# THE IMPACT OF VARIABILITY MECHANISMS ON SUSTAINABLE PRODUCT LINE CODE EVOLUTION

Thomas Patzke, Fraunhofer Institute Experimental Software Engineering (IESE)
Research Group Variation Management

thomas.patzke@iese.fraunhofer.de

SE 2010, Paderborn
26.2.2010

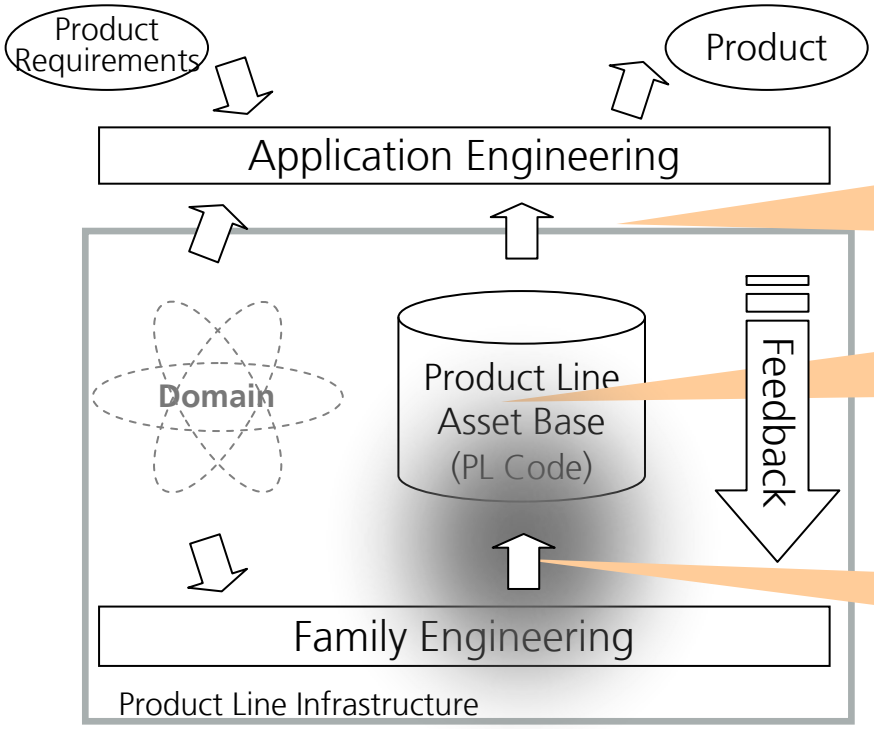


Fraunhofer IESE

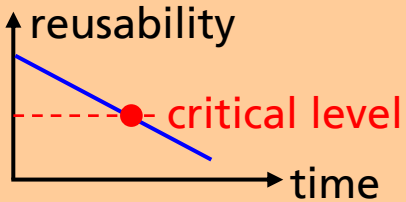
# OVERVIEW

- Background & Problem
- Solution Ideas
- Overall Contributions
- Details
  - PL Complexity Measurement
  - Case Study
- Conclusion

# Background & Problem: Product Line Infrastructure Evolution




**Problem in practice*:**




**Cause: code decay due to**
- drift from changing scope
- increasing internal complexity due to implemented changes

**Our focus: Practical guide to well-behaved evolution of PL code so that its decay is avoided**

- Main challenge: Keeping code reusable!

*Ricoh, POSCO, Bosch, Testo, John Deere, ...

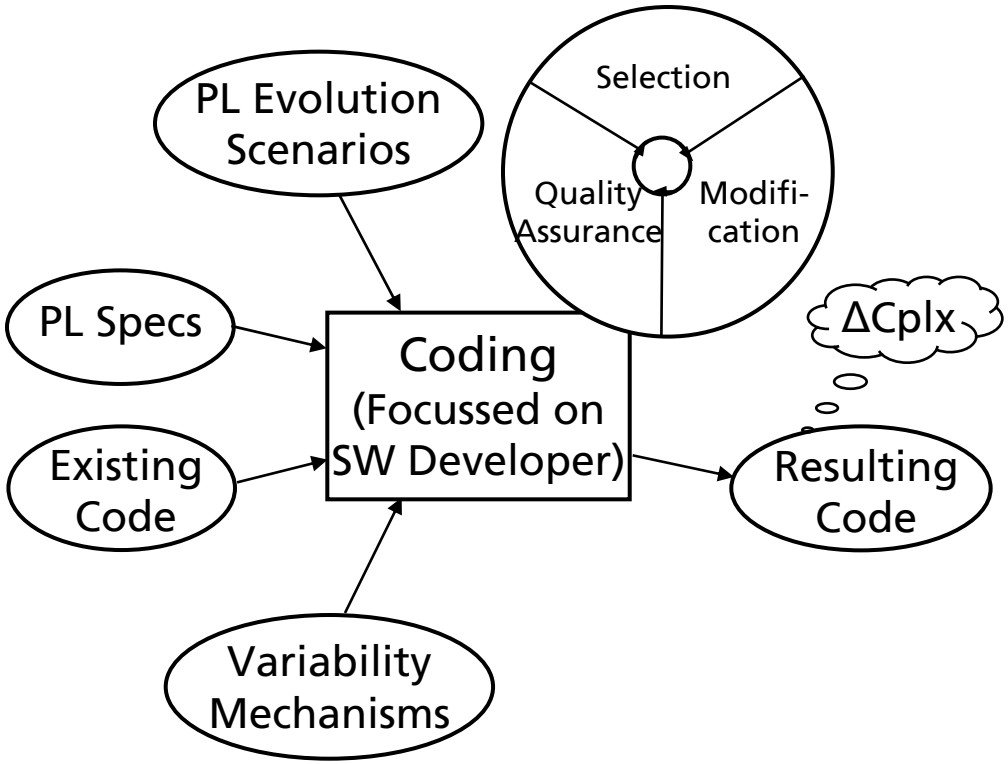
# Towards a Solution (1/2)

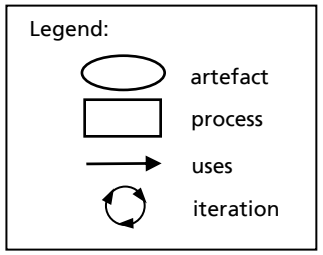
- The evolution problem has been addressed
  - for general systems and single SW systems, in theory and practice
- But it has not been tackled for product lines
  - delta: genericity (common & variable parts)
- We address these novel issues:
  - What makes product line code complex?
  - How can it be evolved well with ‘just enough’ effort?

# Towards a Solution (2/2)

- Code which is used as-is does not pose new challenges
  - because it is not variable, like single systems code
- New challenges lie in adaptable code
  - variability mechanisms make it adaptable
- Variability mechanisms make code more complex, which is unavoidable
  - but the unsystematic use of mechanisms makes code more complex **than necessary**
- Various types of variability mechanisms exist in practice and research
- **Our primary hypothesis:**
  - **Selecting the right combination of variability mechanisms is the key factor for keeping product line code reusable**

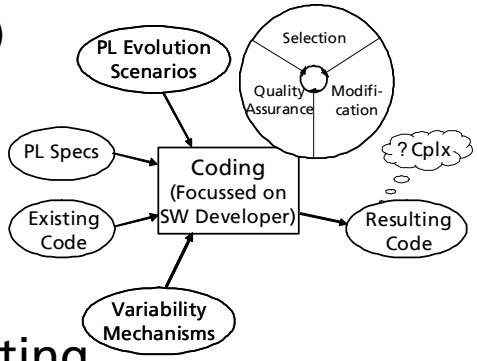
# Solution: Product Line Implementation Process



# Main Contributions to Applied Research in Product Line Engineering

- Development of a method for preventing product line "code aging", consisting of
  - a pattern language of variability mechanisms (novel in this depth): Cloning, Cond. Exec./Compil., Polymorph., Partial Bdg., Aspect-Or., Frame Tech.
  - product line evolution scenarios (as yet unexplored)
  - a method core, consisting of these iterative phases:
    - selection (novelty: PL "code smells")
    - modification (new: PL refactorings)
    - quality assurance (novelties: PL construction testing, **PL complexity measurement**)



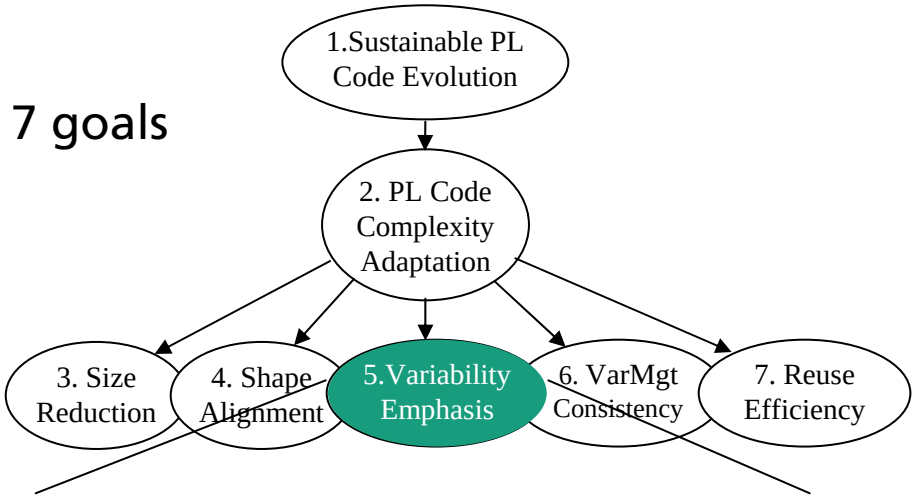

- Validation of vital parts of the method in a **case study**
  - result: there is no silver bullet for PL implementation

Focus of this talk

# Product Line Complexity Measurement (1/2): GQM
## - Goals and Questions

- Goal-oriented approach (application of GQM method)

1. Formulation of goals
   - Result: goal hierarchy of 7 goals




2. Refinement of goals to questions
   - Result: 23 questions
     - excerpt for goal 5:

| | |
|---|---|
| Analyze the | code of software product lines |
| for the purpose of | **emphasizing** |
| with respect to | **variable parts** |
| from the viewpoint of the | software developer |

Q14: How many variable parts are visible at the module level? (How many should be?)
Q15: How many variable parts are visible module-internally? (How many should be?)
Q16: How many variable parts are indistinguishable from common code?

# Product Line Complexity Measurement (2/2)

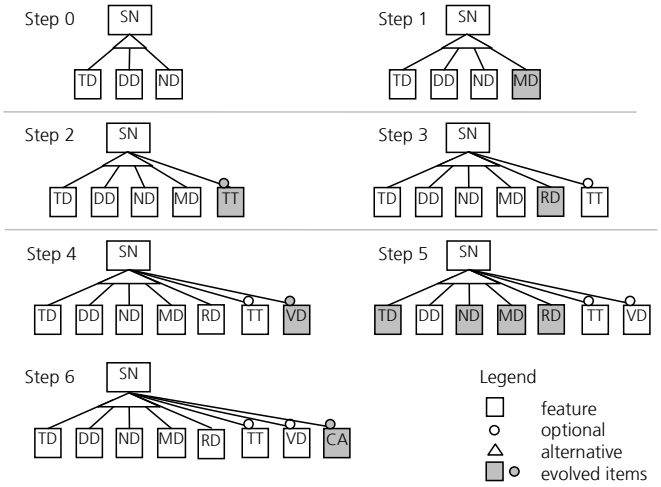
## - Metrics

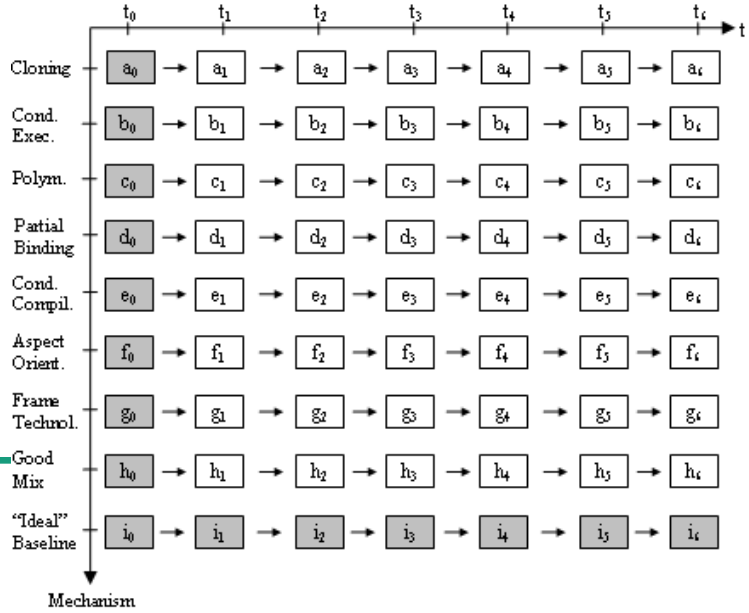
3. Refinement of questions to metrics
   - Result: metrics suite of 21 PL complexity metrics
     - excerpt for goal 5 / questions 14-16:

| G | Q | Metric name | Description |
|---|---|---|---|
| 5 | **Variability emphasis** | | |
| | 14 | **$NVPrt_e$** | Number of **externally visible** variable parts |
| | 15 | **$NVPrt_i$** | Number of **internally visible** variable parts |
| | 16 | **$NVPrt_a$** | Number of **ambiguous** variable parts |

# Case Study (1/5): Setup

- Development & evolution of product lines for resource-constrained embedded systems (Wireless Sensor Nodes)
- PL evolution over 6 steps, covering different PL evolution types
- using all variability mechanisms in monocultures, plus “ideal” baseline and “good enough” mechanism mix







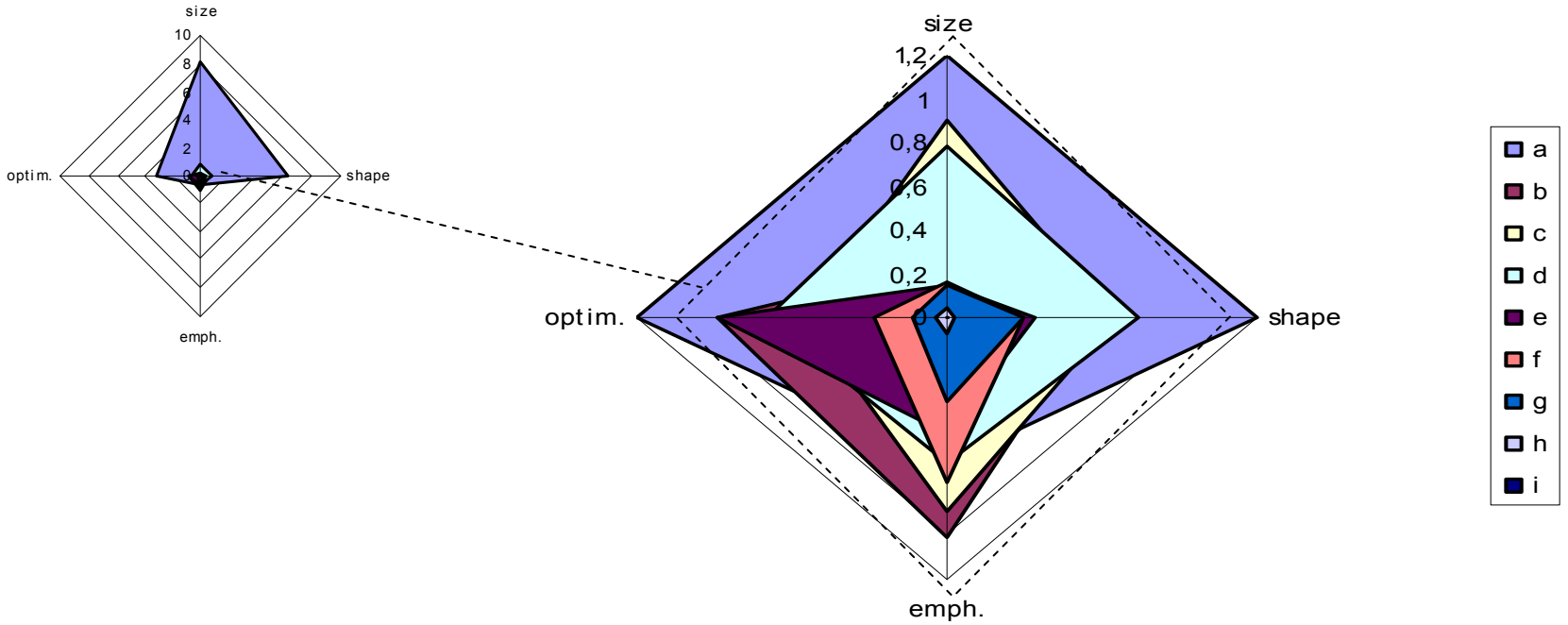

Fraunhofer IESE

# Case Study (2/5): Hypotheses

| No | Hypothesis |
|---|---|
| 1 | Product line code becomes more sustainable by context-specific variability mechanism selection. |
| 2 | Except in the short term, code obtained by Cloning is harder to evolve than code with any other variability mechanism. |
| 3 | In the long term, a monoculture of a variability mechanism is harmful for product line code quality. |
| 4 | Runtime variability mechanisms unnecessarily increase product line code complexity. |
| 5 | As a variability mechanism, Aspect-Orientation is obsolete. |

# Case Study (3/5): Measurement Results

## - Complexity dimensions (after final evolution step 6)

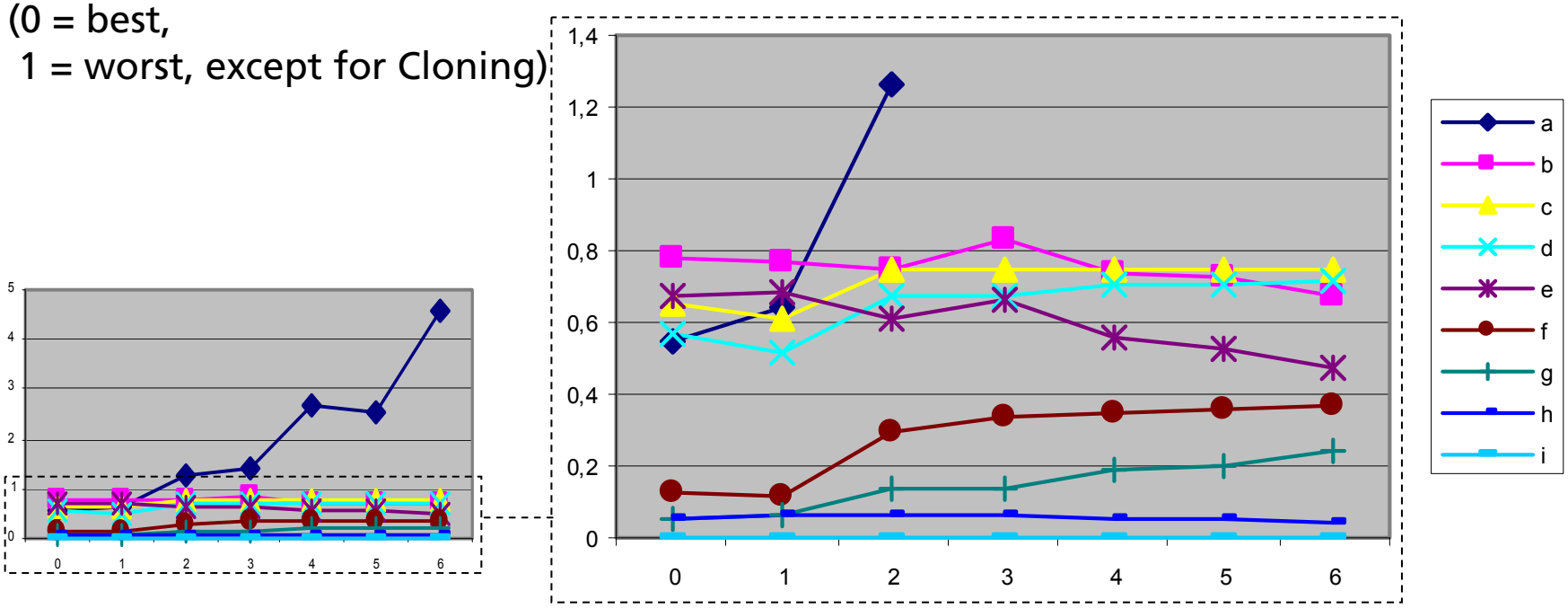



Mechanisms:
a: Cloning, b: Cond.Exec., c: Polym., e: Cond.Compil.,
f: Aspect-Or., g: Frame Techn., h: good enough mix,
i: “ideal” spatial baseline

# Case Study (4/5): Measurement Results

## - Normalized complexity trends

Unweighted average complexity
(0 = best,
1 = worst, except for Cloning)




Mechanisms:
a: Cloning, b: Cond.Exec., c: Polym., e: Cond.Compil.,
f: Aspect-Or., g: Frame Techn., h: good enough mix,
i: "ideal" spatial baseline





Fraunhofer IESE

# Case Study (5/5): Results

| No | Hypothesis | Supported? |
|---|---|---|
| 1 | Product line code becomes more sustainable by context-specific variability mechanism selection. | ✓ |
| 2 | Except in the short term, code obtained by Cloning is harder to evolve than code with any other variability mechanism. | ✓ (strongly) |
| 3 | In the long term, a monoculture of a variability mechanism is harmful for product line code quality. | ✓ |
| 4 | Runtime variability mechanisms unnecessarily increase product line code complexity. | ✓ |
| 5 | As a variability mechanism, Aspect-Orientation is obsolete. | ✓ |

# Conclusion

- We have developed a method for preventing product line “code aging”, consisting of
  - a pattern language of variability mechanisms
  - product line evolution scenarios
  - a method core, consisting of selection, modification and QA phases (containing PL complexity measurement as GQM instance)
- Vital parts of the method have been validated in a case study
- Recommendations
  - Cloning is useful in short-term evolution, but most detrimantal later
  - Monocultures and runtime mechanisms lead to over-complexities
  - A mix of Frame Technology and Conditional Compilation can keep PL code sufficiently simple in the long term

# Thank you!