

# Betriebssystemtechnik

Operating System Engineering (OSE)

---

## Zusammenfassung und Ausblick



# Überblick

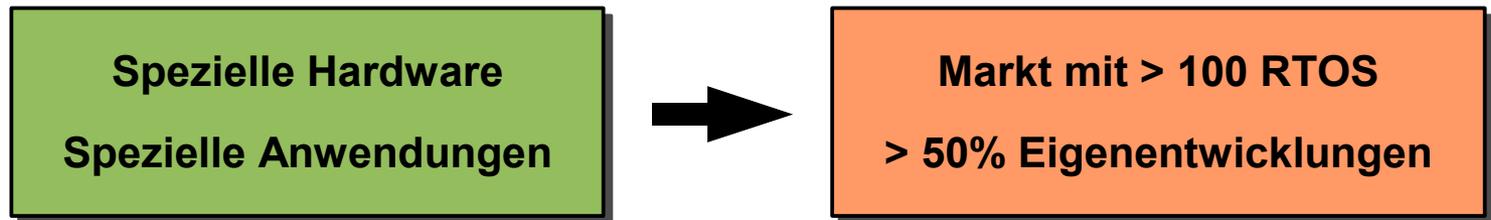
---

- Zusammenfassung: Software-Produktlinien
  - Prinzipien
  - Methoden
  - Werkzeuge
- Herausforderungen für die Zukunft
  - Automatische Konfigurierung
  - Kombination und Integration
  - Beherrschbarkeit nicht-funktionaler Eigenschaften
- C*i*AO



# Motivation

## Betriebssysteme für eingebettete Systeme

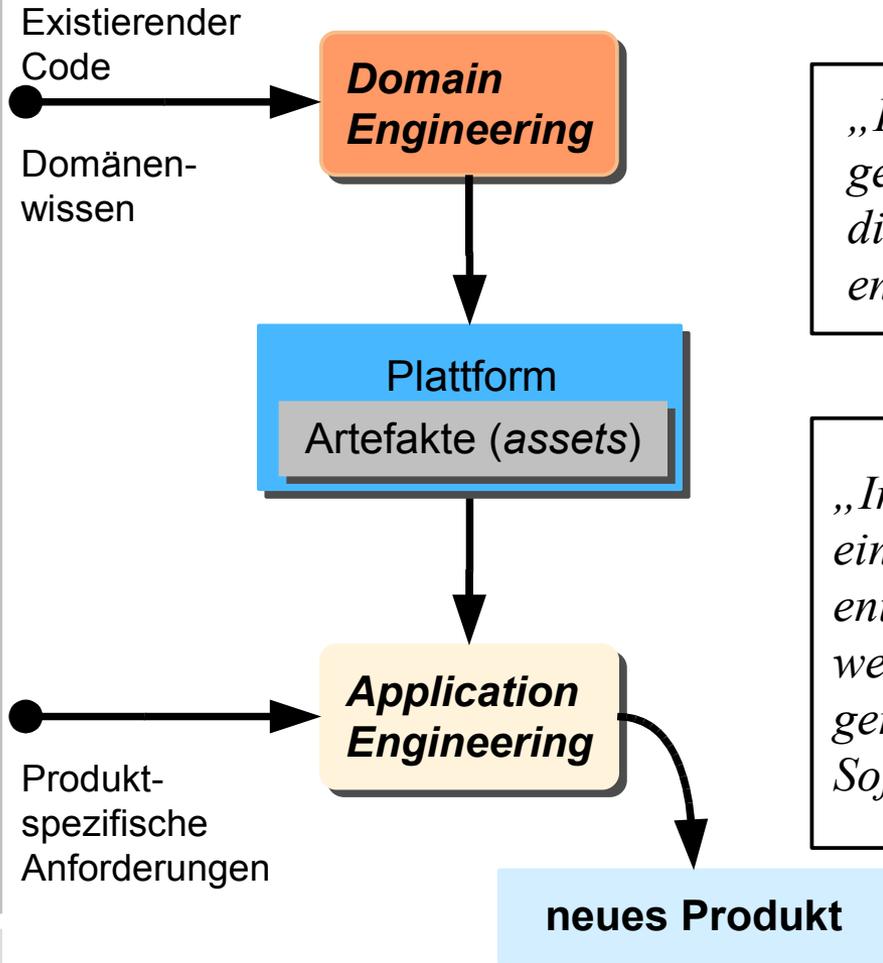


- „das Rad wird neu erfunden“
  - auch die selben Fehler werden wiederholt
- oftmals bietet **ein** BS Hersteller **mehrere** Systeme an
  - mit getrennter Code-Basis
  - getrieben durch die speziellen Anforderungen seiner Kunden



# Software-Produktlinien (Software-PL)

**Organisierte Wiederverwendung** durch die **aktive** Gestaltung einer gemeinsamen Plattform für aktuelle und künftige Produkte



*„Im **Domain Engineering** werden die gemeinsamen und variablen **Artefakte**, die Bestandteile der **Plattform**, entwickelt.“*

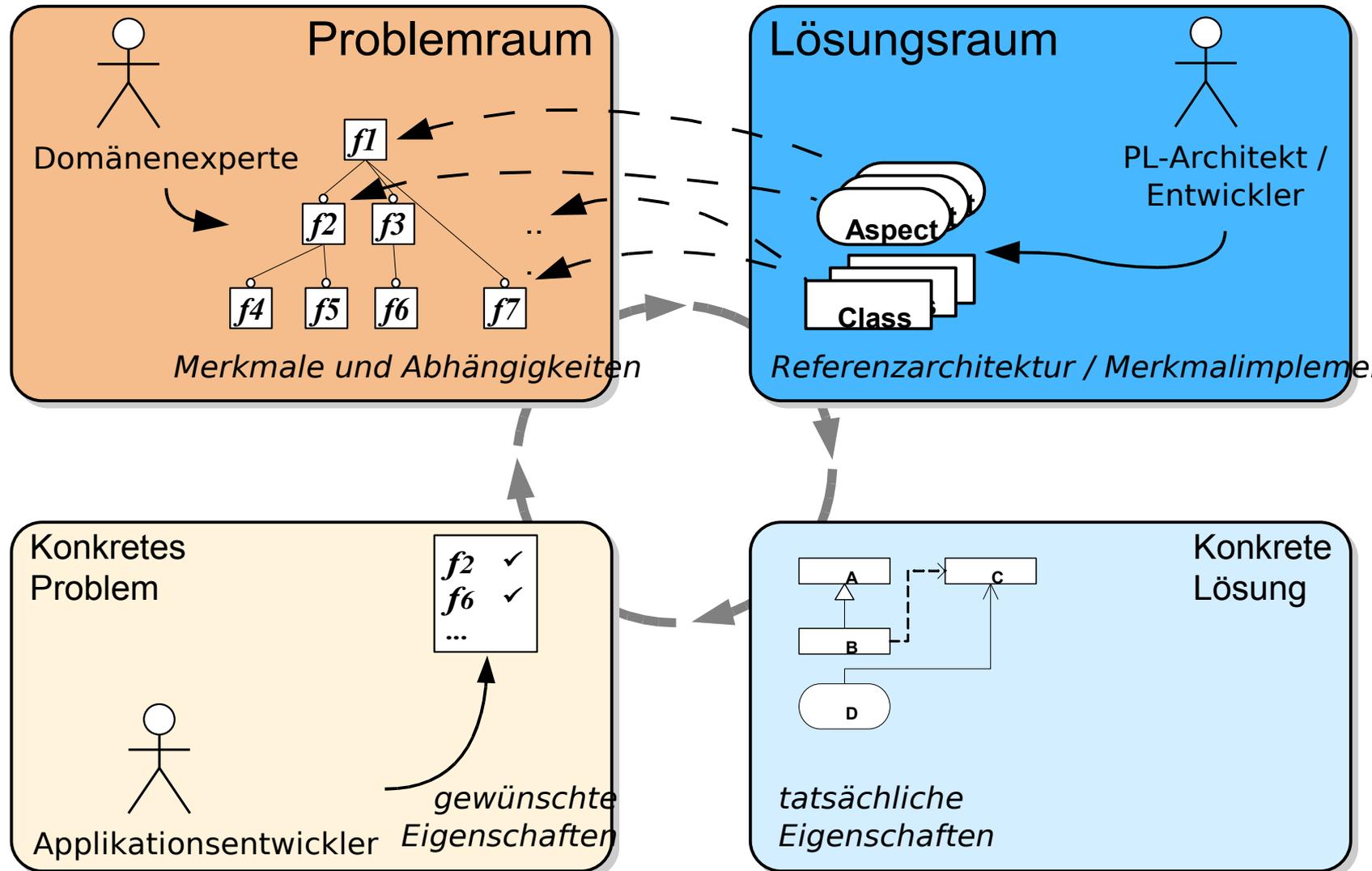
[1]

*„Im **Application Engineering** werden einzelne **Produkte** der Produktlinie entwickelt bzw. abgeleitet. Die Produkte werden [...] konfiguriert, so dass nur in geringem Maße produktspezifische Softwareentwicklung notwendig wird.“*

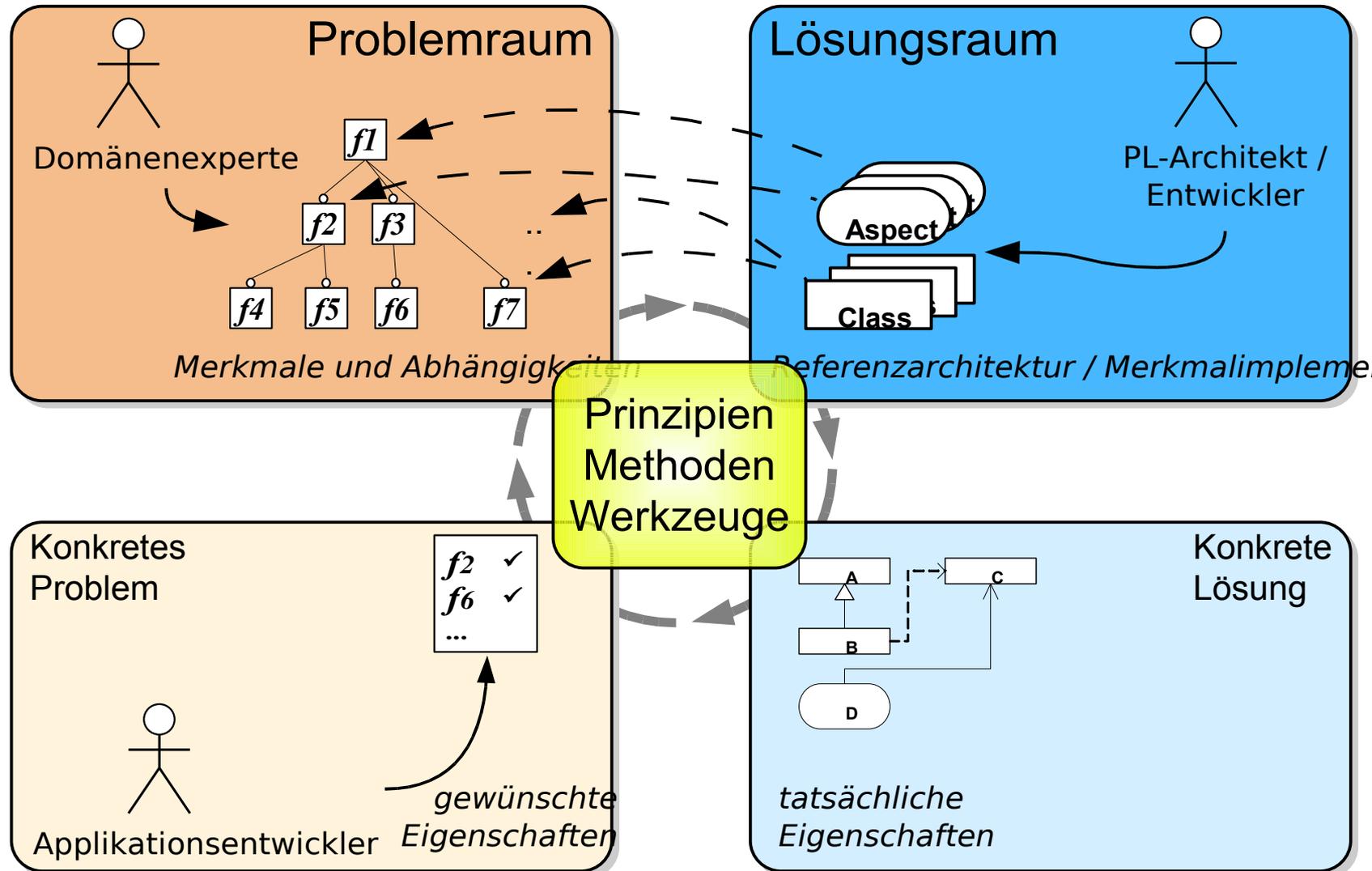
[1]



# Überblick: Software-Produktlinien



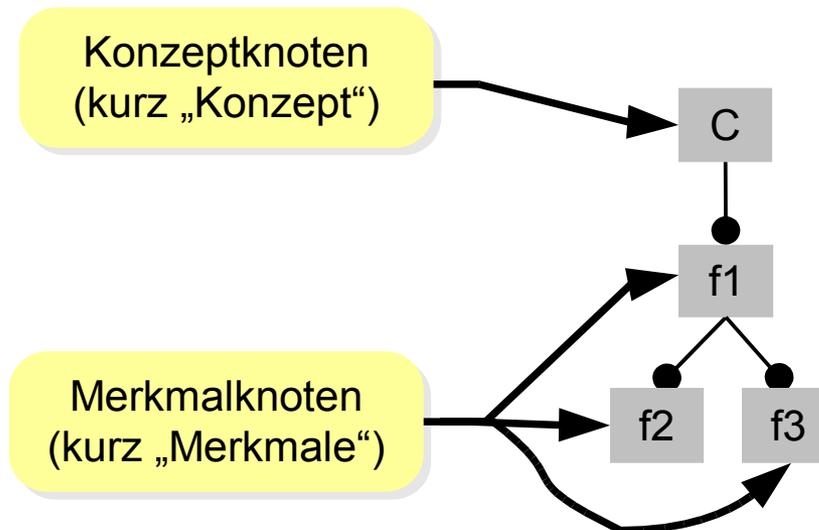
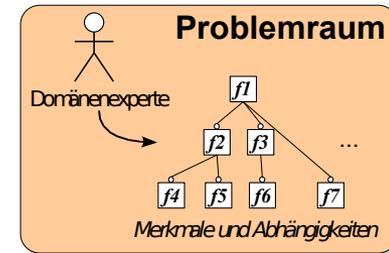
# Überblick: Software-Produktlinien



# Beschreibung des Problemraums

## Merkmalmmodell (*feature model*) [2]

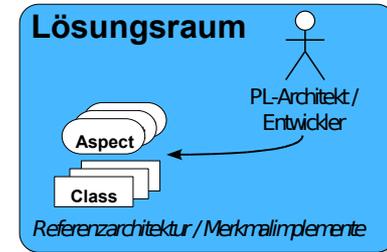
- Gerichteter, azyklischer Graph
- Beschreibt die möglichen **Ausprägungen** eines **Konzepts** anhand von **Gemeinsamkeiten** und **Unterschieden**



# Beschreibung des Lösungsraums

## Plattform

- Komponentenlager / *-framework*
- **Implementierung** der Konzepte und Merkmale



## Familienmodell

- **Abbildung** aus Komponentenlager in den Problemraum (von Artefakten auf Merkmale):

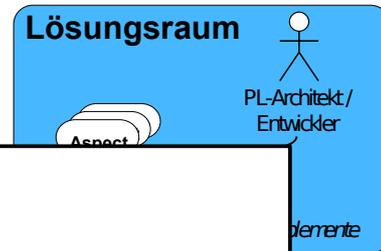
$$A \rightarrow M$$

- Eventuell über zusätzliche **Transformationen**:  
 $(A \times T) \rightarrow M$



# Beschreibung des Lösungsraums

## Plattform



### Problem: Trennung der Belange

- **Ideal:** Jedes Artefakt ist genau einem Merkmal zugeordnet:  $A \rightarrow M$  ist rechtseindeutig

- **Realität:** „`#ifdef`-Hölle“

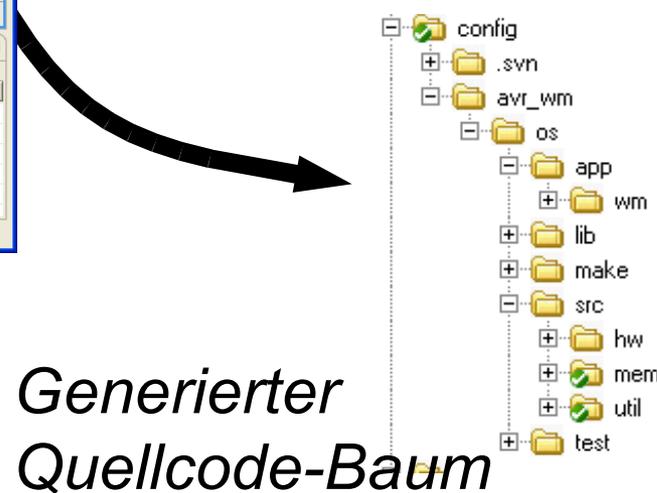
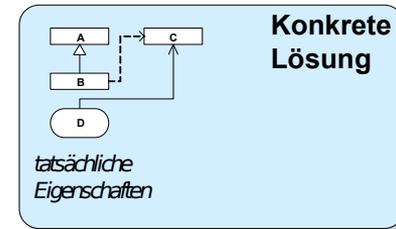
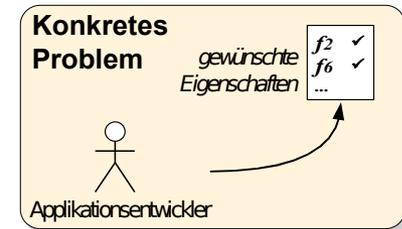
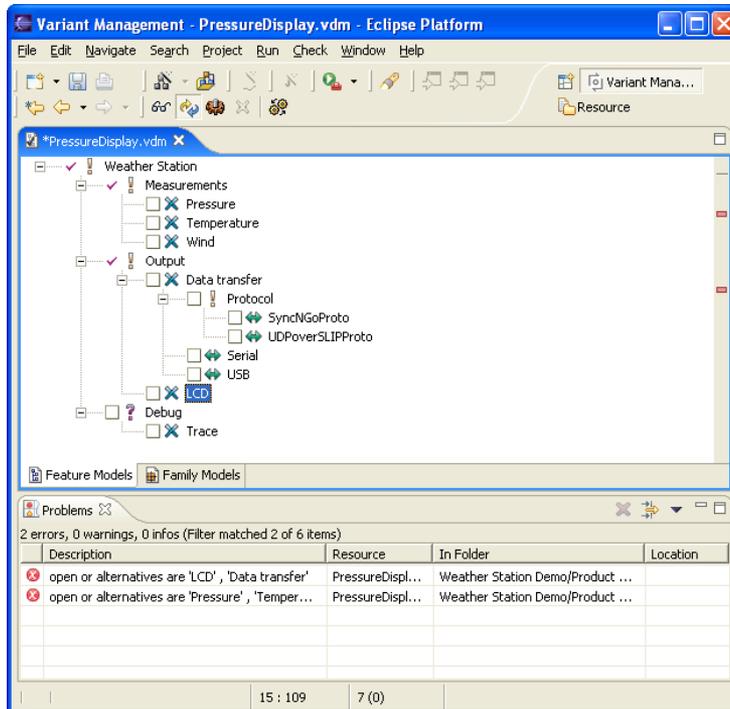
- Im Rahmen von „Betriebssystemtechnik“ wurden verschiedenste Werkzeuge und Sprachen betrachtet, die hier Besserung versprechen. Insbesondere AOP mit C++.

$$(A \times T) \rightarrow M$$



# Konkretes Problem, konkrete Lösung

## Merkmalauswahl



*Generierter  
Quellcode-Baum*



# Bau von Produktlinienkomponenten

---

... erfordert **Mechanismen zur Anpassung**

- **Präprozessoren (PP)**

- XVCL

- **Generatoren (GEN)**

- pure::variants Klassenaliase

- **Weber, Aspektorientierte Programmierung (AOP)**

- AspectC++

- **Generische Programmierung (GP)**

- C++ *Templates*

- **Objektorientierte Programmierung (OOP)**

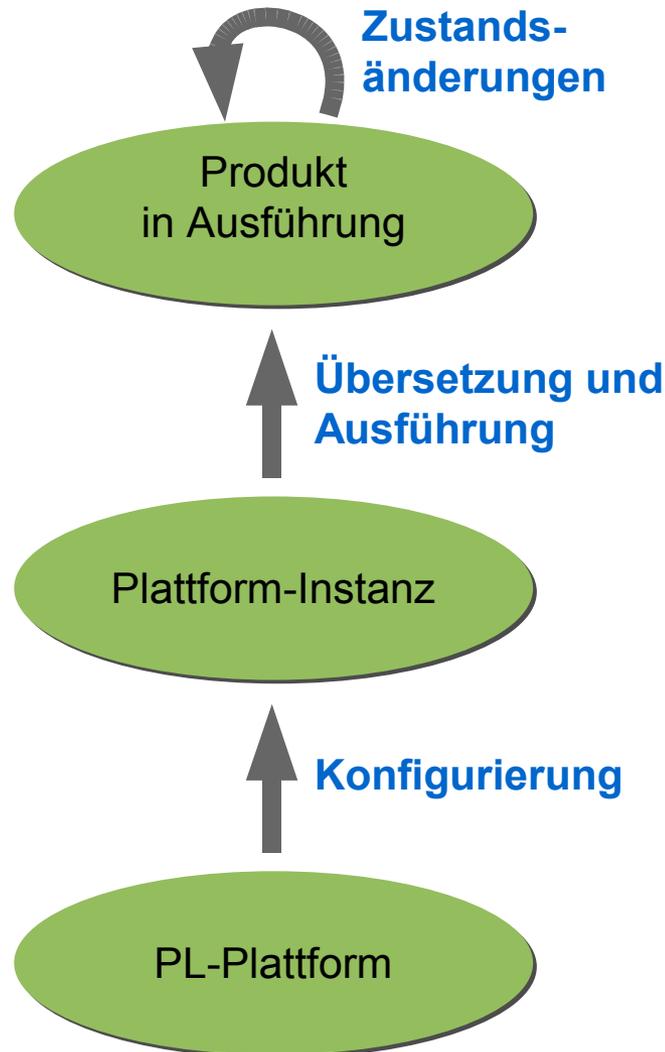
- virtuelle Funktionen und Vererbung in C++

- Was benutzt man wofür?

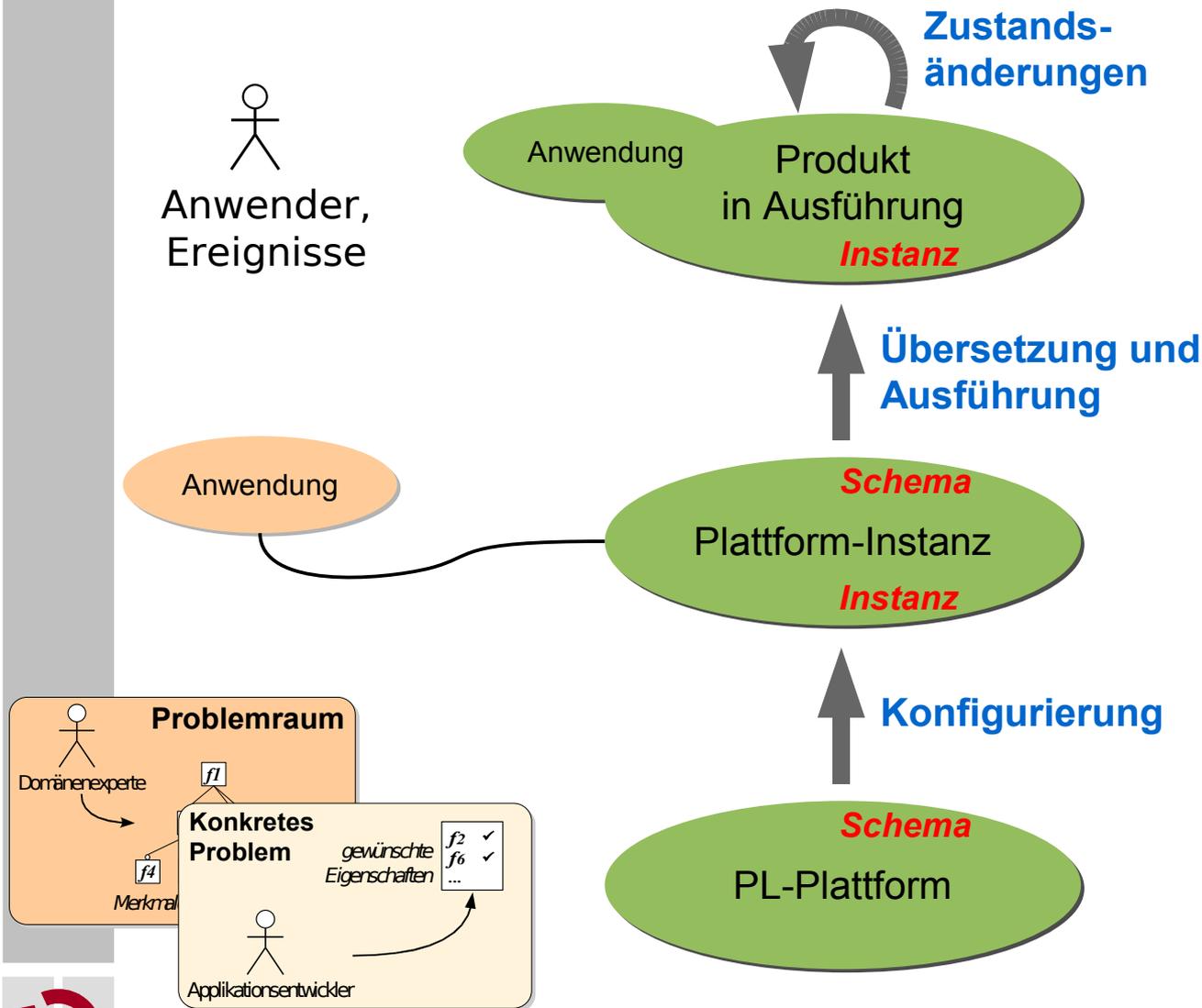


# Welcher Mechanismus ist wofür?

Die Komponenten einer Produktlinienplattform werden konfiguriert, übersetzt und zur Ausführung gebracht. Auf jeder Ebene, auch im laufenden System, gibt es Variabilität und Konfigurierungsentscheidungen.



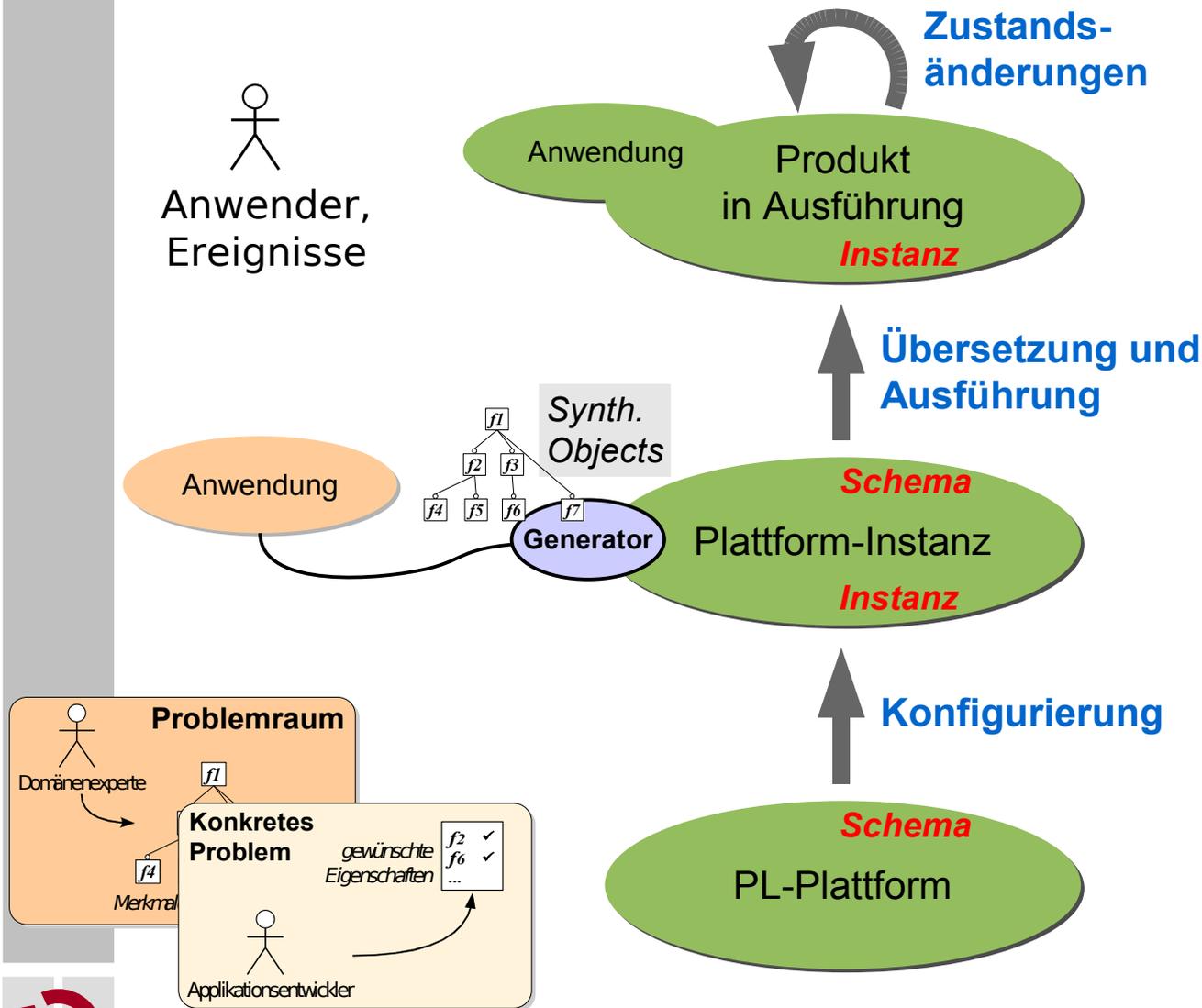
# Welcher Mechanismus ist wofür?



Bei jedem Übergang zwischen den Ebenen gibt es eine Schema-Instanz Beziehung. Welche Instanz gewählt wird, entscheidet (je nach Ebene) z.B. die Merkmalauswahl, die Anwendung durch instanziierte *Templates* und Typen sowie aufgerufene Funktionen oder zur Laufzeit der Anwender bzw. externe Ereignisse, durch die das Programm sein Verhalten ändert.



# Welcher Mechanismus ist wofür?



## geeignete Mechanismen

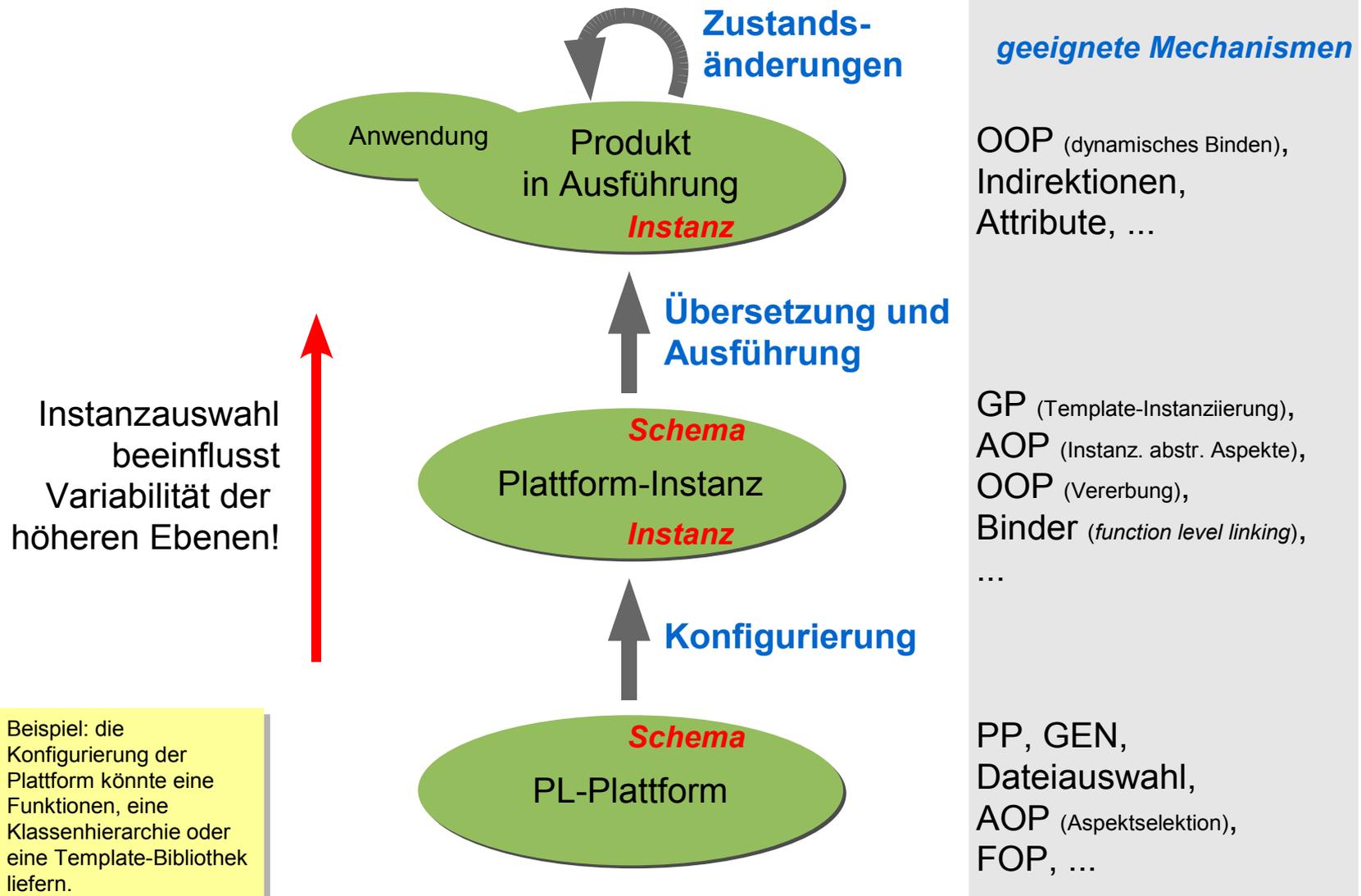
OOP (dynamisches Binden),  
Indirektionen,  
Attribute, ...

GP (Template-Instanziierung),  
AOP (Instanz. abstr. Aspekte),  
OOP (Vererbung),  
Binder (*function level linking*),  
...

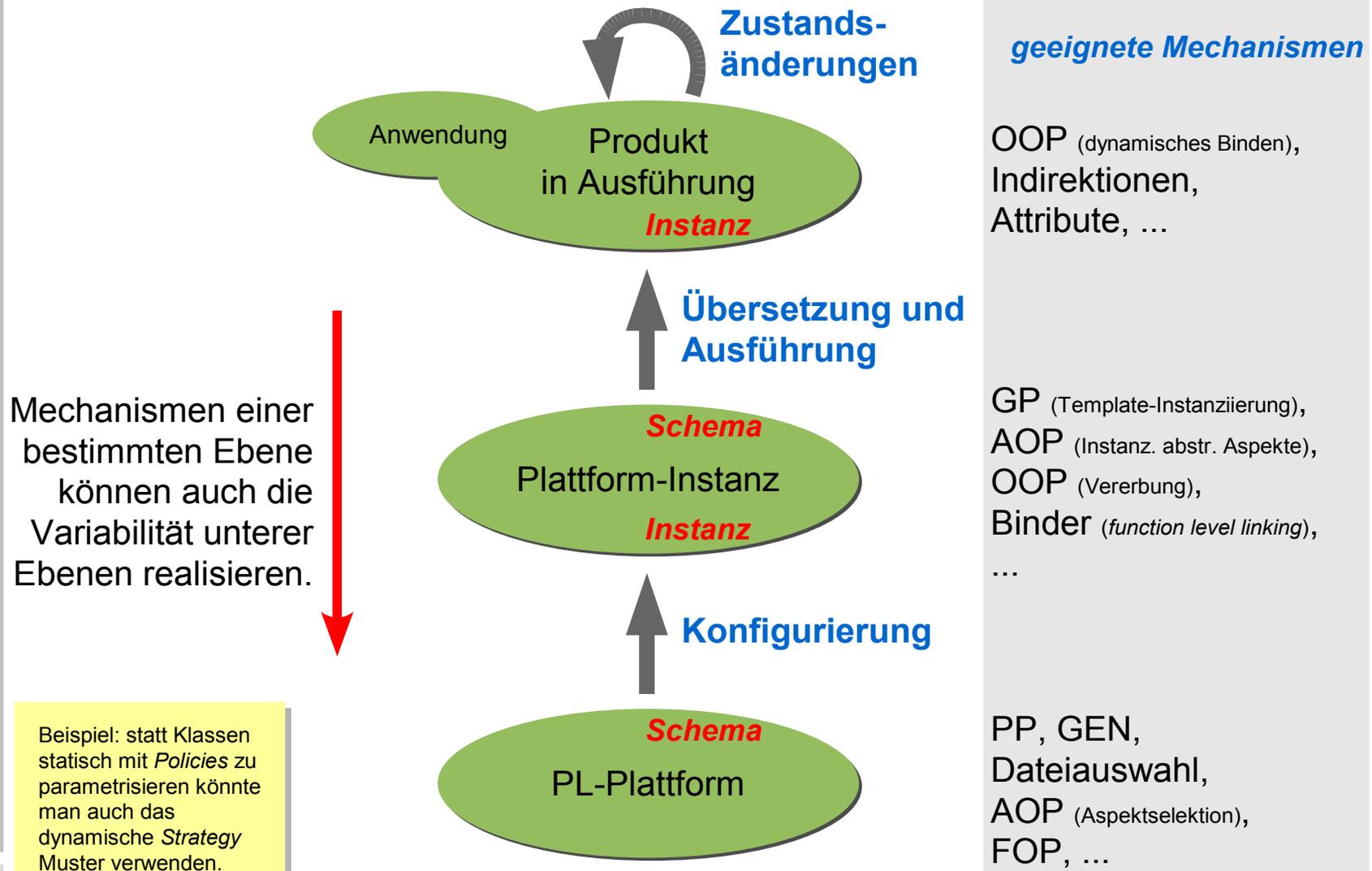
PP, GEN,  
Dateiauswahl,  
AOP (Aspektselektion),  
FOP, ...



# Welcher Mechanismus ist wofür?



# Welcher Mechanismus ist wofür?

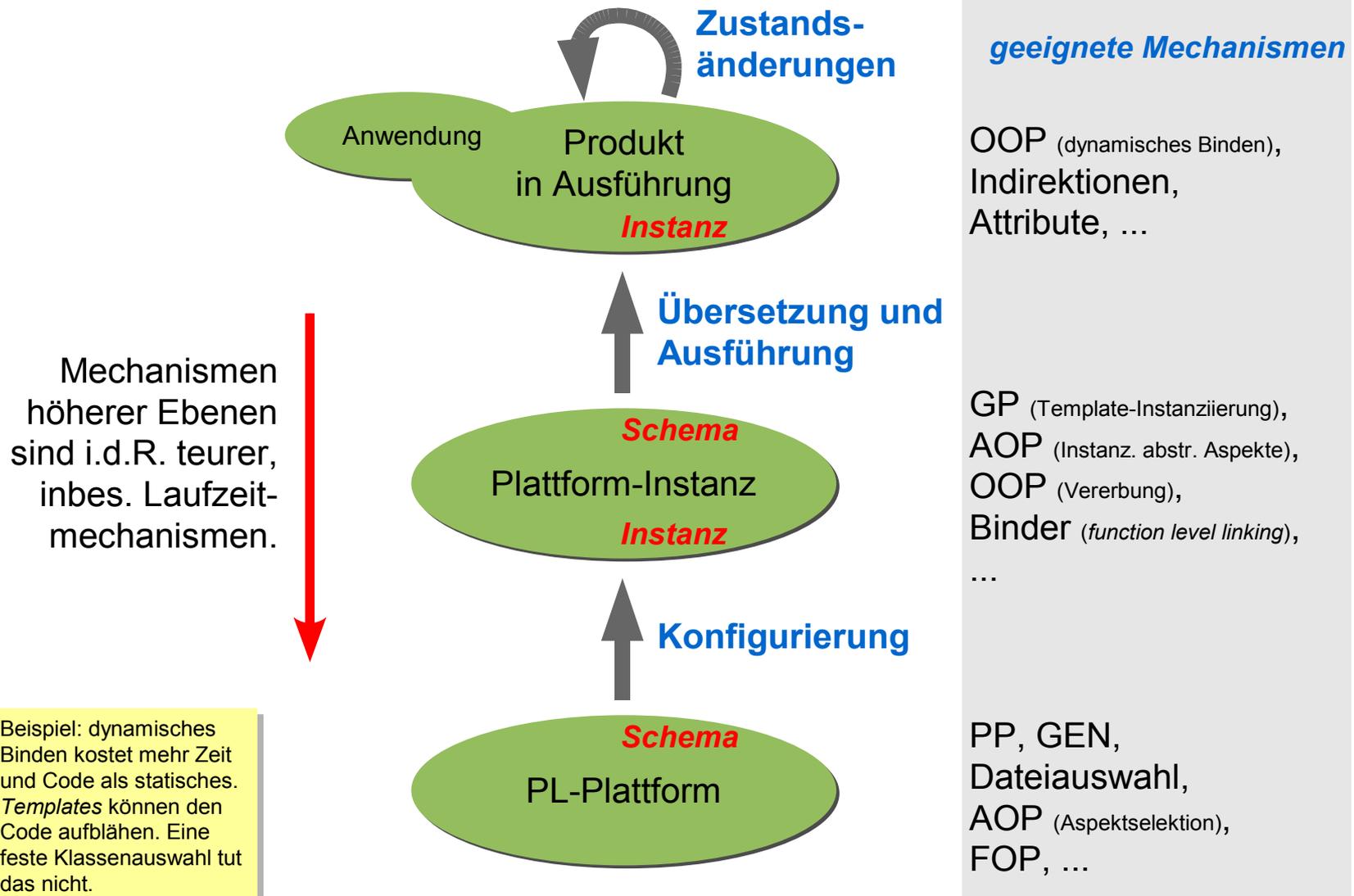


Mechanismen einer bestimmten Ebene können auch die Variabilität unterer Ebenen realisieren.

Beispiel: statt Klassen statisch mit *Policies* zu parametrisieren könnte man auch das dynamische *Strategy* Muster verwenden.



# Welcher Mechanismus ist wofür?



# Zwischenfazit: Stand der Kunst

---

... in der **Betriebssystem-Produktlinienentwicklung**

- **Insgesamt: durchaus erfolgreich**

- OSEK und eCos sind erfolgreiche Beispiele aus dem Bereich eingebetteter Systeme

- **Im Detail: durchaus nicht trivial**

- es gibt viele Möglichkeiten, Plattformkomponenten zu realisieren
- grundsätzlich sollte gelten: Konfigurierungsentscheidungen so früh wie möglich treffen (z.B. statisch statt dynamisch)
  - spart Ressourcen und erspart dem Anwender unnötige Komplexität

- **Es bleiben viele Herausforderungen**

- **(Automatisches) Finden** der optimalen Variante
- **Komposition** von Produktlinien
- Konfigurierung **nicht-funktionaler Eigenschaften**



# Herausforderung: Finden der optimalen Variante

## ■ Gegeben

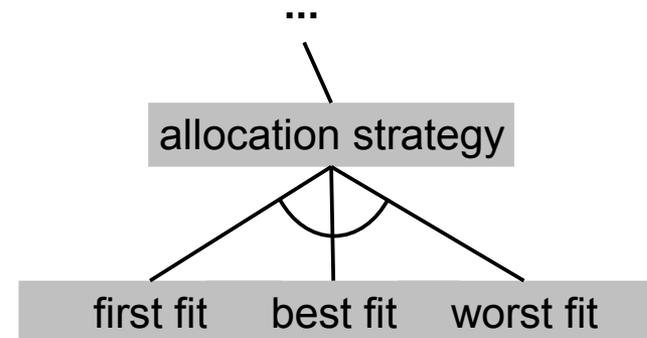
- **Anwendung** (z.B. Steuergerät)
- Feingranular konfigurierbare **Produktlinie** (z.B. PURE: ~**250** konfigurierbare Merkmale)

## ■ Gesucht

- **Optimale Variante** (=Konfiguration)
- Entsprechend den Anforderungen der Anwendung

**Merkmale der PL oft begrifflich „zu weit entfernt“ von den Anforderungen der Anwendung**

**Unterschiede nur für „Insider“ erkennbar**



# Herausforderung: Finden der optimalen Variante

## ■ Gegeben

- **Anwendung** (z.B. Steuergerät)
- Feingranular konfigurierbare **Produktlinie** (z.B.

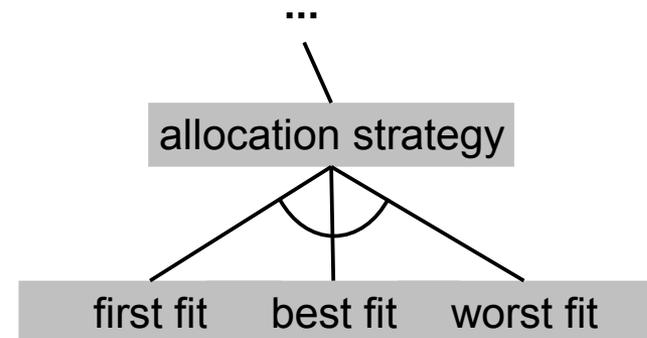
## ■ Gesucht

- **Optimierung**
- **Entscheidung**

**Ziel: Automatische Merkmalauswahl**

Merkmale der PL oft begrifflich „zu weit entfernt“ von den Anforderungen der Anwendung

Unterschiede nur für „Insider“ erkennbar



## Herausforderung: Komposition von Produktlinien

---

### Existierende Produktlinien sind **Insellösungen**

- eng abgesteckte Domäne
- implementierungsnahe Merkmale
- eigene Modellierungssprachen und -konzepte
- eigene Konfigurierungswerkzeuge



# Herausforderung: Komposition von Produktlinien

---

## Existierende Produktlinien sind **Insellösungen**

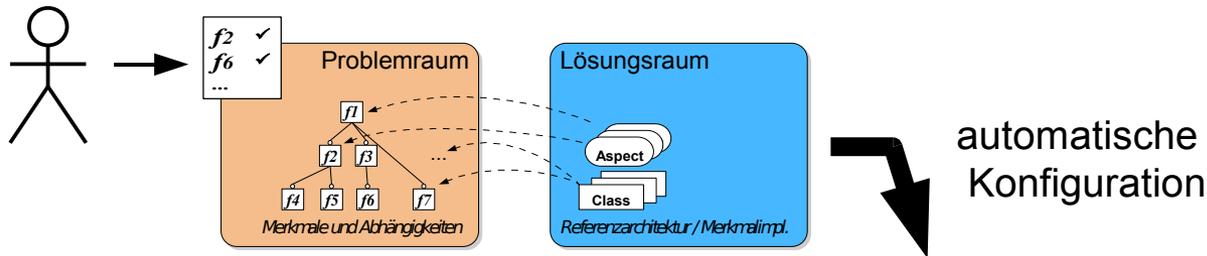
- eng abgesteckte Domäne
- imple
- eigen
- eigene Konfigurierungswerkzeuge

**Ziel: Kombinierbare Produktlinien**

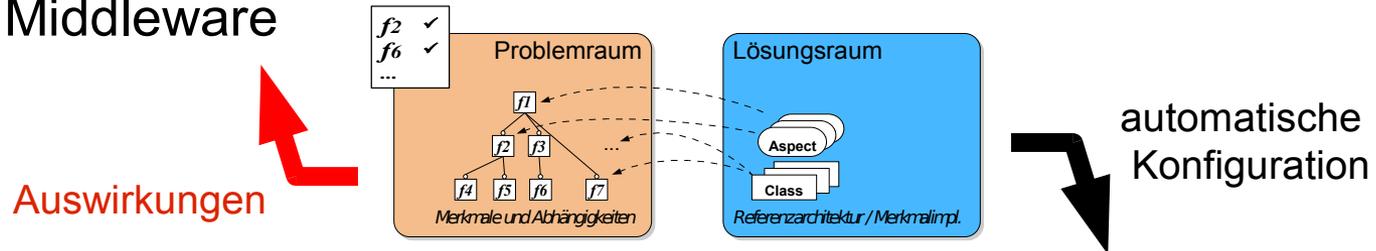


# Szenario: Komposition von Produktlinien

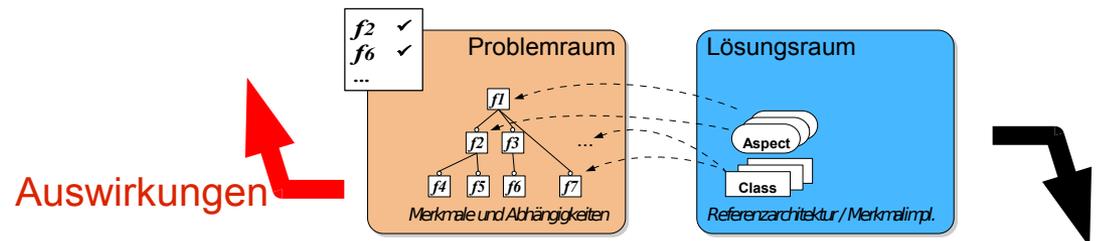
## Schicht 3: Anwendungen



## Schicht 2: Middleware



## Schicht 1: Betriebssystem



## Schicht 0: Hardware



## Herausforderung: nicht-funktionale Eigenschaften

---

Als **nicht-funktional** werden jene **Eigenschaften** eines Softwaresystems bezeichnet, die nicht den eigentlichen Funktionsumfang betreffen, jedoch beim **Betrieb der Software beobachtbar sind**.

- Performanz
- Ressourcenverbrauch
- Skalierbarkeit
- Vorhersagbarkeit
- Latenz
- ...



## Nicht-funktionale Eigenschaften sind...

---

- **erfolgsbestimmend**

- TollCollect 1 *funktionierte* prima...
- oft dominierend über funktionale Eigenschaften

- **emergent**

- nicht sichtbar auf der Ebene einzelner Komponenten
- ergeben sich „plötzlich“ aus der Gesamtkomposition
- **nicht trennbar** von der Implementierung funktionaler Belange
- **Effekte des Lösungsraum** ohne Entsprechung im Problemraum

- **nicht direkt konfigurierbar**



## Nicht-funktionale Eigenschaften sind...

---

- **erfolgsbestimmend**

- TollCollect 1 *funktionierte* prima...

- do

### **Ansatz: Indirekte Beeinflussung**

- über Merkmale mit **bekannt positiven** Auswirkungen auf nicht-funktionale Eigenschaften

- über **konfigurierbare Architektureigenschaften**

### **Rückabbildung in den Problemraum**

- Metriken und Heuristiken erforderlich

- **nic**



# Beispiel: Interrupt Synchronisation

Execution Level

## Configuration 1

- hard synchronisation
- $\mu$ -controller OS

Thread

## Configuration 2

- pro-/epilogues
- Linux, NT, PURE, ...

Thread

## Configuration 3

- driver threads
- L4Ka, ...

Thread

## Configuration 4

- irq pseudo-threads
- Solaris

Thread

**Thread**

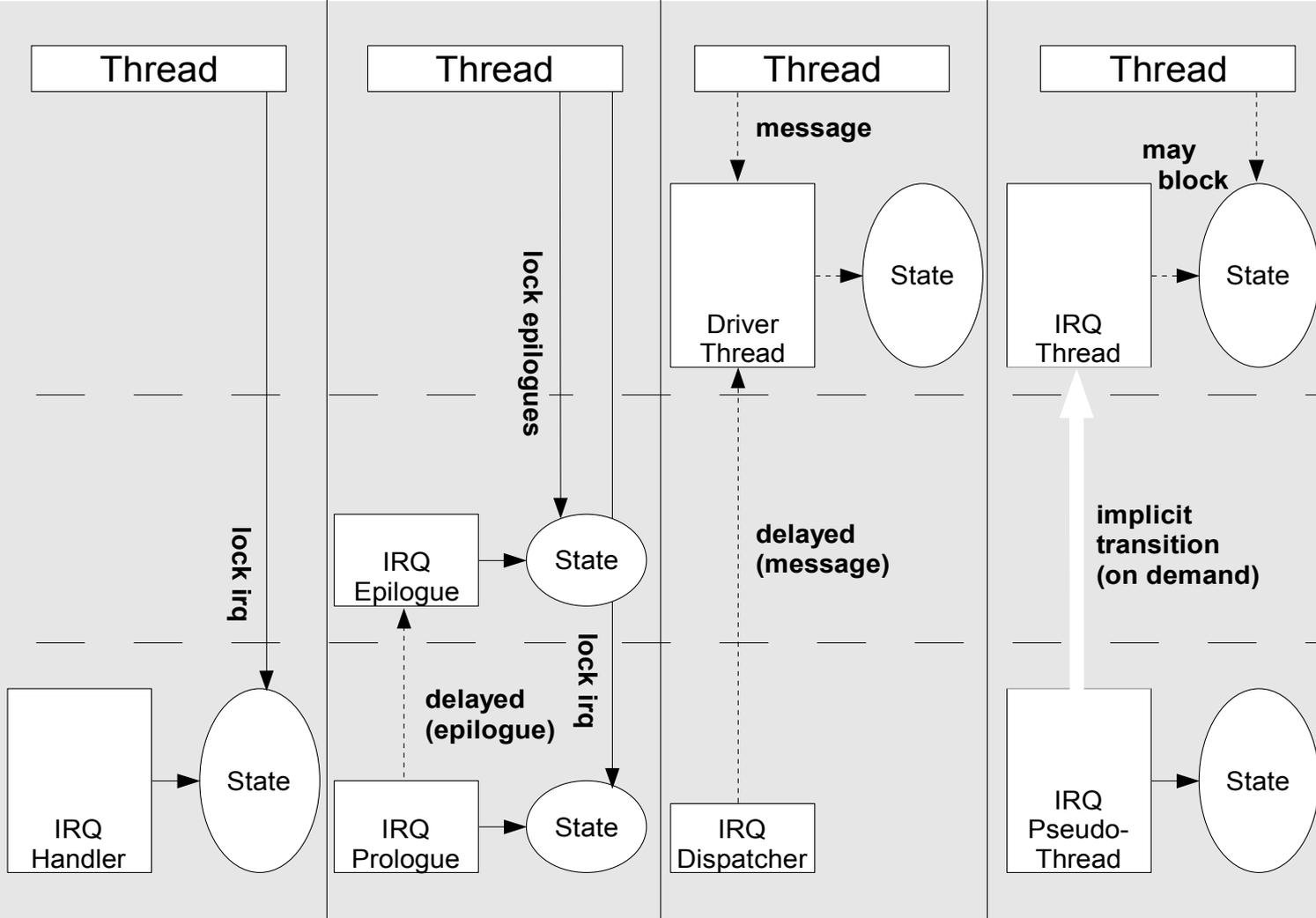
- preemptable
- interruptable

**Epilogue**

- non preemptable
- interruptable

**IRQ**

- non preemptable
- non interruptable



# Architekturneutrales Treibermodell

---

- Methoden werden Synchronisationsdomänen zugeordnet
  - **synchronized**
    - Methoden sind durch ein Software-Protokoll geschützt
    - Aufruf von der IRQ Ebene kann verzögert werden
    - Aufruf von der *Thread* Ebene erfordert Sperren
  - **blocked**
    - Methoden sind auf der IRQ Ebene geschützt
    - Aufrufe von der *Thread* Ebene erfordern eine IRQ Sperre
  - **transparent**
    - Methoden brauchen nicht geschützt werden
    - von jeder Ebene aufrufbar
- Aspekte fügen die nötigen Synchronisationsaufrufe hinzu!



# Beispiel: ein Wecker

---

```
class Timer {
    ... // state
public:
    void init( long time );
    long get() const;
    void add_event(const EventCallback* cb);
private:
    void tick();
    void process_events();
    void handler() {
        tick();
        process_events();
    }

    // what belongs to which synchronization class
    pointcut int_handler() = "% Timer::handler()";
    pointcut blocking() = "% Timer::init(...)"
        || "% Timer::tick()";
    pointcut transparent() = "% Timer::get(...)const";
    pointcut synchronized() = "% Timer::%(...)" && !int_handler()
        && !blocking() && !transparent();
};
```



# Harte Synchronisation

---

```
aspect Configuration1 {
    pointcut block() = Timer::synchronized()
                    || Timer::blocking();
    advice call( block() && !within( block()
        || Timer::int_handler() ) : around() {
        disable_int();
        tjp->proceed();
        enable_int();
    }
};
```



# Pro-/Epilogue Modell

---

```
aspect Configuration2 {
    pointcut block() = Timer::blocking();
    pointcut delay() = Timer::synchronized();

    advice call( block() ) && !within( block()
        || Timer::int_handler() ) : around() {
        disable_int();
        tjp->proceed();
        enable_int();
    }
    advice call( delay() )
        && !within( "% Timer::%(...)" ) : around() {
        lock_epilogues();
        tjp->proceed();
        leave_epilogues();
    }
    advice call( Timer::synchronized() )
        && !within( Timer::synchronized() ) && cflow(
        execution( Timer::int_handler())): around() {
        add_epilog (tjp->action ());
    }
};
```



# Zusammenfassung: Betriebssystem-PL

---

- Anwendung im Kleinen: Erfolgreich
  - das sollten die Übungen zur Lehrveranstaltung gezeigt haben
- Anwendung im Großen: Viele offene Fragen
  - Variantenvielfalt
  - Beherrschbarkeit nicht-funktionaler Belange
  - Integration und Komposition unterschiedlicher PL
  - ...

**Methodisch steht die BS-PL-Entwicklung erst am Anfang!**

**Mithilfe erwünscht!**



# Literatur

---

- [1] G. Böckle, P. Knauber, K. Pohl, K. Schmid (Hrsg.). *Software-Produktlinien*. dpunkt.verlag, 2004. ISBN 3-89864-257-7.
- [2] K. Czarnecki und U.W. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, 2000. ISBN 0-201-30977-7.
- [3] pure-systems GmbH. *Variantenmanagement mit pure-variants*. Technical White Paper, <http://www.pure-systems.com/>.

