

# Variantenmanagement im Kontext von MDSD

Markus Völter, [voelter@acm.org](mailto:voelter@acm.org), [www.voelter.de](http://www.voelter.de)

**Product-Line Engineering, also das Bestreben, Familien von Softwaresystemen zu erstellen erfordert kontrolliertes Verwalten von Variabilitäten zwischen den verschiedenen Produkten der Produktlinie. Wie kann man nun Variantenbildung (beschrieben bspw. durch Featuremodelle) sinnvoll mit klassischen Modellgetriebenen Ansätzen verbinden? Wie beschreibt man Varianten von Modellen? Dieser Artikel soll dazu einige theoretische Anregungen geben, und auch zeigen wie man das Problem praktisch angehen könnte.**

## Product Line Engineering

Im Rahmen von Product-Line Engineering ist das Ziel, verwandte Systeme möglichst effizient zu erstellen. Verwandt sind Systeme dann, wenn sie eine Reihe Merkmale gemeinsam haben, sich allerdings in wohl definierter Art und Weise von einander unterscheiden (man nennt eine Gruppe solcher Systeme dann auch eine Softwaresystemfamilie). Mittels modellgetriebener Softwareentwicklung [SV05] lassen sich solche Produktlinien sehr effizient umsetzen: Dinge, die in allen gleich sind implementiert man als Teil der Plattform, die Unterschiede werden entweder modelliert und dann automatisch generiert, oder, wenn das nicht praktikabel ist, von Hand implementiert.

Nun ist die Frage, wie sich die verschiedenen Systeme voneinander unterscheiden. Aus meiner Sicht gibt es zwei grundlegend verschiedene Arten, wie sich Systeme unterscheiden können: strukturelle Unterschiede, und nicht-strukturelle. Im Folgenden findet sich ein Beispiel für jede Art von Unterschied.

## Strukturelle Unterschiede

Angenommen, Sie haben ein System, welches sie modellgetrieben entwickeln. Dann haben Sie ja in aller Regel ein fest definiertes Metamodell, beispielsweise das in Abbildung 1 gezeigte, extrem einfache Beispiel.

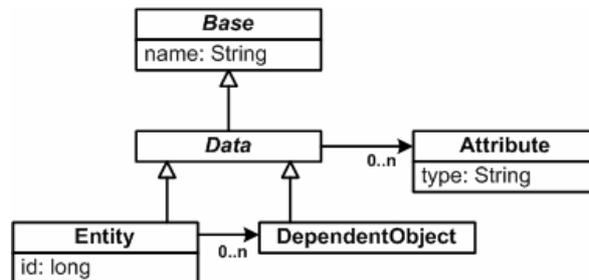


Abbildung 1: Beispielmetamodell

Sie können basierend auf diesem Metamodell nun eine ganze Reihe von Anwendungen beschreiben (und daraus dann selbstverständlich generativ implementieren) die sich eben darin unterscheiden, was für Entitäten, Dependent Objects, Attributes, etc. sie haben. In Abbildung 2 sind zwei verschiedene Anwendungen (bzw. deren Modelle) abgebildet. Sie unterscheiden sich offensichtlich in ihrer Struktur, sind jedoch trotzdem „verwandt“, da sie aus denselben „Arten von Bausteinen“ bestehen – nämlich die durch das gemeinsame Metamodell definierten.

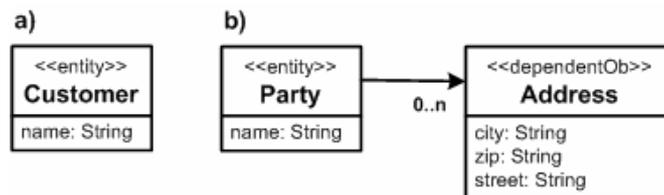


Abbildung 2: Zwei verschiedene Beispielanwendungen

Man beachte, dass man zur Modellierung solcher strukturierter Unterschiede oft auf „Box-and-Line“-Editoren zurückgreift, im Beispiel oben wird UML verwendet.

### Nicht-Strukturelle Unterschiede

Angenommen, Sie möchten Stacks beschreiben. Stacks unterscheiden sich in vielerlei Hinsicht, allerdings *nicht in Ihrer Struktur*. Sie können unterschiedliche Größen haben, unterschiedliche Elementtypen, eine Optimierung der Implementierung auf Geschwindigkeit oder Größe. Im Prinzip lassen sich nicht-strukturelle Variabilitäten immer beantworten mit der Frage: hat das System dieses oder jenes Feature, oder hat es es nicht. Zur Beschreibung solcher Variabilitäten eignen sich Featuremodelle recht gut (siehe z.B. [CE00]).

Abbildung 3 zeigt ein Beispiel. Es beschreibt, dass jeder *Stack* einen *ElementType* haben muss („muss“: dargestellt durch den ausgefüllten Kringle). 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.

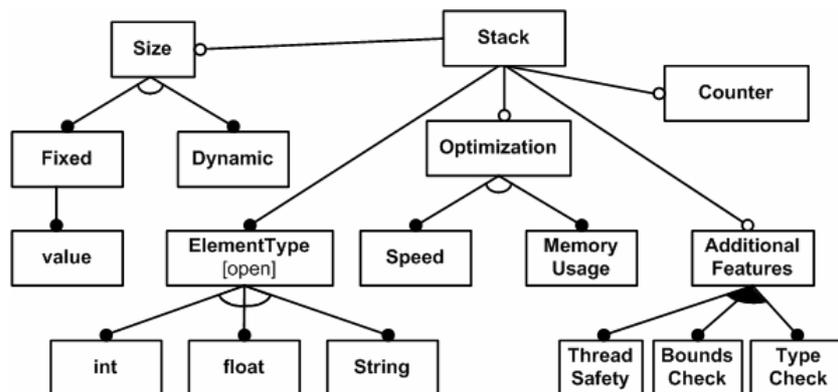


Abbildung 3: Beispiel-Featuremodell für Stacks

Im Prinzip stellt das Featuremodell in Abbildung 3 auch eine Art Metamodell für alle gültigen Konfigurationen dar (eine bestimmte Konfiguration ist dann ein Modell, eine Instanz der Metamodells). Es 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, Geschwindigkeits-optimiert, Bounds-Check

### Kombination Struktureller und Nicht-Struktureller Unterschiede

Interessant wird es nun, wenn man die beiden Arten von Unterschieden kombiniert. Folgendes Beispiel. Angenommen, Sie haben eine fachliche Produktlinie für Geschäftsanwendungen. In solchen Anwendungen spielen „Parteien“ immer eine zentrale Rolle – als Kunde, Vertragnehmer, Angestellter, etc. Je nach Produkt in Ihrer Produktlinie unterscheiden sich die Merkmale diese Parteien – allerdings in wohl geordneter Art und Weise. Natürlich könnten Sie basierend auf dem Metamodell Abbildung 1 für jedes Produkt eine neue Datenstruktur „Party“ definieren. Das ist aber sehr aufwendig. Was Sie

eigentlich wollen ist Folgendes: Sie wollen die (nicht-strukturellen) Unterschiede eines vorgegebenen strukturierten Modells beschreiben, also beispielsweise zum Ausdruck bringen,

- Ob eine Party eine oder mehrere Adressen haben soll
- Ob Sie Telekontakte mit abspeichern wollen oder nicht
- Ob sie beim Telekontakt die Landesvorwahl mit ablegen möchten
- Ob eine Adresse ein Feld *state* haben muss oder nicht (USA!)
- Ob die Datenstrukturen persistent sein sollen, und wenn ja, wie (XML, JDO, Hibernate)

Letztendlich wollen Sie also Varianten von Modellen beschreiben. Genau dies werde ich im Folgenden anhand eines praktischen – und wie üblich mittels openArchitectureWare [OAW] implementierten – Beispiels zeigen.

## Das Ziel

Zunächst stellen wir die Varianten als Featuremodell dar. Der obere Teil von Abbildung 4 zeigt das Ergebnis. Desweiteren definieren wir das strukturelle Modell einer Partei. Dieses Modell enthält alle Modellelemente, die eine Party überhaupt jemals haben kann (also die Obermenge aller Varianten). Durch Verbindung der Modellelemente mit dem Feature von dem seine Existenz abhängt, bringen wir zum Ausdruck, dass ein bestimmtes Modellelement nur vorhanden sein soll, wenn ein bestimmtes Feature selektiert ist. Also:

- Wenn das Feature *NeedsState* nicht selektiert ist, wird das Attribut *state* aus dem Modell von *Address* entfernt. Entsprechend verfahren wir mit dem *countryCode* bei *Phone*.
- Die zu-eins Assoziation von *Party* zu *Address* ist nur vorhanden, wenn das Feature *MultipleAddresses* nicht vorhanden ist. Genau umgekehrt ist die zu-n Assoziation nur genau dann vorhanden.

Aus dem so modifizierten Modell wird dann – mit den üblichen Mitteln – Code generiert. Je nach Selektionen im Featuremodell wird also anderer Code generiert – ggfs. samt Persistenzanbindung.

Was wir also tool-mäßig brauchen ist folgendes:

- Wir müssen das (allumfassende) UML Modell einlesen,
- eine Instanz des Featuremodells einlesen,
- und dann alle Modellelemente aus dem UML Modell entfernen, deren assoziierte Features im Featuremodell nicht selektiert sind.

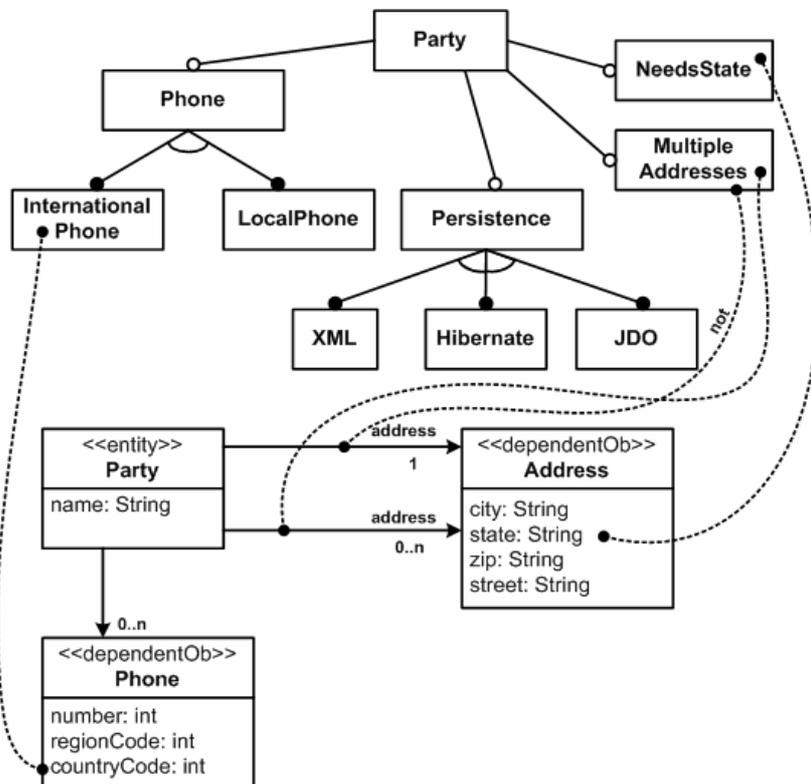


Abbildung 4: Variabilitäten von "Party"

## Umsetzung mit oAW

### UML Modell definieren

Wir beginnen zunächst mit dem „malen“ des UML Modells. Unter Verwendung von Poseidon (wir könnten auch irgendein anderes der von oAW unterstützten UML Werkzeuge verwenden) sieht das dann wie in Abbildung 5 aus. Vorher müssen wir natürlich das Metamodell (hier bestehend aus *Entity* und *DependentObject*) mit den üblichen oAW-Mitteln definieren (also entweder von Hand implementieren oder mittels des Metamodellgenerators generieren).

Zu beachten ist die Annotation der Modellelemente mit dem Feature, von dem sie abhängen. Ich habe mich hier entschieden, den (eindeutigen) Namen des Features als Teil des Namens des Modellelements zu annotieren. Die Attribut *state* in Abbildung 5 hat aus Sicht des UML Tools den Namen *state [NeedsState]*. Der Generator parst das entsprechend auseinander und setzt den Namen wieder auf *state*. Es gäbe in UML auch andere Möglichkeiten, zum Beispiel Tagged Values. Diese werden aber nicht von allen Tools im

Diagram angezeigt. Pragmatisch gesehen ist das Vorgehen, das Feature als Teil des Namens zu notieren das praktikabelste. Was also wie ein Hack aussieht, ist in Wirklichkeit eine einfache, wohldefinierte und elegante Möglichkeit das Problem im Rahmen der Beschränkungen von UML zu lösen (und demonstriert dabei die Stärken des flexiblen Ansatzes von openArchitectureWare)

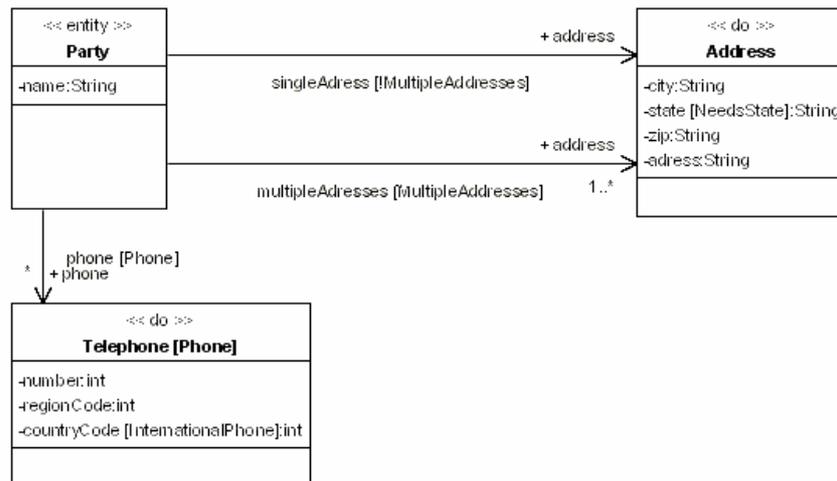


Abbildung 5: Das "allumfassende" Modell in Poseidon

### Featuremodell definieren

Nun müssen wir das Featuremodell beschreiben. Als Featuremodellierungswerkzeug verwenden wir pure::variants, [PS] dessen Modelle von oAW verarbeitet werden können. Das Featuremodell stellt sich dann entsprechend Abbildung 6 dar. Daraus generieren wir mit dem pure::variants Metamodell Generator (ein Modul für openArchitectureWare) die Metaklassen, die das Featuremodell repräsentieren.

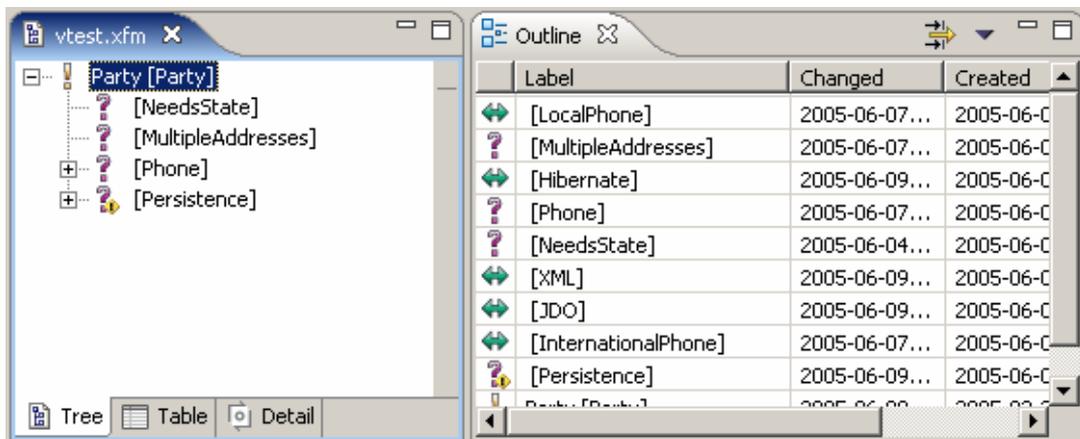


Abbildung 6: Featuremodell beschrieben mittels pure::variants

### Variante definieren

Nun definieren wir eine oder mehrere Varianten (siehe Abbildung 7). Dafür liefert das pure::variants Eclipse Plugin einen entsprechenden Editor, in dem man basierend auf dem vorher definierten Featuremodell nun eine Variante „zusammenklicken“ kann. Man beachte, dass der Editor natürlich die durch das Featuremodell definierten Constraints überprüft und nur gültige Variantenmodelle erlaubt. Natürlich kann man mehrere Varianten definieren und diese unter verschiedenen Namen ablegen und weiterverarbeiten. Die in Abbildung 7 dargestellte heißt *a1*.

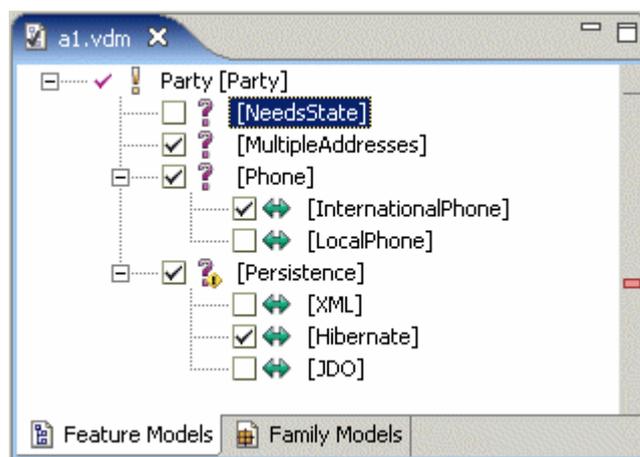


Abbildung 7: Eine beispielhafte Variante

## Modellmodifikation und Generierung

Nun starten wir den Generator. Er ist mittels Plugins so konfiguriert, dass er das UML lädt (dafür konfigurieren wir einen entspr. *XMLInstantiator*) und das Variantenmodell. Der für letzteres nötige Code ist im Folgenden dargestellt. Außerdem deployen wir den sogenannten *VariantModificationMM*, ein *ModelModifier*, der das UML Modell aufgrund der Features im Featuremodell modifiziert.

```
public class VariantPlugin extends GeneratorPlugin {

    public List contributeInstantiators() {
        String modelFile = getProperty("VARIANT.MODEL.FILE");
        PureVariantsVMInstantiator pvi = new
            PureVariantsVMInstantiator( modelFile1,
                new FmFeatureModel("Variant1") );
        return makeList( pvi );
    }

    public List contributeModelModifiers() {
        return makeList( new VariantModificationMM() );
    }
}
```

Das interessante an der Sache ist nun eben genau dieser *ModelModifier*; den möchten wir uns daher etwas genauer anschauen. Wir müssen – wie bei jedem *ModelModifier* – die Operation *modifyModel* implementieren. Dort suchen wir zunächst nach einem instantiierten Featuremodell:

```
public class VariantModificationMM implements ModelModifier {
    public void modifyModel(MetaEnvironment metaEnv) {
        BaseFeatureModel fm = (BaseFeatureModel)
            MMUtil.findSingleInstance( metaEnv,
                BaseFeatureModel.class, "no feature model found." );
    }
}
```

Dann iterieren wir über alle Modellelemente, und rufen für jedes *ModelElement* (*ModelElement* ist die Basismetaklasse aller UML Modellelemente) die Operation *handleElement* auf.

```
ElementSet allElements =
    MMUtil.findAllInstances(metaEnv);
for ( ...allElements.iterator(...) ) {
    Element element = (Element) iter.next();
    if (element instanceof ModelElement ) {
        handleElement((ModelElement)element, fm);
    }
}
```

Dort parsen wir mittels ganz normaler String-Verarbeitung einen eventuell in eckigen Klammern stehenden String heraus und setzen den Namen auf den Wert *ohne* den Teil in eckigen Klammern.

```
private void handleElement(ModelElement element,
                           BaseFeatureModel fm) {
    // parse stuff within brackets, assign this to the
    // variable featureName and set the model element's name
    // to the value without the bracket text.
    handleElementAndFeature(element, featureName, fm);
}
```

Nun kommt das eigentlich spannende. Zunächst überprüfen wir, ob das erste Zeichen des Featurenamens ein Ausrufezeichen ist. Wenn ja, dann bedeutet das logische Negation, das bedeutet, dass wir das betreffende Modellelement löschen müssen gerade wenn das Feature da ist - und nicht andersrum, wie im Default-Fall.

```
private void handleElementAndFeature(ModelElement element,
                                     String featureName, BaseFeatureModel fm) {
    boolean not = false;
    if ( featureName.startsWith("!") ) {
        not = true;
        featureName = featureName.substring(1);
    }
}
```

Dann versuchen wir die Klassen zu resoven, die das betreffende Feature repräsentiert. Der pure::variants Metamodell Generator generiert eine entspr. API. Wird keine Klasse gefunden, so ist der verwendete Featurename ungültig und ein Fehler wird ausgegeben.

```
Class c = fm.getClassByName(featureName);
if ( c == null ) {
    Checks.error(element, "Cannot find a feature "+
                  "named "+featureName );
    return;
}
```

Schlussendlich suchen wir eine Instanz des Features, und wenn wir keine finden, sowie *not* nicht gesetzt ist (also kein Ausrufezeichen im Featurenamen stand) entsorgen wie das betreffende Modellelement durch Aufruf von *dispose*. Eine entspr. Warnung wird ausgegeben. Das wars.

```
BaseFeature f = (BaseFeature)MMUtil.
    findSingleInstance( element, c, null );
if ( ( f == null) && (!not) ) {
    Checks.warn( element, "disposing element, "+
                  "since feature named "+featureName+
                  " is not selected." );
    element.dispose();
}
// the other way round for ( f != null) && (not)
}
}
```

Man sieht: der oben implementiert ModelModifier ist insofern generisch, als er für beliebige Basismodelle (und Metamodelle) sowie beliebige Featuremodelle funktioniert.

Nur die Operation *dispose()* muss vorhanden sein - die wird aber vom oAW Metamodellgenerator automatisch implementiert.

## Template-Relevante Features

Die obige Diskussion konzentrierte sich auf die Frage, wie Featuremodelle die Struktur von „anderen“ Modellen beeinflussen können. Manchmal ist es allerdings auch nötig, in den Templates abhängig von Features zu verfahren. Beispiel Persistenz: die Frage, ob und wie die durch das Modell beschriebenen Daten persistent sein sollen hat keine Auswirkung auf die Struktur der Daten, sondern nur auf den Code, der mit den Daten arbeitet. Das wird - wie bei openArchitectureWare üblich - realisiert durch eine Kombination aus Metamodellimplementierung und Template. Der folgende Code zeigt die Definition der entsprechenden Properties im Metamodell. Deren Implementierung fragt intern ab, ob bestimmte Features im Featuremodell existieren.

```
public class Data extends Class {
    // stuff...

    public boolean isPersistent() {
        return Persistence.exists(this);
    }

    public boolean isXMLPersistent() {
        return XML.exists(this);
    }

    public boolean isHibernatePersistent() ...

    public boolean isJDOPersistent() ...
```

Innerhalb der Templates kann man nun darauf zugreifen, und abhängig von diesen Properties Code generieren, oder auch nicht:

```
<<DEFINE Root FOR Data>>
<<EXPAND ClassAndProperties>>
<<IF isPersistent>>
    <<IF isHibernatePersistent>>
        <<EXPAND HbmXml>>
    <<ENDIF>>
<<ENDIF>>
<<ENDDFINE>>
```

## Fazit

Dieser Artikel zeigt, wie man Variantenmanagement mit "klassischer" MDSO zusammenbringen kann, eine Sache, die insbesondere für Produktlinien von essentieller Bedeutung ist. Zu beachten ist insbesondere auch, dass dieser Ansatz mit den heute verfügbaren Werkzeugen tatsächlich auch funktioniert.

Übrigens handelt es sich bei dem Ansatz letztendlich um das Verweben von Modellen, also eine Art „Modell-AOP“. Weitere Infos zu diesem Thema finden sich unter [MV05].

## Referenzen

- OAW            *openArchitectureWare*, <http://www.openarchitectureware.org>
- SV05            Stahl, Völter, *Modellgetriebene Softwareentwicklung – Technik, Engineering, Management*, dPunkt 2005
- MV05            Markus Völter, *Models and Aspects – Patterns for Handling Cross-Cutting Concerns in the context of MDSD*,  
<http://www.voelter.de/data/pub/ModelsAndAspects.pdf>
- PS              Pure Systems, *pure::variants*, <http://www.pure-systems.com/>
- CE00            Czarnecki, Eisenecker, *Generative Programming*, Addison-Wesley, 2000