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

Main challenge: Keeping code reusable!

Problem in practice*:
- reusability
- critical level
- time

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

*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

PL Specs

Existing Code

Variability Mechanisms

Coding (Focussed on SW Developer)

Resulting Code

ΔCplx

PL Evolution Scenarios

Selection

Quality Assurance

Modification

Legend:

- artefact
- process
- uses
- iteration
Main Contributions to Applied Research in Product Line Engineering

- Development of a method for preventing product line “code aging”, consisting of
  - 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
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:

<table>
<thead>
<tr>
<th>Analyze the code of software product lines for the purpose of emphasizing with respect to variable parts from the viewpoint of the software developer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q14: How many variable parts are visible at the module level? (How many should be?)</td>
</tr>
<tr>
<td>Q15: How many variable parts are visible module-internally? (How many should be?)</td>
</tr>
<tr>
<td>Q16: How many variable parts are indistinguishable from common code?</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>G</th>
<th>Q</th>
<th>Metric name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>Variability emphasis</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>NVPrt_e</td>
<td>Number of externally visible variable parts</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>NVPrt_i</td>
<td>Number of internally visible variable parts</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>NVPrt_a</td>
<td>Number of ambiguous variable parts</td>
<td></td>
</tr>
</tbody>
</table>
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
# Case Study (2/5): Hypotheses

<table>
<thead>
<tr>
<th>No</th>
<th>Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Product line code becomes more sustainable by context-specific variability mechanism selection.</td>
</tr>
<tr>
<td>2</td>
<td>Except in the short term, code obtained by Cloning is harder to evolve than code with any other variability mechanism.</td>
</tr>
<tr>
<td>3</td>
<td>In the long term, a monoculture of a variability mechanism is harmful for product line code quality.</td>
</tr>
<tr>
<td>4</td>
<td>Runtime variability mechanisms unnecessarily increase product line code complexity.</td>
</tr>
<tr>
<td>5</td>
<td>As a variability mechanism, Aspect-Orientation is obsolete.</td>
</tr>
</tbody>
</table>
Case Study (3/5): Measurement Results
- Complexity dimensions (after final evolution step 6)

Mechanisms:
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
## Case Study (5/5): Results

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