Betriebssystemtechnik

Operating System Engineering (OSE)

“Feature- Oriented Programming“ [1]

(D. Batory)

Motivation

- Software wird heute auf einem zu niedrigen Abstraktionsniveau modelliert
  - im Zentrum der Betrachtung sind meist Klassen, Funktionen und Objekte
  - erschwert wird dadurch ...
    - die Konstruktion von Applikationen aus Komponenten und die **automatische Analyse** von Kompositionen
    - **automatisches Generieren** von Applikationen (Produktlinienvarianten) aus Spezifikationen höherer Ebene

→ **Merkmal**- basierte Systembeschreibung und - generierung

Merkmal- basierte Beschreibung

- Anwender beschreiben Programme und deren Varianten mit Hilfe von abstrakten Merkmal. Warum nicht auch die Entwickler beim Entwurf?
- Individuelle Programme werden in Form von Gleichungen repräsentiert:

```
f     // Programm mit Merkmal „f“
g     // Programm mit Merkmal „g“
```

f und g sind „Konstanten“

```
i(x)  // fügt Merkmal „i“ zu „x“ hinzu
j(x)  // fügt Merkmal „J“ zu „x“ hinzu
```

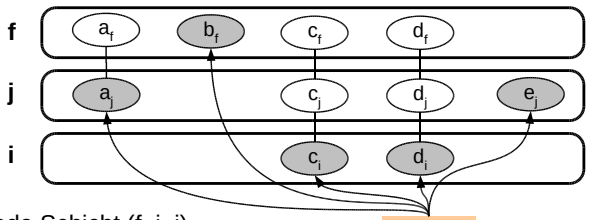
i und j sind „Funktionen“, sog. **„Refinements“**

```
app1 = i(f)       // Merkmale i, f
app2 = j(g)       // Merkmale j, g
app3 = i(j(f))    // Merkmale i, j, f
```

dieses **„GenVoca“** Modell beschreibt eine Produktlinie!

Merkmal- basierter Entwurf

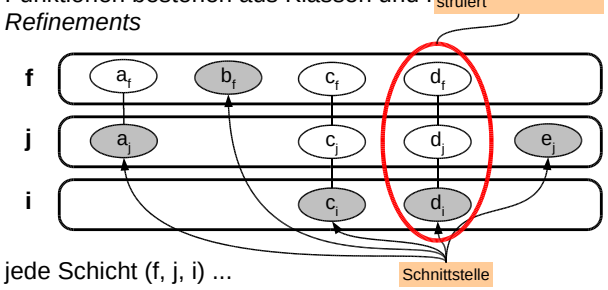
- eine GenVoca- Konstante besteht aus Klassen
- Funktionen bestehen aus Klassen und Klassen-*Refinements*



- jede Schicht (f, j, i) ...
  - realisiert ein bestimmtes Merkmal
  - besteht aus einer „Kollaboration“

Merkmal- basierter Entwurf

- eine GenVoca- Konstante besteht aus K
- Funktionen bestehen aus Klassen und K *Refinements*



- jede Schicht (f, j, i) ...
  - realisiert ein bestimmtes Merkmal
  - besteht aus einer „Kollaboration“

Merkmal- basierte Implementierung

- flexible *„Refinement Chains“* erfordern, dass ...
  - ein *Refinement* Klassen **und** verfeinerte Klassen erweitern kann
  - die Benutzerschnittstelle unabhängig von Verfeinerungen ist
- ein erster Lösungsansatz basiert(e) auf der Kombination von Templates und Vererbung
  - leider bringt die Vererbung aber Probleme mit sich
- der neuere Lösungsansatz basiert auf einem neuen Sprachkonstrukt: *Refinements*

Implementierung mit *Mixin- Layers* (1)

- flexibles Erweitern, ohne die konkrete Basis zu kennen, lässt sich in C++ mittels *Mixin*- Klassen erledigen:

```
template <typename Base>
class Mixin : public Base {
  int _attr; // weiteres Attribut
public:
  // weitere Methode
  void func() {
    something_before();
    Base::func(); // Aufruf der Basismethode
    something_after();
  }
};
```

- die einheitliche Schnittstelle lässt sich durch ein `typedef` erreichen und mit der Konfigurierung kombinieren:

```
typedef Mixin<SomeBase> SomeAPI;
```


Implementierung mit *Mixin- Layers* (2)

- Kollaboration lassen sich durch *Mixin- Layers* realisieren:

```
template <typename BaseLayer>
class MixinLayer_J : public BaseLayer {
public:

  // hier kommen die Verfeinerungen als „nested classes“
  class A : public BaseLayer::A {
    typedef typename BaseLayer::A Base; // die verfeinerte Basis
    int _attr; // neues Attribute
  public:
    void func() {
      something_before();
      Base::func(); // Aufruf der Basismethode
      something_after();
    }
  };
  class C : public BaseLayer::C { ... };
  class D : public BaseLayer::D { ... };

  // hier kommt eine neue Klasse
  class E { ... };
};
```

Implementierung mit *Mixin- Layers* (3)

- Konfigurierung wieder mittels `typedef`:

```
typedef MixinLayer_I<Layer_F> App1;
// oder
typedef MixinLayer_J<Layer_G> App2;
// oder
typedef MixinLayer_I<MixinLayer_J<Layer_F> > App3;
```

- Fazit: Ziel (fast) erreicht
  - mit *Templates* und Vererbung lassen sich Merkmal-Implementierungen flexibel stapeln
  - die konfigurierenden `typedefs` lassen sich leicht aus abstrakten Spezifikationen generieren
- leider hat die Vererbung hier ein paar Nachteile ...

Vererbungsprobleme (1)

- das *Extensibility Problem*:

```
class Buffer { ... };

// zwei Puffervarianten
class FileBuffer : Buffer { ... };
class SocketBuffer : Buffer { ... };

// eine Erweiterung durch Vererbung
class SynchBuffer : Buffer {
  Mutex _lock;
public:
  void put (Item i) { Secure s(_lock); Buffer::put(i); }
  Item get () const { Secure s(_lock); return Buffer::get(); }
};
```

- die Erweiterung `SynchBuffer` wirkt sich **nicht** auf abgeleitete Klassen (Varianten) aus
- Ursache: die Erweiterung heißt nicht `Buffer`

Vererbungsprobleme (2)

- das *Constructor Propagation Problem*:

```
class TheBase {
public:
  TheBase (int, double) { ... }
};

template <typename Base>
class Mixin : public Base {
public:
  // obwohl keine eigenen Initialisierungen nötig sind, muss ein
  // Konstruktor angelegt werden, damit TheBase bedient werden kann
  Mixin (int i, double d) : Base(i, d) {}
};
```

- wenn eine Basis einen speziellen Konstruktor hat, entsteht eine starke Schichtenabhängigkeit
- Ursache: Vererbung propagiert keine Konstruktoren

Implementierung mit *Refinements*

- *Refinements* sind eine spezielle Spracherweiterung:
  - implementiert z.B. in Jak[1] und FeatureC++ [2]

```
class Buffer {
public:
  Base (int) { ... } // ein nicht-Standard Konstruktor
  ...
};

// zwei Puffervarianten
class FileBuffer : Buffer { ... };
class SocketBuffer : Buffer { ... };

// eine Erweiterung durch Refinement
refines class Buffer {
  Mutex _lock
public:
  void put (Item i) { Secure s(_lock); super::put(i); }
  Item get () const { Secure s(_lock); return super::get(); }
};
```

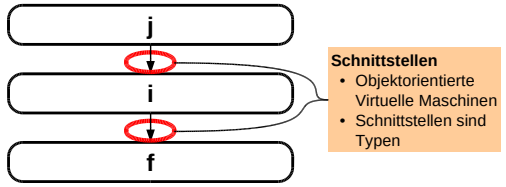
- das *Refinement* wirkt sich auf alle abgeleiteten Klassen aus; der Konstruktor wird automatisch „propagiert“

Zwischenfazit

- FOP versucht Merkmale als *„First Class Entities“* auch bei Entwurf und Implementierung zu behandeln
  - der Merkmalbegriff kommt also in die Welt der Programmierung!
- Programme werden als Gleichungen beschrieben und können leicht aus Gleichungen generierte werden
  - Möglichkeit der Optimierung (*„Automatic Programming“*)
- Offene Fragen:
  - Wie können die Schichten gestapelt werden, d.h. sind die Funktionen typisiert? Reicht Typisierung?
  - Wie findet man bei mehreren funktional gleichwertigen Programmen die beste Lösung unter Beachtung nicht-funktionaler Eigenschaften?
  - Wie sieht die Beziehung zwischen GenVoca Modellen und Merkmaldiagrammen aus?

GenVoca Grammatiken (1)

- GenVoca Architekturen sind immer geschichtet:



- g (x:S):R bedeute, dass das Merkmal g ...
  - eine virtuelle Maschine R implementiert
  - eine Schicht x importiert, die eine VM S implementiert
  - einen Parameter x vom Typ S hat und selbst vom Typ R ist

GenVoca Grammatiken (2)

- Schichten (=Merkmale) mit dem selben Typ bilden ein *„Realm“* (dt. Fachgebiet, Bereich), z.B.:

```
S = { y, z, w }
R = { g(x:S), h(x:S), i(x:R) }
```

- die erlaubten Konfiguration dieses Systems lassen sich durch eine äquivalente GenVoca Grammatik beschreiben:

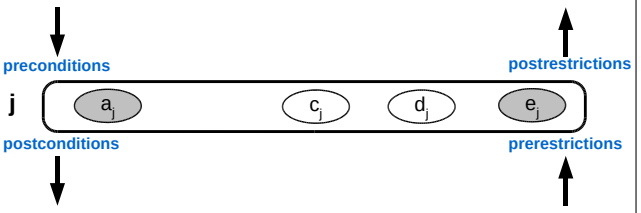
```
S ::= y | z | w ;
R ::= g( S ) | h( S ) | i( R ) ;
```

- gültig wäre z.B.: `g(y)`, `h(z)`, `i(i(h(w)))`
- ungültig wäre z.B.: `g(h(w))`, `i(y)`

eine GenVoca Grammatik beschreibt eine Produktlinie!

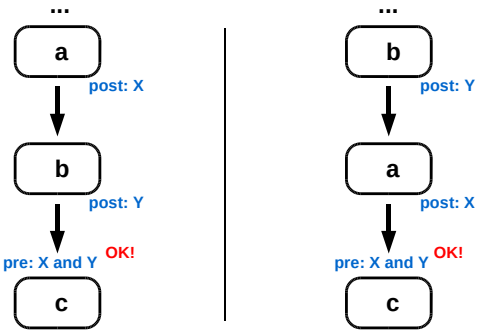
Design Rule Checking (1)

- Grammatiken können nur zur Prüfung der „syntaktischen“ Korrektheite einer Komposition benutzt werden
- Lösung: zusätzliche *„Design Rules“*



Design Rule Checking (2)

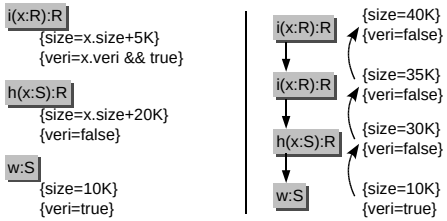
- Wann ist eine Komposition zulässig? Beispiel:



„a“ und „b“ sind vertauschbar

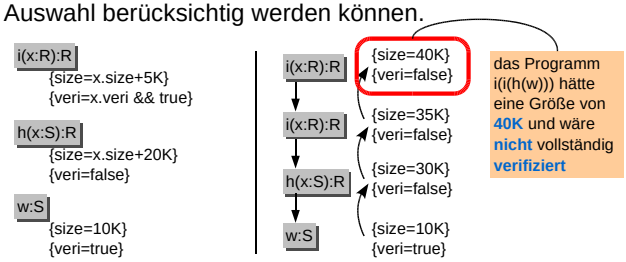
Attributierte GenVoca Modelle

- Welche gültige Konfiguration verbraucht am wenigsten Speicher, ist am schnellsten oder besteht nur aus verifizierten Komponenten?
- Derartige **nicht- funktionale Eigenschaften** können über Attribute erfasst werden, die **automatisch** bei der Auswahl berücksichtig werden können.



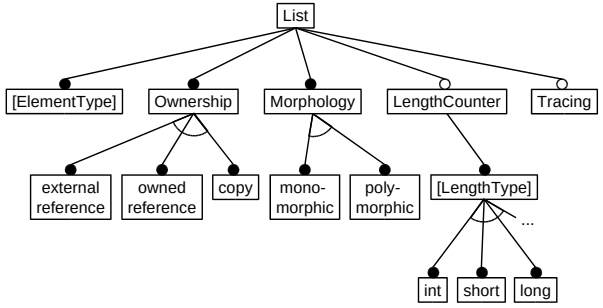
Attributierte GenVoca Modelle

- Welche gültige Konfiguration verbraucht am wenigsten Speicher, ist am schnellsten oder besteht nur aus verifizierten Komponenten?
- Derartige **nicht- funktionale Eigenschaften** können über Attribute erfasst werden, die **automatisch** bei der Auswahl berücksichtig werden können.



Merkmal- Modelle vs. GenVoca (1)

- lässt sich die Variabilität dieser Produktlinie auch als GenVoca- Grammatik ausdrücken?

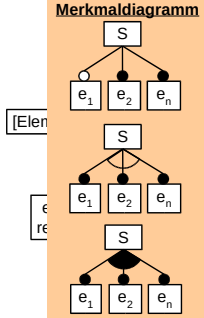


Merkmal- Modelle vs. GenVoca (1)

- lässt sich die Variabilität dieser Produktlinie auch als GenV

JA! Es helfen ein paar einfache **Abbildungsregeln**:

Merkmaldiagramm **Grammatik**



Merkmal- Modelle vs. GenVoca (2)

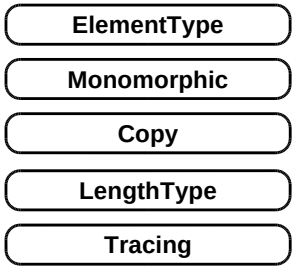
- Ergebnis der Transformation:

```
List ::= Tracing (LengthCounter (Morphology (Ownership (ElementType))));
List ::= LengthCounter (Morphology (Ownership (ElementType)));
List ::= Tracing (Morphology (Ownership (ElementType)));
List ::= Morphology (Ownership (ElementType));

Ownership ::= External Reference | Owned Reference | Copy;

Morphology ::= Monomorphic | Polymorphic;

LengthCounter := LengthType;
```

- Gültige Sätze der Sprache:
  - `Tracing(LengthType(Copy(Monomorphic(ElementType))))`
  - `Monomorphic(External Reference(ElementType))`
- die Grammatik hat im Prinzip die selbe Aussagekraft wie das Merkmaldiagramm
  - nicht- Terminale kommen allerdings in den Sätzen nicht mehr vor
  - Constraints können über Attribute ausgedrückt werden

Merkmal- Modelle vs. GenVoca (3)

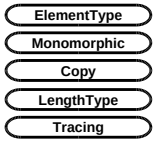
- für die erste Variante ergibt sich z.B. folgende Schichtenstruktur:



- bringt uns diese Hierachie einer vernünftigen Implementierung näher?

Merkmal- Modelle vs. GenVoca (4)

- Probleme (auf den ersten Blick):
  - Mindestens ElementType und LengthType sollten eher keine Schicht, sondern Template-Parameter sein.
  - Wo ist die eigentliche Listenverwaltung?
  - Monomorphic und Copy **scheinbar** unabhängig.



- Gründe:
  - Bei der Transformation vom Merkmalmodell in die Grammatik ist der Kontext der Merkmale verloren gegangen:
    `ElementType` ist `List/ElementType`
    `Monomorphic` ist `List/Morphology/Monomorphic`
    Die Basisebene realisiert also bereits einige Listenfunktionen
  - Der Name einer Schicht im FOP repräsentiert die **Implementierung** eines Merkmals. Ein Merkmal im Merkmaldiagramm hat mit der Implementierung nichts zu tun! Man muss daher **manuell anpassen**.

Merkmal- Modelle vs. GenVoca (4)

- Probleme (auf den ersten Blick):
  - Mindestens ElementType und LengthType sollten eher keine Schicht, sondern Template-Parameter sein.
  - Wo ist die eigentliche Listenverwaltung?
  - Mon
- Grün
  - Bei d atik ist der Kont
    ElementType ist `List/ElementType`
    `Monomorphic` ist `List/Morphology/Monomorphic`
    Die Basisebene realisiert also bereits einige Listenfunktionen
  - Der Name einer Schicht im FOP repräsentiert die **Implementierung** eines Merkmals. Ein Merkmal im Merkmaldiagramm hat mit der Implementierung nichts zu tun! Man muss daher **manuell anpassen**.

ElementType
Monomorphic
Copy
Type
ng

Merkmale im Sinne von FOP sind Elemente des **Lösungsraums**, während Merkmalmodelle den **Problemraum** beschreiben!

Beispiel: ein Liste in *FeatureC++* (1)

```
template <typename ElementType>
class List {
  // Datenelemente
  ElementType *head_; // Zeiger auf 1 Element
  List *tail_;  // weitere Elemente
public:
  // Konstruktor: erzeugt eine neue Liste
  List(SetHeadElementType &h, List *t=0) :
  head_(0), tail_(0) {
    setHead(h);
  }

  // Initialisierung des Elements
  void setHead(SetHeadElementType &h) {
    head_ = &h; // hier wird nichts kopiert
                // oder geprüft!
  }

  // Setter und Getter Funktionen
  ElementType &head() { return *head_; }
  void setTail(List *t) { tail_ = t; }
  List *tail() const { return tail_; }
};
```

List
Monomorphic
Copy
LenList
Tracing

Beispiel: ein Liste in *FeatureC++* (1)

```
template <typename ElementType>
class List {
  // Datenelemente
  ElementType *head_; // Zeiger auf 1 Element
  List *tail_;  // weitere Elemente
public:
  // Konstruktor: erzeugt eine neue List
  List(SetHeadElementType &h, List *t=0) :
  head_(0), tail_(0) {
    setHead(h);
  }

  // Initialisierung des Elements
  void setHead(SetHeadElementType &h) {
    head_ = &h; // hier wird nichts kopiert
                // oder geprüft!
  }

  // Setter und Getter Funktionen
  ElementType &head() { return *head_; }
  void setTail(List *t) { tail_ = t; }
  List *tail() const { return tail_; }
};
```

Dieser Typ ist hier noch unbekannt! Das Fragment funktioniert daher nicht ohne *Refinement*. Sicherzustellen über *Design Rule Check*.

List
Monomorphic
Copy
LenList
Tracing

Beispiel: ein Liste in *FeatureC++* (2)

```
// Erweiterungen für „monomorphe“ Listen
template <typename ElementType>
refines class List {
  // hinzugefügte Funktion: so wird ein Element
  // in einer monomorphen Liste kopiert (evtl.)
  static ElementType *copy(const ElementType &e) {
    return new ElementType(e); // Copy-Konstruktor
  }

public:
  // Initialisierung des Elements erweitern
  void setHead(SetHeadElementType &h) {
    // Typüberprüfung mittels RTTI zur Laufzeit
    assert(typeid(e)==typeid(ElementType));
    super::setHead(h);
  }
};
```

List
Monomorphic
Copy
LenList
Tracing

- die `Monomorphic` Schicht trägt (wie `Polymorphic` auch) eine passende Kopierfunktion durch *Refinement* bei und erweitert `setHead` um eine Typprüfung.

Beispiel: ein Liste in *FeatureC++* (3)

```
// Erweiterungen für kopierende Listen
template <typename ElementType>
refines class List {
public:
  // hinzugefügter Typ
  typedef const ElementType SetHeadElementType;

  // Initialisierung des Elements erweitern
  void setHead(SetHeadElementType &h) {
    if (head_) delete head_;
    // Kopieren mittels copy Funktion
    super::setHead(copy (h));
  }
  ~List () {
    // Speicherfreigabe im Destruktor
    delete head_;
};
```

List · Monomorphic · **Copy** · LenList · Tracing

- die `Copy` Schicht erweitert `setHead` um das Kopieren und fügt einen Destruktor hinzu. Außerdem wird hier der Typ `SetHeadElementType` endlich definiert.

Beispiel: ein Liste in *FeatureC++* (4)

```
// Erweiterungen zum Zählen der Elemente
template <typename ElementType,
          typename LengthType>
class LenList : public List<ElementType> {
  LengthType length_; // Anzahl der Elemente
public:
  typedef List<ElementType> ListType;
  LenList(SetHeadElementType &h, ListType *t=0) :
  ListType(h,t), length_(computedLength()) {}

  void setTail(List *t) {
    ListType::setTail(t);
    length_ = computedLength();
  }

  const LengthType &length() const { return length_; }

  LengthType computedLength() const {
    return tail()? tail()->length()+1 : 1;
  }
};
```

List · Monomorphic · Copy · **LenList** · Tracing

- diese Schicht nutzt Vererbung, um die Länge als weiteren *Template*- Parameter einzubringen.

Beispiel: ein Liste in *FeatureC++* (5)

```
// Erweiterungen zum Verfolgen des Kontrollflusses
template <typename ElementType>
refines class List {
public:
  // Methoden: Wrapper zur Ausgabe des Funktionsnamens
  void setHead(SetHeadElementType &h) {
    cout << "setHead(" << h << ")" << endl;
    super::setHead(h);
  }
  ElementType &head() {
    cout << "head()" << endl;
    return super::head();
  }
  void setTail(ReturnType *t) {
    cout << "setTail(t)" << endl;
    super::setTail(t);
  }
};
```

List · Monomorphic · Copy · LenList · **Tracing**

- das *Refinement* wirkt sich sowohl auf `List` als auch auf die abgeleitete `LenList` aus.
  - leider aber nicht auf die `LenList` Methoden

Beispiel: ein Liste in *FeatureC++* (5)

```
// Erweiterungen zum Verfolgen des Kontrollflusses
// auch in der LenList Klasse
template <typename ElementType,
          typename LengthType>
refines class LenList {
public:
  const LengthType &length() const {
    cout << "length()" << endl;
    return super::length();
  }

  LengthType computedLength() const {
    cout << "computedLength()" << endl;
    return super::computedLength();
  }
};
```

List · Monomorphic · Copy · LenList · **Tracing**

- das *Refinement* wirkt sich sowohl auf `List` als auch auf die abgeleitete `LenList` aus.
  - leider aber nicht auf die `LenList` Methoden

Beispiel: ein Liste in *FeatureC++* (6)

- um den Aufwand zu verringern, integriert FeatureC++ *Feature- Oriented Programming* und AOP mit AspectC++ ;-)

List
Monomorphic
Copy
LenList
Tracing

```
// Erweiterungen zum Verfolgen des Kontrollflusses
aspect ListTracing {
  advice execution(derived(„List“)) : before () {
    cout << JoinPoint::signature() << endl;
  }
};
```

- dabei werden Pointcuts beispielsweise automatisch so angepasst, dass sie sich nur auf Elemente in Basisebenen auswirken.
- die Kombination von AOP und FOP bietet noch viele spannende Forschungsthemen!

Beispiel: Fazit

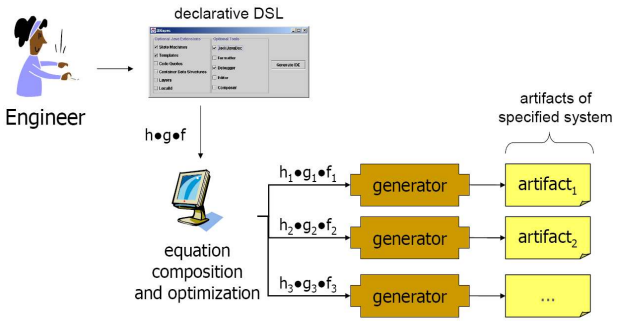
- Vergleich zur generischen Implementierung aus „Synthesizing Objects“:
  - deutlich weniger Klassen
  - keine Template Metaprogramme

ElementType
Monomorphic
Copy
LengthType
Tracing

- Es gibt allerdings auch Nachteile:
  - weniger Konfigurierbarkeit durch den Anwendungscode
  - zusätzliche Spracherweiterung und Werkzeuge nötig
- Generell ist FOP eher für Konfigurierung im Großen gedacht, wo Generizität nicht erforderlich ist.
- FeatureC++ unterstützt Aspekte als eine interessante Ergänzung

FOP Ausblick

- mit der AHEAD Tool Suite (ATS) wird die Vision verfolgt, FOP auf beliegige Artefakte (Klassen, Makefiles, Manuals, ...) anzuwenden [3].



Zusammenfassung

- FOP versucht, eine Verbindung herzustellen zwischen ...
  - Merkmalen bzw. Merkmaldiagrammen und
  - geschichteten Softwarearchitekturen und
  - Entwurf und Implementierung
- Ziel: automatische Analyse, Optimierung und Generierung
- FOP eignet sich speziell für grobgranulare Konfigurierung
  - generische Programme, die viele Varianten gleichzeitig bereitstellen stehen nicht im Zentrum der Betrachtung
  - es geht um Produktlinien von denen z.B. ein Softwarehersteller einzelne statisch konfigurierte Varianten ausliefern will

Literatur

[1] D. Batory, *Homepage of the Product- Line Architecture Research Group*. http://www.cs.utexas.edu/users/schwartz

[2] S. Apel, *Feature C++ Homepage*. http://wwwiti.cs.uni- magdeburg.de/iti_db/fcc/

[3] D. Batory, J.N. Sarvela, A. Rauschmeyer, *Scaling Step- Wise Refinement*. International Conference on Software Engineering (ICSE 2003), May 2003.