt in diesem Zusammenhang eine ganze Reihe von Vorteilen:

- > Die Architektur wird qualitativ hochwertiger, da sie im Laufe vieler Anwendungen in der gleichen Domäne optimiert wird.
- > Durch die Vereinfachung der Anwendungsentwicklung („Programmierung“ mittels DSLs) wird die Entwicklung von Produkten innerhalb der Produktlinie erheblich beschleunigt.
- > Durch die Wiederverwendung von nicht-variablen Bestandteilen und die explizite Beschreibung von Variabilitäten steigt die Qualität des Systems.
- > Außerdem lassen sich Domänenexperten besser in die Entwicklung integrieren.

#### Vorteile

Natürlich sind auch Produktlinien keine Silver Bullets. Der Aufbau einer Produktlinie kostet Zeit und Geld; das Einarbeiten in Techniken und Tools ist nicht umsonst. Die Umstellung des Entwicklungsprozesses und der Organisation ist oft der härteste Brocken – dazu später mehr.

#### Herausforderungen

### 8.2.4 Verstehen der Anforderungen (Domänenanalyse)

Im ersten Schritt sollte man sich zunächst über die Abgrenzung der Domäne Gedanken machen und vor allem klar festlegen, was man nicht mehr als Teil der Domäne – und damit der späteren Produktlinie – sehen möchte.

#### Abgrenzung der Domäne

Wenn wir als Beispiel Motorsteuergeräte (für Automobile) annehmen, so ist es beispielsweise entscheidend, festzulegen, ob

diese nur für Benzin- oder auch für Dieselmotoren verwendbar sein sollen oder ob sowohl PKW- als auch LKW-Motoren dazugehören.

## Glossar und Ontologie

Um die Domäne nun weiter zu verstehen, ist es eine gute Idee, ein Glossar oder eine Ontologie der Konzepte der Domäne zu erstellen. Dies ist ein erster Schritt hin zu einem Domänenmetamodell (siehe Abschnitt 6.2.5.3), welches die Konzepte der Domäne formal beschreibt und diese miteinander in Beziehung setzt. Ein gutes Domänenmetamodell ist die Basis für erfolgreiches Product Line Engineering.

## Architektur-Metamodell

Falls man die Domäne nahe an die technische Architektur der zu erstellenden Systeme anlehnt, spielt das Domänenmetamodell die Rolle des Architektur-Metamodells. Die dadurch entstehende Formalisierung der Architektur ist ein nützliches Mittel zu deren Verbesserung!

## Variabilitätsanalyse

Im weiteren Verlauf ist es nun essenziell, die Variabilitäten innerhalb der Domäne zu erfassen. Für die Anteile, die in allen Produkten identisch sind, müssen entsprechend qualitativ hochwertige Bausteine zur Verfügung gestellt werden. Für die Dinge, die unterschiedlich sind, muss eine geeignete Konfigurations-/Beschreibungsmöglichkeit gefunden werden.

## Feature-Modelle

Es hat sich herausgestellt, dass sich Feature-Modellierung [Eisenecker und Czarnecki 2000 sowie FODA, 05] sehr gut zur Variabilitätsanalyse eignet. Ein Feature-Modell – bzw. ein Feature-Diagramm als seine grafische Repräsentation – beschreibt die Variabilitäten einer SSF, und zwar *frei von Realisierungsaspekten!*

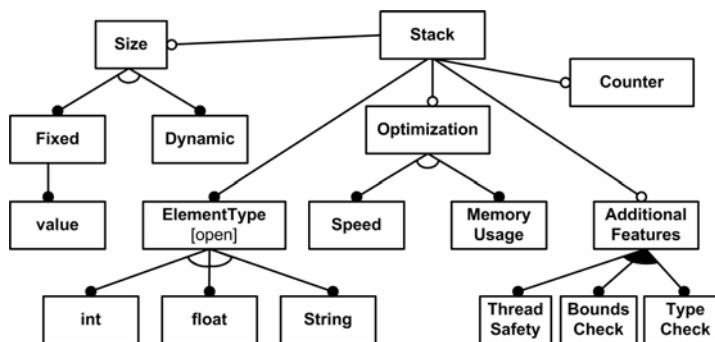


Abb. 8.1-3: Feature-Modell der Systemfamilie Stack.

Abbildung 8.2-3 zeigt das Feature-Modell für eine Systemfamilie Stack. Es beschreibt, dass jeder *Stack* einen *ElementType* haben muss („muss“: dargestellt durch den ausgefüllten Kringel). Dieser kann vom Typ *int*, *float* oder *String* sein (*1-aus-n*: erkennbar durch den nicht-ausgefüllten Bogen zwischen den Assoziationen *ElementType-int*, *ElementType-float* und *ElementType-String*). Die Größe des Stacks kann entweder fix (wobei man dann einen Wert für die Größe angeben muss) oder dynamisch anpassbar sein. Optional (*optional*: leerer Kringel) kann der Stack einen statischen *Counter* mitführen – tut er das nicht, wird beim Aufruf von *size()* die Größe jedes Mal neu berechnet. Weitere Features sind Thread-Sicherheit, Bounds-Checking sowie Typsicherheit. Eines oder mehrere dieser Features können in einem Produkt vorhanden sein (*n-aus-m*: erkennbar durch den ausgefüllten Bogen). Auch kann die Implementierung entweder auf Geschwindigkeit oder Speicherverbrauch hin optimiert werden.

## Beispiel Feature-Modell

Das Diagramm in Abbildung 8.2-3 beschreibt damit den „Konfigurationsraum“ für Mitglieder der Systemfamilie *Stack*. Einzelne Mitglieder müssen gültige Kombinationen aufweisen. Beispiele:

- > Dynamische Größe, ElementType: int, Zähler, Threadsafe
- > Statische Größe mit dem Wert 20, ElementType: String
- > Dynamische Größe, geschwindigkeitsoptimiert, Bounds-Check

## Beispiele für gültige Konfigurationen

Abgesehen von den in der grafischen Notation direkt darstellbaren Angaben kann ein Feature-Modell noch weitere Informationen enthalten, dies werden wir hier aus Platzgründen jedoch nicht näher erläutern.

Das Interessante an dieser Methode und Notation ist, dass sie absolut nichts über die spätere Implementierung der Features aussagt. Würde man eine Systemfamilie in dieser Phase bereits mittels UML modellieren, dann müsste man bereits zu diesem frühen Zeitpunkt Entscheidungen treffen bzgl. Vererbung, Typgenerizität, Assoziationen etc. Dies ist zu dieser Zeit der Domänenanalyse aber weder notwendig noch hilfreich.

## Frei von Implementierungsaspekten

Des Weiteren können Merkmale in obigen Diagrammen zu verschiedenen Kategorien gehören, die das Feature-Modell nicht unterscheidet:

## Aspektuelle Features

- > Klassische Komponentenmerkmale geben das Vorhandensein modularer Features an – d. h. ob ein Produkt einen bestimmten Systembaustein besitzt oder nicht.
- > Aspectuelle Merkmale (siehe Abschnitt 6.2.6) sind im Gegensatz dazu solche, die nicht als Komponente realisiert werden können (z. B. das Merkmal „optimiert auf Performance“ oder „optimiert auf Codegröße“). Solche Merkmale schlagen sich später möglicherweise an vielen Stellen im System nieder, indem verschiedene Komponenten anders implementiert werden.

Feature-Diagramme können beide Arten von Merkmalen nebeneinander aufweisen.

### 8.2.5 Entwerfen der Architektur (Domänen-Design)

#### Implementierung der Plattform

Im Rahmen des Domänen-Designs geht es nun darum, die Softwarestruktur der Domäne festzulegen. Beginnen wird man damit, die gemeinsamen Merkmale der Produkte einer Domäne in Form einer Plattform zu implementieren. Dadurch, dass diese gemeinsamen Merkmale eben bei allen Produkten identisch sind, ist es nicht nötig, sie in irgendeiner Weise generativ zu implementieren. Sie stellen die Basis für die gemeinsame Zielarchitektur dar.

#### Bindungszeiten

Für die variablen Merkmale (in denen sich die verschiedenen Produkte unterscheiden) muss nun entschieden werden, wann ein Merkmal „gebunden“ werden soll – d. h. zu welchem Zeitpunkt man sich bei einem Produkt für oder gegen ein bestimmtes Merkmal entscheidet. Dabei gibt es verschiedene Alternativen:

#### Quellcode

- > Auf Quellcodeebene: Hier wird die Entscheidung für oder gegen ein Feature bereits im Rahmen der Programmierung festgelegt, also im Quellcode „fest verdrahtet“.

#### Übersetzungszeit

- > Zur Übersetzungszeit: Dem Compiler können bestimmte Entscheidungen überlassen werden (Beispiele: überladene Funktionen, Präprozessoren, Code/Aspect Weaver)

#### Linkzeit

- > Zur Link-Zeit: Auch der Linker kann durch Hinzubinden der einen oder anderen Bibliothek dazu dienen, ein Produkt zu konfigurieren (Beispiel: ein Makefile, welches bestimmte Bibliotheken statisch zum Code bindet – oder auch nicht).

- > Zur Installationszeit: Bei Produkten, die einen expliziten Deployment-Schritt enthalten, kann man typischerweise auch hier noch bestimmte Konfigurationen vornehmen (Beispiel: J2EE bietet mittels der Deployment-Deskriptoren die Möglichkeit, bestimmte Einstellungen bei der Installation anzupassen).
- > Zur Ladezeit: Auch beim Laden der Anwendung kann man noch eingreifen, DLLs sind ein Beispiel (Beispiel: das optionale Laden verschiedener DLLs, die die gleichen Funktionen unterschiedlich implementieren).
- > Zur Laufzeit: Zu guter Letzt kann man Entscheidungen natürlich auch zur Laufzeit treffen (Beispiele: Polymorphismus, Dynamic Classloading in Java, interpretierte Konfigurationsparameter).

## **Installationszeit**

## **Ladezeit**

## **Laufzeit**

All diese verschiedenen Möglichkeiten haben ihre Vor- und Nachteile bzgl. Performance, Flexibilität, Codegröße und Komplexität. Beispielsweise sind Dinge, die zur Laufzeit mittels Polymorphismus entschieden werden, sehr flexibel, führen aber auch zu großen Images. Generell kann man sagen, dass je früher im Prozess eine bestimmte Entscheidung getroffen wird, desto performanter und kleiner wird das System sein. Spätere Entscheidungen führen zu erhöhter Komplexität des Codes, größeren Images und meist schlechterer Performance.

## **Vor- und Nachteile**

Auch hier ist wieder eine interessante Verbindung zu MDSD zu sehen. MDSD erlaubt Flexibilität (auf Modellebene), die dann, wenn der Code des Systems einmal generiert ist, aber aus Sicht des Systems statisch realisiert wird. Insofern kann MDSD helfen, die Probleme, die man sich mit erhöhter Flexibilität einhandelt, zu minimieren.

Ein weiterer Aspekt des Domänen-Designs ist die Festlegung der gemeinsamen Architektur der Produkte der PL. Wie oben bereits erläutert, muss diese Architektur in der Lage sein, die Produktlinie „zu tragen“, muss also die Variabilitäten der verschiedenen Produkte abbilden können.

## **Gemeinsame Architektur**

Schlussendlich bleibt noch, den Produktionsprozess zu definieren. Es muss entschieden werden, welche variablen Anteile wie spezifiziert werden. Frameworks, DSLs, Feature-Modelle oder Konfigurationsparameter müssen festgelegt und ihre Auswirkungen

## **Produktionsprozess**

auf die Variabilitäten definiert werden. Vor allem bei der Frage, wie man gute DSLs baut, ist einiges an Erfahrung nötig.

### **8.2.6 Umsetzen der Architektur (Domänenimplementierung)**

Im Rahmen der Domänenimplementierung wird nun das vorher Definierte in die Tat umgesetzt. Komponenten und Frameworks werden implementiert, Editoren für DSLs werden gebaut und Generatoren bzw. Interpreter konfiguriert bzw. programmiert. Hier kommen dann in erster Linie wieder die Techniken der modellgetriebenen Software-Entwicklung zum Einsatz (siehe Abschnitt 6.2.5).

#### **Intelligente IDEs**

Es sei noch erwähnt, dass es neben DSLs und Frameworks noch andere Möglichkeiten gibt, die Entwicklung von Software-Systemfamilien für die Entwickler zu vereinfachen, und zwar insbesondere „intelligente“, domänenspezifische IDEs. Damit sind hier jetzt nicht nur die Editoren für die DSLs und deren Integration in IDEs wie Eclipse gemeint, sondern darüber hinausgehende Features wie zum Beispiel:

- > Verschiedene Perspektiven/Konfigurationen der IDE für die verschiedenen Sichten auf eine Produktlinie, spezifisch für die mit dieser Sicht betrauten Entwickler
- > Speziell dafür angepasste Richtlinien, Muster und Dokumentationen, die den Entwickler in seine Arbeit führen
- > Wizards, die bestimmte wiederkehrende Aufgaben übernehmen, und schlussendlich
- > Aufgabenbasierte IDEs die, abhängig vom Fortschritt der Entwicklung und anderer Informationen in der IDE, den Entwickler auf die nächsten anstehenden Aufgaben hinweisen

Ein Beispiel für die kombinierte Anwendung all dieser Konzepte sind Microsoft's Software Factories [Greenfield und Short 2004].

### **8.2.7 Rollen und Aufgaben**

Um Produktlinien sinnvoll umsetzen zu können, muss natürlich auch die Struktur der zugrunde liegenden Organisation passen.

Dieser Abschnitt spezialisiert also die Aussagen aus Kapitel 7 im Zusammenhang mit PLE.

Makroskopisch muss zunächst zwischen der Entwicklung der Produktlinieninfrastruktur (also all den wieder verwendbaren Artefakten sowie der Implementierung des Produktionprozesses) und der Entwicklung einzelner Produkte basierend auf dieser Infrastruktur unterschieden werden. Es ist dabei zielführend, die beiden Stränge personell zu trennen. Da die Produktentwicklung die Infrastruktur der Produktlinie verwenden soll, ist es essentiell, dass die beiden Stränge eng koordiniert werden. Aus Sicht der Produktlinie sind die Anwendungsentwickler die „Kunden“! Eine iterative Weiterentwicklung, Feedback sowie ggfs. der Austausch von einzelnen Mitarbeitern zwischen den beiden Strängen sind dabei sehr wichtig.

Im Folgenden seien noch einige der für PLE charakteristischen Rollen erläutert; es versteht sich, dass nicht jeder Rolle zwangsläufig von einer separaten Person durchgeführt werden muss:

- > Domänenexperten sind für die Analyse der Domäne zuständig, identifizieren Variabilitäten und Gemeinsamkeiten.
- > Der Domänenarchitekt kümmert sich um die Implementierung der Infrastruktur für die Produktlinie. Dies umfasst die u. a. die Erstellung von Frameworks, DSLs und Generatoren. Bei großen Projekten können diese Dinge möglicherweise eigenständige Rollen darstellen.
- > Aufseiten der Anwendungsentwicklung gibt es nun natürlich den Anwendungsexperten, der mit dem Kunden die Anforderungen für eine Anwendung im Rahmen der Produktlinie definiert.
- > Der Anwendungsarchitekt ist zuständig dafür, die Anwendung basierend auf der Produktlinie umzusetzen. Seine Aufgabe ist es insbesondere auch, neue Features bei der Produktlinieninfrastruktur einzufordern, wenn dies für die Entwicklung von Produkten nötig ist.

## **Trennung Produktlinien- und Anwendungsentwick- lung**

## **Rollen**

### **Domänenexperten**

### **Domänenarchitekt**

### **Anwendungsexperte**

### **Anwendungsarchitekt**