The Impact of Variability Mechanisms on Sustainable Product Line Code Evolution

Thomas Patzke

Product Line Architectures Department
Fraunhofer Institute Experimental Software Engineering (IESE)
Fraunhofer-Platz 1
67663 Kaiserslautern
thomas.patzke@iese.fraunhofer.de

Abstract: Many software development organizations today aim at reducing their development effort, while improving the quality and diversity of their products by building more reusable software, for example using the product line approach. A product line infrastructure is set up for deriving the similar products, but this infrastructure degenerates over time, making reuse increasingly hard. As a countermeasure, we developed a practical method for guiding product line developers in evolving product line code so that its decay caused by reuse is avoided. This paper gives an overview of some of our findings.

Because product line code differs from single systems code only in its genericity, expressed by variability mechanisms, we analyzed to what degree the selection of certain mechanisms affect the code's reuse complexity. Using the Goal-Question-Metric (GQM) approach, we developed a quality model that lead to an extensible product line complexity metrics suite.

A case study compared the evolution qualities of different product line implementations, with the following results: Cloning, the simplest mechanism, leads to similar short-term complexities than more advanced ones, making its interim usage appropriate. In the longer term, any other mechanism has a clearly lower complexity trend, especially if it is selected according to the variability management task at hand. A mix of Conditional Compilation and Frame Technology provides the best long-term evolution potential.

1 Introduction

Many software development organizations today aim at reducing their development effort, while at the same time improving the quality and diversity of their software products by building more reusable software. Due to the diversity of similar software products, it is often insufficient to provide reusable software as fixed building blocks only. Truly reusable software requires adaptation possibilities in order to be reused effectively in different situations. This means that although it contains common parts, it must also provide some means to variation. More recently, more and more reusable software of larger scale is developed in a conscious organization-specific engineering effort alongside the software that reuses it. In these approaches, a set of similar software systems is developed together in a cost-effective way. In these product line engineering approaches, the reusable assets consist of common and variable artifacts, plus descriptions of variability interdependencies, across all stages of the software engineering life cycle, such as requirements, architecture, design or source code. Together they form the organization's product line infrastructure [Mut02].

Once a product line infrastructure has been initiated, individual software products can be derived from it, and if the infrastructure is well-designed initially, it should be easy to derive products from it for some time. However, development in general and product line development in particular do not end with initial construction, but successful development is accompanied by continuous evolution ([Par94], [Som04], [LF06]). In a product line setting, continuous evolution means that gradually, new products are added to the product line and existing common and variable features are changed which were only partially predicted, or which were not predicted at all.

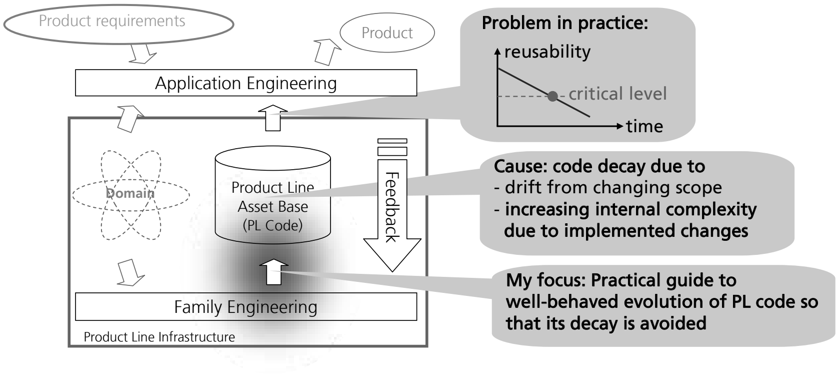
Product requirements
Product
Problem in practice:
reusability
critical level
time
Application Engineering
Domain
Product Line Asset Base (PL Code)
Feedback
Cause: code decay due to
- drift from changing scope
- increasing internal complexity due to implemented changes
My focus: Practical guide to well-behaved evolution of PL code so that its decay is avoided
Family Engineering
Product Line Infrastructure

Figure 1: The product line code evolution problem

Figure 1 shows our observations in practice: Within the product line engineering life cycle, it becomes increasingly difficult to derive products from the product line code over time. The reason for this phenomenon is product line aging, analogous to single systems software aging [Par94]. Code degenerates for two reasons: First, because it is not changed according to the changing scope, so that both drift apart. Second, it ages when it is changed inappropriately [Kru07], for example because the developers are ignorant of product line implementation technology or because they adopt it overambitiously. Our method focuses on the latter issue. It analyzes practical possibilities for implementing variabilities in source code and instantiates a sustainable coding process.

As shown in Figure 2, existing product line code and new product line specifications are the input to our approach, targeting the developer. Other inputs are variability mechanisms (possibilities to realize the new variability-related specifications in the code; Chapter 3) and product line evolution scenarios (elementary ways in which variabilities may evolve). The output of the iterative and incremental approach is new product line code with just enough complexity to be easily evolvable. The iteration consists of selection, modification (refactoring) and quality assurance phases. Quality assurance consists of product line testing and measurement (Chapter 4) sub-activities which ensure that all product line members can be constructed and executed as required, and that the applied variability mechanisms are simple enough for sustainable evolution.

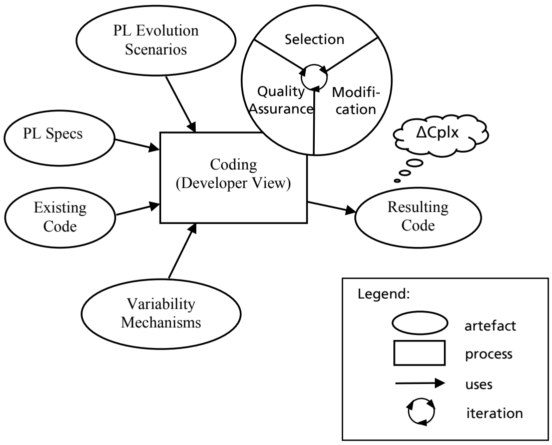

Figure 2: Product line coding approach

The method was developed and applied in a case study (Chapter 5), in which product lines of wireless sensor node software have been developed. Chapter 6 gives a summary and presents open issues and further work.

2 Related Work

Our approach for combines and extends theories and concepts from several different areas, such as software evolution, product line implementation, practical reuse methods, complex systems description and sustainable evolution of complex systems.

Observing the long-term evolution of large single systems used in practice, 8 evolution rules have been proposed [LF06], including the phenomenon of software aging [Par94], and a number of complexity measurements been proposed ([AD07], [Kel06]). Collections of product line implementation mechanisms ([AG01], [SVB05], [CE00], [JGJ97]) have mostly had a theoretical scope, sometimes biased towards a particular solution, but not regarding long-term effects. More practical methods have been proposed ([MP03], [Bas97], [Kru07], [Cop98]), which only partially address sustainable evolution. Work on complex systems description ([Sim62], [Ley01], [CZ03]) suggested that such systems should be described by actions leading to symmetries, rather than just describing the shape of end products. Whereas these are only analytical, work on sustainable evolution of such systems [Ale02] suggests a generic recipe for actively creating well-behaved complex systems, involving certain steps in a particular order. We developed our method as an instance of that approach.

3 Variability Mechanisms

As mentioned in Chapter 1, the additional complexities in product line code evolution compared to single systems code evolution have to do with variability and how variability is managed in the code. A developer may select from a number of different variability mechanisms to realize and configure a new product instance that reuses a given product line infrastructure. In practice, variability mechanisms are often used ad-hoc or in monocultures, which leads to overly complex code. In order to guide the developer in selecting appropriate mechanisms, we have developed a pattern language of variability mechanisms [Pat08], summarized in this chapter. The selected mechanisms include Cloning, Conditional Execution, Polymorphism, Partial Binding, Conditional Compilation, Aspect-Orientation and Frame Technology. We believe that our pattern language captures the major possibilities for organizing variable code for two reasons. First, because each chosen mechanism is frequently used for variability management in practice or has shown new and unique variability management possibilities in practical research. Second, because from a developer's perspective, the mechanism set covers all major variability management situations, as illustrated in Figure 3.

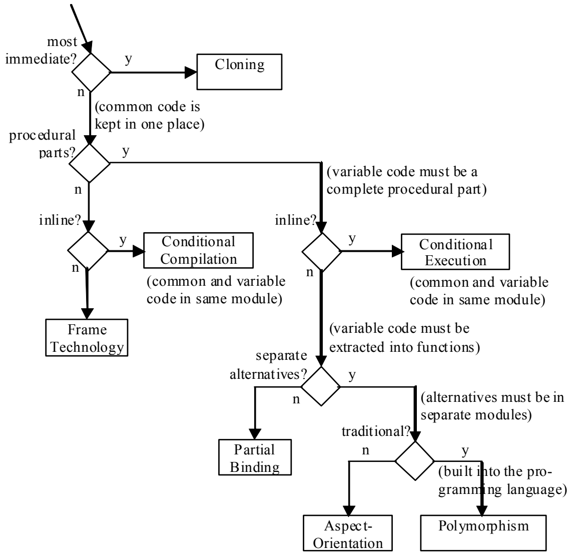
most immediate?
y
n
Cloning
(common code is kept in one place)
procedural parts?
y
n
(variable code must be a complete procedural part)
inline?
y
n
Conditional Compilation
(common and variable code in same module)
Frame Technology
inline?
y
n
Conditional Execution
(common and variable code in same module)
(variable code must be extracted into functions)
separate alternatives?
y
n
(alternatives must be in separate modules)
Partial Binding
traditional?
n
y
(built into the programming language)
Aspect-Orientation
Polymorphism

Figure 3: Variability management decisions resulting in variability mechanisms

When a new product must extend existing infrastructure code, the simplest approach is **Cloning**, i.e. to duplicate the existing code and to modify the new copy. Cloning is simple because at the moment when it is applied it achieves just what is required in the most direct way: the new product is based on existing code, and existing products are not affected. However, the approach is least sustainable, because it leads to maximum duplication of common code, which must be co-evolved consistently throughout the product line's future lifetime, which is most of the time the code spends its life. This way, Cloning it is most detrimental for long-term product line code evolution.

The only sustainable alternative is to sacrifice direct duplication, so that at least some common code is kept in one place, while the new variability is introduced otherwise. A straightforward way for differentiating a variable code part from previously undifferentiated code is to enclose it by a conditional statement. The new variability is added as a new branch of the conditional statement. The resulting variability is bound at execution time, which is why the mechanism is called **Conditional Execution**. A disadvantage of this approach is that common and variable code parts reside and evolve in a single module, because the parameter of variation is closed [Bas97]. This problem is solved in many programming languages by using an indirect rather than a direct conditional statement for differentiating the variable parts, for example through function pointers, Template Methods, or generic types [CE00]; that is, by **Polymorphism**.

If the variation points can be identified as unique procedures in the common code, a non-traditional way is to extract the variable parts into aspects, using **Aspect-Orientation**. A more traditional alternative is to store all variable procedural code parts in a single module, different from the common code. We call this mechanism **Partial Binding**, because the parameters of variation are partially bound: their calling sites do not specify completely which functions they call. For example, they only provide a forward declaration, and the missing function definition is completed by stating the variable module at preprocessing- or link-time.

Except for Cloning, the above mechanisms are programming language-dependent. Conditional Execution and (runtime) Polymorphism are especially problematic in the larger context of product line implementation because they result in runtime binding, which leads to a single bloated system that contains all variable features, rather than a proper product line, a set of similar systems evolved together. Runtime conditionals are also inherent parts of software code, and if they are also used for variability management, "normal" code and variability managing code become deeply entangled. However, it is not necessary to express the condition with programming language code. An alternative is the preprocessor conditional, such as the #ifdef statement of the C and C++ preprocessor, binding the parameters of variation at preprocessing time, in **Conditional Compilation**. This way, variable parts may cover arbitrary code sections, independent of their semantics. Conditional Compilation facilitates reuse in a broader sense, where the preprocessor serves as a construction interpreter [Bas97]; like Conditional Execution it is a closed-parameter mechanism. At least one open-parameter construction-time mechanism exists: **Frame Technology** [Bas97], which uses an advanced preprocessor, such as the FP frame processor developed by the author of this paper [PM03], for assembling construction modules. These may contain source code text, whose default variable parts are annotated and can be overridden.

4 Product Line Complexity Measurement

As mentioned in Chapter 1, one sub-activity of the quality assurance phase of our method is concerned with measurement in the resulting product line code (cf. Fig. 2), more precisely with complexity measurement. Until recently, there had been no such research for product lines [Kna04]; this situation is currently changing ([AD07],

[LT08]). However, none of these consider the measurement purpose. We developed a quality model for evolving product line code that motivates which product line-specific quality attributes must be addressed in the measurement phase. It is customizable to an organization's product line development needs. The quality model has been developed using the Goal-Question-Metric (GQM) approach [S++02], the most popular mechanism for goal-oriented software measurement. GQM depends on the formulation of the following elements: Goals, questions and metrics. First, goals are formulated, which define what shall be achieved. The goals are often decomposed using a goal hierarchy of main goals and corresponding sub-goals. Each goal is refined by questions, whose answers indicate to what extent the goal has been reached. Finally, metrics are given for each question, which makes the questions quantifiable. Figure 4 shows the goals and sub-goals needed in our method.

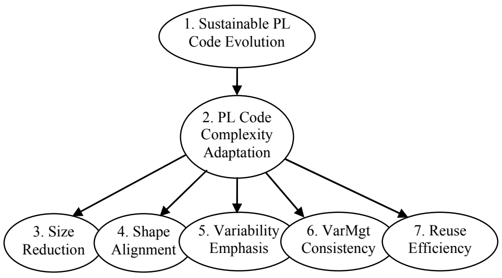

Figure 4: Goal hierarchy of the product line code quality model

The main goal addressed is sustainable evolution of product line code (G1), which aims at making the product line code evolvable over longer periods of time, possibly open-ended. In order to achieve the main goal, product line complexity adaptation is proposed as a sub-goal (G2), which denotes that product line code complexity is reduced, but only as far as necessary in the particular development context. For example, product line code complexity is too high if a module contains more parameters of variation than necessary. There are five mostly orthogonal sub-goals for achieving complexity adaptation. The first is size reduction (G3), which means that the product line code is constrained in size, as required for the variability management purposes at hand. The second sub-goal, code shape alignment (G4), denotes that the shape of common and variable code parts corresponds to the required variability management tasks. For example, closed-parameter mechanisms are usually sufficient for expressing optional variabilities; open-parameter mechanisms would introduce gratuitous complexity. The third sub-goal is concerned with emphasizing variable parts (G5), so that developers can easily see and control those parts of product line code which are different across space or time, while at the same time the common parts are more subdued [Bas97]. Another sub-goal addresses variability management consistency (G6), because inconsistent code is more complex to evolve than necessary, and the last sub-goal is reuse efficiency (G7), which means, for example, that an excess of reusable modules may become as harmful for long-term reuse as a shortage of reusable modules.

According to the GQM method, each goal is refined by questions, and these are then refined by concrete metrics. Table 1 gives an example for the size reduction goal G3. This way, we established a metrics suite of 21 metrics.

Analyze the	code of software product lines
for the purpose of	**reducing**
with respect to	**size**
from the viewpoint of the	software developer

Q7: How large is the code? (Which code size is necessary?)
Q8: How much product line code has changed over time? (How much was necessary?
Q9: How many modules have been used? (How many are necessary?)
Q10: How many variation points are used in the code? (How many are necessary?)

G	Q	Metric name	Description
3	Size reduction		
	7	LOC	Lines of code for entire product line code
	8	$\nabla_{LOC,t}$	Temporal code churn ([HM00], [AD07]) in lines of code
	9	NOM	Number of modules
	10	NVP	Number of variation points

Table 1: Metrics for goal G3: Product line code size reduction

5 Case Study

In order to compare the quality of sustainable code evolution in various product line development contexts, we conducted a case study that monitored and evaluated the evolution of software product lines of small and highly resource-constrained embedded systems. The product line implementations were co-evolved by using each of the mechanisms presented in Chapter 3, applying the overall method shown in Figure 2. In particular, the quality of the results was measured using our quality model (Chapter 4). In this context, four main hypotheses have been investigated, summarized in Table 2.

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

Table 2: Overview of investigated hypotheses

The single systems case, when no variability management is applied, corresponds to the situation when Cloning is used throughout. This results in the first hypothesis, which claims that Cloning has a comparable complexity than the other mechanisms only in initial product line development phases, while it has a more than linear complexity trend compared to other mechanisms in the mid- and long term. The other hypotheses are concerned with groups of mechanisms, except for Cloning. Hypothesis two states that, in the long term, a monoculture of one variability mechanism, for example only Conditional Compilation, as observed in practice, leads to less sustainable code than a context-specific mix of mechanisms. The third hypothesis states that in most cases, runtime variability mechanisms make the code unnecessarily complex, so that the dual construction time mechanisms should be used instead. The fourth hypothesis claims that Aspect-Orientation is obsolete for variability management, as other mechanisms lead to the same result with less effort and complexity.

The subject of the case study is code for small embedded systems - battery-powered wireless sensor nodes which are part of a rapid prototyping platform for Ubiquitous and Pervasive Computing environments [TeCo]. The sensor nodes are able to communicate with internet gateways or with each other, forming an ad-hoc wireless sensor network. They are equipped with various types of actuators and sensors. The case study simulates the evolution of a product line in six steps, as shown in the feature diagram in Figure 5.

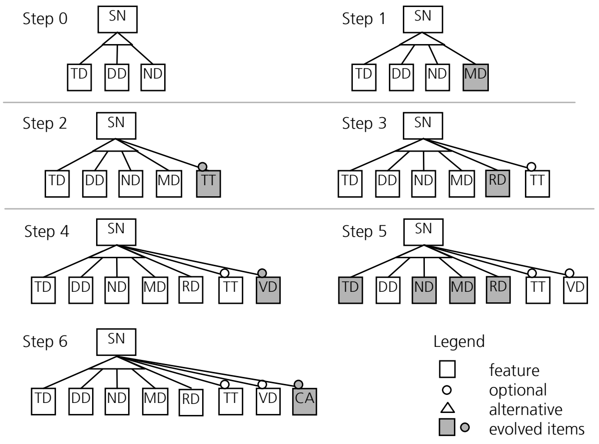

Figure 5: Feature diagram snapshots of the evolving sensor node product line

Initially, the product line consists of the sensor node (SN) and the three alternative features tilt detector (TD), drop detector (DD) and noise detector (ND). In step 1, a new alternative movement detector (MD) is added, step 2 adds an optional time transmission (TT), step 3 a raw data detector (RD), and step 4 a voltage detector (VD). In step 5, most existing alternatives are simplified, and in step 6 a clock adjustment feature (CA) is added.

Figure 6 depicts the evolution trace of the product line code. The sequences “a” to “g” have each been coded using a monoculture of the seven variability mechanisms discussed in Chapter 3. The respective implementations at $t=t_0$ serve as a temporal baseline, to evaluate how complexity changed over time. Sequence “i” represents the “ideal” implementation at each point in time, serving as a spatial baseline, to evaluate how much more complex each implementation is than required. It contains the same C code as in the other implementations, enriched with variability management pseudocode. The pseudocode expresses what activities a human developer or an automated construction interpreter must at least perform in order to assemble all required product instances from the C code parts, e.g. by changing the text at certain lines. It separates what must be performed for "good enough" variability management from how to achieve these tasks. The remaining sequence "h" uses a mix of the available variability mechanisms, staying as close as possible to the baseline "i".

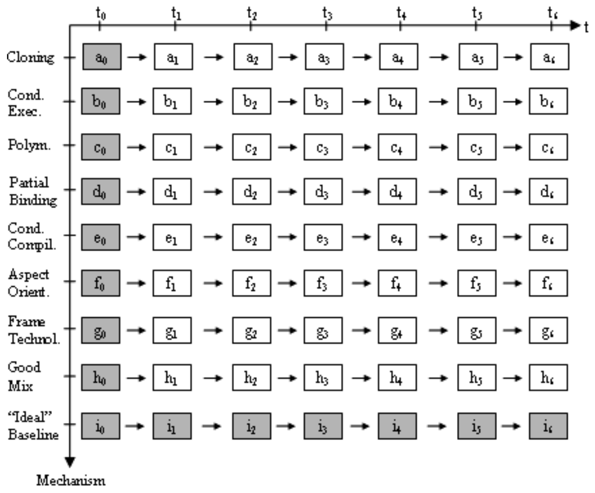

Figure 6: Evolution trace for product line code, with baselines (grayed)

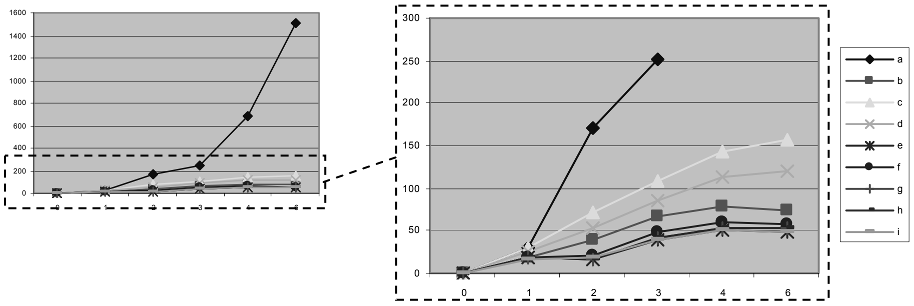

Figure 7: Trends for temporal code size delta (left), and excerpt (right)

While keeping all product line implementations consistent (invariant goal G6), trends for the 17 metrics of all remaining goals were captured, partially automated. As an example, Figure 7 shows the trends for temporal code churn, corresponding to question Q8 in Table 1. The values were then normalized against the worst figures except for Cloning, because its metrics often exceeded the others' considerably. The values ranged between zero for the ideal implementation "i", and one for the worst case. The corresponding values are listed in Table 3. Thereafter, average values were computed for the corresponding goal categories. The same process was repeated for all seven evolution scenarios. Figure 8 summarizes the results.

	LOC	$\nabla_{LOC,t}$	NOM	DRH	WRH	$v(G)_{rt,closed}$	$v(G)_{rt,open}$	$v(G)_{ct,closed}$	$v(G)_{ct,open}$	LOC_{ad}	$NVPrt_e$	$NVPrt_i$	$NVPrt_a$	RR	NOD	$\nabla_{LOC,s}$	K_{var}
a	9,63	11,8	3,18	0,3	5,1	19	2	4	1	12,6	1	1	0	1	1	10	0,3
b	0,23	0,17	0,36	0,3	0,4	0,8	0	0,9	0	0,85	1	1	1	1	1	0,5	1
c	1	1	0,73	0,3	0,6	1	1	0,6	0	0,99	1	1	0,67	0,39	1	0,6	0,18
d	0,68	0,65	1	0,3	1	0,4	0	1	1	1	1	1	0	0,62	1	1	0,17
e	0,05	0,03	0,36	0,3	0,4	0	0	0,4	0	0,62	1	1	0	1	1	0,5	1
f	0,12	0,11	0,27	0,3	0,4	0,5	0	0	0	0,27	1	1	0,5	0,33	1	0,1	0,03
g	0,1	0,09	0,27	1	0	0,4	0	0	0	0,27	1	0	0	0,36	0	0,1	0,05
h	0,1	0,05	0	0	0	0,2	0	0	0	0,06	0	0	0	0,03	0	0,1	0,02
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Name	Meaning
LOC	lines of code
$\nabla_{LOC,t}$	temporal code churn
NOM	number of modules
NVP	number of variation points
DRH	depth of reuse hierarchy
WRH	width of reuse hierarchy
v(G)	cyclomatic complexities
LOC_{ad}	lines of adaptee code
NVPrt	number of variable parts
RR	reuse ratio
NOD	number of defaults
$\nabla_{LOC,s}$	spatial code churn
K_{var}	compression distance of alt. var.

Table 3: Normalized metrics for all mechanisms after evolution step 6

cplx	0	1	2	3	4	5	6
a	0,55	0,64	1,26	1,42	2,65	2,5	4,57
b	0,77	0,77	0,74	0,83	0,74	0,73	0,67
c	0,65	0,61	0,75	0,75	0,74	0,74	0,75
d	0,57	0,52	0,68	0,68	0,7	0,7	0,72
e	0,67	0,69	0,61	0,66	0,56	0,53	0,48
f	0,13	0,11	0,3	0,33	0,35	0,36	0,37
g	0,06	0,06	0,14	0,14	0,19	0,2	0,24
h	0,06	0,06	0,06	0,06	0,05	0,06	0,05
i	0	0	0	0	0	0	0

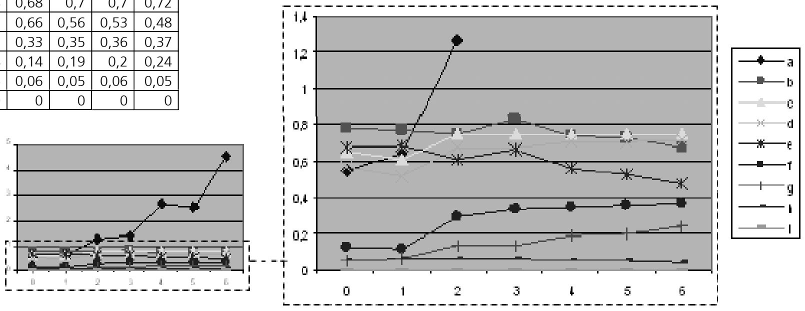

Figure 8: Total complexity trends, compared to ideal implementation

The case study shows that after few product line evolution scenarios, Cloning leads to a considerable excess of complexity, supporting hypothesis 1. The three conventional variability mechanisms Conditional Execution, Polymorphism and Partial Binding result in comparable levels of complexity, with weaknesses in different dimensions. The runtime mechanism Conditional Execution results in 42% higher average complexity than its construction time dual Conditional Compilation, especially due to lack of variability emphasis. This clearly supports part of hypothesis 3. On the other hand, the second runtime mechanism, Polymorphism, only performs 4% worse than its dual, Partial Binding, which does not support hypothesis 3. Conditional Compilation leads to some variability management complexity, mainly due to missing optimization possibilities, but scores best among the closed-parameter mechanisms. Aspect-Orientation performs better, but makes defaults and variation points hard to see. Frame Technology, even in the restricted dialect used in the case study that only supports open parameters of variation, clearly outperforms Aspect-Orientation in almost any category. For that reason, Aspect-Orientation is obsolete when both mechanisms are available, supporting hypothesis 4. Frame Technology performs best among the mechanism monocultures, with a 24% deviation from the ideal implementation after the final evolution step. However, it over-emphasizes variable parts in some cases and leads to a lack of open construction time complexity, due to limitations of the applied frame processor. All monocultures lead to complexity excess, which supports hypothesis 2.

The approach has the following threats to validity: First, the case study has deliberately been kept small, and it can be argued if the found results scale to larger product line code. However, practical experience in various projects indicates that the given hypotheses are useful heuristics for a majority of product line development projects. Second, it may be argued that the chosen scenarios in the case study are non-representative for typical product line implementation problems, for example because they are too few, too simple, too orthogonal, or because they leave out numeric variabilities. While variabilities of the numeric type occur occasionally, they typically do not pose a serious coding problem, and especially, they do not rely on the presence of most discussed variability mechanisms, which is why they have not been covered in the case study. Another threat of validity is that the chosen reference system is biased towards certain mechanisms or mechanism combinations, and that the underlying heuristics for representing alternative and optional variabilities do not hold. While it is true that representation of variabilities in code is situation-dependent, the heuristics acknowledge this fact. On the other hand, if considering complexity in terms of necessity and arbitrariness, there is strong evidence that many optionals do not require as open parameters of variation as alternatives do, which is why open-parameter mechanisms have been used in the reference implementation for the former, while the latter have preferred closed-parameter mechanisms. A final threat to validity is if an appropriate set of goals and metrics has been used, and if the smaller measured differences among many mechanisms beyond Cloning could rather mean that any mechanism may be applied on equal terms. While there are indicators in single systems development that many metrics strongly correlate with code size, lines of code cannot be the only indicator for product line code quality, because not all variability management and evolution problems are solved by just maximally collapsing the code. There must be other factors for distinguishing more suitable variability mechanisms from less suitable ones, because otherwise, for example, it would not make a difference if Conditional Execution or Conditional Compilation is used, which often differ hardly in terms of code size. Here, the differences are more in code shape, or in the visibility of variable parts, which help or prevent the developer from keeping product line code reusable.

6 Summary and Outlook

This paper presented an overview of our approach for keeping product line code reusable in practice. Because product line code becomes more complex than single systems code due to its additional variability, we analyzed the impact of major variability mechanisms on evolvability. A product line quality model was developed according to the GQM approach, and the proposed metrics were applied in a case study. The results show that Cloning can be appropriate in the short term, while a mix of Conditional Compilation and Frame Technology yield the best long-term results.

Further work will extend the approach to single systems development, evaluating which semantic source code elements are essential and which cause arbitrary complexities in specific contexts, in order to deliberately omit the latter elements when there is no reason to use them. A second extension is to broaden approach from coding activities to other software development activities, for example architecting, so that all product line infrastructure artifacts are systematically simplified and become more evolvable. Third, the scalability of the approach will be evaluated by using large-scale code. Fourth, synergies will be evaluated with other product line implementation technologies, especially configuration management.

References

[AD07] S.A. Ajila, R.T. Dumitrrescu: Experimental Use of Code Delta, Code Churn, and Rate of Change to Understand Software Product Line Evolution. JSS 80(1), 74–91, January 2007

[Ale02] C. Alexander: The Nature of Order, Book 2. CES Publishing, 2002

[AG01 M. Anastasopoulos, C. Gacek: Implementing Product Line Variabilities. ACM Software Engineering Notes 26(3): 109-117, May 2001]

[Bas97] P. Bassett: Framing Software Reuse. Lessons From The Real World. Prentice Hall, 1997

[CE00] C. Czarnecki, U.W. Eisenecker: Generative Programming. Addison-Wesley, 2000

[Cop98] J.O. Coplien: Multi-Paradigm Design for C++. Addison-Wesley, 1998

[CZ03] J.O. Coplien, L. Zhao: Symmetry and Symmetry Breaking in Software Patterns. GCSE-2: 37-56 (LNCS2177), Springer-Verlag, 2000

[HM00] G.A. Hall, J.C. Munson: Software Evolution: Code Delta and Code Churn. JSS 54(2): 111-118, October 2000

[JGJ97] I. Jacobson, M. Griss, P. Jonsson: Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley, 1997

[Kel06] D. Kelly: A Study of Design Characteristics in Evolving Software Using Stability as a Criterion. IEEE Transactions on Software Engineering 32(5): 315-329, May 2006

[Kna04] P. Knauber: Managing the Evolution of Software Product Lines. ICSR-8, 2004

[Kru07] C.W. Krueger: The 3-Tiered Methodology: Pragmatic Insights from New Generation Software Product Lines. Proc. SPLC-11: 97-106, 2007

[Ley01] M. Leyton: A Generative Theory of Shape (LNCS 2145). Springer-Verlag, 2001

[LF06] M.M. Lehmann, J.C. Fernandez-Ramil: Rules and Tools for Software Evolution Planning and Management. In N.M. Madhavij, J. Fernandez-Ramil, E.E. Perry (Eds.): Software Evolution and Feedback - Theory and Practice. Wiley & Sons, 2006: 539-563

[LT08] R.E. Lopez-Herrejon, S. Trujillo: How complex is my Product Line? The case for Variation Point Metrics. VAMOS08 Workshop, 2008

[Mut02] D. Muthig: A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines. PhD Thesis, Fraunhofer-Verlag, 2002.

[MP03] D. Muthig, T. Patzke: Generic Implementation of Product Line Components. NetObjectDays 2002: 313-329

[Par94] D.L. Parnas: Software Aging. ICSE-16: 279-287, 1994

[Pat08] T. Patzke: Mechanisms, Processes and Metrics for Implementing and Evolving Reusable Embedded Systems. Fraunhofer IESE Report 021.08/E, 2008

[PM03] T. Patzke, D. Muthig: Product Line Implementation with Frame Technology: A Case Study. Fraunhofer IESE Report 018.03/E, 2003

[S++02] R. van Solingen, V.R. Basili, G. Caldiera, H.D. Rombach: Goal GQM Approach. In J.J. Marciniak (Ed.): Encycl. Software Engineering (2nd Ed.), Wiley & Sons, 2002: 578-583

[Sim62] H.A. Simon: The Architecture of Complexity. In H.A. Simon (Ed.): The Sciences of the Artificial (3rd ed.). MIT Press, 1996

[Som04] I. Sommerville: Software Engineering (7th Ed.). Pearson Education, 2004

[SVB05] M. Svahnberg, J: van Gurp, J. Bosch: A Taxonomy of Variability Realization Techniques. Software - Practice and Experience 35: 705-754, Wiley & Sons, 2005

[TeCo] Particle Computer homepage: particle.teco.edu (Retrieved October 2009)