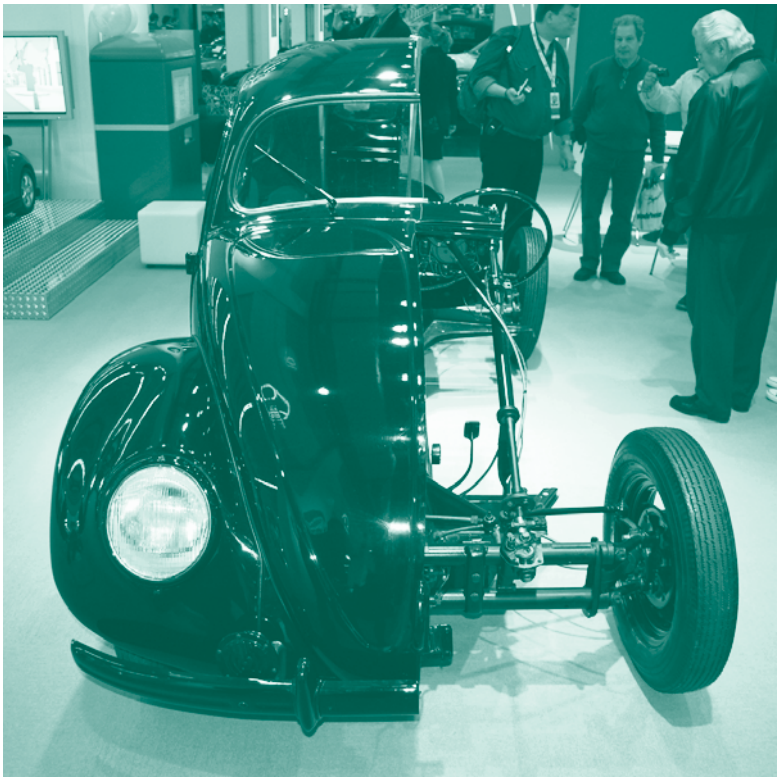
PhD Theses in Experimental Software Engineering

Vol. 36

Thomas Patzke

Sustainable Evolution of Product Line Infrastructure Code



FRAUNHOFER VERLAG

PhD Theses in Experimental Software Engineering

Volume 36

Zugl.: Kaiserslautern, Univ., Diss., 2011

Printing:
Mediendienstleistungen des
Fraunhofer-Informationszentrum Raum und Bau IRB, Stuttgart

Printed on acid-free and chlorine-free bleached paper.

ISBN 978-3-8396-0315-4
Fraunhofer-Informationszentrum Raum und Bau IRB
Postfach 800469, 70504 Stuttgart
Nobelstraße 12, 70569 Stuttgart
Telefon +49 711 970-2500
Telefax +49 711 970-2508
E-Mail verlag@fraunhofer.de
URL http://verlag.fraunhofer.de

Sustainable Evolution of Product Line Infrastructure Code

Beim Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation
von

Dipl.-Ing. Thomas Burkhard Patzke

Fraunhofer-Institut für Experimentelles Software Engineering
(Fraunhofer IESE)
Kaiserslautern

Berichterstatter:	Prof. Dr. Dr. h.c. Dieter Rombach Prof. Dr. Arnd Poetzsch-Heffter Prof. Dr. Jürgen Nehmer
Dekan:	Prof. Dr. Arnd Poetzsch-Heffter
Tag der Wissenschaftlichen Aussprache:	24.06.2011

D 386

PhD Theses in Experimental Software Engineering

Volume 36

Zugl.: Kaiserslautern, Univ., Diss., 2011

Printing:
Mediendienstleistungen des
Fraunhofer-Informationszentrum Raum und Bau IRB, Stuttgart

Printed on acid-free and chlorine-free bleached paper.

ISBN 978-3-8396-0315-4
Fraunhofer-Informationszentrum Raum und Bau IRB
Postfach 800469, 70504 Stuttgart
Nobelstraße 12, 70569 Stuttgart
Telefon +49 711 970-2500
Telefax +49 711 970-2508
E-Mail verlag@fraunhofer.de
URL http://verlag.fraunhofer.de

Summary

A major goal in many software development organizations today is to reduce development effort and cost, while improving their products' quality and diversity by developing reusable software. An organization takes advantage of its products' similarities, exploits what they have in common and manages what varies among them by building a product line infrastructure [Bayer++99, Muthig02]. A product line infrastructure is a reuse repository that contains exactly those common and variable artifacts, such as requirements documents, architecture, or source code, which are needed to produce all required products.

The life of successful software does not end after initial development. Every real-world software system must continually evolve in order to remain useful for its end-users [Lehman80]. Likewise, every real-world product line must continually evolve in order to remain satisfactory for its reusers. However, a problem we have often seen in various product line engineering projects in practice is that the product line infrastructure, and especially its code, the product line infrastructure code, becomes increasingly difficult to evolve and reuse over time because it degenerates.

Code decay has two causes [Parnas94]: lack of change and inappropriate change. The focus of this thesis is to prevent inappropriate change. In single-systems engineering, changes are made to improve functionality, efficiency, or ease-of-change. Trade-offs must be made among these goals. Those types of changes are also necessary in product line engineering, but they are not sufficient because most work products [Jalote05] of product line engineering are developed to be reused. Additional types of changes in product line engineering are necessary to improve variability, reuse efficiency, or ease-of-configuration. Trade-offs must not just be made among these goals, but also between product line goals and single-system goals. These issues make product line development more complex than single-systems development. Within the product line engineering life cycle, these issues arise in a process in which artifacts are developed for reuse: family engineering [Muthig02]. The main contribution of this thesis is the development of a readily applicable, reactive product line evolution method for family engineers in practice. The method's goal is to guide family engineers to keep product line infrastructure code sustainable by reducing unnecessary complexities in variability management [Bosch++02].

Benefits of the method are controlled complexity reduction of existing product line infrastructure code, whole life cycle cost and effort

reduction, protection of investment, short-term increase of variability management productivity, and customizability to specific organizational contexts.

Product line infrastructure code evolution is still an unexplored topic in product line research. This thesis investigates what makes code overly complex that is developed in family engineering. It explores how this code can be evolved as required under real-world constraints in family engineering, with 'just enough' product line-specific complexity. Product line infrastructures contain artifacts that capture the products' required variability. This is the key characteristic that differentiates these artifacts from less efficiently reusable or single system artifacts. Family engineers have various possibilities to realize variability in artifacts, and in particular in code. If they apply them without consideration, as we and others have often seen in practice [Krueger07], product line infrastructure code becomes unnecessarily complex. As a remedy, I devise a set of tactics for effective family engineering in this thesis. I present a pattern language [Gamma++95] of plain and practically relevant types of variability mechanisms that cover all relevant combinations of these tactics.

The variability mechanisms address the solution domain in which variation is realized. They are one input to a product line realization process I develop in this thesis. I identify product line evolution scenarios as another input, concerned with the problem domain. They characterize different basic types of changes in future product line requirements that cause a product line infrastructure to evolve. The realization process itself consists of three sub-processes for which I have identified variability-related and non-variability-related sub-activities that are ordered in a particular way as to optimize productivity. While I explain the first two sub-processes, Selection and Modification, I identify a set of product line-specific code defects, and I invent a larger set of refactorings for removing these and other product line-specific defects, also beyond code. As part of the third sub-process, Quality Assurance, this thesis contributes to the unexplored discipline of product line measurement by developing a goal-oriented measurement scheme which characterizes complexity in the code of evolving product line infrastructures.

Using this measurement approach, I evaluate the impact of all discussed variability mechanisms on complexity under typical product line evolution scenarios in a case study, with three main results: First, contrary to popular belief in product line engineering, code duplication does not always over-complicate variability management. Second, the two factors Late Binding and Programming Language-dependence significantly increase the complexity of product line infrastructure code. Third, this type of complexity is decreased if a variability mechanism supports Defaults and both open and closed variation.

Table of Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Many software development organizations today aim at reducing their development (Def.12) effort and cost, while improving the quality and diversity of their software products by engineering (Def.41) reusable software. A set of reusable artifacts (Def.44), for example a product line infrastructure (Def.62), and ultimately its source code, is created, but it decays over time unless it is evolved (Def.66) in a sustainable way. Thus, the engineering goal is to keep the product line infrastructure reusable over long periods of time. To support this goal with a focus on source code, a method is developed in this thesis which guides product line engineers in practice in the sustainable evolution of source code in a product line infrastructure, or product line infrastructure code.

The remainder of this introduction is organized as follows: Section 1.1 introduces the problem of product line evolution and sketches a solution. Section 1.2 presents a historical perspective of product lines and especially how the state-of-the-art in coding for reuse has emerged. Section 1.3 discusses the solution idea of minimizing complexity excess through a well-defined product line evolution method in more detail. Section 1.4 lists the benefits of the approach. Section 1.5 summarizes the introduction and gives an outline of the remaining chapters.

1.1 Evolution as a Product Line Engineering Challenge

A goal in most software organizations is to develop high-quality software in a timely and cost-effective manner, for example by large-scale software reuse. Reusable software reduces development effort if new applications can be constructed by using pre-existing elements again, rather than always developing them anew. Software which has successfully been reused in several products is also likely to contain fewer defects than newly-developed software. This improves the quality of each new product that reuses these elements.

An organization typically develops a set of similar software systems for the same market segment, so that it makes sense to aim at the mass production (Def.30) of software, tailored at individual customer needs (Def.24), instead of always developing new similar systems from scratch. Tailoring software to individual needs is only economically useful if the systems are similar enough, contain enough commonality (Def.45), but also provide sufficient means for required diversification.

In the past, reusable software has often been built independent, and much earlier than software which reuses it. Reusable software meant fixed blocks of small-scale artifacts, for example programming language libraries or "third party" libraries. A problem with these approaches is that they usually provide unbounded commonality, too much than required for each specific product, and at the same time, they offer insufficient means for product-specific variation (Def.47). Both problems have arisen because development for reuse and development with reuse [Karlsson95] have not been aligned with each other.

More recently, more and more reusable software of larger scale is developed in a deliberate organization-specific engineering effort alongside the software that reuses it. In these approaches, a set of similar software systems is developed (conceived, designed, constructed, and evolved) [Shaw05] as a product line [Withey96] (Def.23) by capitalizing on the products' required commonality and predicted variation [Weiss+99]. All artifacts created for reuse during the engineering sub-phases, such as requirements, architecture, or code, but also methods and tools, constitute an organization's product line infrastructure [Bayer++99, Muthig02].

After building up a product line infrastructure, an organization is able to rapidly instantiate the required individual products by consuming elements of the product line infrastructure. However, successful products in practice must be changed over time in order to remain satisfactory for end-users [Lehman80], and likewise, the product line infrastructure must accommodate changes in order to remain satisfactory in reuse. For example, the product line may need to accommodate new products that had initially not been planned, or existing product line characteristics may need to be changed in ways that were not entirely foreseen.

Figure 1 shows how evolution problems propagate through the product line engineering (Def.49) sub-processes, and which interaction is the focus of this thesis. Product requirements for similar products are the input to the product line engineering (PLE) process. The output is a set of products. Within product line engineering, there are two sub-processes that execute in parallel, interfaced by a reuse repository called the product line infrastructure. As mentioned above, the product line infrastructure contains different types of reusable artifacts. For the sake of brevity, Fig.1 only highlights code artifacts. The family engineering (Def.60) process is responsible for producing the product line infrastructure, and the application engineering (Def.61) process consumes the product line infrastructure during the production of products.

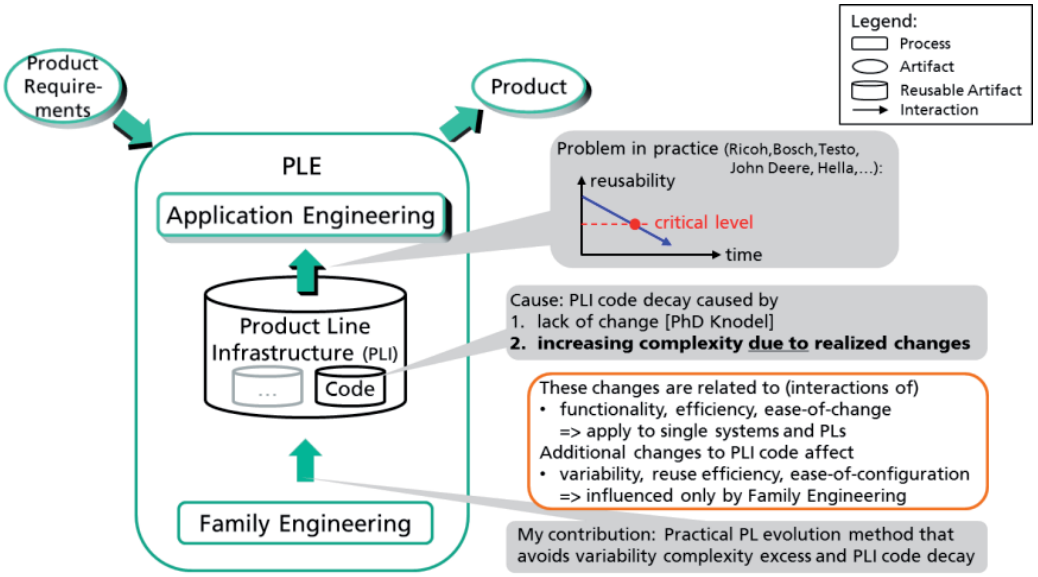


Figure 1: The problem of product line infrastructure code evolution

A problem in practice we have observed in numerous industry projects is that over time, it becomes progressively harder for application engineers to reuse artifacts from the product line infrastructure, especially code. Below a critical level of reusability the engineers prefer to rewrite code from scratch, rather than to reuse it. For example, in a particular project, application engineers struggled with adding new functionality to software for digital entertainment systems because the product line infrastructure code did not support these changes. In another project, the effort to reconfigure existing automation system code became excessive because the product line infrastructure provided too many combinations of configuration options. In a product line engineering context, this problem is caused by the product line infrastructure, and in particular by its code artifacts, the product line infrastructure code.

For conventional single system software, this type of phenomenon has been the topic of software evolution research since the late 60s [Lehman02] and is known as software aging [Parnas94], software decay [Mens+08], or code decay. As shown in Fig.1, two reasons for code decay have been found [Parnas94]: The first reason is lack of change, which means that there are new product requirements, but these are not realized[1] in the code, so that the code and the requirements documents drift apart. This issue can be resolved by keeping the code in sync with other product line infrastructure artifacts. While others have treated this issue of creeping architectural mismatch in single systems extensively [Garlan++95, Knodel10], it is not in the focus of this thesis.

[1] In accordance with [Krueger92, Pohl++05], the term realization is used throughout this thesis for the activity also known as implementation or coding.

The other reason is that the internal complexity (Def.43) of the code increases [Lehman80], not due to omitted changes, but due to committed changes which have been realized inappropriately. This makes the product line infrastructure code harder to understand, evolve and reuse than necessary. Fig.1 illustrates that product line infrastructure code is developed and evolved in the family engineering process, and this is where complexity arises and where it must be tamed. For that reason, this thesis concentrates on the interface between family engineering and product line infrastructures, investigating which factors contribute to complexity excess in evolving product line infrastructure code. The focus is not only on passively measuring complexity attributes because analyzing alone does not make the code less complex. Instead, the focus is on the entire process for actively counteracting the degeneration process. The result is a practical guide, aimed at family engineers, to evolve product line infrastructure code, balancing effort and complexity in such a way that code decay is reduced or avoided.

What is the key difference between single system artifacts and artifacts in a product line infrastructure, which makes its code more complex than single system code? As will be shown in Section 2.3, it is variability (Def.46). Product lines capture variability [Synthesis93, O'Connor++94], a concept that is not significant in single systems. Variability is realized in artifacts of the product line infrastructure, such as code, by variability mechanisms [Jacobson++97] (Def.64). In this thesis, a set of orthogonal family engineering tactics is developed which can be used to rank variability mechanisms. The tactics concept is inspired by architectural tactics [Bass++03], and the set of tactics extends a list of guidelines for effective reuse, proposed in the reuse literature [Bassett97]. Based on these tactics, a pattern language of plain and orthogonal types of variability mechanisms is presented from a family engineer's viewpoint. In order to describe why variability arises, this thesis identifies evolution scenarios which are types of changes in product line requirements that may lead to over-complexities in future product line infrastructure code. Both the variability mechanism patterns and the evolution scenarios are inputs to the product line realization process developed in this thesis whose goal is to systematically guide family engineers in practice in the efficient and long-term evolution of product line infrastructure code, as introduced in Section 1.3.

1.2 A Brief History of Product Line Realization

Today's notion of software product lines has gradually evolved during more than half a century of software development. This section presents some larger milestones which illustrate how the current understanding of software product line realization and variability mechanisms (Chapter 4) emerged. Chapter 2 will define the resulting concepts in more detail, as used in the remainder of the thesis.

Software reuse is concerned with using software development artifacts again. Reuse activities are performed because a reuser who solves a new development problem is faced with a problem that has already been solved by an existing artifact. In particular, the activities for creating the reused artifact do not have to be repeated again. Typically, this saves development effort and provides quality improvements.

Software development activities have always included realization activities, and this is why source code reuse has always played a role in software reuse. As Figure 2 shows, one of the earliest reuse concepts are subroutines, collected in subroutine libraries, Their first introduction can be traced back to at least 1951 [Wilkes++51]. Subroutines are useful in a reuse context because they allow functionality-related solutions to be reapplied if the identical development problem arises again.

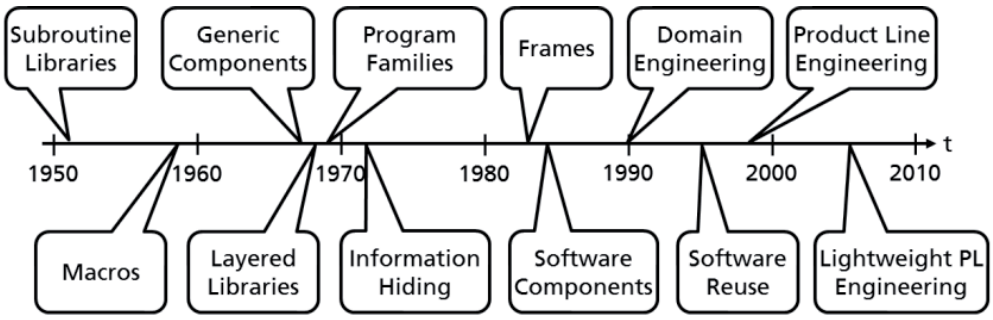


Figure 2: History of product line engineering

By the end of the 50s, macros had been introduced [Greenwald+59] as another code reuse concept which is still in use today. Like subroutines, macros relieve a reuser from redeveloping code artifacts. Unlike subroutines, the reused artifacts are typically not restricted to algorithms only, but may consist of arbitrary text elements.

Two novel reuse ideas were addressed at the 1968 NATO Software Engineering Conference [Naur+69]: organization of libraries and software components. On the one hand, an extension of the subroutine library idea by the layering concept was proposed which enables interdependent libraries to be organized in hierarchies according to function call relations [Dijkstra68]. These ideas were later generalized to hierarchies organized according to other types of relations [Parnas74]. In a reuse context, this idea is valuable because hierarchies make it possible to organize elements according to various criteria, e.g. according to reusability or change frequency.

On the other hand, the idea of generic function libraries as software components was introduced [McIlroy68]. This paper first addressed that reusable code must solve a family of similar problems. In other words, this paper generalized the reuse concept which had formerly only been concerned with sameness to one which is concerned with similarity. It

suggested that truly reusable components must be adaptable in order to be used effectively. This means that they do not only require common, but also variant elements, either as predefined options or as open-ended extension possibilities. In a product line context, one major contribution of that paper is that it highlighted the importance of variability for reuse.

In the following year, the concept of program families was first presented [Dijkstra69] which was later refined [Parnas76]. Dijkstra motivated the need for entire software systems to be viewed as similar artifacts because they evolve over time, so that they form generations of a single product. Different versions can then be seen as having a common ancestor which is only partially complete because some development decisions are still left open. This way, the concept of generic function libraries was extended to entire software systems. Parnas generalized this idea by not only considering the shape of software code families as end-products, but by suggesting a process for constructing them. This construction process happens in a well-defined order in which decisions concerning the more common elements are made earlier than decisions for more variant elements.

In 1972, information hiding had been proposed as a systematic approach for module decomposition in a reuse context [Parnas72]. Using this concept, the developer of a reusable artifact suppresses certain realization details from a developer who reuses the artifact. This simplifies reuse because information about a reused artifact is reduced in such a way that the more common details about the artifact that only its builder needs are suppressed, while those elements which are important for the reuser are highlighted.

In the early 80s, problems in reusable subroutine libraries were identified that arise when common, fixed software elements must evolve [Bassett84]. A solution called frame technology was suggested in which reusable modules of source code text are customized exactly as required in each reuse situation. In a later publication [Bassett97], it was stressed again that not only subroutines qualify as reusable source code artifacts, but that any kind of source code text is reusable, independent of its meaning in a programming language context.

The 90s started with an extensive survey of state-of-the-art and state-of-the-practice reuse approaches [Krueger92]. These comprised, among others, high-level languages, copy-and-paste programming, software components and code generators. They were analyzed according to different criteria, especially abstraction (information hiding, as used in [Parnas72]) and specialization (genericity, envisioned in [McIlroy68]). By the same time, domain engineering was proposed as an approach for more productive development of similar software systems [Campbell++90], and the feature concept was suggested to document such systems [Kang++90].

The late 90s brought a consolidation of the idea of software reuse as ad hoc development of small- to medium-sized code artifacts. Object-oriented and component-based development focused on constructing small-scale building blocks in the late software engineering life cycles [Meyer97, Szyperski98]. Developing software from Lego block components, analogous to engineering physical objects, was also criticized as an insufficient strategy for efficient software reuse in practice, preventing unpredicted changes [Bassett97] and neglecting architectural issues [Ran99]. The concepts of patterns and pattern languages, initially conceived in the context of building architecture [Alexander++77], were adopted in various phases of the software engineering life cycle [Gabriel96, pp.33], for example in realization [Coplien91, Beck96], design [Gamma++95, Buschmann++96], or architecture [Shaw+97], to systematically describe recurring solutions to common problems that arise in software development contexts in practice. It was suggested to analyze and represent similar systems by considering their commonalities and differences [Synthesis93, O'Connor++94], and the proposal was made to take advantage of these concepts in domain engineering [Neighbors80, Withey96]. The variation point concept was suggested for representing differences in reusable software artifacts [Jacobson++97], which rediscovered and refined the concepts of engineering change point [Bassett87] and hot-spot [Pree94]. Multi-paradigm design was suggested as a development approach for common and variable code [Coplien99]. Generative programming proposed new realization solutions [Czarnecki+00], for example as provided by mixins [Smaragdakis+02] or Aspect-Orientation [Kiczales++97]. The traditional concept of reuse, informally taking place in an unplanned fashion, was renamed opportunistic reuse, as opposed to planned reuse activities, called systematic reuse [IEEE1517]. As envisioned in the mid-90s [Prieto94], the traditional notion of software reuse disappeared in academia by the turn of the millennium.

Since the end of the 90s, the software family idea [Dijkstra69, Parnas76] has been refined in several respects, leading to the concepts of software product lines and product line engineering [Weiss+99, Bayer++99, Bosch00, Atkinson++01, Clements+01, Gomaa04, Pohl++05, Käkölä+06, Linden++07, Northrop+07, Kang++10]. As conceived in the 90s, reusable artifacts are now considered across the entire engineering life cycle, beyond source code. Unlike in the domain engineering approaches of the early 90s, reusable artifacts are now engineered based on precisely defined system boundaries, due to additional scoping activities in product line engineering. This leads to complexity reduction by strategically avoiding development effort for artifacts that will not be reused. Whereas traditional families were seen as single system artifacts changing over time (now termed product populations [Ommering04]), product lines comprise multiple similar artifacts existing simultaneously, for example a standard and an extended application. Most recently [Elsner++10], as in this thesis, both of these aspects, variation in time

and variation in space, are considered in product lines. This evolution aspect of product lines is also becoming increasingly important as product line engineering enters industrial practice, necessitating lightweight processes that are customizable to individual development contexts [Bayer++99, Krueger02a, Krueger02b, Muthig02, Kolb++06, Krueger07, Hanssen+08, Bosch09, Codenie++10, Kolb+10, Krueger10, McGregor++10, Mohan++10].

1.3 Solution Idea: Complexity-Aware Family Realization

As discussed in Section 1.1, the approach developed in this thesis is keeping the product line infrastructure, and especially its code artifacts, reusable (cf. Figure 1). At least two orthogonal dimensions of product line realization exist [Muthig++02]: configuration management and generative techniques within source code. While the former is addressed by others [Anastasopoulos++09], the present thesis is concerned with the latter. Figure 3 highlights again that this work focuses on the family engineering activity, not on application engineering, within the classical product line engineering life cycle [Bayer++99, Weiss+99, Clements+01, Pohl++05]. The goal is to support the product line engineer, especially the family engineer, in the long-term evolution of code contained in the product line infrastructure.

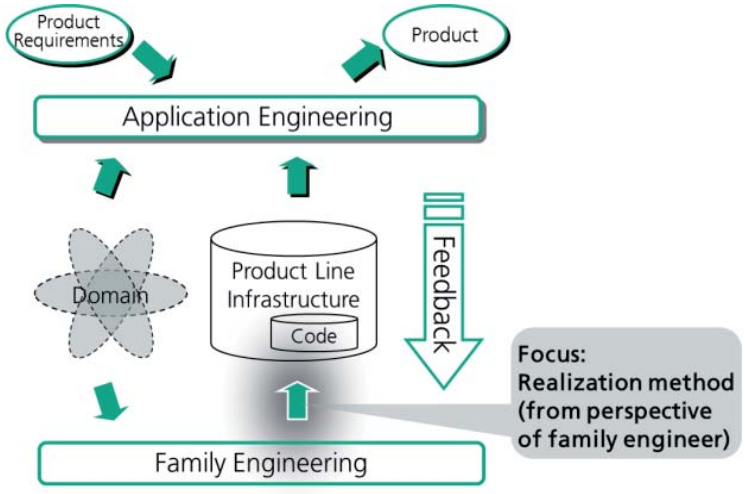


Figure 3: Focus of the product line evolution method

As part of the product line infrastructure, the evolving code consists of common and variant code elements (Def.55), configured (Def.28) by configuration artifacts, for example configuration scripts or Makefiles. An interdependency of code and configuration artifacts exists, but both may be optimized independently for evolution. While consistent configuration is also a problem in practice [Krueger07], this thesis concentrates on the main realization artifacts which are common and variant source code elements.

As will be shown in Section 3.4, a problem in practice is that evolving systems in general and software artifacts in particular become complex. This phenomenon has been observed both in academia and in practice. Sometimes, concrete suggestions have been made on how to tame this complexity, for example by software refactoring [Fowler99]. However, extensive research does not yet exist on how software product line realizations may be evolved in such a well-behaved way that they remain evolvable and reusable. As indicated in Section 1.1 and explained in Section 2.3, product line development adds a dimension of complexity compared to single system development, the reuse dimension with its co-evolving common and variant elements. This raises novel research challenges which are addressed by this thesis, for example, what makes product line infrastructure code complex and how can it be evolved in a well-behaved manner in practice, with "just enough" effort.

As will be shown in Section 2.1, reuse (Def.21) extends use (Def.6), and use is a single-system concept. For that reason, code which is just used during product line reuse does not pose new evolution challenges compared to single-systems practices. The new challenges of product line infrastructure code evolution have to do with those properties that become relevant in reuse, as opposed to unmodified use. These challenges are variability, reuse efficiency, and ease-of-configuration. In particular, variability mechanisms are responsible for a type of complexity in product line infrastructure code which does not exist in single system code [Bosch++02].

This type of complexity is unavoidable, essential. However, a large proportion of software complexity is of another type, called arbitrary (or accidental) complexity [Brooks95] (see Def.43 in Sec.2.3). Arbitrary complexity exists in all artifacts and processes that are not essential to solve the current software development task, but which unnecessarily make the artifact more difficult to evolve. For example, arbitrary complexity may be introduced during realization if clear software requirements or consistent software architectures are missing. More generally, complexity is propagated through the software engineering life cycles, accumulating in the software realization phase.

In product line engineering, unnecessary complexity arises due to the inclusion of unneeded commonality (which can be avoided by proper scoping activities), and due to inadequate management of variation. During all phases of family engineering, and especially during the realization of product line infrastructures, the provided mechanisms which allow variation to be included, contribute to product line-specific over-complexity. Such complexities in the artifacts arise during the process in which the artifacts are evolved, and this is why they are reduced or avoided by complexity-aware engineering processes. Such a process is developed in this thesis for family engineering, and in particular for family realization (Fig.3). Figure 4 gives an overview of the

proposed approach whose details will be developed in Chapter 5. Existing code of a product line infrastructure and new product line requirements are the inputs of a product line realization process that targets the product line engineer, in particular the family engineer. The output of the process is new product line infrastructure code which has undergone variation in time and whose complexity is well-managed, so that the code is kept evolvable.

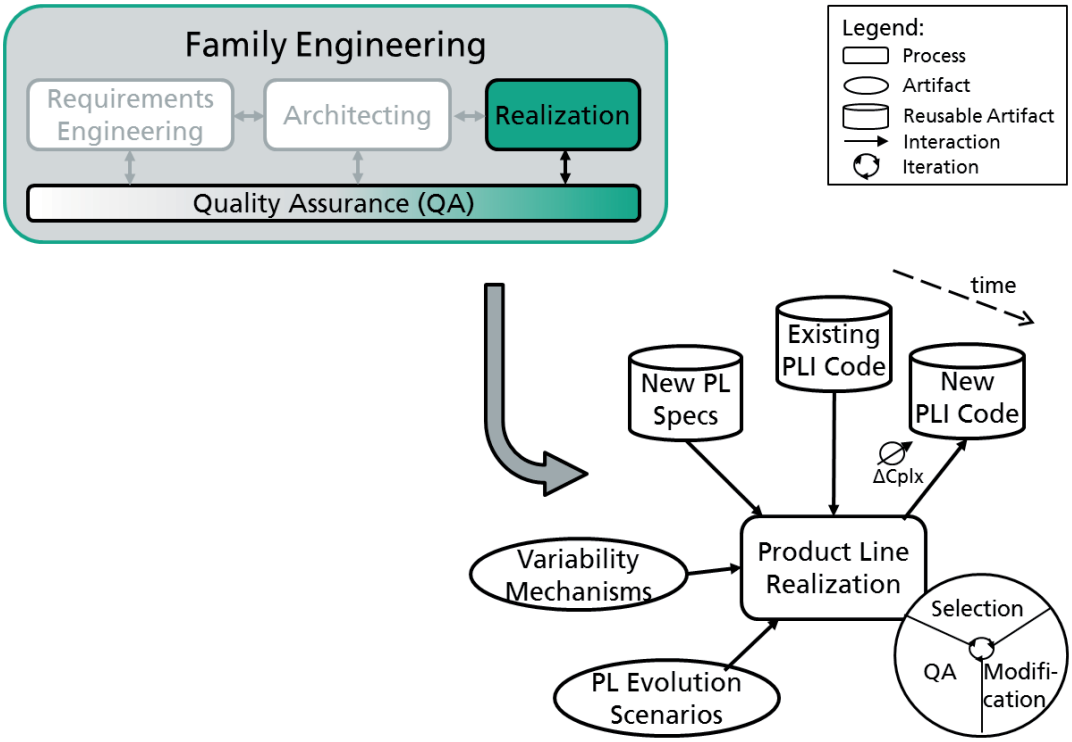


Figure 4: Product line evolution method

Product line infrastructure code complexity is determined and can be controlled by variability mechanisms, and this is why another process input is a collection of variability mechanisms, presented in form of a pattern language, a format well-known in practice. The set of presented mechanisms has deliberately been limited to least complex ones which we have seen in projects in practice, and which possess disjoint reuse characteristics, according to an extended set of criteria initially proposed in the reuse literature [Bassett97]. The goal of setting up this pattern language is to establish a conceptual toolset of product line realization mechanisms that the family engineer may use and customize.

Another process input that augments the family engineer's mental tool set are product line evolution scenarios. They serve to describe the possible next types of variability-related changes that result from predicted or unpredicted changes in product line requirements. One such requirement is, for example, to support a new product which requires a new alternative variation, in addition to an existing set of alternatives. The presented product line evolution scenarios are

elementary ways in which variabilities may evolve, based on a small set of disjoint elementary scenarios.

The product line realization process consists of three iterative and incremental processes. These processes and their sub-activities are organized to optimize evolution efficiency, producing the most important results as early as possible, while mistakes may be undone with least effort. The goal of the first of these processes, selection, is for the family engineer to understand the new requirements and to identify variation candidates in the code, while possibly detecting variability defects. In the second process, modification, the family engineer performs the changes in a specific order, possibly removing identified defects by variability refactorings. The final process, quality assurance, serves to provide feedback that product line characteristics have not suffered. It consists of product line testing and product line measurement sub-processes. As part of product line testing, a novel testing approach is proposed, based on Bassett's idea of separating reuse from use [Bassett97]. The testing approach ensures that all product line members (Def.48) can be constructed and executed as required. For the following sub-process of product line measurement, a nearly unexplored discipline in product line engineering, an extensible variability complexity measurement scheme is developed, based on the GQM approach [Solingen++02]. The measurement scheme serves to ensure that the resulting product line infrastructure code has remained simple enough for sustainable evolution.

Using that measurement approach, a case study is performed that evaluates the impact of all presented variability mechanisms on evolvability, taking into account major classes of product line evolution scenarios. The case study supports the hypothesis that there is no "silver bullet" [Brooks95] of a single variability mechanism for keeping product line infrastructure code reusable, but instead the key factor for sustainable product line infrastructure code evolution lies in an appropriate process for applying mechanisms, according to the development context (Def.20) [Patzke10a].

1.4 Contributions and Benefits

This chapter introduced the novel problem of code aging in product line infrastructures. It was shown that product line concepts have evolved during more than half a century of software development. It was discussed that there are particular complexities when product line infrastructure code is evolved which may be mitigated by guiding the family engineers in selecting appropriate variability mechanisms.

The primary contribution of this thesis is a reactive product line evolution method which supports family engineers in practice to keep product line

infrastructure code evolvable throughout its life. Besides a presentation of fundamental product line concepts (Ch.2), the following method elements make the following contributions to product line engineering: A customizable product line realization process is identified (Sec.5.2) which consists of novel variability refactoring and quality assurance activities. The process is deliberately subdivided into variability-related and non-variability-related sub-activities which can be reused and whose sequence has an impact on development productivity. As part of quality assurance, a GQM-based variability complexity model with concrete product line metrics is developed (Sec.5.3), as well as a novel product line testing approach. Two types of artifacts are identified that support the process (and which can also be described by reusable sequences): variability mechanisms (Ch.4) and product line evolution scenarios (Sec.5.1). Based on a set of product line realization tactics (Sec.2.3), a pattern language of plain variability mechanism types is presented whose goal is to guide family engineers in practice. The goal of the identified product line evolution scenarios is to characterize future possibilities of product line requirements changes. A case study (Sec.6) validates important method elements under quasi real-world conditions in embedded systems development. The results show that code duplication can be beneficial in the short term, while it is most detrimental in long-term evolution. It is also found that late binding and programming language-dependence significantly increase the complexity of product line infrastructure code, while Defaults and support for both open and closed variation decrease it.

Several types of benefits can be expected by applying the product line evolution method developed in the current thesis.

First, the method results in controlled complexity reduction of existing product line infrastructure code, which not only leads to at least 30% complexity reduction after the third iteration, compared to a non-controlled approach, but also reduces overall complexity growth which tends to become exponential in ad hoc approaches, to an acceptable rate, as shown in Fig.5a.

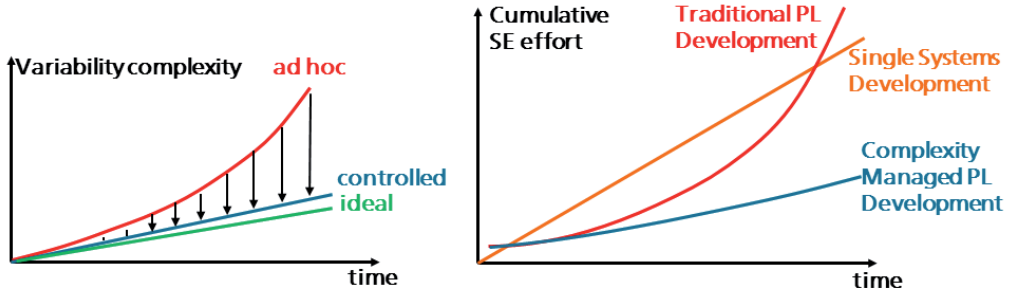


Figure 5: a) Complexity reduction in product line infrastructure code; b) whole life cycle effort reduction

At the same time, the method provides whole life cycle cost and effort reduction in product line engineering in practice because it actively counteracts product line infrastructure degeneration and protects existing investment by avoiding the premature retirement of product line generations [Ganesan++06]. This is illustrated in Fig.5b.

Another benefit of the method is that it leads to an increase of variability management productivity by supporting the practically important factor of development speed [Kolb+10] without compromising other quality attributes. The method is also customizable to the respective development context of an organization, and it is future-proof because it builds on 2nd generation product line methods.

1.5 Outline

The remainder of the thesis is structured as follows:

Chapter 2 explains the background of this thesis, presenting a novel consistent taxonomy of 68 product line concepts. It is explained that besides the classical concept of reuse as unmodified usage, a more general dual concept of reuse with modification exists which is important for efficiently realizing variability in product lines. As a result of this duality, the concept of a reuse hierarchy is developed which complements the classical architectural style of layering that is often based on unmodified usage relations. Product line concepts are explained with a focus on variability, artifacts and processes. When discussing the variability mechanism concept, a classification scheme of five criteria is developed for guiding variability mechanism selection. The interrelation of complexity and evolution is shown in a product line engineering context.

Chapter 3 presents related work on product line realization artifacts, product line engineering processes, empirical studies questioning the harmfulness of code duplication, complexity and evolution in single systems, and complexity and evolution in product lines. It is shown that most work that concentrates on product line realization artifacts has neglected process issues which are important for engineering product lines in practice. On the other hand, research on product line engineering processes has often focused on introducing product lines in a proactive way, neglecting incremental transition and evolution strategies in the presence of existing artifacts, especially code. It is also shown that, according to a growing number of empirical studies from single systems software engineering in the last decade, code duplication cannot generally by considered harmful in any software development context. Work on complexity and evolution in single software systems and single systems in general has resulted in guidelines for passively characterizing and actively evolving them, aiming at long-term quality

improvement while possibly tolerating short-term degradation. It is also shown that product line evolution has been recognized as variability in time, but that synergies to classical complexity and evolution research are missing.

Using the classification scheme from Chapter 2, Chapter 4 presents a pattern language of seven basic variability mechanism types for source code, primarily targeted at family engineers in practice using the C/C++ programming language, for example in embedded systems development. As a novel contribution to product line engineering, cloning, which has traditionally been considered harmful in all reuse situations, is also included as a variability mechanism. This is also backed-up by a growing number of recent studies in single systems engineering that have started to reject the universal harmfulness of cloning. Each mechanism is discussed using a slightly modified Design Pattern [Gamma++95] template well-known in practice which includes construction dynamics, variants, advantages, disadvantages and relations to other mechanisms. As an extension to current view-based architecture or design descriptions [Kruchten95] also found in traditional design patterns, the structures of each discussed mechanisms is described in terms of processes creating these structures. This is a novel approach for describing the evolution trace of software artifacts, documented as snapshots of the development process.

The same representation is also applied in Chapter 5 in order to document the evolution trace of requirements artifacts. The presented product evolution scenarios are a novel contribution to product line engineering, as they allow a product line engineer to predict future product line requirements that result in an increase of unavoidable product line-specific complexity. A concise number of basic evolution steps are identified that can be combined to yield more complex evolution scenarios.

The main part of Chapter 5 presents the overall process for keeping product line infrastructure code reusable by consistently using the variability mechanisms introduced in Chapter 4. The process has been developed by transferring a process for sustainable evolution of complex systems to product line infrastructure code development. The process sub-activities are ordered in such a way that backtracking is minimized which optimizes the efficiency of the overall method. When presenting the first two method phases, a contribution is made to the novel discipline of variability refactoring by identifying a set of 23 product line infrastructure code smells and 37 variability refactorings.

As part of the quality assurance phase of the product line realization process, the thesis contributes to the unexplored discipline of product line measurement by developing a customizable goal-oriented scheme for variability complexity measurement. This thesis also uses a novel

complexity concept in which complexity is only seen as the absence of simplicity (see Def.43). This complexity concept is further refined to the product line-specific concept of variability complexity (Def.65). As a result, novel metrics for measuring reuse, similarity, variability and evolution are developed and applied. As another contribution to product line quality assurance, the thesis develops a two-staged product line testing approach, based on the two dual concepts of unmodified usage vs. reuse with modification, introduced in Section 2.1. The first testing phase, construction testing, is novel because, in contrast to all other product line testing approaches [Pohl++05, Neto++11], it tests if all members of the product line can be constructed as expected, whereas previous testing approaches have only considered if all product line members execute as expected. The testing approach has been applied in the case study in Chapter 6.

The case study is performed on small resource-constrained embedded systems product lines developed in the C programming language. The goal of the case study is to analyze, by applying all variability mechanisms from Chapter 4 and the measurement scheme from Chapter 5, if the selection of variability mechanism types has a major impact on product line infrastructure code evolvability. A set of factors is validated that contributes to variability complexity. In the case study, a combination of Conditional Compilation and Frame Technology led to least variability complexity in the long term.

Chapter 7 gives a summary, identifies open issues and gives an outlook at future challenges. The following section lists the references.

Appendix A contains a glossary of the discussed product line concepts. Appendices B to E contain material used or produced in the case study.

2 Background

Section 1.2 has shown that the current understanding of product line concepts emerged gradually during the history of software development. Many concepts were defined and refined, partially in consistent ways, but often inconsistently. This chapter presents a consistent terminology, based on a survey of over 300 documents from the software engineering literature that provided over 500 definitions of product line engineering terms [Patzke10b]. Note that the terminology is not as important as the underlying concepts, as observed in [Northrop+07].

Based on the reuse model from [Bassett97], Section 2.1 illustrates why reuse is more than unmodified use, as traditionally understood in framework development, object-oriented design patterns and component-based development. The goal of the section is to show why component-based techniques lead to unnecessary restrictions in family engineering. Based on this understanding of reuse as a dual concept to use, Section 2.2 shows why it makes sense to decompose reusable artifacts according to their degree of reusability. The goal of the section is to present criteria for family engineers to organize hierarchies of reusable modules. The two sections focus on underlying reuse concepts and postpone a more detailed discussion of product line engineering concepts to Section 2.3. It explains why variability is the key issue that differentiates product line engineering from conventional single systems engineering. Variability-related concepts are defined, and an overview of processes and artifacts in product line engineering is given. In particular, the variability mechanism concept is discussed, and criteria are evaluated for effective variability mechanism usage in family engineering. Product line-specific issues of complexity, evolution and refactoring are presented as well. As a whole, the goal of Chapter 2 is to motivate why the product line evolution method developed in the remainder of this thesis is relevant for efficient product line infrastructure evolution in practice.

2.1 The Duality of Use and Reuse

Artifacts developed for reuse in product lines have different shapes [Pohl++05, Ch.4]. In order to prescribe or describe how to systematically create such shapes in source code, a distinction must be made between shape-related concepts of unmodified use vs. reuse [Bassett97]. In this section, the following issues are first discussed in general, before addressing them in a reuse context: What is the physical shape of source code (modules), who creates them (interpreters), and why are they created (in order to be executed).

The artifacts created and changed during the realization activity are modules which contain source code, usually in textual form.

Definition 1: Module

A module is an *artifact (44)* containing a group of symbols "that can be consistently referenced as a unit" (adapted from [Bassett97]).

In this and the following definitions, references to other defined terms are written in italics, followed by the identifier of that term[2]. In this case, the artifact concept is defined in detail later (Def.44). For now, it denotes a tangible software item, e.g. a UML class diagram or a source code file.

Modules are created and changed, and the containing symbols are understood by a software engineer or by an automated device such as a compiler or code generator. These are collectively called interpreters.

Definition 2: Interpreter

An interpreter is "an agent capable of interacting with a *module*" *(1)* [Bassett97]. Note that this concept has a more general meaning than interpreters of programming languages such as the Python interpreter: an interpreter may also be a compiler, a preprocessor, an aspect weaver, a frame processor [Bassett97], or a human engineer.

Within software development, the final output of interpreters is machine code that runs on computer hardware. As for interpreters, the traditional concept of execution can also be generalized, so that it does not only denote the running of machine code, but also the transformation of source code into machine code.

Definition 3: Execution

Execution is the *interpretation (2)* a) of a binary module by computer hardware, or b) of a module "by a compiler-linker-computer trio, or by any functionally equivalent interpreter" (adapted from [Bassett97]). Contrast with: *Construction (27)*.

The input of the execution activity is called an executable module, and the corresponding interpreter is an execution interpreter.

[2] Also, as in [IEEE610], the terms Contast with, Synonym, and See are use, which respectively refer to an opposite concept, the same concept, and a related concept.

Definition 4: Executable Module

An executable module [Bassett97] is a) a binary module that can run on computer hardware, or b) a module that can be compiled and linked to run on computer hardware. Contrast with: *Constructible Module (26)*.

Definition 5: Execution Interpreter

An execution interpreter [Bassett97] is an *interpreter (2)* whose input consists of *executable modules (4)*. Contrast with: *Construction Interpreter (25)*.

This means that an execution interpreter is either computer hardware, or a compiler/linker or an equivalent tool in addition, but it is not a preprocessor or frame processor.

Once a source code element has been written, especially if it was written manually, the software engineer does not want to repeat the same activity again. This is why source code is made persistent, usually in a file. This file is then used as input to the execution interpreter which treats the source code contained in the file as if it had been issued directly.

The input of the execution interpreter usually does not consist of a single file, but of several files. This facilitates parallel multi-person development. When source code elements are stored in several files, they may also be used again when similar systems are needed later. This saves the effort of redevelopment. The activity of using again is called use, the corresponding property is usability, and the agent is a user.

Definition 6: Use

Use is the process of reapplying an *executable module (4)* in unmodified form. Syn.: Use-as-is [Bassett97], Unmodified Reuse. See also: *Reuse (21)*.

Definition 7: Usability

Usability is the capability of an *executable module (4)* to be *used (6)* again. Usability depends on functionality, efficiency and ease-of-change (see [Bassett97, IEEE1517]). See also: *Reusability (19)*.

Definition 8: User

A user is an agent capable of *using (6)* an *executable module (4)* (see [Bassett97]). Note that the same term also denotes a completely different concept: an end-user, a person running the resulting machine code [Synthesis93, Campbell07]. In this thesis, a user is a software engineer exercising reuse without modification. See also: *Reuser (22)*.

Fig.6 illustrates that the role of an execution interpreter is to create machine code from executable modules. These are composed by a user, in unmodified form, for example when he develops application logic and glue code to connect them [Krueger10, p.42].

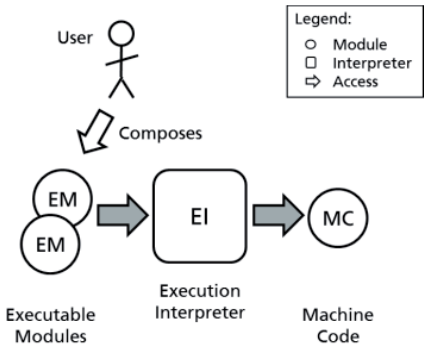


Figure 6: Use-specific transformation of executable modules into machine code

Definition 9: Composition

Composition is a) the *activity (10)* of a *user (8)* who combines *executable modules (4)* without modifying them internally (see [Krueger10, Northrop++06]), or b) the result of the activity in a). Contrast with: *Configuration (28)*.

The concepts of activity and the underlying concepts of process and development are defined as follows:

Definition 10: Activity

An activity is "a set of cohesive tasks of a *process*" *(11)* [IEEE12207].

Definition 11: Process

Process defines, in a repeatable and consistent way, how "*development (12)* is - or should be - performed, i.e. the specific *activities (10)* that need to be conducted" (adapted from [Linden++07]).

Definition 12: Development

Development covers all the *activities (10)* associated with a software product, from conception through client negotiation, design, realization, validation, operation, and *evolution (66)* (adapted from [Shaw05]). Synonym: Development Process.

Execution interpreters restrict their input modules to be spit according to certain rules. Usually, executable modules must contain a set of inseparable primitives that the execution interpreter understands, for example functions, data or objects, in case of a programming language.

For that reason, users may only decompose executable modules into primitives prescribed by the execution interpreter. These are usually programming language primitives, and the executable modules become subroutine libraries.

Substituting a function call by the function's realization is an example of a process called binding: Before binding a function, the option of how that function is realized is still open (i.e., it may still vary), but after binding, that decision is closed.

Definition 13: Binding

Binding is "the act of assigning a value to a variable in a *module*" *(1)* [Bassett97].

Binding is carried out at a particular moment, the binding time.

Definition 14: Binding Time

Binding time is the moment when *binding (13)* happens.

Examples of binding time are development time (the earliest possible binding time in the code development process, when values are bound while the program text is written), compilation time (values are bound during compilation) or runtime (the latest possible binding time, when values are bound while the program runs) [Krueger04; Coplien99, p.73]. Figure 7 gives an overview of binding times in software development.

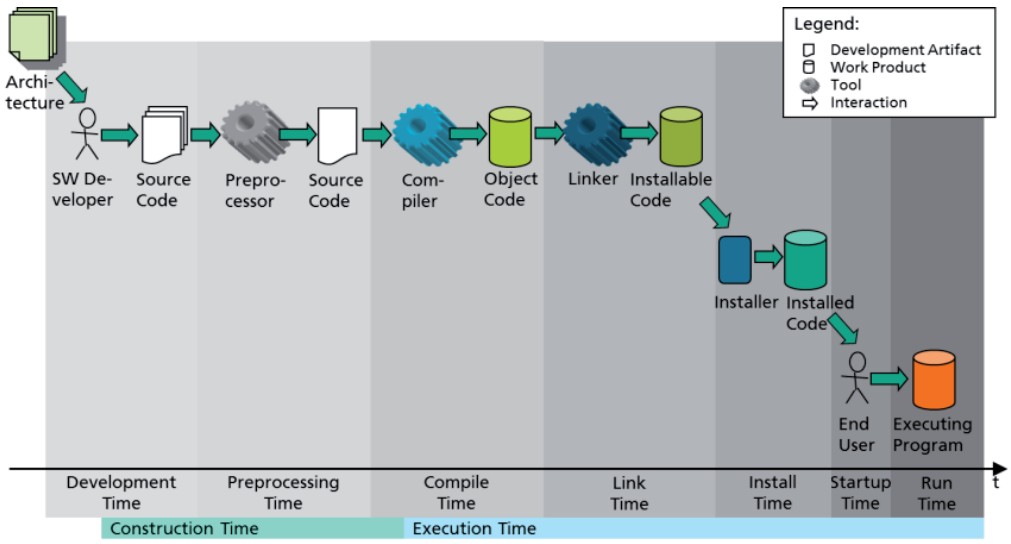


Figure 7: Overview of binding times

In contrast to [Bassett97], this thesis makes a distinction between execution time and runtime in order to classify binding possibilities more precisely.

Definition 15: Execution Time

Execution time is the *binding time (14)* during which an *execution interpreter (5) interprets (2)* an *executable module (4)*, emitting machine code. Contrast with: *Construction Time (31)*, *Runtime (16)*.

Definition 16: Runtime

Runtime is the *binding time (14)* during which machine code runs on computer hardware. Contrast with: *Execution Time (15)*.

In traditional reuse approaches, reusable elements are collections of executable modules, for example class libraries or function libraries. These libraries typically contain more information than necessary for just using (Def.6) them. In order to reduce the complexity of using these artifacts, abstractions are created, for example provided interfaces [Bosch00] which hide module internals and, e.g., only provide essential function or class interfaces to users. More generally, an abstraction has this meaning [Krueger92, Bassett97, Northrop++06]:

Definition 17: Abstraction

An abstraction of an *artifact (44)* is a succinct description which suppresses details that are unimportant for the purpose at hand, while emphasizing properties that are important to this purpose (adapted from [Krueger92, Bassett97, Northrop++06]). See also: *Specification (33)*, *Realization (34)*.

An abstraction of an executable module includes those details that the developer of that module has regarded as relevant for later use. According to the principle of parsimony [Atkinson++01], a well-designed abstraction should be as simple as possible, which means that it should not contain extra properties that a user will never need.

One possibility for making systems less complex to understand and use is to organize them hierarchically. For example, if an executable module abstraction does not offer enough functionality to a user, he may extend it by creating another abstraction which uses and extends the first one. The resulting layers isolate different sources of change, so that modules in different layers become nearly decomposable: they become independent in short-term evolution. The resulting abstraction hierarchy is organized according to function call relations [Dijkstra68], where unimportant details are suppressed by encapsulation.

Definition 18: Encapsulation

Encapsulation hides the elements of an *executable* (3) *abstraction (17)* that its *users (8)* do not need to know (adapted from [Bassett97]). Synonym: Information Hiding.

Encapsulation leads to total invisibility, a property which is stronger than, but often confused with, simple access control provided by object-orientation [Booch91, p.49]: While 'private' elements of an object are inaccessible to a developer using an object, they are not invisible to him. For example, he may still see the signature of private methods, or details of private inner classes, so that he can see more information than he requires. On the other hand, component-based approaches such as the separation between specification and realization in the KobrA method [Atkinson++01], the architectural style of Information Hiding Systems [Shaw+97], or the design patterns Bridge [Gamma++95] or Whole-Part [Buschmann++96] facilitate encapsulation and optimize usability (Def.7) because they hide elements that their users should not know.

In many software development situations, the executable module does not offer its user exactly what is required. For example, a function in a function library might provide too much functionality, so that after calling the function, a user must undo part of the functionality again. A function might also offer not enough functionality, so that each user must add the same missing elements. Or it might offer not quite the required functionality, so that a potential user must rewrite the executable module from scratch. A common problem in all these cases is that a user needs to adapt the executable module exactly for his particular use situation: the module must be more reusable.

Definition 19: Reusability

Reusability is the capability of a *module (1)* to be adapted in order to become *usable (7)* in a specific *context (20)*. Reusability depends on usability, *variability (46)* and adaptability (adapted from [Bassett97, IEEE1517]). See also: Usability.

The context concept used in Def.19 is defined as follows:

Definition 20: Context

Context is "the setting in which [software] *engineering (41)* is practiced. Examples include the working styles of software developers, the values held by a development team, the cultural background of the developers, the paradigm of the code, and the kind of industry for which the software is being developed. Context […] is a multi-dimensional space with an infinite number of points, each point defining a particular software project at a particular time" [Murphy++10].

More examples of different development contexts, for example single- vs. multi-developer or open-source vs. commercial, are also given in the empirical studies on Cloning in Section 3.3.

Like usability, reusability has two closely related concepts:

Definition 21: Reuse

"Reuse is the *process (11)* of adapting" a *module (1)* "in order to make it *usable*" *(7)* (adapted from [Bassett97]). See also: *Use (6)*.

Definition 22: Reuser

A reuser is an agent capable of *reusing (21)* a *module (1)*. See also: *User (8)*.

These definitions imply that the most basic type of reuse is use, and reuse is a superset of use. However, the two concepts are still frequently confused. As will be clarified in Sec.2.3, the concept of commonality in product lines is associated with use, while reuse is both concerned with commonality (its use subset) and variability. Use and reuse also address different development goals, and this is why they are duals [Bassett97, pp.78ff.]. For each of the concepts introduced so far in a use context (e.g. executable module or composition), a corresponding concept exists in a reuse context, presented next, and this is most relevant for effective product line realization, where a product line is defined as follows:

Definition 23: Product Line (PL)

A product line [Synthesis93] is a set of similar systems that "share a common, managed set of *features (63)* satisfying the *needs (24)* of a particular market segment […], and that are developed from a common set of *core assets (56)* in a prescribed way" [Clements+01]. Note that "product lines do not mean fortuitous, small-grained reuse, single-system development with reuse, just component-based or service-based development, just a reconfigurable architecture, releases and versions of single products, or just a set of technical standards" [Northrop+07].

In this definition, features and needs are associated with early engineering processes such as requirements engineering. The feature concept will be defined in detail later; for now, it means an abstract requirement, as suggested in [Pohl++05].

Definition 24: Needs

Needs are "the considerations that customers identify as desired capabilities, perceived weaknesses, or desired improvements in a system of interest" [Campbell07]. See also: *Requirements (35)*.

In order to clarify the distinction between use and reuse, the same set of issues that have been discussed at the start of this section for single-systems engineering are now discussed in a product line context. What is the physical shape of reusable code (constructible modules), who creates them (a construction interpreter), and why are they created (in order to be constructed).

As in a reuse-agnostic context, the artifacts developed during product line realization contain source code text. They may be created manually or automatically. However, in product line development their text is not only concerned with the functionality of a single software system, but simultaneously with the similarity of a set of systems. The primary goal of developing this code is to transform it into product-specific code, and only a secondary goal is to transform it into machine code. As in the case of use, a reuser may also need to do this automatically. A corresponding type of tool for transforming reused code into a product-specific form is called a construction interpreter.

Definition 25: Construction Interpreter

A construction interpreter [Bassett97] is an *interpreter (2)* whose input consists of *constructible modules (26)*. Contrast with: *Execution Interpreter (5)*.

Construction interpreters are preprocessors, such as the C preprocessor *cpp*, frame processors (Sec.4.7), or similar tools. The concepts of constructible module and construction have these definitions:

Definition 26: Constructible Module

A constructible module is a *module (1)* that is *interpreted (2)* by a *construction interpreter (25)*. Synonyms: Component [Bassett97], Meta-Component [Bassett02]. Contrast with: *Executable Module (4)*.

Definition 27: Construction

Construction is the *interpretation* of a *constructible module (26)* by a *construction interpreter (25)*. Contrast with: *Execution (3)*.

Fig.8 shows a setting dual to the use situation from Fig.6: A reuser configures constructible modules which are then constructed into executable modules by a construction interpreter.

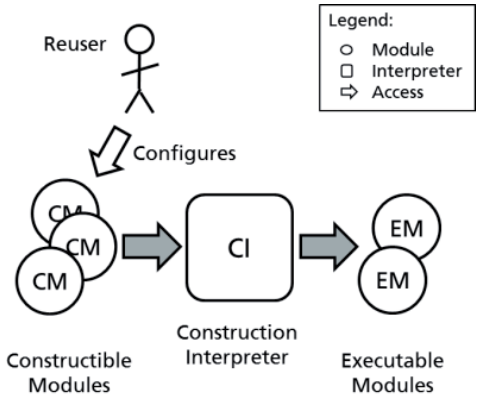


Figure 8: Reuse-specific transformation of constructible modules into executable modules

Like executable modules, constructible modules can be organized hierarchically in order to make it easier for reusers to understand and reuse them (see Section 2.2). The activity is called configuration.

Definition 28: Configuration

Configuration is the *activity (10)* of a *reuser (22)* adapting *constructible modules (26)* to modify them internally via manual techniques or automated mechanisms (see [Krueger10]). Contrast with: *Composition (9)*.

Configuration is the dual activity to composition. Whereas composition is applied by a user in order to change an executable module externally, configuration is applied by a reuser in order to change a constructible module internally. Both activities are instances of customization [Krueger10, Codenie++10]. In product line development, mass customization leads to the mass production of individualized products.

Definition 29: Mass Customization

Mass customization [Davis87, Pine93] focuses on the means of efficiently *producing (30)* and *evolving (66)* multiple similar products, "exploiting what they have in common and managing what varies among them" (see [Krueger02a]). Synonym: Software Manufacturing [Bassett97].

Definition 30: Production

Production is "the process used for building all products in a *product line*" *(23)* [Northrop+07]. Synonyms: Instantiation [Synthesis93], Product Derivation [Deelstra++05], Production Process [Northrop+07]. Note that "the production activity can be fully automated, completely manual, or somewhere in between" [Krueger04].

As in the architectural style Pipes and Filters [Shaw+97], a construction interpreter and an execution interpreter may be concatenated (Fig.9), so that constructible module input is directly transformed into product-

specific machine code. Alternatively, the two tools may be combined in a single tool, as in C/C++ preprocessor-compiler-linker trios.

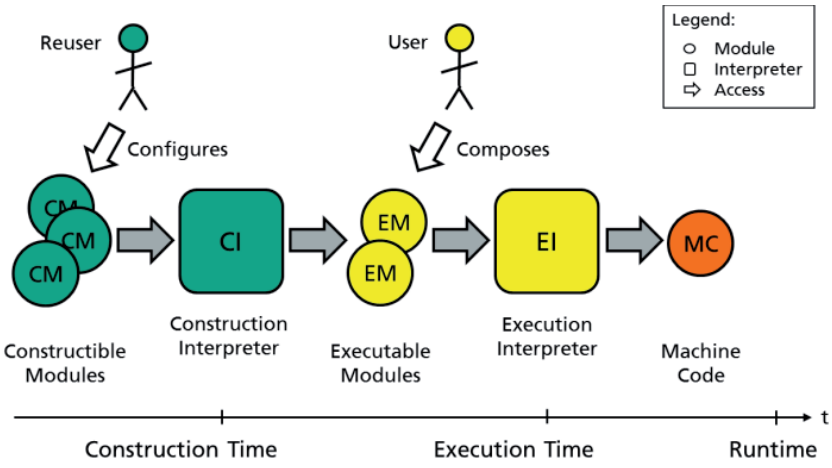


Figure 9: Reuse includes use: construction and execution of modules

Once a code element has been written in order to be reused, its developer usually makes it persistent, for example in a file. The construction interpreter then accepts the file as a surrogate of its contents, the code. Such code may be provided to a construction interpreter in several files, so that work may be assigned to several software engineers. Input files of a construction interpreter do not need to be split according to programming language primitives, but only to primitives of the construction interpreter's language. A construction interpreter can be totally independent of an execution interpreter. It may process text of arbitrary programming languages or any type of textual artifact. Constructible modules are bound at construction time which is always earlier than execution time or runtime (Figs.7, 9).

Definition 31: Construction Time

Construction time is the *binding time (14)* during which a *construction interpreter (25) interprets (2)* a *constructible module (26)*, emitting *executable modules* (4). Contrast with: *Execution Time (15)*.

Abstraction (Def.17) is an important property for reuse [Krueger92] because it allows a constructible module to highlight its essential adaptation possibilities, while deemphasizing what always remains common. This makes it easier to reuse the module because the reuser can concentrate on the module's configuration interface [Bosch00] without paying attention to module details that do not vary among all reuse situations. While encapsulation (Def.18) is important for effective use because it separates the elements a user does not need to know about from those he needs to know, it restricts reuse because it prevents reusers from adapting modules in ways its developers have not foreseen.

2.2 The Reuse Hierarchy

As indicated in Section 2.1, layering makes it possible to organize reusable elements according to their reusability. However, hierarchies are meaningless if the relations among the layers are not specified exactly [Parnas74]. This section shows that a more general type of hierarchy exists for artifacts of a product line, which extends the hierarchy concept known in traditional unmodified reuse.

In single-system evolution, it has been shown that hierarchies organized according to the use relation are beneficial [Parnas76]. The use relation has been defined as follows:

Definition 32: Use Relation

"We say of [two modules] A and B that A *uses (6)* B if correct *execution (3)* of B may be necessary for A to complete the task described in its *specification (33)*. That is, A uses B if there exist situations in which the correct functioning of A depends on the availability of a correct *realization (34)* of B" (adapted from [Parnas79]). See also: *Reuse Relation (39)*.

The specification and realization concepts from Def.32 and their associated concepts are defined as follows:

Definition 33: Specification

A specification serves to state *requirements (35)*, and represents the higher of the two levels [of an *abstraction (17)*] [Krueger92]. Contrast with: *Realization (34)*.

Definition 34: Realization

Realization is a) the lower, more detailed level [of an *abstraction (17)*] [Krueger92], or b) the process of developing the artifact in a).

Definition 35: Requirements

Requirements are "the criteria, consistent with *needs* (24) and constraints, that determine whether a product is acceptable as a *solution (37)* to a *problem*" *(36)* [Campbell07].

Definition 36: Problem

A problem is "the gap between a system as it exists and the system as would better enable a customer in achieving objectives" [Campbell07]. See also: *Problem Space (50)*. Contrast with: *Solution (37)*.

Definition 37: Solution

A solution is "a means of transforming a system to resolve an identified *problem" (36)* [Campbell07]. See also: *Solution Space (51)*. Contrast with: Problem.

Section 2.1 has discussed that use alone is often insufficient to organize common and variable artifacts of a product line. For a reuser, it is important that the provided code is adaptable to his specific development situation, which means that it must be reusable, not just usable. This means that the resulting hierarchies must be organized according to a different relation, the reuse relation. The artifacts realizing a product line can then be divided into a set of constructible modules that are configured to become a product line member, produced by a construction interpreter. Each constructible module has a specification, a production plan which describes precisely the steps of the production process.

Definition 38: Production Plan

A production plan is a guide to show how products in the product line will be *composed (9)* and *constructed (27)* from *modules (1)* (adapted from [Clements+01, Krueger10]).

Transferring Parnas' concept of use relations (Def.32), I define reuse relation this way:

Definition 39: Reuse Relation

We say of two modules A and B that A *reuses (21)* B if correct *construction (27)* of B may be necessary for A to complete the *production process (30)* described in its *specification (33)*. That is, A reuses B if there exist situations in which the correct production of A depends on the availability of a correct *realization (34)* of B. See also: *Use Relation (32)*.

For two modules A and B, reuse and use coincide when A does not need to perform construction of B in order to meet its specifications. This is either the case if B does not provide a configuration interface [Bosch00] to A, or if A does not need to use B's configuration interface because appropriate defaults (Def.55) are provided for all configuration options.

The given reuse definition extends all reuse definitions suggested previously because it includes two novel aspects: First, it explicitly considers that product line modules require a production plan as part of their specification. Second, it addresses correctness of reuse.

Reuse hierarchies are defined next, as an extension of Parnas' use hierarchy concept [Parnas74]. Compared to use hierarchies which only consist of fixed modules, reuse hierarchies have the advantage that they may also contain adaptable modules which are necessary to efficiently realize product lines (see Sec.2.3). The graph of reuse relations is directed and loop-free. The constructible modules which are linked by reuse relations form a hierarchy. Consider a reuse relation R(A,B) among the constructible modules A and B, where A reuses (Def.21) B. If a hierarchical order exists, it forms a reuse hierarchy.

Definition 40: Reuse Hierarchy

In the reuse hierarchy which is formed when a *constructible module (26)* A *reuses (21)* a constructible module B, there exist reuse levels with the following properties:

1. Level 0 is the set of all constructible modules A such that there does not exist a constructible module B for which R(A,B)
2. Level n is the set of all constructible modules A such that
   a) there exists a constructible module B at level n-1 such that R(A,B), and
   b) if R(A,C) then C is at level n-1 or lower (adapted from [Parnas74]).

In other words, no constructible module at level 0 reuses another constructible module, and a constructible module at level n>0 reuses at least one constructible module at level n-1 and no constructible module above level n-1.

I have identified the following consequences of reuse hierarchies:

- The constructible modules at level 0 have the highest reusability. They realize commonalities or quasi-commonalities in a product line. They do not need to know about any situation in which they are reused, and so they are most context-free (i.e., most independent of the context in which they are reused).
- Constructible modules at level 1 are less reusable because they need to have knowledge about the constructible modules at level 0 which they reuse. They cannot be reused by constructible modules at level 1 or level 0.
- Constructible modules at the highest level are completely context-specific. They are completely dependent on a specific reuse situation and thus determine each product line member (Def.48).

One benefit of a reuse hierarchy is that the artifacts realizing a product line can be tested incrementally, starting with the most reusable ones at level 0. Another advantage is that some artifacts realizing the product line, those constructible modules at the same level of the hierarchy, can be tested and evolved in isolation.

Another advantage of a reuse hierarchy is complexity reduction by separating context-sensitive from context-free elements, highlighting novelty and hiding sameness. In particular, when a system is divided into two constructible modules A and B so that A reuses B, then A, the more product-specific element, becomes simpler because it reuses B. B does not become significantly more complex because it is not related to A. There is a reusable subset which requires B but which does not need A. There is no conceivably reusable subset which requires A, but not B.

These properties help to decide if two constructible modules should be organized in a larger constructible module C (Fig.10a), if both should be siblings (Fig.10b), or if one should reuse the other (Figs.10c,d).

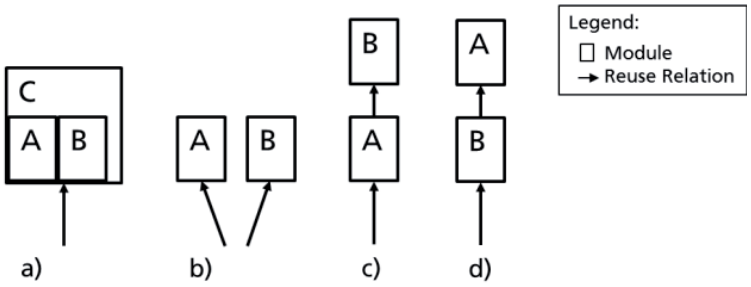


Figure 10: Four possibilities for organizing two constructible modules A and B [Bassett97, p.173]

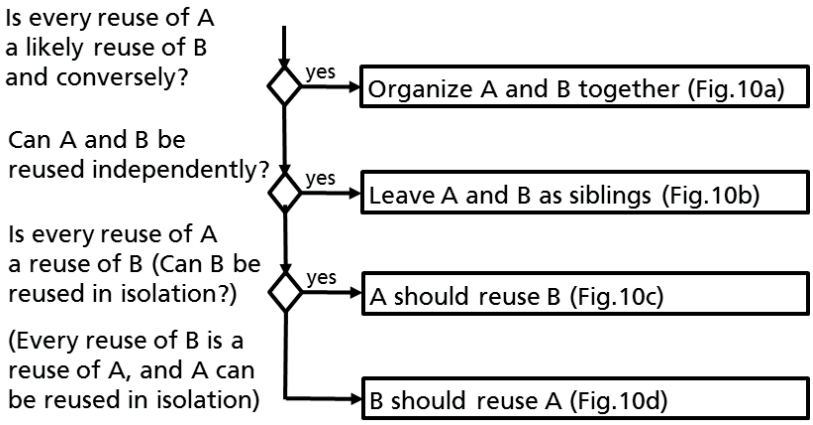


Figure 11: Decision tree for organizing reuse hierarchies

Fig.11 shows a decision tree I invented for guiding family engineers in organizing reuse hierarchies. If every reuse of A is likely also a reuse of B and conversely, they belong in the same larger constructible module C. Otherwise they belong in separate constructible modules. In this case, if A and B can be reused independently, they should remain in sibling constructible modules. Else they form two levels of a reuse hierarchy. If every reuse of A is a reuse of B but not conversely, the constructible module A should reuse B. Otherwise, every reuse of B is a reuse of A, so A should reuse B.

2.3 Evolution in Product Line Engineering

As indicated in Section 1.1, the goal of product line engineering is "to provide customized products at reasonable costs" [Pohl++05, p.9]. This is done by developing artifacts that are not more complex than necessary because they offer only what must be reused. In this section, some single systems concepts are defined first to clarify which activities are addressed (engineering), whom they serve (stakeholders), what problems arise (complexity), and which items are affected (artifacts). Based on these concepts, product line concepts are then explained.

Definition 41: Engineering

Engineering is a *process (11)* "governing the total technical and managerial effort required to transform a set of […] *needs (24)* [of *stakeholders (42)*] into a *solution (37)* and to support that solution throughout its life" [ISO24765]. The goal of engineering is to support "practical, cost-effective solutions to *problems (36)* [in system *development (12)*] in a timely and predictable manner, preferably by applying scientific knowledge" [Shaw05]. Synonym: Systems Engineering [ISO24765].

Definition 42: Stakeholder

A stakeholder is someone who has a vested interest in a system and who is entitled to contribute to *requirements (35)* (adapted from [Jackson01, Clements++03, Bayer04]). End users or customers are typical stakeholders.

Definition 43: Complexity

Complexity is the absence of simplicity [Alexander02] in an *artifact (44)* or *process (11)*. This defect makes the artifact more difficult to *develop (12)* than necessary. It arises when elements have been realized in *engineering (41)* that are not immediately required by *stakeholders (42)*. Complexity reduction aims at making the artifact easier to understand and change. Synonyms: Arbitrary Complexity [Brooks95], Excess Complexity. See also: *Variability Complexity (65)*. Contrast with: Parsimony.

Definition 44: Artifact

An artifact is the output of an *engineering (41)* process. An artifact may be a requirements specification, an architecture, a source code module, a test case, or any other useful process result (see [Clements+01, Pohl++05]). Synonyms: Development Artifact [Pohl++05], Development Asset [Linden++07], Work Product [Jalote05].

Artifacts become overly complex if they contain more elements than they need to have. Product lines need to have, and take advantage of, their products' commonality and predicted variation [Weiss+99]. This distinguishes them from single systems [Muthig02, Krueger04]. Principles of variability in product line engineering have been extensively discussed in [Pohl++05, Ch.4]. In the current thesis, the concepts of commonality and variability are defined as follows:

Definition 45: Commonality

Commonality [Synthesis93] of a *product line (23)* prescribes what needs to be identical among a set of *product line members (48)*. The goal of commonality is to facilitate rapid, cost-effective *development (12)*. Contrast with: *Variability (46)*.

Definition 46: Variability

Variability [Synthesis93] of a *product line (23)* prescribes what may differ among a set of *product line members (48)*. "The goal of variability […] is to maximize return on investment [for *developing (12)* products] over a specified period of time or number of products" [Bachmann+05]. Variability is concerned with a) differences in artifacts at the same time (variability in space), b) different temporal versions of an artifact (variability in time, *evolution (66)*), or c) a combination of a) and b) (variability in time of variability in space). The major types of variability are optional and alternative variability. Synonym: Variability Subject [Pohl++05]. Contrast with: *Commonality (45)*.

Commonality and variability have also been defined as follows:

Let $P(R_i) = \{p \mid \forall (r \in p) : p \text{ satisfies } R\}$ be the set of all products characterized by the product requirements $R \subseteq D$, where D is the application domain, and let S be the product line scope, then

$Com(S) = \bigcap_{i=1}^{n} P(R_i)$, ($R_i \in S^o$) contains all commonalities in S, that is, all commonalities of the system family defined by S.

Let S be the product line scope, then Var(S) = S - Com(S) contains the variabilities of S (adapted from [Muthig02, John10]).

A related concept to variability is variation.

Definition 47: Variation

Variation [Bachmann+05] of a *product line (23)* is a particular instance of *variability (46)*. The goal of variation is to define how *product line members (48)* have to differ conceptually from each other. Synonyms: Variability Object [Pohl++05], Parameter of Variation [Weiss+99].

Definitions 45 to 47 depend on the concept of product line members.

Definition 48: Product Line Member

A product line member [Withey96] is a "deployed software-intensive system or software" [Northrop+07] "that has been defined [by *stakeholders (42)*] to be built [from a *product line infrastructure (62)*]" [Metzger++07]. Note: A software-intensive system is "a system in which a significant degree of essential behavior is realized through software" [Campbell07].

The product line infrastructure concept will be defined in detail soon; for now, it means a set of reusable artifacts.

The concepts of commonality, variability and variation are now illustrated in a running example of a wireless sensor node product line, an example which will be used throughout this thesis. A wireless sensor node is a small embedded systems device that detects environmental conditions and transmits them to other sensor nodes. It is equipped with sensors and a wireless transceiver. Stakeholders have defined that product line members shall be developed whose commonality is reliable wireless communication, whose variability is the particular monitored variable [Parnas+95] of the physical environment, and whose variations are tilt detection (has the sensor node been tilted), drop detection (has the sensor node been dropped), and noise detection (what noise level exists near the sensor node). The sensor nodes are developed using the product line engineering paradigm.

Definition 49: Product Line Engineering (PLE)

"Product line engineering is an *engineering (41)* approach that subsumes all *processes (11)* […] supporting the *development (12)* […] of a *product line (23)*" [Muthig09].

A related definition of product line engineering is:

Product Line Engineering PLE=(S, FE, AE, I) is a software engineering approach that consists of a scope $S \subseteq D$ of an application domain D, a family engineering approach FE, an application engineering approach AE, and a product line infrastructure I (adapted from [Muthig02]).

In product line engineering, variability and the corresponding variations occur in the product line's problem space. Later engineering activities are called the solution space. Together, the problem and the solution space are the development phases of product line engineering (Fig.12).

Definition 50: Problem Space

The problem space [Czarnecki+00] of a product line refers to early activities in *product line engineering (49)* where *product line members (48)* are *specified (33)*. See also: *Problem (36)*. Contrast with: *Solution Space (51)*.

Definition 51: Solution Space

The solution space [Czarnecki+00] of a product line refers to later activities in *product line engineering (49)* where *product line members (48)* are *realized (34)*. See also: *Solution (37)*. Contrast with: *Problem Space (50)*.

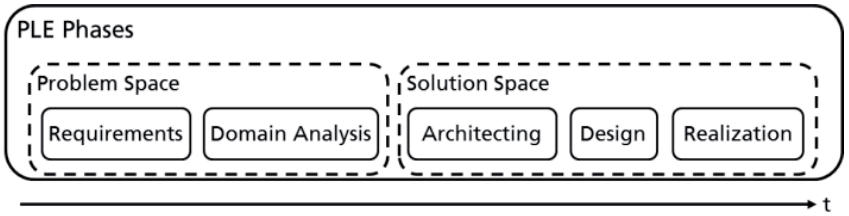


Figure 12: Problem space and solution space in product line engineering

Variability and variations are identified and specified in the problem space, in early engineering activities of scoping and requirements engineering, when stakeholders and engineers interact. In later engineering activities of the solution space, such as architecting and realization, a particular product line is realized. This is reflected in a dual pair of concepts to variability and variation: variation point and variant, as proposed in [Pohl++05].

Definition 52: Variation Point (VP)

A variation point [Jacobson++97] is a particular *realization (34)* of *variability (46)* within *product line assets (59)*. The main purpose of variation points is to highlight where variability occurs within the realized *commonality (45)*, making the realized *variations (47)* easy to see and control. Synonyms: Hot-Spot [Pree94], Engineering Change Point [Bassett87]. Contrast with: Variability, *Variant (53)*.

Definition 53: Variant

A variant [Jacobson++97] is a *realization (34)* of *variation (47)* within *product line assets (59)*, at a particular *variation point (52)*. A variant consists of one or more *variant elements (54)*. The purpose of variants is to realize how *product line members (48)* differ from each other.

Definition 54: Variant Element

A variant element is cohesive part of a *variant (53)*.

The concept of product line assets used in Def.52 and Def.53 is discussed in detail later; for now, it denotes artifacts that realize a product line's commonality and variability.

Table 1 summarizes how the four concepts of variability, variation, variation point and variant are related. It combines the problem space terminology used in [Bachmann+05] with the solution space terminology used in [Jacobson++97] in a way proposed in [Pohl++05].

	Concept that expresses what varies	**Concept that expresses how it varies**
Problem Space	Variability	Variation
Solution Space	Variation Point	Variant

Table 1: Variability concepts in the problem and solution space

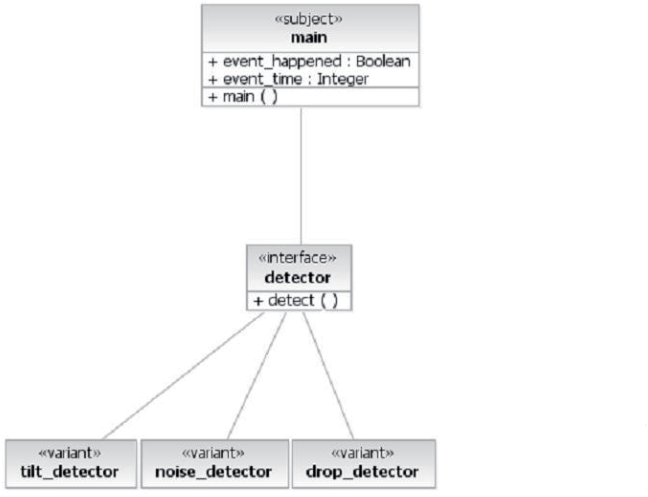


Figure 13: Structural architectural model of a sensor node

To illustrate these concepts, consider Fig.13 of the running example. It shows an excerpt of a structural architectural model of a sensor node product line in form of a UML class diagram in KobrA notation [Atkinson++01]. In this model, the *detect* method of the abstract detector class serves as a variation point, realizing the detection variability. The corresponding variants realize the variations of tilt, drop,

and noise detection as *detect* methods of the concrete subclasses tilt detector, drop detector, and noise detector. The KobrA and PuLSE [Bayer++99] methods developed at Fraunhofer IESE consistently use the tag <<variant>>, or its abbreviation <<var>>, for tagging variants in all types of artifacts (see Fig.13). The proposed terminology is consistent with the terminology in these methods.

A special type of variant which serves to optimize variability management [Patzke08, Savolainen++09] is called a default.

Definition 55: Default

A default [Bassett97] is a *variant (53)* that is automatically chosen if no other variant is selected in its place (adapted from [Gomaa04]). The purpose of defaults is to simplify *production (30)*, decreasing the number of *configuration (28)* options.

The concept of product line assets has been used in the definitions of variation point (Def.52) and variant (Def.53). In order to define what a product line asset is, two other concepts characterizing a product line asset must be introduced first: core asset and variability asset.

Definition 56: Core Asset

A core asset [Bass++97] is a *reusable (19) artifact (44)* that is developed for reuse in more than one *product line member (48)*. Core assets explicitly capture the product line's *commonality (45)* and predicted *variability (46)*. The task of core assets is to support the efficient *production (30)* of all product line members. Synonyms: Product Line Artifact [Muthig02], Reuse Asset [IEEE1517], Software Asset [Withey96]. Contrast with: *Variability Asset (57)*.

Core assets contain the same types of elements known in conventional single-systems engineering. Core assets alone have turned out to be insufficient for capturing variability, especially in a consistent, traceable and unambiguous form [Muthig02, Bachmann++04, Berg++05, Pohl++05, John++07]. Traceability of variability is achieved by a different type of artifact that is developed independent of core assets, and that only captures variability information. This type of artifact does not yet have a general name; it is called variability asset in this thesis.

Definition 57: Variability Asset

A variability asset is an *artifact (44)*, such as a Decision Model [Synthesis93, Bayer++99], a Variability Diagram [Pohl++05], or a Product Model [Krueger10], that captures the relationships, constraints and resolutions of *variability (46)* in *core assets (56)* in an integrated form. The task of variability assets is to facilitate *traceability (58)* of variability throughout the engineering life cycle. Contrast with: Core Asset.

Traceability, as used in the definition of variability assets, is defined according to definitions given in [IEEE610, Bayer+02, Berg++05]:

Definition 58: Traceability

Traceability is the ability to establish a relationship between two *artifacts (44)* developed in different *engineering (41)* phases, for example between a requirements specification and a design. The purpose of traceability in *product line engineering (49)* is to efficiently identify dependencies between *core assets (56)* that exist due to *variability (46)*.

Core assets and variability assets together are called product line assets.

Definition 59: Product Line Asset

A product line asset [Brownsword+96] consists of a set of *core assets (56)* and the corresponding *variability assets (57)*. The task of product line assets is to capture the output of *family engineering (60)* in an integrated form. Synonym: Domain Artifact [Pohl++05].

The separation of product line assets into core assets and variability assets leads to two model types, the core asset model (also called main or basic system model [Linden++07]), and the variability model (synonyms: Variation Model [Bachmann++04, Käkölä+06, Linden++07], Conceptual Variability Model [Berg++05]). The Orthogonal Variability Model [Pohl++05], the PuLSE Decision Model [Bayer++99, Muthig02] or the CVV [Sinnema++04] are variability model instances. More details and examples of orthogonal variability modeling are shown in [Atkinson++01, Ch.15; Muthig02; Pohl++05; Bayer++06].

As an example, Fig.14 shows an excerpt of the wireless sensor node product line assets, with the variability model in the upper part and the core asset model in the lower part. The variability model consists of variability assets and refers to the core asset model by means of variation point references. In this example, variability assets are organized hierarchically, realized by decision models. The high-level decision model refers to all low-level decision models, and these reference variation points and variants of the associated core assets, for example features in the feature model, classes in the UML class diagram, or *#ifdefs* in the C

source code. Each colored element in the depicted core assets represents a variant element. All same-color variant elements belong to the same variant. Traceability of variability is achieved by the variability model, for example, time transmission is traceable from the high-level decision model to all core assets by following the corresponding references.

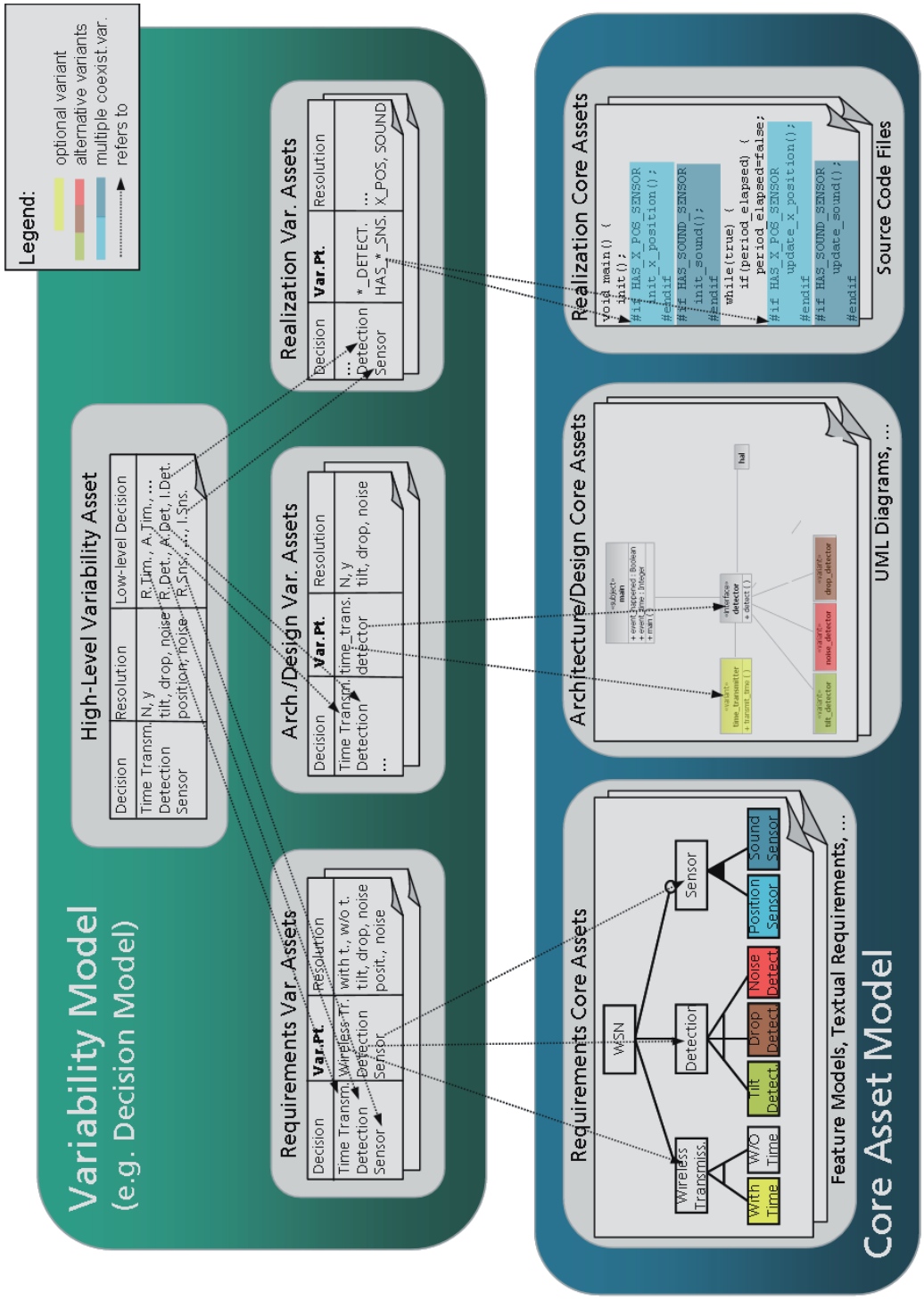


Figure 14: Product line assets of a wireless sensor node product line

Product line assets are the output of family engineering.

Definition 60: Family Engineering (FE)

Family engineering [Muthig02] is the process of *product line engineering (49)* in which *product line assets (59)* are developed for a given scope, i.e. according to sharp domain boundaries determined by *stakeholders (42)*. Domain Engineering [Campbell++90] consists of family engineering and Scoping [Schmid03]. The aim of family engineering is to reduce *variability complexity (65)* by developing just the required product line assets. Synonym: Development for Reuse [Karlsson95]. Contrast with: *Application Engineering (61)*. Note: Family engineering and product family engineering [Synthesis93, Gomaa04] (*product line engineering (49)*) are different concepts.

An organization often requires more than family engineering and product line assets to efficiently apply product line engineering.

Definition 61: Application Engineering (AE)

Application engineering [Campbell++90] is the process of *product line engineering (49)* in which a particular *product line member (48)* is *produced (30)* by consuming elements from the *product line infrastructure (62)*. The aim of application engineering is to efficiently produce all required product line members. Synonym: Development with Reuse [Karlsson95]. Contrast with: *Family Engineering (60)*.

Family engineering and application engineering have also been defined as follows:

"Family Engineering FE: D→**I**: is a set of activities that constructs and evolves a product line infrastructure I, that is, a reuse infrastructure for products in the application domain D. Thus, I contains artifacts that are individually related to concepts in the application domain to guide the definition, documentation, classification, and evolution of all artifacts in I" [John10].

Application Engineering AE: $D \times I \rightarrow P$ is an engineering approach that constructs a concrete application p characterized by the product requirements by using the product line infrastructure I. The product p is valid if it satisfies R, the requirements of a particular customer or customer group (adapted from [Muthig02]).

The artifact interconnecting the two processes of family engineering and application engineering is called the product line infrastructure. Family engineering produces the product line infrastructure, and application engineering consumes it.

Definition 62: Product Line Infrastructure (PLI)

A product line infrastructure [Bayer++99] is a repository of all *product line assets (59)* of an organization, including common methods [Synthesis93] and tools for developing these assets in *family engineering (60)*, and for reusing them in *application engineering (61)*. The main tasks of a product line infrastructure are a) to capture all types of elements relevant in the product line engineering life cycle, and b) to provide an explicit interface between family engineering and application engineering. Synonyms: Infrastructure [Synthesis93], Core Asset Base [Bass++98], Product Line Asset Base [Brownsword+96], Reuse Infrastructure [Bassett97], Platform [Meyer+97].

Fig.15 shows the metamodel for product line infrastructures used in this thesis.

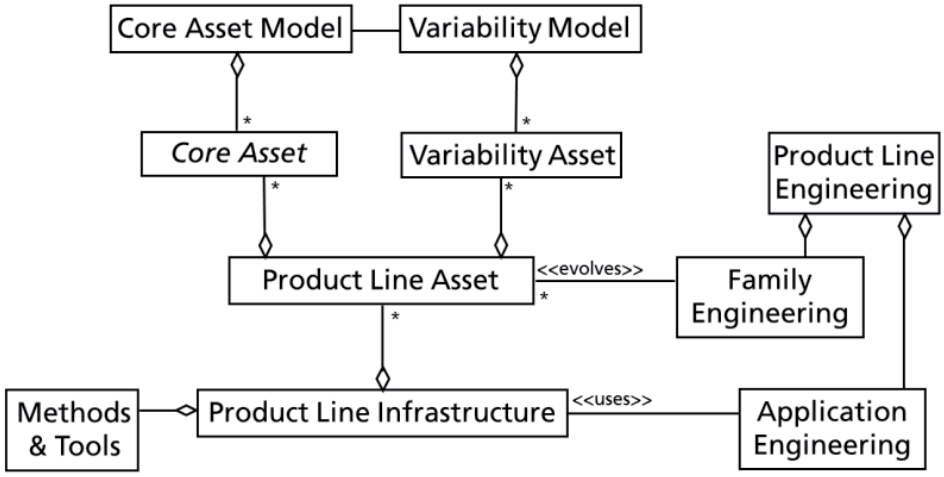


Figure 15: Metamodel for product line infrastructures

Product line infrastructures are a central concept in the PuLSE method developed at Fraunhofer IESE which supports the different engineering sub-processes Product Line Infrastructure Construction, Product Line Infrastructure Evolution and Product Line Infrastructure Usage [Bayer++99]. Fig.16 gives an overview of the product line engineering life cycle in PuLSE (compare Fig.3). The scoping activity creates a well-bounded domain for the product line by only considering immediate product requirements. Family engineering develops a product line infrastructure which is reused by application engineering.

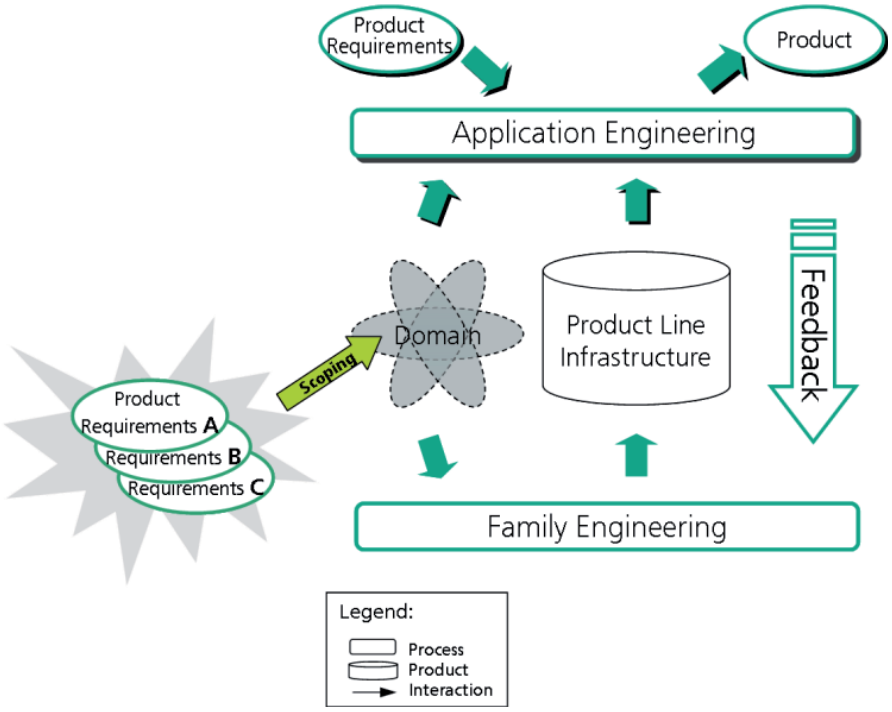


Figure 16: Product line engineering life cycle

Fig.17 shows more details, as developed in this thesis, of the interaction between family engineering and application engineering, facilitated by a product line infrastructure. The shown life cycle model extends existing state-of-the-art models [Muthig02, Pohl++05] by refining the processes and core assets associated with quality assurance: Product line inspection and in particular product line measurement are introduced as quality assurance sub-activities where previous work only considered testing. Quality assurance is also applied to all types of core assets (requirements, architecture, and even variability assets), not just to code (to see the extension, compare Fig.17 to [Pohl++05, Fig.2.1]).

Both product line engineering processes consist of the sub-processes of requirements engineering, architecting, realization and quality assurance. Family engineering produces the corresponding core assets, for example requirements, architecture, code and test cases, plus variability assets, such as decision models, within the variability model. Together with common methods and tools for managing product line assets, for example for adapting core assets or for tracing variability, these artifacts constitute the product line infrastructure which is consumed by application engineering in the production of product line members.

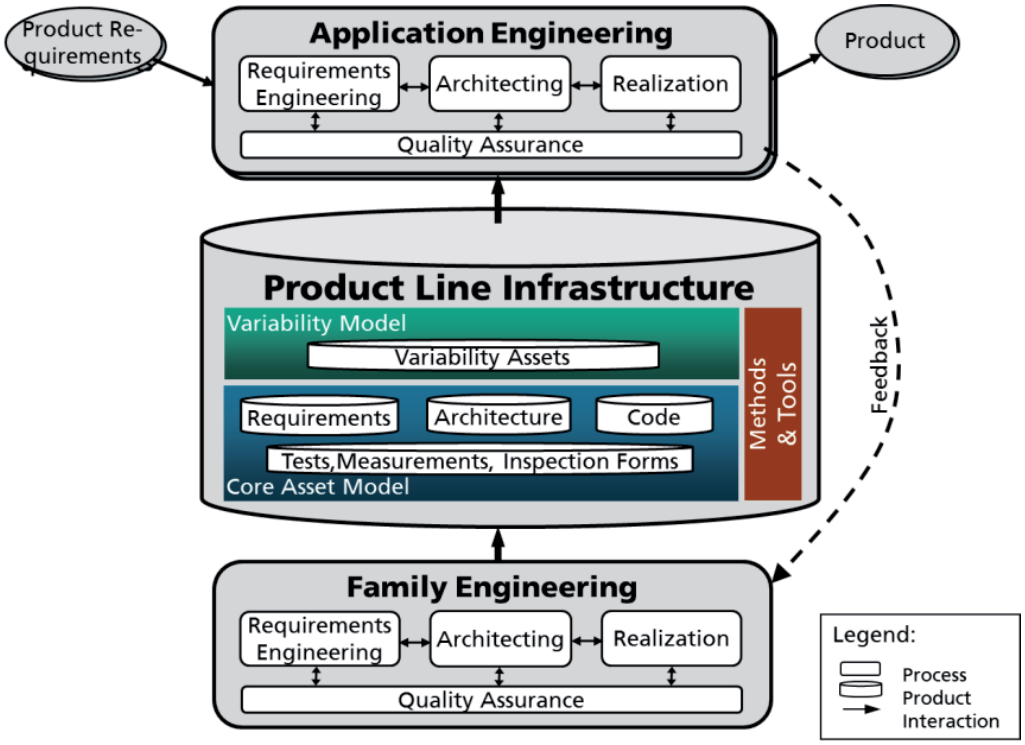


Figure 17: Product line engineering life cycle details

In the definition of product lines (Def.23), the feature concept has been introduced denoting an abstract requirement. As features are also depicted as part of the feature models in Fig.14, they will now be defined in detail.

Purely requirements-related definitions of features, originally proposed by Kang et al. [Kang++90], have been accepted in the product line engineering literature. They characterize a feature or features as

- "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems" [Kang++90],
- "a use case, part of a use case or a responsibility of a use case" [Jacobson++97],
- "a logical unit of behavior that is specified by a set of functional and quality requirements" [Bosch00],
- "a common language between many stakeholders. They communicate the high-level functional requirements from the marketing to the development" [Savolainen+01],
- "an aspect valuable to the customer" [Riebisch03],
- "a functional requirement; a reusable product line requirement or characteristic. A requirement or characteristic that is provided by one or more members of the software product line" [Gomaa04],
- "product capabilities and characteristics that are important to the user" [Berg++05],

- "an abstract requirement. Features describe the functional as well as the quality characteristics of the system under consideration" [Pohl++05],
- "a triplet, f = (R, W, S), where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification" [Classen++08],
- "any kind of system property or requirement that is considered important enough to be part of a general product characterization" [Muthig09],
- "a product requirement $R \subseteq D$ that is visible to a user of the product P [in the application domain D]" [John10].

In contrast, other work also partially or completely subsumes elements of the solution space in their notion of features, for example as

- "an increment of functionality, usually with a coherent purpose" [Zave99], cited in [Batory05, Batory++06, Trujillo07],
- "a product characteristic that is used in distinguishing programs within a family of related programs" [Batory++04],
- "a property of a domain concept, which is relevant to some domain stakeholder and is used to discriminate between concept instances" [Czarnecki+00],
- "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems. An asset type that is used to model functional aspects of a product" [Hotz++06],
- "a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option" [Apel++07],
- "a characteristic or trait in the broadest sense that an individual product instance of a product line may or may not possess. A feature only describes what is variable, not how this variability is realized." [Reiser08].

These definitions either correspond to the concept of software feature which subsumes both problem and solution space, or to the concept of technical feature [Savolainen+01] which only relates to the solution space. The software engineering terminology standard defines software feature as "(1) a distinguishing characteristic of a software item; (2) a software characteristic specified or implied by requirements documentation" [IEEE610]. In the same standard, the feature concept only refers to the second, problem-oriented element. The first, solution-oriented element is sometimes referred to as a technical feature, as observed by Savolainen et al.: "Not all requirements can or should be considered to be features. Since features are usually used as a method to communicate between marketing and development, technically aligned small pieces of system functionality should not be modeled as features.

Naturally, the functionality that cannot be observed externally or is not important for the customer or the user of the system should be modeled as functional requirements. In system-family context, the term technical feature is often used for functional requirements and for lower level features" [Savolainen+01]. The lack of a clear feature concept has also been criticized elsewhere ("features – an overused and underdefined term" [Parnas07]; "this notion [of a feature in the product line context] appears to be confusing, mixing various aspects of problem and solution" [Classen++08]).

As in the software engineering standard, the feature concept used in this thesis is only related to requirements, as outputs of the problem space, but not to the solution space. Feature models, as requirements artifacts, could be regarded just as relevant for single systems engineering as for product line engineering. However, there has not yet been a pressing need for feature models in single systems engineering, where the notion of variability is meaningless. Only in product line requirements engineering, feature models and features found a wider acceptance, for communicating to stakeholders what may differ among a set of product line members: the variability of the product line. With this background, the following feature definition is used in this thesis:

Definition 63: Feature

In *product line engineering (49)*, a feature [Kang++90] is an end-user visible functional or non-functional characteristic of a *product line member (48)*. The goals of features are associated with *variability (46)* and the *problem space (50)*: a) to communicate variable characteristics between *stakeholders (42)* and software engineers, and b) to document variability in the form of abstract *requirements (35)*. In practice, product line requirements are often captured in feature models. A feature model is the feature profile of the products in a product line, hierarchically organizing the products' end-user visible characteristics. Its goal is to document end-user requirements, not active variability management. Active variability management is usually done through *variability assets (57)*. Note that the feature term is also used for describing artifacts in the *solution space (51)*, but that concept is not in the scope of the current definition.

This definition is reflected in Fig.14, where features, as part of feature models that represent requirements core assets, serve as inputs to the architecting and realization phases shown in Fig.17. A more detailed classification of features in product lines is presented in [Lee+04, Lee+10]. The reasons why feature models are insufficient for managing variability are discussed in detail in [Bühne++04, Berg++05, Pohl++05, Metzger++07].

The variability that is specified, for example, in feature models, is realized in product line infrastructure code, or more generally in core assets, by variability mechanisms.

Definition 64: Variability Mechanism

A variability mechanism [Jacobson++97] is a particular way of intentionally realizing *variability (46)* in *core assets (56)*. The purpose of variability mechanisms is to balance *reuse (21)* effort and *evolution (66)* effort by efficiently organizing common elements and *variants (53)*, as appropriate in the particular *context (20)* of *product line engineering (49)*. Synonym: Variation Mechanism [Krueger04, Bachmann+05].

In contrast to older definitions [Jacobson++97, Muthig+03], variability mechanisms are nowadays not seen as restricted to source code only [Bachmann+05, Clements+06]. All variation points in core assets have a variability mechanism-specific representation, for example as a tag, a comment, an *#ifdef* statement, or a function call. Variability mechanisms cover a wide spectrum of automation, they "can be as simple as an empty block in source code that an application developer must fill in, or as complex as a translation system that generates source code from high-level requirement specifications" [Krueger04]. Note that in a product line engineering context, the usage, not the structures, of a technique qualifies it as a variability mechanism, so that primitive techniques, applied predictively and intentionally, are regarded as variability mechanisms, whereas advanced techniques, used without consideration, are not seen as variability mechanisms (see also the discussions on [Krueger04, Krueger07] in Sec.3.2).

In analogy to the development context definition (Def.20), the above mentioned product line engineering context denotes the setting in which product line engineering, and in particular family engineering, is practiced. The goals of variability mechanisms in family engineering are to balance reuse effort and evolution effort. Bassett identified three prerequisites for effective reuse, from the perspective of development:

- "Effective reuse highlights novelty - makes exceptions easy to see and control - while hiding what is routinely the same" [Bassett97, p.87].
- "Effective reuse involves components of all size scales, down to bits" [Bassett97, p.77],
- "Effective reuse involves construction-time variability that is sensitive to differing contexts of use" [Bassett97, p.78].

I have refined these points into realization tactics for product line evolution, inspired by the concept of a tactic in software architecture, which is "a design decision that influences the control of a quality attribute response" [Bass++03]. Table 2 gives an overview of the tactics used for classifying variability mechanisms (Ch.4) in the remainder of this

thesis. The set of tactics is not meant to be complete, but sufficiently orthogonal. It is meant to be customized in a particular development context, according to the most relevant quality attributes there.

Tactic	Rationale	Values
Increase VP explicitness	Increasing the visibility of variation points makes variants easier to detect and product line assets easier to evolve.	explicit, implicit, ambiguous
Allow appropriate variant granularities	Associated variant elements of different sizes should be realized by mechanisms that support a corresponding spectrum.	wide range, narrow range
Limit late binding	Later binding times lead to less degree of freedom for realizing variants.	constr. time, exec. time, runtime
Isolate variants	Separating common and variant modules allows them to evolve independently.	open, closed
Provide automated production	Automation reduces application engineering effort.	automatic, manual
Provide defaults	Defaults reduce the number of variants	default support, no default support

Table 2: Tactics for effective family realization

The first tactic, Explicitness Increase, helps a family engineer detect all relevant variation points in a core asset quickly and unambiguously, possibly without the availability of variants or defaults. Many variability mechanisms result in non-explicit or ambiguous variation points which are harder to detect than necessary in this family engineering context.

A second tactic for a family engineer is to allow the variants to cover the required size scales, but not more. Variants may be small, for example realized by single characters or lines in a textual core asset, they may be medium-sized, for example fragments of data structures or functions in a syntax tree, or they may be large, such as subsystems which are realized as entire sub-directories. The tactic is to allow appropriate granularity, so that for example a small variant element can be managed together with its associated large variant element, using the same wide-range variability mechanism. On the other hand, a narrow-range mechanism that only supports medium-sized variation would be inappropriate in this context.

The third tactic, Limit Late Binding, allows a family engineer to vary arbitrary core asset elements, especially code elements, regardless of the meaning of these elements. As discussed in Section 2.1 and [Bassett97], the family engineer has a higher degree of freedom in creating variants if he may neglect programming language semantics (i.e., if he applies reuse, and not just use) which is only possible in early binding. For example, he may make part of an algorithm optional if he uses a construction time mechanism such as an *#ifdef* statement, without further refactoring effort. Conversely, an execution time or runtime mechanism that enforces the variant to be a function will require

additional refactoring effort for the variant candidate. The corresponding variability mechanisms are either construction time, execution time or runtime mechanisms. Note that according to definitions 31, 15 and 16, construction time is that binding time during which the properties of executable modules that are invariant at execution time or runtime can still be changed, whereas only the properties of runnable modules can still be changed at execution time, "when a module can be interpreted by a compiler-linker-computer trio, or by any functionally equivalent interpreter" [Bassett97].

A fourth tactic, Isolate Variants, means to keep variants independent of each other, and independent of common elements, so that they may evolve in isolation. In particular, alternative variants should be appropriately decoupled from each other to facilitate their co-evolution. The tactic is associated with open or closed variation points [Gurp++01, Svahnberg++05] whose set of variants may be increased without changing existing artifacts. The corresponding variability mechanisms are either open or closed. Independent of a particular engineering activity, open and closed variability mechanisms have also been called selection and substitution [Becker00], selection and new definition [Kim++05], adaptation and replacement [Linden++07, pp.40ff.], or annotation and composition [Apel++09].

The fifth tactic for balancing reuse effort and change effort, beyond family engineering, is to automate production. The corresponding variability mechanisms are either automatic or manual. The sixth tactic is to provide defaults, in order to reduce the number of variants.

The given tactics were inspired, and subsume some of the architectural tactics for achieving variability [Bass++04] which are

- Limit options,
- Isolate the expected changes,
- Raise the abstraction level,
- Maintain semantic coherence,
- Abstract common services,
- Hide information,
- Maintain existing interfaces,
- Separate the interface from the implementation,
- Use an intermediary, and
- Limit communication paths.

In my approach, the five tactics are meant to be applied in a particular family engineering context, as to avoid complexity excess. As introduced in the beginning of this section, complexity in general (Def.43) means to have unneeded elements in artifacts, and in product line engineering these artifacts form a product line infrastructure. In this thesis, variability-related complexity is called variability complexity.

Definition 65: Variability Complexity

Variability complexity is the absence of *variability (46)*-related simplicity in a *product line infrastructure (62)* or product line engineering process. This defect makes the product line infrastructure more difficult to *evolve (66)* than necessary. It arises when variability-related elements have been realized in *family engineering (60)* that are not immediately required by *application engineering (61)*. Variability complexity reduction aims at making the product line infrastructure easier to evolve, especially the *variants (53)* within the *core assets (56)*. See also: *Complexity (43)*.

Variability complexity may occur for two reasons: On the one hand, it arises due to over-complexities in previous family engineering sub-phases, for example when architecting has defined unnecessary variation points which are then realized (Variability Mismatch [Deelstra03]). On the other hand, variability complexity arises because of improper use of variability mechanisms [Bosch++02]. This is the focus of the current thesis.

The evolution concept used in my variability complexity definition (Def.65) has the same meaning as in Shaw's definition of development (Def.12). In contrast to maintenance which is concerned with the elimination of defects, the purpose of evolution is the adaptation of existing systems due to new requirements [Endres+03, p.160; Eden+06]. Maintenance as a separate concept, analogous to the repair of physical systems, is increasingly considered obsolete [Neighbors80, Bassett97, Lapham06, Godfrey+08], as well as the traditional distinction [Rajlich+00] between initial development and evolution [Lehman02, Sommerville04, p.82; Boehm10, Kirby++10]. More details on quality models for evolution can be found in [Breivold++08, Brcina++09].

My definition of evolution subsumes the above mentioned ideas, and other definitions proposed in the software engineering and product line engineering literature, where evolution, or sustainment, is seen as

- "the repair, adaptation, and enhancement of a software system" [Neighbors80],
- "the life of the software after its initial development cycle" [Jacobson++97],
- "adding to and improving a product or product line over time" [Svahnberg03],
- "the change of software artifacts over time" [Ommering04],
- "variability in time" [Pohl++05],
- "advancements to a product family" [Hotz++06],
- "the changes performed to any asset or a set of them with respect to time, including expectations for future changes" [Käkölä+06],

- "the processes, procedures, people, materiel, and information required to support, maintain, and operate the software aspects of a system" [Lapham06],
- "a process of progressive, for example beneficial, change" [Lehman+06b],
- "change that is guided and constrained by rules and policies that allow local needs to be satisfied in local ways without destroying the integrity and value of the overall system" [Northrop++06],
- "modifying or replacing a product due to changing needs or technology" [Campbell07],
- "activities performed to ensure that a product or service remains operational" [ISO24765].

In the current thesis, evolution is defined as follows:

Definition 66: Evolution

Evolution [Lehman80, Lehman02] is the sub-activity of *development (12)* during which changes occur in the *problem space (50)* over an extended period of time which lead to changes in real-world *artifacts (44)* in the *solution space (51)*. The goals of evolution are to explicitly address long-term issues, such as unpredicted changes, but "it is also appropriate to apply the term evolution when long-term change trends are beneficial even though isolated sequences of changes may appear degenerative" [Lehman+06b]. Synonyms: Sustainment, Variability in Time, Reuse across Time [Bassett97]. Contrast with: Decay [Mens+08]. See also: Evolvability [Breivold09, Mäntylä09], Sustainability [Wirfs-Brock09, Lutz++10], *Variability Evolution* (67).

I call variability-specific evolution in product lines variability evolution.

Definition 67: Variability Evolution

Variability evolution is the sub-activity of *product line engineering (49)* during which changes occur in the *problem space (50)* over an extended period of time, for example changing *needs (24)* or technology, which lead to changes in *solution space (51)* artifacts of the *product line infrastructure (62)*. The goals of variability evolution are to explicitly address long-term issues, such as unpredicted changes in the variability of the product line. See also: *Evolution (66)*.

Efficient evolution is not achieved abruptly, but in incremental steps. Eden and Mens defined the evolution step concept as follows:

"Let us represent the set of problems as P, the set of solutions as S. A step in the process of system evolution can be represented as a mapping of the combination of the old problem $p_{old} \in P$, the evolved problem

$p_{new} \in P$, and the old solution $s_{old} \in S$ into the evolved solution $s_{new} \in S$. This mapping can thus be represented as the evolution function, a mathematical function E which maps each tuple $\langle p_{old}, p_{new}, s_{old} \rangle$ to the evolved solution s_{new}. Evolution step is a pair $\varepsilon = \langle \langle p_{old}, p_{new}, s_{old} \rangle, E(p_{old}, p_{new}, s_{old}) \rangle$" (adapted from [Eden+06]).

Evolution steps are defined for both types of evolution:

Definition 68: Evolution Step

An evolution step is a smaller sequence of changes during the larger *evolution (66)* or *variability evolution* (67) of a system. The purpose of evolution steps is to break down the evolution activity into more manageable sub-activities that keep the evolving artifact maximally stable. Synonym: Minimally Invasive Transition [Krueger10].

Consistent with my definition of evolution steps, the concept of minimally invasive transitions has been characterized this way:

"The software product line development methodology of minimally invasive transitions is distinguished by its focus minimizing the cost, time and effort required for organizations to adopt software product line practice. A key characteristic of this methodology is the minimal disruption of ongoing production schedules during the transition from conventional product-centric development practice. Minimally invasive transitions take advantage of existing software assets and rely on incremental adoption strategies" [Krueger10].

As this thesis focuses on evolution in product line engineering, and in particular in family engineering, the investigated evolution steps are only concerned with what primarily distinguishes family engineering from single systems engineering: variability ("The primary distinction between software product line engineering and conventional software engineering is the presence of variation in some of the software artifacts" [Krueger04]). As inputs to the architecting and realization sub-processes (Fig.17), features represent differences in user requirements (variability in space, Def.46) which evolve over time (variability in time, Def.66) [Savolainen+01]. As will be discussed in more detail in Section 5.1, the three basic evolution steps are addition, removal and change. Consequently, the considered evolution scenarios will only consist of evolution steps that are caused by the need to add, remove, or change features that vary in space. The same observation has been made by Elsner et al.: "As requirements change over time, the product line must evolve as well. For a product line this means adding, removing, or changing features, as well as adding, removing, or changing variability dependencies (e.g., mandatory, optional, alternative)" [Elsner++10, p.132].

Evolution leads to changes in some system properties that are relevant to stakeholders, users (Def.8), or reusers (Def.22) of the system, while keeping other system properties invariant. For example, evolution may affect the functionality or efficiency of the system, which are system properties relevant to stakeholders. It may also make system artifacts easier to understand and change, which are properties relevant to the software engineers themselves or other users of the system. At the same time, other properties are not changed, or their change is not relevant to any of the above mentioned parties.

The presence of variability is the only property that distinguishes product line infrastructures from single system artifacts, or from executable modules (Def.4). Variability is relevant to both types of product line engineers, but family engineers are the only engineers that change variability properties in product line infrastructures during evolution (which is why the current thesis focuses on family engineering). Two types of such family engineering-specific changes can be distinguished: those that alter variability properties, and those that preserve them, while improving other properties of reuse, such as adaptability [Bassett97]. The latter type of change is called a variability refactoring in this thesis.

Definition 69: Variability Refactoring

Variability refactoring is a specific *family engineering (60)* activity by which a *product line infrastructure (62)* is changed in order to *evolve (66)* or *reuse (21)* it in a more cost-effective way. The distinction between conventional and variability refactorings is that conventional refactorings make existing *artifacts (44)* easier to *use (6)*, preserving their functionality, while variability refactorings make existing *product line assets (59)* easier to reuse, preserving not only the functionalities of all *product line members (48)*, but preferably also keeping their *executable modules (4)* invariant.

In other words, for a conventionally refactored system, if its executable modules are passed through an execution interpreter twice (before and after refactoring), the resulting machine code will be the same (see Fig.6). For a variability refactored product line infrastructure, if its constructible modules are passed through a construction interpreter twice (before and after refactoring) the resulting executable modules will be the same (see Fig.8). Note that the replacement of a construction interpreter and a corresponding change of the constructible modules is also a variability refactoring. Also note that both conventional and variability refactorings may be performed together in family engineering. Section 5.2 presents variability refactorings in more detail.

3 Related Work

The approach for sustainable evolution of product line infrastructure developed in this thesis uses, combines and extends existing research from five mostly orthogonal areas, discussed in this chapter:

- reusable code artifacts (Sec.3.1),
- product line engineering processes (Sec.3.2),
- usefulness of code duplication (Sec.3.3),
- complexity and evolution in single systems (Sec.3.4), and
- complexity and evolution in product lines (Sec.3.5).

Section 3.1 presents research which has focused on code artifacts that may realize product lines. Some work in this area has listed various collections of variability mechanisms, while other work has propagated a single mechanism or a particular combination of two mechanisms as a universal solution for realizing product lines. All of this work has focused on the end-result only. The engineers creating or using these artifacts are out of scope. Although this thesis also presents a set of variability mechanisms, these are not just seen as isolated solutions, but they are discussed in a deeper engineering context, as inputs to a product line realization process. This is also why they are presented from the perspective of a family engineer.

Section 3.2 shows research which is related to this thesis because it considers the larger line engineering process which results in artifacts such as product line infrastructure code. Much work in this area has focused on the problem space and earlier solution space activities such as architecting and designing, often under the assumption that the product line is introduced proactively, i.e. in a context in which a new product line is created from scratch. In that work, product line realization and the resulting artifacts are usually out of scope. More reactive work in this area exists that addresses incremental transition strategies. However, that work does not consider in detail how family engineering could employ existing mechanisms for "good enough" variability management.

Section 3.3 presents recent studies that have challenged the software engineering myth that code duplication, or cloning, is universally harmful. A growing number of empirical studies are demonstrating that cloning can be an effective engineering tool in some contexts, and this thesis explores for the first time if there are contexts in product line

engineering, and particularly in family engineering, in which cloning can be tolerated or used as a desirable evolution strategy.

Section 3.4 shows previous work concerned with describing and controlling complexity in evolving system artifacts in general, and software artifacts in particular. Empirical research in the evolution of large industrial software has led to the formulation of rules for software sustainment. It is described that a certain type of complexity, also found in software artifacts, has been found to be a major cause for lack of effective system evolution. This thesis explores the goal of keeping artifacts simple for elements of a software product line infrastructure, in particular its code.

Section 3.5 presents other work that has started to explore product line evolution and measurement. In contrast to this thesis, that evolution work is mainly concerned with earlier activities than realization in the product line engineering life cycle. That work also does not consider the connection to product line-specific complexity. Unlike this thesis, but similar to the solution-oriented work describing reusable code artifacts (Sec.3.1), existing work on product line measurement has not yet investigated it in a goal-oriented context to actively support product line infrastructure evolution in an organization-specific context.

3.1 Reusable Code Artifacts

In order to develop a sustainable product line infrastructure, as depicted in Figure 1, an organization must acquire product line engineering capabilities across all engineering life cycles which may not have existed in the organization's previous single system development. According to the 3-tiered methodology for reactive product line adoption [Krueger07], the base capability that provides most immediate benefits in this context is the ability of software engineers to manage variation in the organization's existing artifacts, such as design or code. In previous work, we have identified several dimensions of product line realization technologies, especially configuration management and programming language/generator techniques [Muthig+03]. While branching and configuration management in a product line context has been covered elsewhere [Parnas76, Krueger02a, Anastasopoulos++09], this thesis only addresses the code dimension.

Previous work covering this dimension often focused on the solution only, either presenting collections of reuse techniques or variability mechanisms, or propagating a single or a hybrid mechanism only. For reasons of space, the discussion in this section is restricted to collections of variability mechanisms. The conclusions also apply to solution-oriented work on a single mechanism or combinations (e.g. Aspect-Orientation [Kiczales++97, Tarr++99], Frame Technology [Bassett87, Wong++01,

Sauer02], VSL [Becker04], Collaborations [Smaragdakis+02], Feature-Orientation [Batory+04, Trujillo07], Aspect-Orientation and Frame Technology [Loughran+04], Feature-Orientation and Aspect-Orientation [Apel07], Change-Orientation [Ebraert09], Conditional Compilation [Kästner10], Feature-Orientation and Frame Technology [Zhang+10]).

Krueger's early work on software reuse [Krueger92] has compared seven approaches for reusing software artifacts of different life cycle stages, including code. The approaches include high-level languages, code scavenging, source code components or application generators. Code scavenging, source code components and application generators roughly correspond to the three mechanisms of Cloning, Module Replacement and Frame Technology presented in this thesis. However, variability management was not yet in scope of Krueger's mechanisms, and the goal of discussing the mechanisms in this thesis is not to present reuse techniques across all engineering life cycles, but to provide a language-independent toolset for engineers during the family realization process.

The reuse book by Jacobsen et al. [Jacobson++97] was one of the first publications that explicitly illustrated the concept of variability mechanisms in reusable artifacts. Their list of mechanisms includes Inheritance, Uses, Extension and Extension Points, Parameterization, Configuration and Module-Interconnection Languages, Generation, and Template Parameters. For each mechanism, a brief recommendation is given. As the authors admit themselves, their list is incomplete and unorganized. In contrast, the mechanisms presented in this thesis form a complete and disjoint set, according to the criteria introduced in Section 2.3. Two of the mechanisms listed by Jacobson et al., Inheritance and Template Parameters, are subsumed by a mechanism suggested in this thesis (Polymorphism), the others are out of scope, as they mostly address specific design issues.

In the context of Generative Programming, Czarnecki and Eisenecker [Czarnecki+00] gave a detailed overview of various academic techniques developed in the 90s that could be used for realizing solution space artifacts, especially code. These include C++-specific template techniques, different flavors of polymorphism, Aspect-Orientation, collaboration-based approaches, and Intentional Programming. Although Aspect-Orientation and Polymorphism are also included in the list of mechanisms in this thesis, the goal of presenting these mechanisms is fundamentally different: It is not to give a broad overview of coding solutions using a variety of tools, or automatically map problem space concepts to the solution space, but instead to provide a compact, deliberately limited set of language-independent approaches that can rapidly be applied by average software developers in real-world development contexts for incrementally simplifying variability management. For that reason, the focus is not on solution-space

artifacts, but instead on the holistic process of continually evolving the entire product line infrastructure, as discussed in the next chapters.

Various other solution- and technology-focused publications have since presented variability management solutions for source code. For example, Anastasopoulos and Gacek [Anastasopoulos+01] have suggested Aggregation, Aspect-Orientation, Conditional Compilation, Dynamic Class Loading, Dynamic Link Libraries, Frames, Inheritance, Overloading, Parameterization, Delphi Properties, and Static Libraries, discussed their mapping to programming languages, and compared them according to interface, realization, initialization, timing, and other criteria. Especially the sub-criteria they call Scalability and Traceability will also be evaluated in more detail in this thesis, as Granularity and Explicitness (cf. Tab.2). Whereas that work only gives a 3-staged ranking whether it is possible, difficult or impossible in general to satisfy a particular criterion, this thesis evaluates if it is necessary to apply a certain mechanism or class of mechanisms in a particular development context, or if a simpler approach exists that would also suffice. These considerations have not yet been completely in the scope of our early work [Muthig+03] which ranked Conditional Compilation, Subtype Polymorphism, Parametric Polymorphism, Ad-hoc Polymorphism, Collaborations, Aspect-Orientation, and Frame Technology according to their support for product line realization. However, that work was already not completely solution-focused, but considered these mechanisms as part of a larger product line adoption process (see related work in Sec.3.2).

The variability management survey by Myllymäki [Myllymäki01] includes the mechanisms Aggregation, Inheritance, Parameterization, Overloading, Macros, Conditional Compilation, Configuration, Generation, Static Libraries, Dynamic Class Loading, Dynamic Link Libraries, Reflection, and Patterns, and gives some examples of their usage. However, in contrast to this thesis, no criteria are given why each mechanism has been included, and consequences of using them in real-world software development are not discussed. Bachmann and Clements published a similar collection of mechanisms [Bachmann+05] which includes Inheritance, Component Substitution, Plug-ins, Templates, Parameters, Generator, Aspects, Runtime conditionals, and Configurator, and lists their cost and prerequisites both in family and application engineering. Inheritance, Templates and Plug-ins map to Polymorphism described in this thesis, Component Substitution is the same as Module Replacement, Aspects correspond to Aspect-Orientation, and Runtime conditionals denote Conditional Execution. This thesis focuses on family engineering, for which Bachmann and Clements only make rather vague observations.

Other work has also discussed programming language-specific collections of variability mechanisms. For the Java language, Hunt

[Hunt06] has proposed Parameterization, Inheritance, Java Language Interface, Aspects, and XVCL, while Alves [Alves07] has discussed Frameworks, FOP, JPEL, AOP, JaTS, XVCL, and Conditional Compilation. Again, these publications do not discuss if the presented solutions are orthogonal, or if they scale. These and most other publications mentioned in this section have in common that they just characterize, and sometimes compare, what is possible as a solution, from a general, purely technical perspective, such as binding time or separation of concerns. They do not consider what is meaningful to limit reuse complexity in a real-world development situation, especially when code already exists that has to be integrated rapidly and seamlessly into a product line by average developers, using rapidly available methods and tools. For that reason, hardly any of the mentioned publications so far has included two of the most frequently used mechanisms in practice, Conditional Execution and Cloning (also see Section 3.3).

The product line engineering book of van der Linden et al. lists a set of three product line architecture mechanisms, with six associated variation mechanisms [Linden++07] which comprise Inheritance, Patching, Configuration, Code Generation, Component Replacement, and Plug-ins. Although concrete code details of these mechanisms are missing, some of them can be mapped to the mechanisms shown in this thesis: the Component Replacement mechanism corresponds to Module Replacement in this thesis, and the Plug-in mechanism can be realized by Polymorphism.

3.2 Product Line Engineering Processes

Unlike most work discussed in Section 2.1, the product line engineering literature is not only concerned with describing product line artifacts, but it also takes the corresponding processes into consideration which produce and consume these artifacts. Domain engineering approaches such as FODA and FORM [Kang++90, Kang++98], or Synthesis [Campbell++90, Synthesis93] have had a strong process focus which also heavily influenced their successors, product line engineering methods such as PuLSE [Bayer++99], KobrA [Atkinson++01], and others [Weiss+99, Pohl++05, Krueger07, Linden++07, Northrop++07]. However, none of these approaches provide detailed realization and explicit evolution processes, such as those developed in this thesis. For example, the Synthesis method describes the realization of reusable assets mainly in terms of activities, sub-activities and interactions which result in reusable artifacts, while presenting artifacts in less detail. All mentioned product line engineering methods have usually been concerned with both processes and artifacts. For example, PuLSE [Bayer++99] both presents processes for developing a product line infrastructure, and artifacts, such as product maps or decision models, which are created in these processes. The Framework for Product Line

Practice [Northrop++07] gives an overview of product line concepts and practice areas, but does not address realization and evolution. The product line book by Pohl et al. is another example [Pohl++05]. It describes both how variability is documented in the artifacts at different life cycle stages (part 2 of the book), and the corresponding engineering processes which produce them (part 3) and consume them (part 4). Although that book gives a detailed overview of product line engineering, code artifacts are not within the scope of that work (see [Pohl++05, p.136] which refers to our early work [Muthig+03] concerning realization technology). The current thesis sees all artifacts as end-products of processes which develop or evolve them because this mode of description has been identified as necessary for controlling complexity during evolution [Alexander02; Pressman10, pp.60f.]. As mentioned before, the current thesis also refines the quality-assurance processes and artifacts discussed in [Pohl++05] by explicitly applying quality assurance in all life cycle stages, by introducing product line measurement, and by adding a construction testing phase (cf. Figs.17, 41, and 42).

As indicated in Section 3.1, our early work [Muthig+03] described different types of product line artifacts, but already considered them as part of a larger development process. Later work [Patzke+03] refined this approach with a focus on Frame Technology, comparing different scenarios for evolving existing single system code into product line infrastructure code using conventional object-oriented mechanisms versus Frame Technology. Whereas a conventional artifact description was shown for the conventional solution, the Frame Technology solution was described both by a list of ordered activities and the resulting artifacts. We also used process descriptions when documenting refactoring activities of industrial single systems code into product line infrastructure code [Patzke+04, Kolb++06]. More recent work has described an incremental approach to improve variability management capability in practice by augmenting existing technology [Patzke07]. That paper shows how liabilities of Conditional Compilation can be counteracted by augmenting the C preprocessor, in that case study the cpp preprocessor front-end of the GNU gcc compiler, with basic frame technology capabilities. A technology-independent transition path is presented which introduces reuse hierarchies and explicit variation points into evolving artifacts, and an example of embedded systems code written in C is given.

Svahnberg et al. [Svahnberg++05] have suggested a process for introducing variability into product line artifacts. The process consists of the four steps Identify Variability, Constrain Variability, Implement Variability, and Manage Variability. The first process step corresponds to the Selection activity proposed in this thesis, while the other three steps are sub-steps of the Modification activity (see Sec.5.2). An important element of my approach is the Quality Assurance step which is missing in

their approach. As inputs to the four activities, Svahnberg et al. present thirteen variability realization techniques which cover different stages of the software engineering life cycle. Three of these techniques are related to realization: Condition on Constant, Condition on Variable and Code Fragment Superimposition. These roughly correspond to the mechanisms Conditional Execution, Conditional Compilation and Aspect-Orientation in this thesis, while the two Binary Replacement design mechanisms are subsumed by Module Replacement in this thesis.

The work of Weiss et al. [Weiss+99] and Coplien [Coplien99] also includes both process and artifact descriptions. The proactive development of a weather station product line is illustrated in [Weiss++99, Ch.5]. That example shows the successive activities in the requirements, architecting and realization life cycles, plus the resulting artifacts, such as the code of a specific code generator for that product line which is written in Perl and which produces Java code. In contrast, this thesis focuses on the realization activity and its artifacts independent of a particular programming language, and at the same time it concentrates on evolution activities beyond initial development.

Multi-Paradigm Design [Coplien99] is a method for designing and realizing common and variable code according to a certain process. In this approach, the problem and solution areas are first analyzed independently, and then both results are mapped. Each analysis is performed separately on commonalities and variabilities. The problem analysis is concerned with the design of the application domain. The solution analysis covers the variability management possibilities of the applied programming language. The solution domain analysis is illustrated for the C++ language by presenting commonality and variability analysis tables that recommend *#ifdef*s (Conditional Compilation) for fine-grained algorithmic variation at compile time, and virtual functions (Polymorphism) for algorithmic variation at runtime. The approach developed in the current thesis is similar; it also suggests a practical developer-oriented process that aims at context-specific selection of solution mechanisms. However, this thesis takes a more general view, with a problem space that consists of new product line requirements, existing code and knowledge, a set of customizable realization tactics, and a solution space of groups of plain and programming language-independent types of variability mechanisms. Other differences are the inclusion of quality assurance processes, the focus on complexity reduction, and the restriction to family engineering activities.

Iterative Design Refinement (IDR) [Bassett97] is a reuse method which explicitly attacks the component-based software development model which treats software artifacts like fixed hardware blocks. In the IDR method (Fig.18), single-system requirements and software engineering standards are inputs to a co-dependent pair of life cycle processes. The

Domain Analysis and Meta Component Design and Development pair evolves core assets, while the pair Domain Analysis and System Design and Development evolves the product line members. The process is iterative and contains feedback loops. The development processes produce and consume core assets, realized by frames using the Frame Technology mechanism.

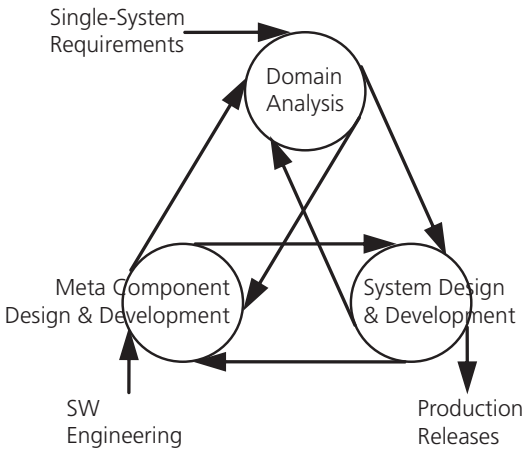


Figure 18: Iterative Design Refinement [Bassett97]

A stepwise process for creating and organizing these frames is presented [Bassett97, pp.170ff.] which consists of the activities Match to Existing Use-as-is Parts, Match to Existing Same-as-except Parts, Match to Other Behavioral Archetypes' Parts, Frame the Most Reusable Pieces First, Normalize, and Frame Context-Related Deltas. The fourth process step explicitly involves metrics for identifying most reusable assets, and similar metrics (LOC, LOC_{ad}) are also used in the measurement process developed in this thesis in order to reduce code size. An evolution process is also suggested [Bassett97, p.212, pp.223ff.], covering the change of reusable artifacts over time, which is reflected in the product line evolution scenarios presented in this thesis. The IDR method is similar to the method developed in this thesis because it is also concerned with the incremental evolution of code artifacts. However, my approach does not suggest frame technology as a solution in advance, but delays the engineering decision for applying a particular mechanism until an informed decision can be made. This is also the reason why Cloning is accepted as a variability mechanism as long as the development context permits it.

As observed by Krueger [Krueger10], traditional product line engineering methods have often been proactive (waterfall-like), whereas new product line methods predominantly needed in practice require more reactive (agile) adoption strategies (discussions of the different product line adoption models can be found in [Krueger02a, Krueger02b], and case studies of reactive adoption approaches are presented in [Buhrdorf++04, Kolb++06]). The above mentioned iterative approaches

by Bassett and Coplien [Bassett97, Coplien99] already contain some reactive ingredients, for example by prioritizing which reusable elements to extract [Bassett97, p.170], postponing this decision until a need is proven, or by mostly focusing on the realization phase [Coplien99]. Explicitly non-reactive product line approaches that speed up development are demanded by industry [Kolb+10]; their development has already started [Krueger02a, Muthig02, Krueger04, Krueger07, Patzke07, Codenie++10, Krueger10].

The need for simplified, non-proactive transition strategies for mass customization is well-known issue in product line engineering research [Krueger02a, Krueger02b]. It was shown that there is often an adoption barrier to traditional product line engineering approaches, in which a new product line infrastructure is proactively engineered from scratch, requiring substantial transition time and effort. In many real-world development contexts, however, these delays are unacceptable because ongoing evolution of existing products cannot be delayed. Mass customization technology is presented which may be used in both proactive and non-proactive product line adoption. Non-proactive approaches are either reactive or extractive. In a reactive approach, the product line infrastructure is grown incrementally, as the need arises for new products. In between the proactive and the reactive approach, the extractive approach reuses one or more existing systems by extracting their common and variable elements into a single product line infrastructure. Likewise, the product line evolution method developed in this thesis is customizable to each of the three adoption models. However, as in [Buhrdorf++04], the focus is more on non-proactive approaches in which existing artifacts must be evolved.

Krueger has also presented a product line taxonomy [Krueger04] which describes different dimensions of product line concepts, solutions, and processes, for example binding and binding times, variation across space and time, production artifacts and sub-processes, product line evolution sub-activities, and adoption approaches. Transition scenarios are characterized by having an initial state which is void in the case of proactive approaches, and a target state. The conclusion is "that if the initial state has multiple products using even the most primitive, ad hoc, conventional techniques such as clone-and-own or IFDEFs, it is still a product line that can be characterized using the taxonomy" [Krueger04, p.331]. This thesis builds on these ideas, as it considers any mechanism as a variability mechanism if it results in multiple sufficiently similar products that are evolved together, using a common process.

This idea is also supported by a publication that develops a 3-tiered methodology for introducing software product lines in practice [Krueger07] whose elements are depicted in Figure 19.

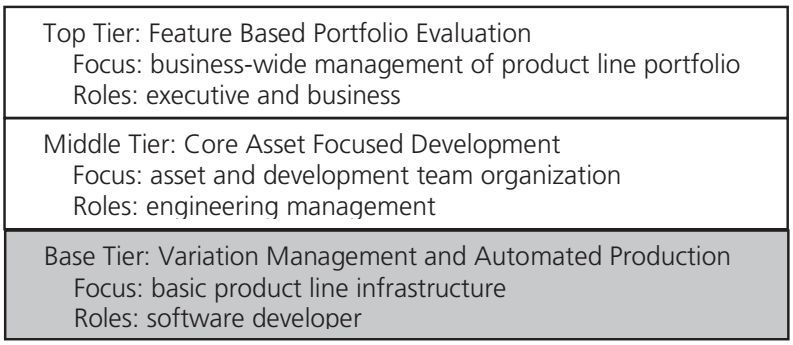


Figure 19: The 3-tiered product line methodology [Krueger07]

Each tier expresses capabilities and benefits of transitioning from product-centric to product line development. The higher tiers depend on capabilities of lower tiers, so that an incremental transition strategy starts at the base tier which is called Variation Management and Automated Production. It is concerned with the capabilities for setting up a product line infrastructure in architecture, design and source code artifacts. The base tier is concerned with software developers. As highlighted in Fig.19, this tier is addressed by the current thesis. Base tier problems include inconsistent variability management, for example large-scale duplication and configuration management branches, or unsystematic usage of multiple home-grown variability management techniques, such as *#ifdefs* without clear naming conventions, controlled by non-localized compiler flags, mixing of application and variation logic, controlled by dispersed configuration options in configuration files, non-volatile memory, and databases, or custom build and installer scripts, file or configuration management conventions. This accidental usage of mechanisms is addressed by the current thesis which aims at teaching software engineers to incrementally improve their variability management habits. Standard definitions of product lines from major publications [Weiss+99, Pohl++05, Northrop++07] do not exclude any mechanism, as long as it is used predictively and intentionally, in order to capitalize on commonality and clearly managing the variation (cf. the goal of variability in [Bachmann+05]). This is also reflected in Krueger's software asset definition ("any legacy source code or other software asset [that] can serve as a core software asset […] so long as it consolidates commonality, contains zero or more variation points, and can be used […] to instantiate products" [Krueger07, p.101]) which is consistent with the core asset definition (Def.56) used in this thesis.
The other two tiers are out of scope of this thesis. The middle tier, Core Asset Focused Development, addresses engineering management, focusing on the organization of assets and development teams around a product line infrastructure. The top tier, Feature Based Portfolio Evolution, is concerned with business-wide management of the entire product line portfolio, addressing executive and business personnel.

Three types of best practices have recently been presented that are recurring in new product line initiatives in practice [Krueger10], and which have also influenced this thesis: software mass customization, minimally invasive transitions, and bounded combinatorics. The software mass customization process (see Def.29), invented in the manufacturing of physical systems, and later adopted as a software engineering process supporting reuse in practice [Bassett97, Krueger02a], composes and configures existing product line artifacts of a product line infrastructure in different ways to yield the required products. As in this thesis, the focus is on family engineering, not on application engineering. The paper even aims at an integrated engineering approach, as application engineering as a separate process is considered harmful. The new integrated approach better supports evolution, taking into account both variability in space and in time because real-world development does not just require the latest products to be evolved together, but also to evolve older product versions, sometimes by back-propagating newer changes, and sometimes deliberately ignoring them. These evolution issues are also in the scope of this thesis, for example as part of the development context which sometimes leads to a deliberate short-term evolution step that may appear degenerative (see Def.66 or Commonality Realization in Sec.5.2). Minimally invasive transitions denote that evolution proceeds as smoothly as possible, not altering more than necessary. In other words, it aims at simple steps that lead to complexity reduction which is a major concern of this thesis (see also the related work in Sec.3.4 and 3.5). Bounded combinatorics has a similar intention, especially in a product line engineering context, because it deliberately aims at developing a product line infrastructure that only supports the required products. This means that the goal of a good product line is not to support the maximum combinations of features, but to limit the combinations to the required ones only. This is one example of product line-specific complexity reduction, a major topic addressed by this thesis.

Another non-proactive product line engineering method which heavily influenced this thesis is the lightweight product line transition method developed by Muthig [Muthig02]. It covers all stages of the product line engineering life cycle, but focuses in particular on the architecture and design processes and the associated artifacts of the product line infrastructure. The method explicitly works out the difference between single-system and product line development by focusing on variability-related activities and their differences to single-system activities, as in the current thesis. Like this thesis, the lightweight method distinguishes between artifact descriptions and process descriptions, but unlike this thesis, it does not consider to exclusively use process descriptions for documenting the evolution trace of artifacts. The lightweight method presents a metamodel for product line infrastructures (Fig.20), whose concepts are refined in this thesis with regard to the use-reuse duality (Sec.2.1), reuse hierarchies (Sec.2.2), and complexity and evolution considerations (Sec.2.3). It mainly consists of a metamodel for core

assets (called product line assets in [Muthig02]) and a metamodel for decision models. The product line infrastructure metamodel developed in the current thesis (Fig.15) is based on that metamodel, but extends it by engineering processes and variability assets.

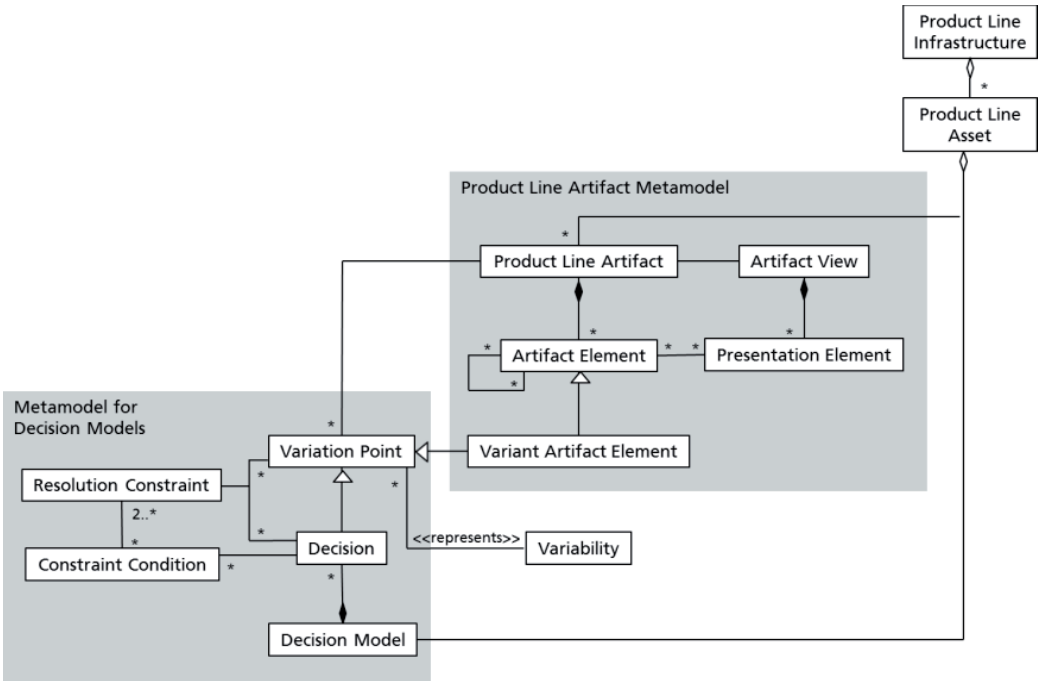


Figure 20: Metamodel for product line infrastructures [Muthig02]

The second main element of the lightweight product line method is a description of the method's process activities which cover the four sub-processes Initialization, Incremental Product Line Modeling (Incremental Family Engineering), Evolution and Management, and Application Engineering. Figure 21 gives an overview of the Incremental Product Line Modeling process which consists of the sub-activities Commonality Modeling, followed by Variability Identification and Variability Modeling.

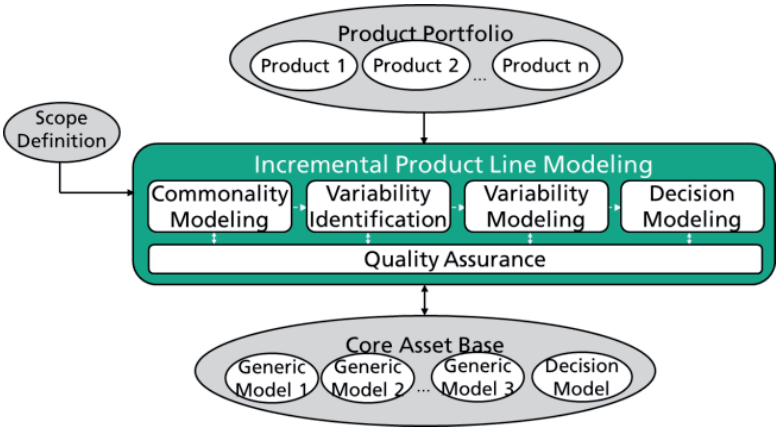


Figure 21: Incremental Product Line Modeling sub-process [Muthig02]

Similar to the separation of the two variability-related processes in that work, this thesis develops a product line realization process that consists of successive Selection and Modification sub-activities (Fig.36). Another similarity is that the Modification sub-activity consists of successive commonality- and variability-related sub-processes (Commonality Realization and Variability Realization). The same order of process steps (first a commonality-related step, and then a variability-related one) has also been suggested in the original family paper ("we consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by *first* studying the common properties of the set and *then* determining the special properties of individual family members") [Parnas76]. In this thesis, I have identified the reason why it is economical to these sub-steps in the given order: because only in that order, they result in backtrack-avoiding sequences which provide incremental evolution possibilities (see also Sec.3.4).

3.3 Usefulness of Cloning

The practice of code duplication (Cloning) has traditionally been regarded as universally harmful, in software engineering in general, in software reuse, and in product line engineering. For example, Cloning has been ranked as the top "code smell" indicating the need to refactor software [Fowler99], and it motivates the need for automated clone detection tools [Demeyer++02]. Due to the presence of Cloning, "a software developer spends more time locating, understanding, modifying, and debugging a [cloned] code fragment than the time required to develop the equivalent software from scratch" [Krueger92]. Cloning "creates gratuitous complexity" and makes software engineers "drown in a sea of look-alikes" [Bassett97]. Despite these considerations, Cloning is used as one of the most popular approaches for realizing variation in practice [Bassett97, Krueger07]. In our early paper on variability mechanisms, we explicitly excluded Cloning by stating that "goals of variability mechanisms are to minimize code duplication, reuse effort, and maintenance effort" [Muthig+02]. Nowadays, I would rephrase the first goal to "reduce redundant development activities as required" (see also Def.65).

Many investigations on Cloning and clone prevention have been performed by the software evolution and software reengineering communities, addressing single systems [Demeyer++02, Mens+08, Roy++09]. For single systems, a growing number of empirical investigations have found that the conventional assumption of the universal harmfulness of Cloning cannot be supported anymore and that there are development contexts in which Cloning is not a disadvantage or where it is even beneficial. These ideas have not yet been investigated in product line engineering. A contribution of the current thesis is to include Cloning as a variability mechanism and an integrated target of

certain evolution sub-processes, and to show its usefulness in some product line evolution contexts.

In the remainder of this section, a number of independent studies and surveys from the past decade are presented which give convincing evidence that Cloning is not detrimental, or even beneficial, for some software qualities, such as evolvability (variability in time), in certain real-world development contexts.

In an early empirical study on this subject, Monden et al. investigated the relation between code clones and software reliability and maintainability of a 20 year old software system written in a COBOL dialect, consisting of 2000 modules and one million lines of code. They found that on average, cloned modules were 1.7 times as reliable as non-cloned modules, but that modules containing large clones were less reliable. They also found that cloned modules were less maintainable than non-cloned modules, and that modules containing larger clones were less maintainable than those with smaller clones.

In an investigation that explored the industrial resistance to adoption of software maintenance automation, such as clone detection and refactoring tools, Cordy summarizes the realities he observed in 6 years of automation services in financial software, involving more than 4.5 GLOC of code [Cordy03]. A surprising result was that clone removal is risky. As the data processing programs across an organization were very similar, it was practice to create new applications by cloning an existing custom clone. It was also found that discovered errors in a clone were not a problem because the common practice was only to remove them in the respective product, and deliberately leave the others unaltered, tolerating the error there because the risk was too high that removing the error led to new errors (I have made the same observations in several larger embedded systems projects from the automotive domain, where refactoring clones would need re-certification of systems, which was infeasible due to time constraints). Only on rare occasions, when a fundamental change was necessary for a central element, automated clone detection and removal was essential. As one conclusion, the author recommends to "emphasize agile, lightweight techniques that provide timely answers as needed".

In a study of usage patterns of Cloning, Kim et al. observed the Cloning practices of expert software developers [Kim++04]. Partially by direct observation, and partially by an instrumented Eclipse development environment, the nine subjects, mainly developing Java software, were observed over a period of 60 hours. It was found that Cloning saved typing effort, captured important design decisions made by programmers, were useful for program understanding, and in the short term were useful for deciding when to refactor. Some specific insights were gained, for example that certain programming language limitations

result in unavoidable clones, that programmers sometimes deliberately do not refactor clones because the result would not match their conceptual code organization, that refactoring clones is often postponed until Cloning has been practiced several times, and that copied text is often used as a template that is customized in the pasted context. The case study in this thesis also shows that cloning does not lead to significant complexity increase in the short term, making it useful in that context.

A follow-up empirical study investigated code clone genealogies, the history of how each element in a group of clones has changed with respect to other elements in the same group [Kim++05b]. The goal was to show if Cloning is inherently bad and if refactoring is a solution. A model of clone genealogies was presented which describes how groups of clones change over time, consisting of the relationships Same (no cloned elements have changed), Add (one or more elements have been added to the clone), Subtract (one or more elements have been removed from the clone), Consistent Change (all cloned elements have been changed consistently), Inconsistent Change (at least one element has been has been changed inconsistently), and Shift (one or more cloned elements partially overlap). Supported by tool automation, these evolution patterns were detected in a source code repository of two open-source Java projects (37-224 versions). It was found that aggressive, immediate refactoring is unnecessary for many volatile clones (48-72% of the clones in the study disappeared within 8 versions), and that conventional refactoring techniques cannot easily remove many long-lived clones (49-64% of the clones in the study could not easily be removed). It was concluded that Cloning can be useful in the short term, and that "refactoring may not always improve software with respect to clones" [Kim++05b, p.187]. These insights have been considered when developing the product line evolution method in this thesis.

The two above mentioned studies are summarized in [Kim08].

Another set of studies on the harmfulness of cloning practices were conducted by Kapser et al. The first study [Kapser+06] investigated cloning in the Apache web server C code, and identified over 13000 clones. It was found that "platform-specific code often had a high degree of cloning" and that this design strategy "can be an advantage in the initial stages of development when appropriate abstraction levels and degrees of commonality between subsystems are unclear". Cloning was also found reasonable in case of experimental additions to the system. During early iterations of the Variability Realization sub-process presented later in this thesis (Fig.39), the family engineer is exactly in this development context. This makes Cloning a viable variability management strategy.

The following publication [Kapser+08], one of the most frequently cited publication on the subject, evaluated the harmfulness of Cloning and found significant evidence that cloning is often a "principled engineering tool". The paper describes four classes of patterns of cloning with eleven sub-patterns, discusses their advantages and disadvantages, and evaluates their frequencies in two larger open-source systems, the Apache web server and the Gnumeric spreadsheet. The four classes of patterns are Forking (cloning larger portions of code in order to evolve them independently), Templating (cloning an existing solution in order to parameterize it in simple ways), Customization (realizing a very similar problem for which a solution already exists, but which requires more involved additions, removals and modifications than in Templating), and Exact Matches (cloning without modification). As suggested in the paper, explicit links may be used to manage Boilerplating, a sub-pattern of Templating. These explicit links may be code annotations, for example specific comments which explain how to modify a neighboring code element. In other words, they make variation points explicit. They may also facilitate automation, by means of custom code generators as mentioned above [Weiss+99, Ch.5], or by Frame Technology which is indicated in the paper [Kapser+08, p.654]. In the current thesis, this technique is also considered for Replicate and Specialize, a Customization sub-pattern which has been identified most frequently in Kapser's case studies. The different types of Cloning were identified in the two case studies, and were ranked either as good, incidental harmless, or harmful. It was found that on average there were more useful instances of Cloning (33-71%) than harmful ones (14-57%). Two particular Cloning patterns, Boilerplating (100%) and Replicate and Specialize (56-94%), were predominantly beneficial. This is important in the current thesis because exactly these two patterns solve typical product line development problems.

Two summaries of these observations were published in [Kapser09, Godfrey+10].

Another set of empirical studies first investigated how clones evolve [Aversano++07]. Using a refined version of the classification scheme by Kim et al. [Kim++05b], the code evolution in two open-source projects, ArgoUML and DNSJava, was observed over more than 5 years. It was found that 45-74% of the clones were changed consistently, and an additional 13-16% underwent late propagation, i.e., consistent change did not happen immediately, but in the long term. The method developed in this thesis does not assume refactoring activities to happen immediately after a potential Cloning activity in Commonality Realization (Fig.39), but that both may happen independently of each other, with a possible delayed refactoring. The authors conclude that with regard to consistency, cloning was not harmful in these projects. Follow-up work [Thummalapenta++10] extended that paper by developing an automatic approach for classifying the clone evolution patterns of Consistent

Evolution, Late Propagation, and Independent Evolution, and analyzing the code of four open-source systems (ArgoUML, JBoss, OpenSSH, and PostgreSQL). It was found that clones are often propagated immediately in these systems, which makes clone refactoring unnecessary. Another observation was that Templating has been commonly used in all systems, leading to co-evolution. It was also found that clone characteristics, such as programming language, clone radius or clone detector, do not influence the evolution pattern, and that high proportions of defect removals occur for Late Propagation clones, which indicates that this type of clone tends to be more defect-prone. For the approach presented in the current thesis this means that the engineer must be educated that such types of clones require more caution.

Using similarity measures, an empirical study investigated how certain types of "clone smells" evolved in the source code of the Mozilla Firefox web browser [Bakota++07]. The investigated evolution scenarios are Vanished Clone Instance (the clone disappeared in subsequent versions), Occurring Clone Instance (a new instance of the clone emerged), Moving Clone Instance (the original clone broke apart), and Migrating Clone Instance (a clone was later consolidated again). This work is related to the current thesis because it investigates variation across space, realized by the Cloning mechanism in a real-world software system.

Another investigation [Krinke07] analyzed to what extent consistent and inconsistent clones have propagated in source code of five successful open-source projects (ArgoUML, CAROL, jdt.core, Emacs, and FileZilla). The results show that about 50% of the clones were inconsistent, but that these were not problematic because they usually remained unaltered in later versions. A follow-up study [Krinke08] used most of the same systems to analyze if cloned code is more stable as non-cloned code. This hypothesis was confirmed. There have also been quantitative approaches to stability in software evolution ([Kelly06], see Sec.3.4), but the novelty of the current thesis is that it investigates these issues for product lines, in order to arrive at stable processes.

In two other publications on the topic, Lozano et al. evaluated the harmfulness of cloning, with a focus on changeability. In an initial study [Lozano++07], a custom tool was used to analyze the evolution of an open-source Java application (DnsJava) over 99 months by analyzing its CVS repository. It was found that methods changed more, and more frequently, when they contained cloned code. A follow-up study [Lozano++08] compared maintenance effort of cloned methods vs. non-cloned methods by testing the hypothesis that if a method has clones, the effort spent in changing it increases. In an empirical study of the 3-6 year evolution of four open-source Java projects (ganttProj, jEdit, freecol, and a jboss subsystem), measurements were performed on the likelihood, impact, and effort of changes. It was found that Cloning did not affect the likelihood of changes, but increased the number of

changes, and that in at least 50% of the cases, being cloned did not increase changeability measures, but when it increased the changes were significant. More details can be found in [Lozano09]. The current thesis also investigates evolution qualities by measuring code characteristics, but not at method granularity and not just for variability across time, but also for variability across space.

Other empirical work investigated clone refactoring possibilities in 17 real-world web applications [Rajapakse+07]. Although clone refactoring was technically feasible most of the time and resulted in code size reductions up to 78%, it also caused many trade-offs that would be unacceptable in real-world development contexts, such as rapid evolution, a topic explored in this thesis. In a following position paper [Jarzabek+10], it was mentioned that there is no definite answer if cloning is harmful because this depends on the context, balancing various software qualities or engineering goals. Clones may be created deliberately to improve reliability or performance, to avoid complicating the code, or because of programming language limits, and there may not be a clone-free alternative. In another survey paper [Hordijk++09], a quality model for Cloning was developed which was applied to categorize literature on the subject. No definite answers in favor or against Cloning were found, which led to the conclusion that more research is needed.

Another recent empirical study analyzed the impact of inconsistent changes on software quality, as perceived by the end user [Bettenburg++10]. The evolution history of three open-source systems (Apache Mina, jEdit, ArgoUML) was observed over 7-50 releases, using different types of clone detection tools. It was found that only 1-4% of the clone genealogies caused an end-user-visible defect. Again, it was found that the majority of long-lived clones (44-68%) were of the Replicate and Specialize type (cf. [Kapser+08]) and were deliberately introduced. The conclusion is that for the studied systems, clones do not have a large impact on post-release defects. For the current thesis, this means that quality assurance must be an integral part of the product line evolution method, whereas the selection of a particular variability mechanism tends to be less important for product quality, from an end-user perspective.

A similar study analyzed the source code of 17 open-source systems written in C, C++, C#, and Java [Saha++10]. Using a custom clone genealogy extractor, four types of genealogies were extracted: alive genealogy (containing at least one clone group in the latest release), dead genealogy (the opposite), syntactically similar genealogy (in which only identifiers were changed, but no lines were added or deleted), and consistently changed genealogy (in which all clone groups were at least changed consistently once). It was found that the proportion of live and dead genealogies was similar for all systems, independent of the

programming language, and that clones appeared to be more manageable in smaller systems than in larger ones. For the current thesis, this means that Cloning as a variability mechanism may be more effective for product lines for smaller systems, such as many embedded systems. It was also found that a large proportion of genealogies were alive and long-lived, that most of the clone groups that do not change syntactically are unlikely to be removed during evolution, and that many volatile clones disappeared quickly.

A further empirical study on clone evolution dynamics investigated if clones have a higher change frequency than non-clones [Hotta++10], which would be detrimental for software evolution. 15 different open-source systems written in C, C++, and Java, from different application domains and of different sizes were analyzed for modification frequency of clones. It was also found that in short periods, cloned code was modified more frequently than non-cloned code, whereas in the long term cloned code was modified less frequently than non-cloned code, for all analyzed systems. The conclusion is that Cloning did not have a serious impact on the evolution of the studied systems. The results apply to variation in time. If the same results also apply to variation in space, as found in product lines and investigated in this thesis, then Cloning could also be a viable long-term realization strategy.

A recent empirical study in the presence of four commercial systems written in C# and COBOL showed frequent inconsistent changes to code, resulting in a significant number of defects [Jürgens++09]. This means that code clones mattered for software correctness in these systems. However, as the authors admit, they did not investigate the impact on evolution, a major topic addressed by the current thesis. In a follow-up publication [Jürgens+10], an analytical cost model was developed for quantifying the cost of cloning on maintenance. Eleven industrial systems were analyzed, with mixed results. Whereas some subjects could benefit from clone detection and removal, it is not cost-effective for others.

In contrast, another recent empirical study analyzing the impact of Cloning on defect-proneness [Rahman++10] came to the opposite conclusion. The study analyzed four major open-source projects (Apache, Evolution, Gimp and Nautilus) over 116-155 versions. Three research questions were addressed: To what extent does cloned code contribute to defects? Do clones occur more often in defective code than elsewhere? Are prolific clone groups (clones with many copies) more defective than non-prolific clone groups? It was found that most of the defects in both liberal and conservative clone detector settings contained hardly any cloned code. This means that only a small number of defects were caused by Cloning. Using statistical methods, it was also found that across all analyzed projects the overall clone ratio was significantly lower than clone ratio in defective code. This indicates that clones are not a

major defect source in these projects. Prolific clone groups also had lower defect density (defects per lines of code) than non-prolific clone groups. This means that making more copies of a clone did not introduce more defects.

Yet another empirical study analyzed in more detail which clone characteristics have a particularly high impact on defects [Selim++10]. The code history of two large open-source projects (Apache Ant and ArgoUML) was analyzed using two clone detectors. Two research questions were addressed: Can we model the impact of clones on defects with high accuracy? What are the most important predictors of defects in cloned code? It was found that cloned code is not always more risky than non-cloned code, but that the risk is system dependent. This supports the approach taken in the current thesis of selecting variability mechanisms, or removing initially introduced clones, based on the development context.

In another investigation, Olbrich et al. have recently analyzed if code smells in general, not just clones, are harmful [Olbrich++10]. They analyzed the history of 3 open-source projects (Lucene, Xerces, Log4j) with regard to God Classes and Brain Classes and found, after normalization with respect to size, that these were more stable and contained fewer defects than other classes. This is another indicator supporting the strategy proposed in this thesis of deliberately using plain mechanisms, but in a well-defined and well-understood manner (cf. the discussion in the context of Fig.19). A survey on this topic has also been presented in [Zhang++11].

3.4 Complexity and Evolution in Single Systems

Successful software products evolve [Parnas94]. While much research has been done on observing how source code is organized at a particular moment in time, research has rarely investigated how code characteristics change over time, and even fewer suggestions have been made how to systematically counteract long-term code degradation.

According to [Sommerville04], the majority of work on software evolution has been carried out by Lehman and Belady. Lehman et al. proposed a classification of software systems according to their evolution dynamics [Lehman80]. E- (Evolution-) type systems are the most common form developed and evolved in practice. A software system is an E-type system if it is actively and regularly used to solve a problem in a real-world domain. Other types are S- (Specified-) type systems whose only criterion is correctness, and P- (Problem-) type systems which share some properties of both E-type and S-type systems. This thesis only addresses product lines of E-type systems because of their predominance in practice.

Evolution is intrinsic for E-type systems. Based on long-term observations of E-type systems, eight hypotheses on their evolution characteristics have been proposed which are known as the laws of software evolution. These are summarized in Table 3.

Name	Meaning
Continuing Change	An E-type system must be continually adapted or else it becomes progressively less satisfactory in use.
Increasing Complexity	As an E-type system is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity.
Self-Regulation	Global E-type system evolution is feedback regulated.
Conservation of Organizational Stability	The work rate of an organization evolving an E-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime.
Conservation of Familiarity	In general, the incremental growth (growth rate trend) of E-type systems is constrained by the need to maintain familiarity.
Continuing Growth	The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime.
Declining Quality	Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining.
Feedback System	E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.

Table 3: Laws of software evolution [Lehman+06a]

Although these laws were formulated for single systems only, they may be translated to product lines. Table 4 lists the corresponding rules, as relevant for this thesis.

Name	Meaning
Continuing Change	An E-type product line infrastructure must be continually changed or else it becomes progressively less satisfactory in reuse.
Increasing Complexity	As an E-type product line infrastructure is changed its complexity increases and it becomes more difficult to evolve unless work is done to maintain or reduce the complexity.
Continuing Growth	The adaptation capability of E-type product line infrastructures must be continually enhanced to maintain reuser satisfaction over product line lifetime.
Declining Quality	Unless rigorously changed and evolved to take into account changes in the operational environment and product line engineering context, the quality of an E-type product line infrastructure will appear to be declining.

Table 4: Laws of product line infrastructure evolution

The first law in Table 4 has been translated from single systems (Tab.3) to product lines by taking into account the duality of use and reuse (Sec.2.1).

The complexity phenomenon addressed in Lehman's second law has also been discussed by Brooks [Brooks95, pp.182ff.] who observes that not all complexity is inevitable [Brooks95, p.211]. For this reason, he explicitly distinguishes between the two types of essential and arbitrary complexity. Whereas essential complexity cannot be reduced in the code without violating the requirements, arbitrary complexity is reducible, and this excess complexity is what I mean when speaking of complexity in general, as defined in Section 2.3 (Def.43). The corresponding law for product lines refers to variability complexity (Def.65).

Lehman's sixth law is mapped to the third law for product lines by replacing the actors causing the change (Lehman's users can denote both end-users and application engineers that exercise unmodified reuse (Def.6) by reusers who are application engineers in product line engineering. Whereas Lehman addresses functionality in this law, it is mapped to adaptability, the corresponding reuse property [Bassett97], in Tab.4.

The software aging phenomenon mentioned in the discussion of Lehman's law was coined in the paper by Parnas mentioned at the start of this section [Parnas94]. In that paper, it is claimed that the two reasons for software aging are lack of movement (the system is not evolved enough) and ignorant surgery (the system is changed by incompetent developers), and I used the same two reasons to motivate the need for a product line evolution method (see Fig.1). The paper suggests a number of countermeasures, for example stopping the deterioration, retroactive documentation, retroactive incremental modularization, amputation or restructuring. These techniques have become popular as refactorings [Opdyke92, Roberts99, Fowler99, Kolb++06]. The goal of conventional refactorings is to reduce the complexity of how the artifact (e.g. code) is composed (Def.9), keeping the runtime behavior of the resulting code invariant. Besides these refactorings, the evolution of a product line infrastructure (e.g. its code) requires additional variability refactorings (see Def.69), of which examples will be shown in Section 5.2. The corresponding rule of declining quality in product line infrastructures (Table 4) adds the product line engineering context to the factors that cause the change. The product line engineering context does not just mean application engineering needs, but also changes in the tools and methods in the product line infrastructure.

Three possibilities for measuring code complexity in an evolving single system were suggested in [Hall+00]. Some of these metrics have independently been used for defect prediction in practice [Munson96,

Nagrappan+05], and have also been suggested for product line measurement [Ajila+07]. The first metric, code delta, "indicates how the system as a whole has increased or decreased in terms of the chosen measure" [Hall+00], for example in terms of lines of code per module. Code delta is defined thus:

$$\Delta_{ABC}^{t,t+1} = \sum_{c \in M_c} \left(m_c^{t+1} - m_c^t \right) - \sum_{a \in M_a} m_a + \sum_{b \in M_b} m_b \ ,$$

where ABC denotes a chosen metric (e.g. LOC), t and t+1 characterize two consecutive points in time, m is the metric value, M_A is the set of modules removed between t and t+1, M_B were added in this period, and M_C were changed. A limitation of code delta is that it does not indicate how much has changed because if an equal number has been added and removed, the delta is zero. One can measure the change, however, if only the absolute values are taken. The resulting measure is called code churn, and it measures the sum of the added, changed or deleted items, for example lines of code. The definition of code churn is

$$\nabla_{ABC}^{t,t+1} = \sum_{c \in M_c} \left| m_c^{t+1} - m_c^t \right| + \sum_{a \in M_a} m_a + \sum_{b \in M_b} m_b \ .$$

The third measure describes how to compare average values against a baseline, so that these values can be compared. This is achieved by standardizing the measured value x against the mean μ and standard deviation σ of the reference, using the standard score z:

$$z = \frac{x - \mu}{\sigma} \ .$$

The z value is a dimensionless quantity which indicates to what extent the new value has changed, in standard deviations of the old values. If there has been no change, z is 0. In the paper [Hall+00], the reference values for μ and σ are obtained for the first version of the system, at $t=t_0$. The values in later evolution steps are compared against these, using the standard score. In cases when there is only a single sample for the baseline value, σ is set to 1. In Section 5.3, this metric will be extended for product line measurement, so that it does not only compare against a base version in time, but also against reference product line infrastructure code in space.

Numerous other definitions and models of complexity have been proposed in the software engineering literature. For example, three types of complexity have been distinguished in [Laird+06]: structural, conceptional, and algorithmic complexity. Complexity has also been classified as a sub-characteristic of the internal product attribute size,

alongside the other sub-characteristics length and functionality [Fenton96]. That work identified four complexity dimensions: problem complexity, algorithmic complexity, structural complexity, and cognitive complexity. The first two dimensions address issues such as efficiency in space and time (space and time complexity: resources used by the running machine code), or Big-O complexity (constant, logarithmic, linear complexity, quadratic or exponential complexity: scalability of function calls). Structural complexity is concerned with issues of control flow (e.g. McCabe complexity), data flow (coupling, cohesion, fan-in, fan-out), or data structure (morphology: size, depth, width). Some of the metrics invented in the current thesis were inspired by those metrics. Cognitive complexity denotes how easy software can be understood (it remains unclear from which perspective, from that of a developer or an end-user). The author concludes that complexity is a combination of different attributes, and that a single measure for these sometimes conflicting goals is dangerous. I share this view in the current thesis, but use a different overall complexity concept, more alongside that suggested by Brooks ([Brooks95], see the remarks above and Def.43), where complexity is a relative measure associated with excess of artifact elements.

According to Sneed et al. [Sneed++10, pp.54-56], a different complexity concept has been suggested in the software measurement community by Kokol et al. [Kokol++99]. That concept is similar to the complexity concept used in this thesis. They reject usual complexity metrics such as those mentioned above because they only measure complexity of the representation but not the complexity of the system itself. The measure is called Alpha Complexity Metric. Their complexity concept is based on entropy, a measure for lost energy in physical systems which does not directly serve its purpose. Applied to software systems, these are all system elements which do not directly contribute to the desired result. Likewise, the current thesis investigates those elements in product line infrastructures which do not directly contribute to the production of products, and so make the infrastructures complex. A widespread measure of entropy or complexity in different disciplines is Long Range Power Law Correlation (LRC) which refers to anything that leads to unnecessary bloat. For conventional code this could be unnecessary algorithms or temporary variables (these also appear among the code smells for conventional refactorings [Fowler99]). The approach was validated for source code of successive versions of Microsoft Windows which showed increasing Alpha Complexity, indicating a growing amount of unneeded code. The challenge in measuring Alpha Complexity Metric is to distinguish between essential and non-essential elements. In the current thesis, I have invented two-dimensional baselining for describing limits for variation in space and time.

Other research has not only investigated how complexity can be passively observed in systems, but proposed concrete processes to

counteract complexity growth [Alexander02]. That work is concerned with general types of systems and their environment. Transferred to product lines, this corresponds to the product line infrastructure and its environment (engineers, customers, money, etc.), which are also known as the product line ecosystem [Bosch09, McGregor+10]. A number of orthogonal properties are identified that occur when non-complex systems are built. The development process is incremental and development decisions are consciously taken as to avoid backtracking effort in case of errors. The process basically exists of the three steps of observation, modification and quality assurance, which inspired the family engineering process developed in this thesis. In [Alexander02], a precise complexity concept is also suggested as elements that unnecessarily complicate an artifact. This matches the ideas mentioned above and influenced complexity considerations in the current thesis. An important observation, transferred to product line engineering is that the simplest realization of core assets is one in which the degree of commonality and variation exactly matches the needed degree of commonality and variability.

An approach to measure single-system evolution characteristics of software artifacts over time was given in [Kelly06]. The aim of that work was to detect artifacts which remain stable during long-term evolution. A software artifact is regarded as stable if, when observed over two or more versions, the differences in a metric associated with the artifact are small. As a difference metric, the distance function D(x,y) is proposed, where D(x,x)=0 (its value is 0 if two artifacts are identical), and D(x,y)>0 if $x \neq y$. Figure 22 shows that a reference version of an artifact at time $t=T_0$ evolves into three artifacts at T_1, T_2 and T_3. Two possibilities for variation (V) or stability emerge: First, there is temporal stability which is defined by the maximum distance V_{BT} between the initial version T_0 and one of its successors T_1, T_2 and T_3 (solid arrows). Second, there is a spatial stability, defined as the maximum distance V_T among the successors T_1, T_2 and T_3 (dashed arrows).

The current thesis extends this approach by not just considering a baseline for variability in time, as suggested by the artifact T_0 in Fig.22, but also proposing a baseline for variability in space (see Fig.47).

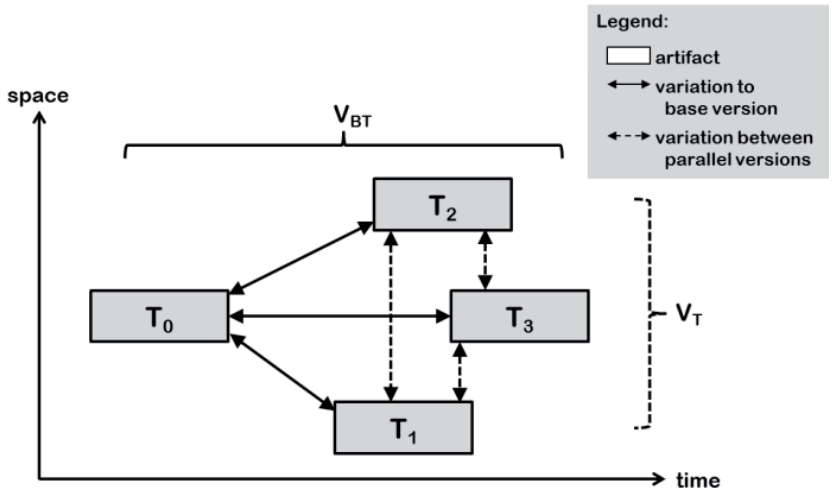


Figure 22: Temporal stability V_{BT} and spatial stability V_T in the evolution of software artifacts [Kelly06]

3.5 Complexity and Evolution in Product Lines

Publications on product line-specific complexity are rare. In an early publication on the topic, Bosch et al. have identified and classified core issues of variability management [Bosch++02]. The classification lists general issues, family engineering issues in the architecting, design, and realization phases, application engineering issues, and issues in the evolution of variability. The current thesis focuses in particular on family engineering issues and related issues in the evolution of variability.

Complexity of software variability has been declared as a topic in [Deelstra+08], with nearly exclusive focus on application engineering. However, the complexity concept remains undefined, and it is not made clear how it differs from single-system complexity. The number of variation points and variants are identified as main issues of complexity, and two other briefly mentioned factors are obsolete variation points and non-optimal realization of variability. It is not made clear why only these factors were chosen and how they may be detected and removed in a systematic process. In contrast, the current thesis focuses on product line-specific complexity from a family engineering perspective. This is why I can clarify how variability complexity differs from single-systems complexity. Product line-specific complexity issues such as variability, reuse efficiency and ease-of-configuration are driven by family engineering, not by application engineering. Application engineering only consumes the product line infrastructure, so that particular novel complexity issues do not arise there, compared to single systems engineering.

Another publication on the topic [Lopez+08] observes that variation points distinguish product line assets from conventional assets, and proposes to use Cyclomatic Complexity as a variability complexity metric.

In contrast to [Deelstra+08], process issues are out of scope. In contrast to the current thesis, the complexity concept is not discussed at all, only one solution for a particular type of realization is presented, and an underlying quality model is missing. Although I also use Cyclomatic Complexity as a product line metric, I have extended it to four different metrics that cover the two dimensions of binding times and variant isolation.

With regard to product line measurement, a case study investigating product line evolution [Ajila+07] applied the classical evolution metrics of code churn, code delta and change rate (see Sec.3.4) to core assets and non-core asset code of a commercial product line. It was found that code size increased continually, although developer productivity varied. It was also found that the majority of changes resulted in increased code complexity, and that code churn and number of modules was low. Product line-specific results were not discussed.

In another publication on the topic [Berger++10], the following metrics were suggested: size of commonality, impact of commonality, product-related reusability, impact of product-related reusability, reusability benefit, relationship ratio, and individualization ratio. These were applied to a feature model of a small product line. The results were used for recommending which products should first be supported by the product line. In contrast to the current thesis, variability, the main characteristic which distinguishes a product line from single systems, was not discussed.

A publication which considered variability in product line measurement [Zhang++08] suggested four dimensions of metrics: commonality metrics, variability metrics, reusability metrics, and complexity metrics. The following basic metrics were identified: number of common components, number of variable components, number of variation points, number of independent variation points, number of weak coupling variation points, and number of product line members. A number of aggregate metrics and complexity metrics were suggested. Unlike in the current thesis, the usage of these metrics in order to achieve a certain goal was out of scope.

Few publications have been concerned with removing product line-specific complexity by means of variability refactorings. Two of our earlier mentioned publications have presented case studies of product line development in practice where code smell detection and refactoring support were issues [Patzke+04, Kolb++06], for example to improve support for Conditional Compilation.

Alves et al. [Alves++05] have suggested a number of activities for converting conventionally written code for mobile games written in Java into an aspect-oriented realization. The listed activities are Extract

Method to Aspect, Extract Resource to Aspect, Extract Context, Extract Before Block, Extract After Block, Extract Argument Function, Change Class Hierarchy, and Extract Aspect Commonality. They call these activities refactorings, although they do not motivate what specific goals their refactorings have in a product line engineering context, except for converting code to Aspect-Orientation, and which quality attributes are improved. In a later publication, some of the same authors define software product line refactoring as "a change made to the structure of a SPL in order to improve (maintain or increase) its configurability, make it easier to understand, and cheaper to modify without changing the observable behavior of its original products" [Alves++06]. This end-result-focused definition lacks product line engineering process elements and does not reveal the difference to conventional refactorings. A number of feature model change operations are presented which do not make clear which quality attributes, if any at all, are improved. The list of refactorings consists of Convert Alternative to Or, Collapse Optional an Or, Collapse Optional and Alternative to Or, Add Or Between Mandatory, Add New Alternative, Convert Or to Optional, Convert Mandatory to Optional, Convert Alternative to Optional, Pull Up Node, Push Down Node, Remove Formula, and Add Optional Node. The two publications are summarized in [Alves07].

Another publication [Lösch+07] addresses the problem of obsolete variants in product lines. Based on Concept Analysis, three different refactoring strategies for removing unused variants are shown (Merge Variants, Remove Variants, Mark as Alternative. The paper only addresses complexity due to lack of change, which is not the focus of the current thesis. Moreover, only an extremely limited set of refactorings for a particular realization technique based on composition is seen, and process issues with regard to product line engineering are not addressed at all.

Many publications have been concerned with product line evolution [Svahnberg+99, Savolainen+01, Bosch02, Pussinen02, Deelstra03, McGregor03, Knauber04, Patzke+04, Kolb++06, Ajila+08, Anastasopoulos++09, Elsner++10, Estublier++10, Lutz++10, Guo+10, Krueger10, Ramasubbu+10].

Product line evolution categories and their interdependencies in different product line engineering phases have been studied in [Svahnberg+99]. The identified requirements evolution categories are New Product Family, New Product, Improvement of functionality, Extend Standard support, New version of infrastructure, and Improved quality attribute. The mentioned types of architecture evolution are Split of software product line, Derivation of product line architecture, New component, Changed component, Replacement of component, Split of component, New relation between components, and Changed relation between components. Evolution categories for product line realization are New

framework implementation, Changed framework implementation, Decreased framework functionality, and Solving in external component. Many of these scenarios are not specific to product lines, and in particular are not specific to variability. In contrast, the set of product line evolution scenarios suggested in Sec.5.1 of this thesis focuses on product line-specific evolution scenarios which are related to variability in time of variability in space.

Another paper on the subject [McGregor03] claims that the difference between evolution of single systems and evolution of products in (proactive) product lines is that for single systems, anticipated evolution is possible and unanticipated evolution is very likely, whereas for product lines, anticipated evolution is very likely and unanticipated evolution is less likely. In the current thesis, a distinction between anticipated and unanticipated evolution is not made, that is, all evolution is seen as unanticipated. Anticipated evolution is deliberately not considered because it leads to speculative design decisions and extra complexity which is not needed in the respective product line infrastructure version. This view is consistent with non-proactive product line methods [Krueger10].

The distinction between proactive and reactive product line evolution has been discussed in [Knauber04]. Two main differences between single-systems evolution and product line evolution are identified: First, single systems are evolved in situations when not all requirements have been known before, whereas product line evolution happens in situations when a stable product line infrastructure exists. Second, incremental development of single systems extends functionality, whereas incremental product line development is concerned with improving the product line infrastructure, when the complete functionality of some products already exists, possibly redundant. The same distinction is made in the problem statement of the current thesis (Summary and Sec.1.1), considering more affected properties than just functionality. The paper also recommends strategies for proactive vs. reactive evolution: Whereas the former should be concerned with product line infrastructure development first, followed by product development, the latter should proceed in the opposite order. Whereas these recommendations address product line adoption in situations when a product line infrastructure does not yet exist, the current thesis starts with the assumption that a product line infrastructure already exists and concentrates on how this infrastructure is going to be changed during evolution. A recent case study about evolution of long-lived, sustainable systems [Lutz++10] illustrates how anticipated and unanticipated changes to the Voyager spacecraft can be handled with product line engineering methods.

Another publication on the subject [Elsner++10] investigated the different notions of evolution, or variability in time, in product line engineering. Three different categories of variability in time are found

(variability of linear change over time, multiple versions at a point in time, and binding time over time), all of which are also addressed in this thesis. The first two categories are of particular importance to this thesis: The first characterizes situations in which either artifact versions change over time or their variability dependencies change over time. It applies to situations when each version invalidates the previous one, so that only the current version is regarded as valid.

The second more general characteristic is exactly what the current thesis defines as variability evolution (Def.67). It is concerned with multiple valid product line versions at the current moment in time, a situation I have often seen in practice: Due to legal or other organizational issues it is often not desired to evolve a certain set of products produced from a product line infrastructure. Companies tolerate that these products lack certain features or contain defects because immediate countermeasures are too expensive. In that case, both an older and the current version of a product line infrastructure are valid at the current moment. This situation is both supported by configuration management [Anastasopoulos++09] which is out of scope of this thesis, or by variability techniques such as the versioning idiom mentioned in the Details sub-section of Conditional Compilation (Sec.4.5). The same issue has also been shown in [Krueger10], as illustrated in Fig.23.

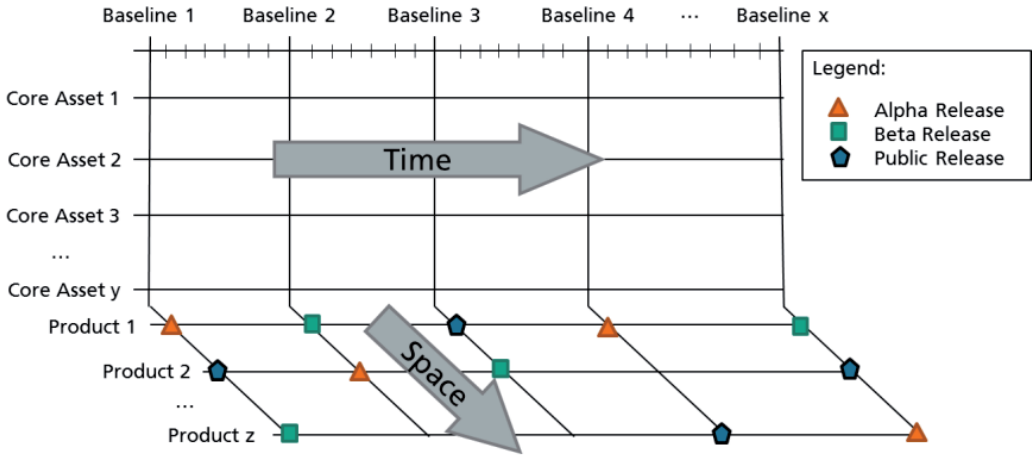


Figure 23: Product line evolution in time and space [Krueger10]

As illustrated in the figure, both the core assets and the products evolve in time and in space. Product line members may have different evolution rates in time, so that at a particular moment in time (e.g., at baseline x) the product line members have different maturity (product 1 is in the beta release phase, product 2 in public release, and product z in alpha release. This means that if all public releases must be supported at this time, versions of the core assets for baseline 3 and 4 must also be available, which lead to the production of public releases of product 1 and product z.

4 Variability Mechanisms

As shown in Section 2.3, variability mechanisms (Def.64) are used in product line engineering to realize variability in core assets with the intention of balancing reuse effort and evolution effort. They are adopted in family engineering for efficiently packaging common elements and variants, reducing product line-specific complexity.

The problem with variability mechanisms in product line infrastructure code in practice is that there are too many ways to realize variability in space and time. As criteria for their strategic application have not yet been given in product line literature, the code of each product line infrastructure is often realized with various inconsistent flavors and incompatible combinations of variability mechanisms [Krueger07]. This unnecessarily increases product line-specific complexity, makes the product line infrastructure code less sustainable and leads to avoidable whole life cycle effort and cost in family and application engineering.

To overcome these difficulties, I have developed an extensible list of tactics for effective family engineering (Section 2.3, Tab.2) which consists of the tactics

- Increase variation point explicitness,
- Allow appropriate variant granularities,
- Limit late binding,
- Isolate variants, and
- Provide automation.

All types of variability mechanisms may be classified according to these dimensions, allowing a family engineer to select them in his specific engineering context. For example, if a variant must be realized that consists of both small and large variant elements, the tactic would be to allow variant granularities of wide range, as opposed to those of narrow range. I have identified seven plain types of variability mechanisms that cover different combinations of these tactics:

- Cloning,
- Conditional Execution,
- Polymorphism,
- Module Replacement,
- Conditional Compilation,
- Aspect-Orientation, and
- Frame Technology.

Table 5 shows the mapping between tactics and variability mechanisms, and that each mechanism is assigned to a characteristic combination of tactics, which represents its profile of strengths and weaknesses. In the evolution process described in Section 5.2, this table helps the family engineer to select appropriate variability mechanisms or to refactor existing ones so that the product line infrastructure code becomes less complex. For example, if binding time restrictions do not exist and a new optional variability has to be realized, the relevant family engineering tactics (Tab.2 in Sec.2.3) could be to increase variation point explicitness, to limit late binding, and not to isolate variants (as only a single variant exists for optional variabilities, so that variant isolation would lead to unnecessary complexity). According to Tab.5, this combination of tactics is best matched by Conditional Compilation.

Property Mechanism	VP explicitness	Granul.	Earliest binding	Variant isolation	Production	Default support
Cloning	implicit (explicit for Templating)	wide	constr. time	yes (open)	manual	no
Conditional execution	ambiguous	narrow	runtime	no (closed)	automated	no
Polymorphism	ambiguous	narrow	mostly runtime	yes (open)	automated	no
Module replacement	ambiguous	narrow	exec. time	yes (open)	automated	no
Conditional compilation	explicit	wide	constr. time	no (closed)	automated	yes
Aspect-orientation	ambiguous	narrow	exec. time (+runtime)	yes (open)	automated	yes
Frame technology	explicit	wide	constr. time	yes (open, & often closed)	automated	yes

Table 5: Characterization of least complex types of variability mechanisms

Figure 24 illustrates in which module types and at which binding times each variability mechanism is typically employed (comp. Fig.9).

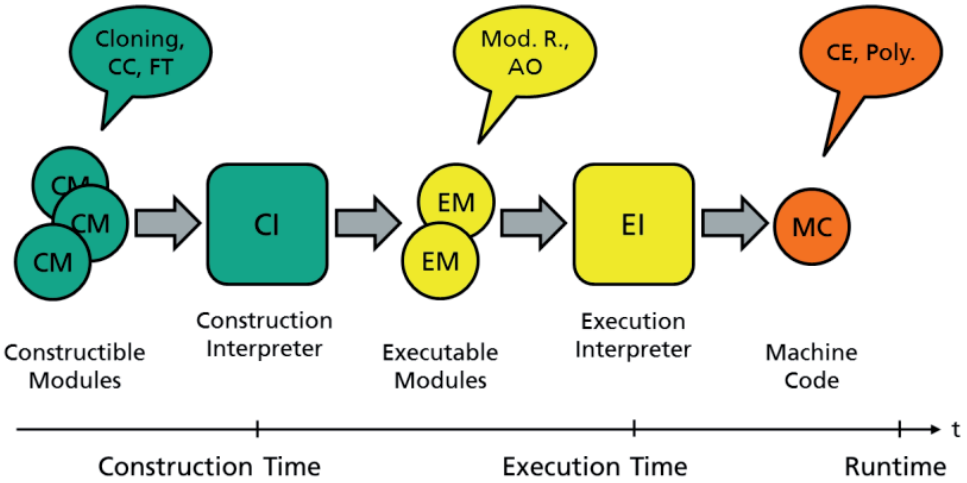


Figure 24: Mass customization by variability mechanisms

In the following sub-sections, each mechanism is discussed in more detail. Details on advantages and disadvantages of each individual mechanism can be found in the respective "applicability" and "consequences" sub-sections, and more detailed pros and cons compared to other mechanisms are shown in the respective "related patterns" section (the reason why I organize the sections in such a way is mentioned below).

Each of the mentioned types of variability mechanisms comprises a family of closely related particular variability mechanisms, so that the given list actually covers more than just seven mechanisms. Each of the mechanism types has been included in the list because

- it satisfies the tactic combinations in a plain (potentially the simplest) manner,
- it is known and frequently used in practice, eliminating the adoption barrier and avoiding disruption of ongoing development, or
- it has empirically shown new and unique variability management possibilities in practice or practical research.

The mechanisms are described independent of a particular programming language. As indicated in Fig.4, they are one of the input elements of the product line evolution method developed in this thesis. The mechanisms are presented in form of a pattern language [Alexander++77, Gamma++95] addressing family engineers in practice. For didactic purposes, the mechanisms are presented in a form shared by many software pattern catalogs used in practice, the *GoF* format [Gamma++95], using a wording that closely resembles that style. The format helps the family engineer to rapidly familiarize with the given pattern language and helps him to identify relevant pattern sub-topics, such as example realizations and pattern variants. The format also ensures that none of the essential software pattern characteristics [Vlissides98] have been omitted, which are problem, context, solution, recurrence, teaching, and naming.

Each pattern description consists of the eight items listed in Table 6. As a slight variation of the original GoF style, the items Structure, Participants, and Collaborations found in that style have been replaced by a more compact section explaining the development process of each variability mechanism, from the perspective of a family engineer.

Name: Point of reference to the pattern. Fosters communication among software engineers. Becomes part of the pattern language vocabulary.
Intent: Concise description of the pattern's purpose, formulated as an imperative, showing the software engineer if the pattern could be relevant for his problem.
Motivation: Example scenario which shows the software engineer how the pattern is typically applied. Exemplifies the Intent.
Applicability: Situations in which the pattern helps most.
Process: Construction process dynamics of applying the pattern, involved artifacts, tools and stakeholders. Guides the software engineer in which order to apply the pattern.
Consequences: Focal point of the pattern. Shows the software engineer the positive and negative effects of applying the pattern, so that he can use or reject the pattern based on informed decisions. Enumerates which system qualities and complexities are affected.
Details: Pattern-specific technical details, variants, tool support and known uses.
Related Patterns: Discussion of similar patterns from the current pattern language. Helps the software engineer to find alternatives.

Table 6: Variability mechanism pattern elements and their purpose

4.1 Cloning

Cloning is the most basic and by far most common form of reuse [Bssett97, p.86; Thörn10]. The mechanism is also known as Code Scavenging [Krueger92], Copy and Modify [Bassett97], Copy-Paste [Bosch00] or Clone and Own [Clements+01]. It is simple to introduce, which makes it popular for development in practice: Trusted code can readily be introduced, rather than rewriting it, custom modifications are easy, and there is no danger of breaking existing code which uses the unmodified clone reference. These short-term benefits are soon reversed as all cloned copies co-evolve independently.

Intent

Given a source code element which has proven its usability in existing software systems, adapt it to suit the changing needs of a new system. Cloning allows you to rapidly evolve common code without affecting its existing users.

Motivation

Consider the running example of a wireless sensor node product line (Sec.2.3, Fig.14). The left part of Listing 1 shows a real-world realization of one product line member, with variant elements color-coded as in Fig.14, and arrows denote variation points.

```
#include<string.h>
#include<stdio.h>
#include<stdbool.h>
#include<stdint.h>

// hardware initialization
void init();

// wireless transmission
// string to send
extern char send_buffer[61];
// sends send_buffer
void send();

// actuator abstractions
// switches led 2 on or off
void set_led_2(bool);
// toggles led 2: on <-> off
void toggle_led_2();

// clock abstraction
// clock value
extern int32_t the_clock;
// periodically set by ISR every sec
extern volatile bool period_elapsed;
// converts clock value to string
char* timetoa(int32_t);

// sensor values
extern int16_t x_position;

// sensor operations
// init_.. call before first use
// update_.. refeshes sensor value
void init_x_position();
void update_x_position();

bool event_happened=false;
int32_t event_time=0;
int16_t tilt_count=0;
int16_t tick=0;
```

```
void main() {
  init();
  init_x_position();
  while(true) {
    if(period_elapsed) {
      period_elapsed=false;
      update_x_position();
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        if(event_happened) { // a tilt has started
          event_time=the_clock; // start one-shot tm.
        }
        else { // a tilt has ended
          // has the device been tilted btw 1 and 5s?
          if(the_clock-event_time>0
             && the_clock-event_time<=5) {
            toggle_led_2();
            tilt_count++;
          }
        }
      }
      tick=tick+1;
      if(tick%5==0) {
        tick=0;
        sprintf(send_buffer,
                "drink=%d",tilt_count*25);
        send();
      }
    }
  }
}
```

```
void main() {
  init();
  init_x_position();
  while(true) {
    if(period_elapsed) {
      period_elapsed=false;
      update_x_position();
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        if(event_happened) { // a tilt has started
          event_time=the_clock; // start one-shot tm.
        }
        else { // a tilt has ended
          // has the device been tilted btw 1 and 5s?
          if(the_clock-event_time>0
             && the_clock-event_time<=5) {
            toggle_led_2();
            tilt_count++;
          }
        }
      }
      tick=tick+1;
      if(tick%5==0) {
        tick=0;
        sprintf(send_buffer,
                "drink=%d",tilt_count*25);
        if(event_happened) {
          strcat(send_buffer,",time=");
          strcat(send_buffer,
                 timetoa(the_clock-event_time));
        }
        send();
      }
    }
  }
}
```

Listing 1: Simple Cloning: Sensor node realization without (left), and with time transmission support

The time transmission feature has not yet been realized (yellow area). A second product shall be realized with this feature (optional variability). The simplest way to realize this, as shown in the right-hand code fragment in Listing 1, is to duplicate the code and to add the respective variant. Similarly, the other products may also be realized by cloning the original (Listing 1, left) and replacing the colored parts by alternatives (they correspond to alternative variabilities). This will probably be more difficult for the sensor (blue sections), as this variant consists of several variant elements, whereas the other variants only consist of a single variant element.

As the code changed by the family engineer does not have the shown colors and arrows, it is complicated for him to see the variation points and variants in Simple Cloning because the variation points are implicit (see Tab.5). However, the engineer can make variation points more explicit by annotating the variants, as shown in Listing 2 (gray sections), and then cloning the annotated sections. This more advanced type of Cloning is known as Templating [Kapser+08] which can also be automated for efficient production in product lines, as shown in [Weiss+99, Ch.5]. This shows that Cloning has several variants, some of which have been and are being used in product line development, which is another reason why I have included Cloning as a variability mechanism (also see the discussions on [Krueger04, Krueger07] in Sec.3.2).

```
#include<string.h>
#include<stdio.h>
#include<stdbool.h>
#include<stdint.h>

// hardware initialization
void init();

// wireless transmission
// string to send
extern char send_buffer[61];
// sends send_buffer
void send();

// actuator abstractions
// switches led 2 on or off
void set_led_2(bool);
// toggles led 2: on <-> off
void toggle_led_2();

// clock abstraction
// clock value
extern int32_t the_clock;
// periodically set by ISR every sec
extern volatile bool period_elapsed;
// converts clock value to string
char* timetoa(int32_t);

/* ADD SENSOR VALUES HERE */
extern int16_t x_position;
/* END */

/* ADD SENSOR OERATIONS HERE */
// init_.. call before first use
// update_.. refeshes sensor value
void init_x_position();
void update_x_position();
/* END */

bool event_happened=false;
int32_t event_time=0;
int16_t tilt_count=0;
int16_t tick=0;
```

```
void main() {
  init();
/* ADD SENSOR INIT HERE */
  init_x_position();
/* END */
  while(true) {
    if(period_elapsed) {
      period_elapsed=false;
/* ADD SENSOR UPDATE HERE */
      update_x_position();
/* END */
/* ADD DETECTION & TRANSMISSION HERE */
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        if(event_happened) { // a tilt has started
          event_time=the_clock; // start one-shot timer
        }
        else { // a tilt has ended
          // has the device been tilted between 1 and 5s?
          if(the_clock-event_time>0
             && the_clock-event_time<=5) {
            toggle_led_2();
            tilt_count++;
          }
        }
      }
      tick=tick+1;
      if(tick%5==0) {
        tick=0;
        sprintf(send_buffer,"drink=%d",tilt_count*25);
/* END */
/* ADD PRE-TRANSMISSION BEHAVIOR HERE */
        if(event_happened) {
          strcat(send_buffer,",time=");
          strcat(send_buffer,
                 timetoa(the_clock-event_time));
        }
/* END */
        send();
      }
    }
  }
}
```

Listing 2: Sensor node realization with Templating

Applicability

Use Cloning

- when it is easier or faster to slightly adapt mature existing code in a new context than to realize it anew or to thoroughly refactor it,
- to avoid the risk of damaging existing products when modifying reused code, or
- to explore short-term evolution possibilities.

Process

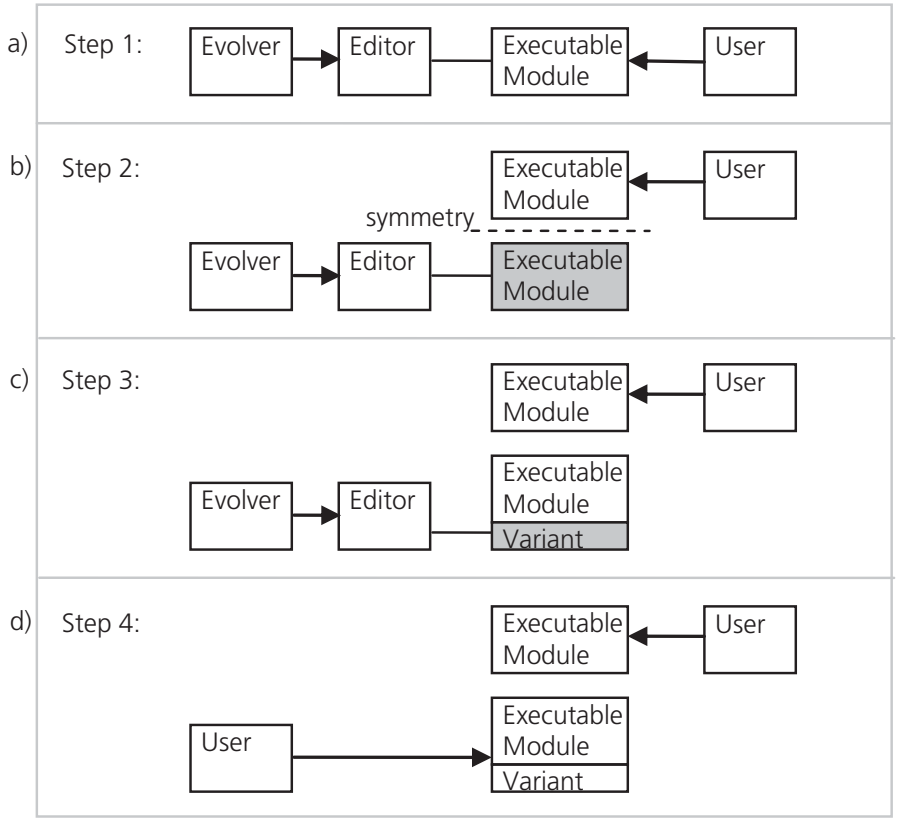


Figure 25: Snapshots of realizing a new variability with Cloning

As shown in Figure 25, a software engineer who needs to evolve an executable module according to his context accesses it through an editor (step 1). The executable module may represent an entire product or a sub-component. The module is also accessed by a previously existing user. In step 2, the evolving agent duplicates the executable module and now refers to it as his local copy. Note that in Figure 25 and the following figures depicting snapshots, elements are highlighted in gray if they have predominantly been changed in the previous step. The module still has considerable similarity to its reference module (shown as a symmetry axis in Fig.25b), but it has gained a new difference (variability) to the original because it now has its own identity, for example a different file name. In step 3, the evolver changes the executable

module, so that a variant builds up. The changes may involve additions, deletions or alterations in the text of the variant. The similarity to the reference module decreases, shown in Fig.25 by the disappearing symmetry axis, and asymmetry increases. In step 4, a new user who might be the evolver himself, accesses the new executable module, while the old user still accesses his copy. There is no explicit coupling between the two copies: they are now seen as if they had no commonalities.

Consequences

Cloning has two main advantages [Kapser+08]: First, working source code can be easily and quickly obtained for a similar context than the required one. This can be particularly useful in situations when new prototype code is created, or when it is hard or uneconomical to refactor the code. Larger mistakes in an evolving cloned module can quickly be undone by deleting the module and cloning the original again. The second advantage is that existing systems are protected from being modified.

One cloning aspect is both advantageous and disadvantageous: The working source code must not be entirely understood in order to make use of it. This is an advantage for a developer because of less cognitive load, but a disadvantage because he gives up control.

Cloning also has clear disadvantages: First, evolution costs, especially in the long run, often increase significantly, as the cloned elements must co-evolve consistently. Synchronization errors are hard to avoid. Common and variant elements are maximally coupled: they become indistinguishable. The original clone group [Kim++05b] becomes untraceable. Second, there is the danger that cloned code which was initially meant to be removed soon is not discarded and persists in the code base. Third, the code will become larger than necessary, and there are risks that obsolete code is propagated.

Details

Evolution. Cloning is frequently conceived to be beneficial for rapid and short-term evolution, as it immediately splits a stable code artifact into two identities (see Section 3.3). However, as these evolve in parallel, Cloning has the long-term risks of inconsistent co-evolution. As new clones emerge, it will become increasingly harder to consolidate them all later. Cloning in code development corresponds to branching in configuration management.

Cloning classifications. Kapser and Godfrey proposed three categories of Cloning [Kapser+08]: Forking, Templating and Customization (see Sec.3.3). Forking (branching) often involves large-scale cloning as in the

motivating example of removing the time transmission variant. Templating examples are boiler-plating due to programming language constraints, API/library protocols which demand functions to be called in a certain order, and language idioms which are used again and again by cloning. Customization includes bug workarounds, and replication and specialization. The latter can be useful in code evolution for preparing code for deprecation.

Kim et al. [Kim++05b] identified, formally characterized and empirically investigated the following evolution patterns associated with Cloning: Same, Add, Subtract, Shift, Consistent and Inconsistent Change. A recent survey on clone detection research is given in [Roy++09].

Symmetries. Cloning is often easily visible in symmetries of source code elements at all levels of granularity, for example similar directory structures and names, similar file names, similar functions, similar algorithms, or similar variable names. This is because Cloning leads to emphasizing common elements. However, as common and variant elements become indistinguishable, it is hard to decide if and where differences exist, and which clone group contains the original, trusted code.

Tools. Several groups of tools have been developed to mitigate the negative long-term effects of Cloning: The Unix tools *diff* and *patch* help in detecting and consolidating inter-module and inter-directory clone pairs and triples. Given two modules, *diff* computes their differences with adjustable graininess, producing compact output in alternative human-readable forms (diff may also produce conditional compilation statements as output, which is an automated refactoring possibility, see Sec.4.5). Using this difference report and one of the compared modules as input, *patch* can reproduce the other. A patch can even be successfully applied if there have been slight changes in the input module, such as the addition or removal of single lines. As mentioned in Sec.3.1, Patching has been classified as a variability mechanism [Linden++07]. As an automated variant of Cloning, it is frequently applied in short-term evolution of open-source software, either for suggesting bug-fixes or for contributing improvements.

Other tools for managing clones are clone detectors which differ in their clone detection algorithms, and which present clones in various ways. One clone detector is *DupLoc* [Ducasse++99] which presents lines of textual clones as dots in a two-dimensional plane, so that lines are produced for successive cloned lines, and both intra- and inter-module clone groups can be detected. Several clone detection approaches have been developed: text-based, token-based, abstract syntax tree (AST) based, program dependence graph (PDG) based, and metrics-based approaches [Bruntink++04, Roy++09].

Related Patterns

Cloning can be considered the archetype of all variability mechanisms. This is because at the very moment cloning is performed, the end result is exactly what the developer wants to achieve by variability management: to tailor existing code exactly to the new development situation, without compromising other code. However, due to the evolution difficulties caused by Cloning, other mechanisms are often required especially in the long run which achieve the same or a similar result than Cloning, while consolidating common elements with moderate extra development effort:

Using **Conditional Compilation** (Sec.4.5) or **Frame Technology** (Sec.4.7), you can customize existing code in such a way that the compiler input becomes indistinguishable from manually cloned code, while variants remain visible in the manually written code. Whereas you cannot store common and variant elements in separate modules if you use Conditional Compilation alone, you can do this when you use Frame Technology or **Module Replacement** (Sec.4.4). However, Module Replacement usually requires each variant element to be extracted into functions, which causes additional refactoring effort. This also leads to compiler input which is no longer identical to cloned code. The same is true when you use **Aspect-Orientation** (Sec.4.6) as a variability mechanism. Like Frame Technology, it requires additional tool support that is not provided by the programming language alone. However, the tool support of Aspect-Orientation is always bound to the programming language, which makes variability management impossible if you use language dialects (e.g. for interrupt service routines in embedded systems C code) or multi- language development (e.g. in C and assembler code). If you use **Conditional Execution** (Sec.4.2) or **Polymorphism** (Sec.4.3), you will even get less similar end results compared to Cloning (e.g. notable resource penalties), although both are applied in similar ways as shown for Conditional Compilation or Module Replacement.

4.2 Conditional Execution

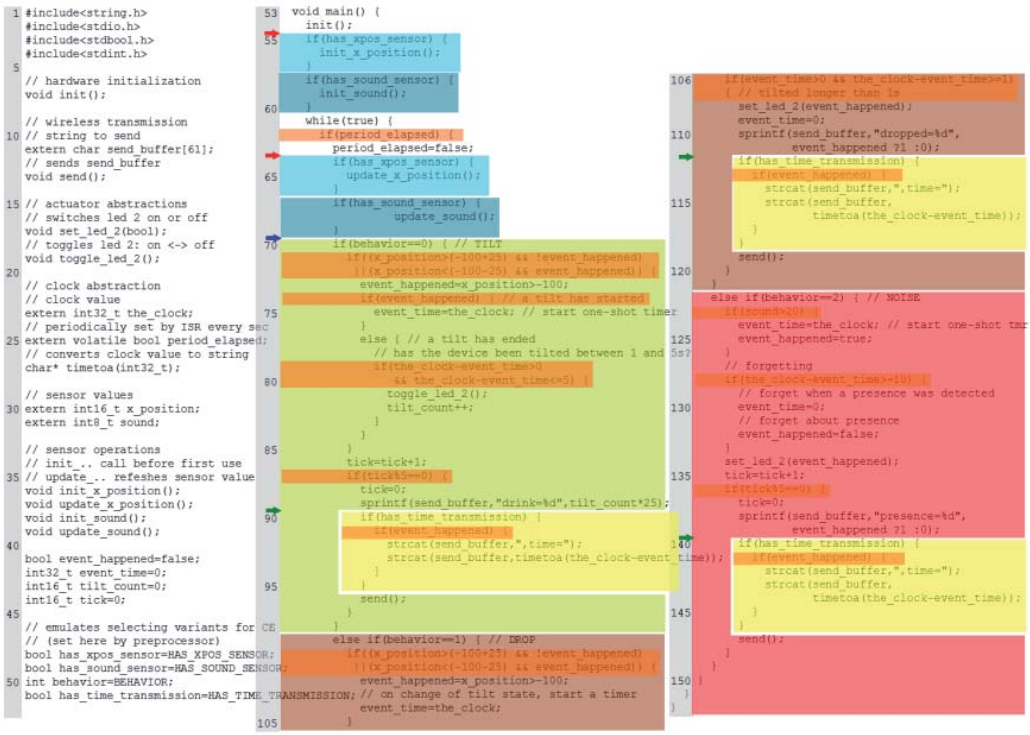
In order to avoid Cloning, you may identify cohesive variants associated with functionality and activate them by conditional programming language statements such as *if* statements. The approach is called Conditional Execution. It is often relatively simple to use, as existing common and variant code elements may remain in their original executable modules. However, the approach is costly to evolve and particularly leads to one single monolithic product realizing all variants, but not to a product line infrastructure that supports mass customization (Def.29) of individualized product line members.

Intent

Separate common from variant algorithmic elements by extracting variant elements into cohesive procedural elements which are conditionally invoked by the common elements, depending on runtime parameter states. Conditional Execution allows you to manage predicted optional or alternative variants, without introducing new modules.

Motivation

Continuing the running example, consider that the realizations of the wireless sensor nodes (Listing 1) are consolidated by Conditional Execution. Listing 3 shows the end result.



Listing 3: Sensor node realization with Conditional Execution

Again, variant elements are color-coded according to Fig.14. The green, brown and red elements are alternative variants of Detector, and the yellow elements are optional variants of Wireless Transmission. The alternative variants of Sensor (bright and dark blue sections) consist of several variant elements, like in the code used for Cloning (Listings 1 and 2). Again, multiple variant elements will likely require more evolution effort than simple variants realized with the same mechanism, but compared to other mechanisms that have realized the respective variation with the same number of variant elements, there is no difference in variability complexity.

Arrows in Listing 3 denote variation points, realized by conditional *if* statements. The orange-colored sections highlight numerous other conditional statements in the code that realize application logic, not product line variation logic. Because the same programming language constructs (*if* statements) have been used for two different purposes, variation points become ambiguous (Tab.5), and it is likely that they may be mixed even more when if statements are consolidated (as in the classical refactoring Consolidate Conditional Expression [Fowler99]). Other problems are that not all variants, such as the variables and forward declarations highlighted in light blue in Listing 1 and 2, can be expressed as variant elements in Conditional Execution, leading to larger common elements than necessary, and that variant elements may become nested and redundant (yellow sections in Listing 3).

Although variation points are ambiguous, they can be seen in the code (see the arrows in Listing 3). Conditional Execution is also a particular way of intentionally realizing variability, which qualifies it as a variability mechanism (Def.64). Note that binding time has never been part of any variability/variation mechanism definition [Jacobson++97, Muthig+02b, Krueger04, Wijnstra04, Bachmann+05, Clements06, Clements+06], and that Conditional Compilation (Sec.4.5), the dual mechanism of Conditional Execution with only an earlier binding time (see Tab.5), is definitively one of the most applied variability mechanisms. The product line literature also agrees in this respect (Conditional Execution has been mentioned as a variability mechanism for example in [Bachmann+05, Svahnberg++05, Krueger07], as discussed in Sec.3.1 and Sec.3.2).

Applicability

Use Conditional Execution

- to consolidate common and variant code, especially when a new optional procedural variant needs to be added, without requiring larger refactorings in advance,
- if an integrated software system is needed with several fixed modes of operation which must be configured after the software development phase, or
- if no element of the source code shall be visible, not even APIs.

Process

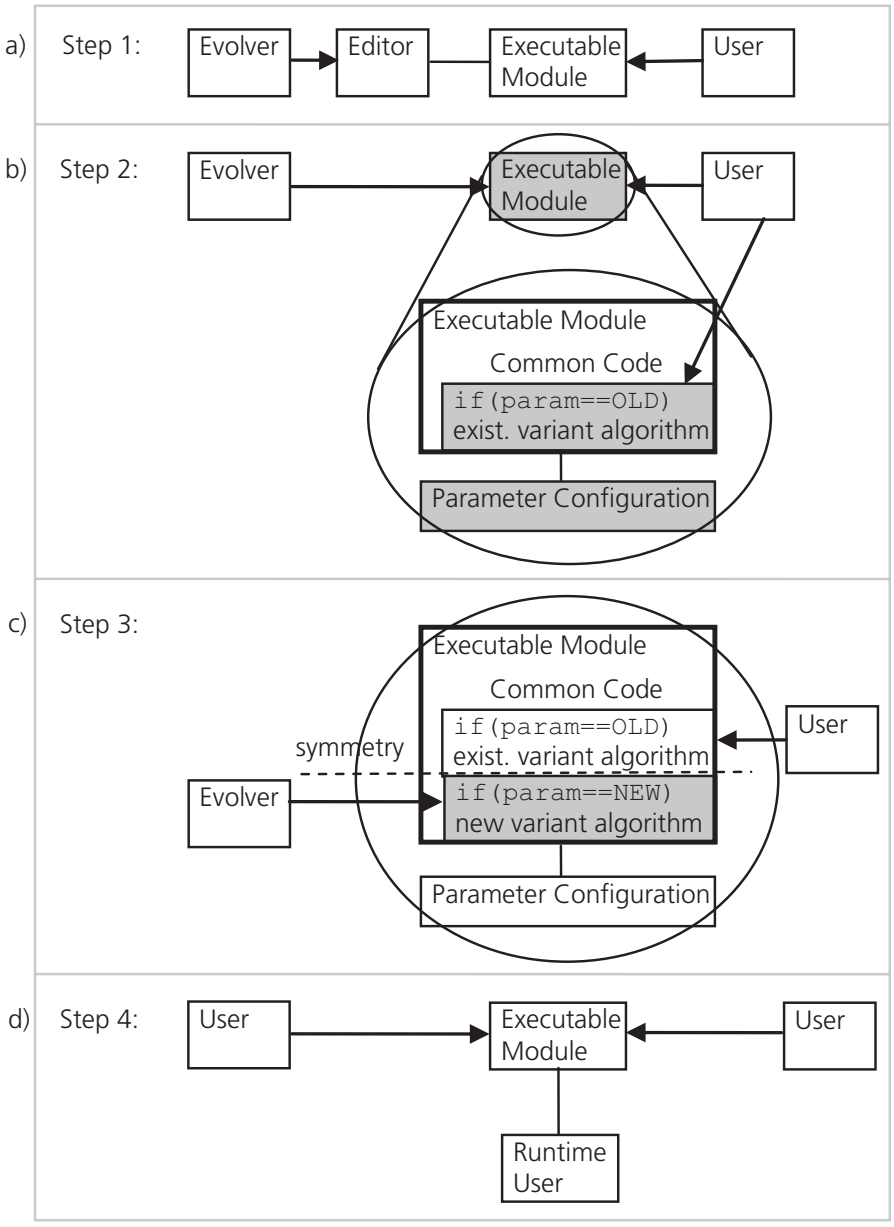


Figure 26: Snapshots of realizing a new alternative variability with Conditional Execution

As Figure 26 illustrates, the first development step is the same as for Cloning (Figure 25a): a software engineer whose role is to evolve the executable module accesses it through an editor tool, while a previous user accesses the module in the existing context. In step 2, the evolver refactors the executable module. This is shown in more detail in the magnified part. The existing variant is extracted as a variant algorithm. This algorithm is enclosed by a condition whose predicate is configured by a new parameter configuration. Taken together, an asymmetry is built up inside the executable module. In step 3, the evolver extends the condition by a new algorithm, activated by a similar predicate. This nearly identical developer activity as in step 2 creates a similar variant

(shown as a symmetry axis as in Fig.25b). In step 4, both the previous and the new developers become users (Def.8) of the executable module, and the variant behavior is configured by a runtime user. Depending on the realized type of parameter configuration, configuration may happen at startup-time, for example by retrieving configuration settings from persistent store (non-volatile RAM, configuration files, or a database), or at runtime. Thereafter, when the consolidated system is executed, the values of these parameters are used to decide which of the variant algorithms to execute.

Consequences

Conditional Execution has two advantages for product line realization: First, some common elements may be used again, rather than duplicating them entirely as in Cloning. Second, Conditional Execution is easy to realize if the variant elements already exist as consolidated algorithms, or if they can easily be consolidated.

With regard to variation points, Conditional Execution has both advantages and disadvantages: The advantage is that, in contrast to Simple Cloning, Conditional Execution realizes non-implicit variation points, so that common and variant elements are somewhat separated. However, variation is closed, so that all evolutionary changes still happen in only one module.

Conditional Execution has four main weaknesses: First, it enforces a realization which contains the subset of all variants, even if they are never used in the specific product. This maximally decreases compilation speed and leads to maximal runtime efficiency penalties. The resulting system only realizes one fixed product instance, not separate products. Second, configuration logic becomes indistinguishable from application-specific functionality because the same language mechanisms are used. It is even possible that the same language conditional contains both code sections for configuration and for application-specific functionality. Cyclomatic complexity increases with each new variant. Unused code remains undetected. Individual products become extremely hard to evolve and test. Third, variants are limited to procedural ones because Conditional Execution depends on conventional programming language semantics. This means that a variant must always contain self-contained algorithms, which often requires additional refactoring effort, for example by applying the classical refactorings Extract Method or Move Method [Fowler99]. Fourth, Conditional Execution does not decouple product line infrastructure code with lower change frequencies from code with higher change frequencies. All code undergoes the same change rates, and no code elements are protected against corruption when others are modified.

Details

Configuration Modules. As mentioned in the process subsection, parameter definitions reside in persistent store. Whereas in IT systems this type of persistence is often realized with configuration files, embedded systems often use non-volatile memory for making configurations persistent. To realize multiple coexisting possibilities in embedded systems, configuration parameters are often realized as bit fields. Frequently, these parameters are not set during software development, but at a later stage in product development (for example when embedding the device in its environment and calibrating it), which is done by re-flashing parts of non-volatile memory. At startup-time, the appropriate behavior is set by using these values. One main disadvantage of this approach is that both memory and runtime resources are wasted, when variables which never change during program execution are treated as if they were conventional variables (see [Bassett97]).

Naming. Naming conventions are often applied to differentiate between conventional variables and configuration variables. For example, the same naming conventions are used as for macros in Conditional Compilation (uppercase, with underscore separation), as illustrated in Listing 3 (HAS_XPOS_SENSOR etc.). As for all naming conventions, ensuring consistency is important, but difficult to enforce. Tools which automate naming consistency checks, such as splint[3], can help you here.

Optimizations. If the number of alternatives exceeds two, *if*-statements should be refactored to *switch*-statements. The number of similar conditional statements may be reduced by nesting conditionals.

Related Patterns

You may use **Conditional Compilation** (Sec.4.5) as an alternative to Conditional Execution in order to conditionally include or exclude algorithms. In addition, Conditional Compilation helps you to manage random variant code elements because it does not rely on programming language semantics. While Conditional Execution enforces runtime binding, Conditional Compilation has construction time binding. Both mechanisms have the disadvantage that they do not help you to extract variants into separate modules, as they only support closed variation. Subtype **Polymorphism** (Sec.4.3) is an alternative runtime binding mechanism to Conditional Execution which allows you to realize open variants. If you require rapid results, you may also consider **Cloning** (Sec.4.1), but then you have to produce the products manually.

[3] www.splint.org (retrieved August 2009)

4.3 Polymorphism

Another possibility to avoid Cloning larger identical elements when you need to realize a new procedural variant is to extract the variant algorithm into functions, to store these in one or more separate variant modules, and to call the variant elements from the common ones indirectly, for example via function pointers in C, or via virtual functions or template parameters in C++. This approach is called Polymorphism [Booch91, Bassett97, Czarnecki+00, p.177]. It is more complicated to realize than Cloning or Conditional Execution because it requires additional refactoring steps and more advanced programming language capabilities. Although it helps you to separate common and variant code elements, it still increases variability complexity to a similar degree than Conditional Execution.

Intent

Decouple common from variant algorithmic elements of product line infrastructure code by extracting variant elements into functions, stored in one or more separate modules, and by calling them indirectly from common code through Template Methods [Gamma++95] or function pointers. Subtype Polymorphism allows you to consolidate common code and add new variants, for example a new alternative, without changing existing common elements.

Motivation

Listing 4 shows a how the running example of a wireless sensor node (Fig.14) can be realized in C by means of the Polymorphism mechanism. For reasons of space, two alternative variants (brown and red elements) have been omitted, but the listing already indicates that all variants are cohesively stored in separate modules.

As before, variant elements have been emphasized by color-coding as in Fig.14, arrows denote variation points, and gray code sections correspond to non-implicit variation point realizations. As in Conditional Execution, the Sensor variation is realized at two variation points (red arrows in Listing 4), which makes it more complex to evolve within the core asset main.c than the Detection variant, realized at a single variation point (blue arrow in Listing 4) within the same core asset using the same mechanism. But again, evolution complexity among variability mechanisms is independent of variation point multiplicity. What really makes a difference is variation point explicitness.

Listing 4: Sensor node realization with Polymorphism

In contrast to Conditional Execution (Listing 3), Polymorphism allows all variants to be stored in separate modules, but the variants must be complete algorithms again. The realization of optional variation (yellow section) requires a second module (send.c in Listing 4) for the "empty" variant. Variation points are realized by invoking function pointers which have been initialized at startup time (not completely shown here). Function pointers may also realize application functionality, in which case the realized variation points become ambiguous, but to a smaller degree compared to Conditional Execution.

As mentioned in Sec.3.1 and 3.2, product line engineering literature has often classified Polymorphism as a variability mechanism, often in the form of plug-ins [Jacobson++97; Bachmann+05; Pohl++05, p.253; Linden++07].

Applicability

Use Polymorphism

- as an alternative to Conditional Execution, in order to separate common and variant code in distinct modules,
- to obtain a software system whose existing variants can be replaced by similar ones without changing common and existing variant elements, or
- to facilitate the parallel evolution of alternative variant modules.

Process

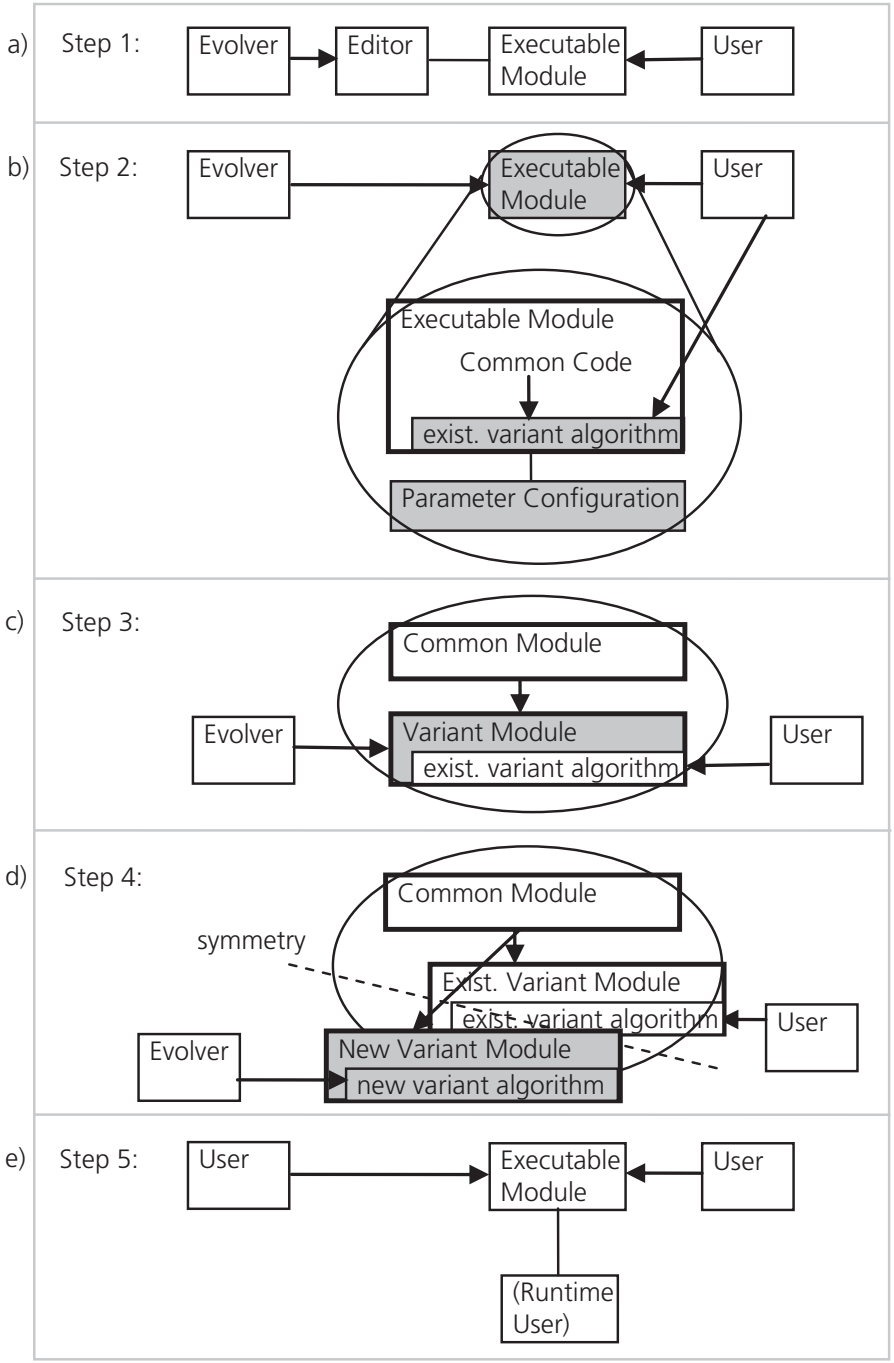


Figure 27: Snapshots of realizing a new alternative variability with Polymorphism

As shown in Figure 27, the first development step is the same as for Conditional Execution (cf. Figure 26a). The second step is similar, in that an existing variant algorithm is extracted from the common code, and a configuration is established. However, in contrast to Conditional Execution, the variant element is referenced indirectly, which is indicated by the arrow from the common to the variant element in Figure 27. In the third step, a variant module is created, and the variant algorithm is

moved there. This corresponds to a split of the executable module from step 2 into a common and a variant module. In step 4, a new variant module is created that is similar to the existing one, as indicated by the symmetry axis (Fig. 27d), as was the case for the two variant elements in the third step of Conditional Execution (Figure 26c). The fifth step for both mechanisms is identical.

Consequences

Polymorphism has three main advantages: First, some common elements may be used again, rather than duplicating them entirely, as in Cloning. Second, common elements are partially decoupled from variant ones, so that both can evolve in isolation, as long as their interface does not change. Common elements and variants always form a clearly visible contrast because they reside in different modules. Third, alternative variants may be isolated from each other, so that they can evolve in parallel (see Fig.10b).

Polymorphism has the following advantageous and disadvantageous property: Like Conditional Execution, it has the advantage of non-implicit variation points, but due to the additional level of indirection in Polymorphism, using these variation points tends to be harder.

Polymorphism has five main types of disadvantages: First, it does not support variants of arbitrary granularity, but enforces variant elements to be medium-sized, forming functions. As in the case of Conditional Execution, this requires additional refactoring effort in many cases. Second, optional variabilities are harder to realize with Polymorphism because additional empty functions must be provided to support the missing variant elements. Third, Polymorphism usually has efficiency penalties, which in the case of Subtype Polymorphism are even more severe than for Conditional Execution. Fourth, the distinction between configuration logic and application logic is blurred when the same language mechanisms are used for both. Fifth, when polymorphism is realized with function pointers, it increases the risk of software defects because errors due to illegal pointer references cannot be ruled out. This is why some industrial embedded systems standards such as MISRA[4] disallow the usage of function pointers.

Details

Binding Time. As a variation mechanism for embedded systems, three different types of Polymorphism are in wider use: Subtype Polymorphism, Parametric Polymorphism and Overloading. With the more frequently applied mechanism of Subtype Polymorphism,

[4] www.misra.org.uk (retrieved August 2009)

configuration happens at runtime (see Figure 27e), whereas Parametric Polymorphism binds at compile time. Subtype Polymorphism is either realized with function pointers, as shown above, or Template Methods [Gamma++95]. For mainstream development, the C and C++ programming languages both support the former, whereas the latter is only available in C++. Parametric Polymorphism is realized with C++ templates [Czarnecki+00]. In programming languages such as C that do not support Overloading, this mechanism is often realized by naming the functions in a similar way. An example is the group of *printf* functions in the C standard library (printf, sprint, vprintf, fprintf).

Evolution. Polymorphism facilitates adding new alternative variants because of its open variation. However, removing unwanted common elements is often more difficult than adding new variants. This is because adhering to the Open-Closed-Principle [Martin02] or the Liskov Substitution Principle [Liskov+94] may be enforced by the programming language (e.g. in Java or Python, but not in C++).

Defaults. The Null Object pattern [Woolf98] discusses various possibilities to realize defaults by means of Polymorphism.

Design Patterns. Several of the behavioral design patterns mentioned in [Gamma++95] rely on Polymorphism in their realization, most notably Strategy and Template Method. They are usually realized with Subtype Polymorphism, but can also be realized with Parametric Polymorphism, as shown in [Czarnecki+00, pp.229ff., p.234, p.287; Duret++01, Alexandrescu01].

Known Uses. Many schedulers in embedded systems operating systems use Subtype Polymorphism to decouple the scheduler realization from user (Def.8) code. For example, [Pont01] shows how a cooperative scheduler for an embedded operating system for the *8051* processor can be realized with function pointers. Similarly, the real-time operating system *µC/OS-II* [Labrosse02] executes tasks in its scheduler, which are referenced via function pointers. Another typical use of Polymorphism is the realization of the Model-View-Controller pattern [Buschmann++96] in order to decouple user interfaces from application logic, as both may have different evolution rates.

Related Patterns

Both **Module Replacement** (Sec.4.4) and Polymorphism require you to use syntax elements of the programming language to decouple common code from variant elements. However, the two mechanisms often differ in binding time. Module Replacement usually has earlier binding than Polymorphism and should be preferred if binding at runtime or startup time is not required. However, if such binding is required, you may

consider using **Conditional Execution** (Sec.4.2) as a simpler alternative, but this will result in stronger coupling of common and variant elements because of the missing polymorphic interface. If you require open variation, **Frame Technology** (Sec.4.7) may be an alternative, as unlike Polymorphism it also supports variant optimization possibilities due to Default support. As a construction-time mechanism it does not suffer from efficiency penalties, and it allows you to realize variants that do not have to be self-contained programming language elements.

4.4 Module Replacement

As an alternative to Polymorphism, when common and procedural variant elements are stored in separate modules, you may also call the variant elements from the common ones directly, and let the preprocessor, compiler or linker bind them. Variation is then achieved by replacing one variant module with an alternative one, which is why this variability mechanism is called Module Replacement.

Intent

Decouple common from variant algorithmic elements of product line infrastructure code by extracting variant elements into functions, stored in one or more separate modules, and by calling them directly in common code. Module replacement allows you to consolidate common code and add new variants without runtime penalties and without changing existing common elements.

Motivation

The running example of a wireless sensor node product line, realized with Module Replacement, is shown in Listing 5. For reasons of space, two alternative variants (brown and red elements) have been omitted, and variants are cohesively stored in separate modules.

main.c:

```
#include<stdbool.h>

// hardware initialization
void init();

// clock abstraction part
// periodically set by ISR every second
extern volatile bool period_elapsed;

// the following functions serve as VPs
void post_init(); // for sensor init.
void pre_update(); // for sensor update
void update(); // no default possible!

void main() {
  init();
  post_init();
  while(true) {
    if(period_elapsed) {
      period_elapsed=false;
      pre_update();
      update();
    }
  }
}
```

xpos_sensor.c

```
// sensor operations
void init_x_position();
void update_x_position();

void post_init() {
  init_x_position();
}

void pre_update() {
  update_x_position();
}
```

sound_sensor.c:

```
// sensor operations
void init_sound();
void update_sound();

void post_init() {
  init_sound();
}

void pre_update() {
  update_sound();
}
```

tilt_detection.c:

```
#include<stdint.h>
#include<stdbool.h>
#include<stdio.h>

// tilt detection behavior

// wireless transmission and reception
// string to send
extern char send_buffer[61];

// sensor values
extern int16_t x_position;

// actuator abstractions
// toggles led 2: on <-> off
void toggle_led_2();

bool event_happened=false;
int32_t event_time=0;
int16_t tilt_count=0;
int16_t tick=0;

void send_wrapper();

void update() {
  if((x_position>(-100+25) && !event_happened)
   ||(x_position<(-100-25) && event_happened)) {
    event_happened=x_position>-100;
    if(event_happened) { // a tilt has started
      event_time=the_clock; // start one-shot tmr.
    }
    else { // a tilt has ended
      // has the device been tilted btw. 1 and 5s?
      if(the_clock-event_time>0 &&
         the_clock-event_time<=5) {
        toggle_led_2();
        tilt_count++;
      }
    }
  }
  tick=tick+1;
  if(tick%5==0) {
    tick=0;
    sprintf(send_buffer,"drink=%d",tilt_count*25);
    send_wrapper();
  }
}
```

send.c:

```
void send();

void send_wrapper() {
  send();
}
```

send_time.c:

```
#include<string.h>
#include<stdbool.h>
#include<stdint.h>

// wireless transmission and reception
// string to send
extern char send_buffer[61];
// sends send_buffer
void send();

// clock abstraction
extern int32_t the_clock;
char* timetoa(int32_t);

extern bool event_happened;
int32_t event_time=0;

void send_wrapper() {
  if(event_happened) {
    strcat(send_buffer,",time=");
    strcat(send_buffer,
           timetoa(the_clock-event_time));
  }
  send();
}
```

Listing 5: Sensor node realization with Module Replacement

Variant elements are color-coded as in Fig.14, arrows correspond to variation points, gray code sections highlight non-implicit variation points, and orange-colored sections highlight other elements in the code that could be mistaken for variation points. In contrast to Polymorphism (Listing 4), no function pointer definitions, initializations and usages are necessary. If function pointers are exclusively used for variability management as in Listing 4, no ambiguities with regard to variation points can arise, which is easy to achieve. However, if direct calls are used for variability management, such ambiguities are hard to avoid, as any function call seen by the family engineer may potentially be related to variability management (the orange colored sections in Listing 5), which results in a similar degree of ambiguity as in Conditional Execution (Listing 3 also contains various orange colored elements).

Again, one optional and one alternative variability is realized with cohesive variants (yellow and green elements), referring to a single variation point each (green and blue arrow), while another alternative variant refers to two variation points (red arrows). Again, the variability complexity in core asset code compared to other variability mechanisms is independent of variation point cardinality.

Module Replacement represents the traditional composition mechanism and has been ranked as a variability mechanism in the reuse and product line architecture literature [Krueger92, Bosch00, Bachmann+05, Svahnberg++05, Linden++07], as mentioned in Sec.3.1 and 3.2. In [Bachmann+05], the mechanism is called Component Substitution, whereas in [Linden++07] it is called Component Replacement. The concept of a component denotes a constructible module in this thesis (see Def.26), which is different to the component concept used in [Bachmann+05, Linden++07], where it is an executable module (Def.4). In order to avoid confusion, I use the more general term ‘module’, not ‘component’, in the name of this pattern.

Applicability

Use Module Replacement

- as an alternative to Polymorphism, also in order to separate common and variant code in distinct modules, but with less effort,
- to obtain a software system whose existing variants can be replaced by similar ones without changing common and existing variant elements,
- to realize larger behavioral variations without affecting runtime performance or memory size,
- to decouple common modules under the developer's ownership from other common and variant modules which are not under his ownership, for example in preparation to replace 3rd-party code, or
- to facilitate the parallel evolution of alternative variant modules.

Process

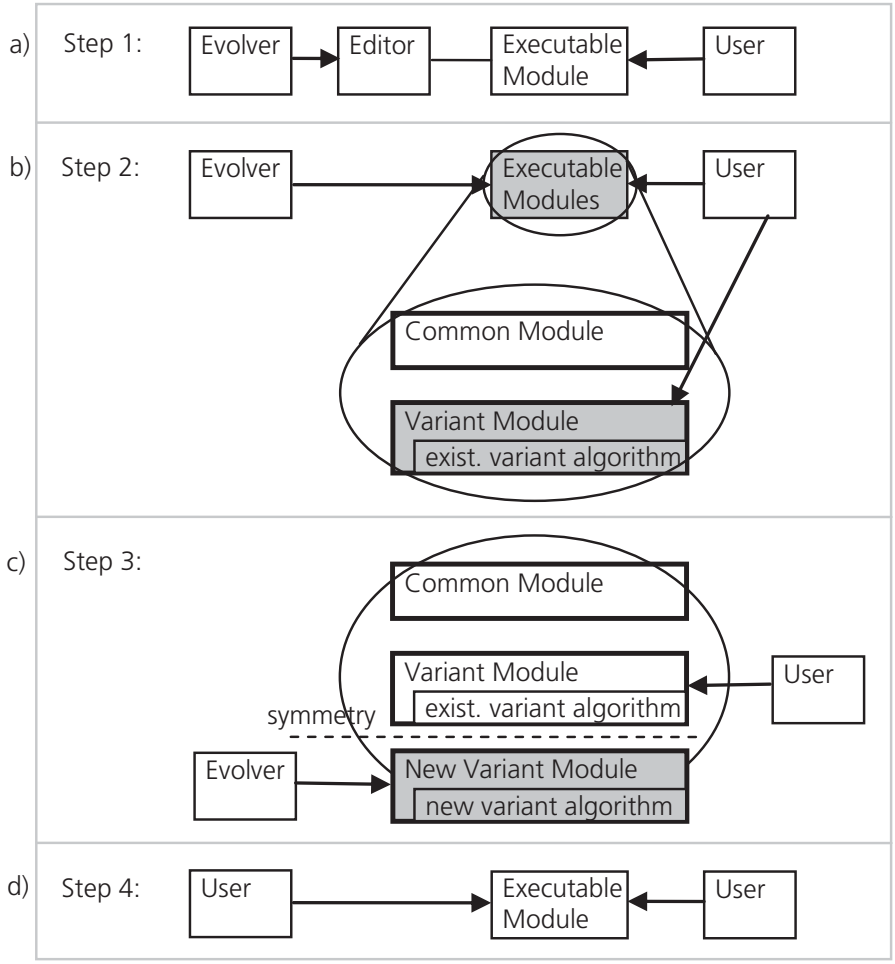


Figure 28: Snapshots of realizing a new alternative variability with Module Replacement

As Figure 28 illustrates, the initial evolution step is indistinguishable from the first step for Polymorphism (Figure 27a). In step 2, the existing variant algorithm is extracted which may be an empty function in the case of a newly introduced optional variability. The existing executable module is split. The common module only refers indirectly to variant elements: as a realization of a software product, it is only partially complete. The missing element is configured by selecting the variant module for compilation. In contrast to Polymorphism, the connection of common and variant elements is more implicit because no direct programming language capabilities such as function pointers are used to couple the two. This step is similar to the combined second and third step in the introduction of Polymorphism (Figure 27b and c). A difference is that configuration tends to be easier because it is not done in the code but in the build process.

In a third step, the evolver realizes the newly required feature as a function in a new variant module. This module is similar to the existing variant module, indicated by the symmetry axis in Fig.28c. In fact, the

new module may be created by cloning. The final step results in the desired setting, in which different users (Def.8) access the same executable module as part of the product line infrastructure, rather than accessing separate copies (compare Fig.28d to Fig.25d).

Consequences

Module Replacement has four main advantages: First, like Conditional Execution, it is a well-known mechanism that is easy to realize if the variant elements can be refactored into functions. Second, it consolidates common elements and decouples them from variants. Third, it decouples variants from each other, so that they may evolve in isolation, as long as the interface of the respective common elements does not change. Fourth, source code realizing alternative functionality becomes easy to exchange, without runtime efficiency penalties.

Two consequences of Module Replacement are both positive and negative: First, it supports open variation, but only at a mid-sized level of granularity. Although the variant elements tend to be easier to use and evolve than those of Polymorphism, they must all be in separate modules in case of Module Replacement, while Polymorphism also allows them to share a single module. Second, variant elements are mostly restricted to functions because Module Replacement depends on compile-time semantics. For example, a variant module usually cannot contain partial functions.

Module Replacement has the following main disadvantages: First and foremost, defaults cannot be realized by Module Replacement alone because it requires a strict separation of common and variant elements, with no intermediate gradients, as offered by reuse hierarchies. For the same reason, negative variabilities are also unsupported. Second, as in Polymorphism, an extra empty function must be provided in order to support optional variability. However, unlike in Polymorphism, this empty function cannot reside together with its sibling because both must share the identical function signature in Module Replacement. Third, variation points are not entirely visible in the core assets because they are represented by normal function calls.

Details

Binding Time. Module Replacement can be realized at preprocessing-, compile- or link-time. In C and C++, for example, a common module may specify (via an `#include` statement) which variant module realizes a missing functionality. Alternative variants of that module may exist in different directories, and only during preprocessing the required alternative is selected by specifying the include path through the

compiler's `-I` option. Similarly, link paths to precompiled variant modules are specified through the `-L` option.

However, the later the binding, the less adaptation possibilities exist for the involved modules. For example, at preprocessing time, the common code may still be altered at arbitrary variation point locations using Conditional Compilation or include path adjustments, while at compilation-time only alternative source code modules are selectable whose source code, however, is still visible. At link-time, this is often not the case anymore, so that the participating object code modules are totally closed against modification.

Alternative Selection Options. There are several alternatives to adjusting include or link paths. The most common ones in C and C++, as described above, are the `-I` and `-L` compiler options. Instead of selecting include or link paths by compiler options, they may also be adjusted through symbolic links, if supported by the operating system of the development machine. Variant modules may also be distinguished by their name alone, rather than by the directory they reside in. In case of linking, the module name is specified by the `-l` option (rather than the `-L` option for the directory). A corresponding dual option to `-I` does not exist. However, at least the *GNU cpp* preprocessor offers a corresponding capability called computed inclusion: instead of specifying the file name to include, a macro may be provided, as in `#include MACRO_H`. Alternative modules may be included this way by redefining the macro name.

Known Uses. Module Replacement is often used for selecting among larger subsystems. In C, these subsystems are realized as identically-named .c and .h files which are stored in sibling directories. An entire subsystem is selected for compilation by providing the respective directory, as shown above. For example, Module Replacement is used internally by the *SDCC* compiler which targets a large number of different microprocessor types, such as Zilog Z80 or Microchip PIC. Hardware-specific functionalities are offered in several executable modules in sibling directories, for example in z80 for the *Z80* processor or in pic16 for the *PIC16* processor. By providing the option "-I pic16", the preprocessor includes the header files for the PIC16 processor, not for the Z80. Similarly, the respective libraries are selected through the "-L pic16" option.

Related Patterns

Only if runtime binding is a must, while open variants are desired, use **Polymorphism** (Sec.4.3) instead of Module Replacement. Because both of these mechanisms lead to similar variants (compare the colored elements in Listing 4 with those in Listing 5), refactoring effort among

these mechanisms can be low. However, Polymorphism results in extra core asset complexity due to its runtime configuration and indirect calls. As mentioned in the Process section, you may use **Cloning** (Sec.4.1) as a sub-mechanism of Module Replacement if you have to realize a new alternative variant. Instead of using the preprocessor flavor of Module Replacement, you may opt for **Conditional Compilation** (Sec.4.5), especially if open variation is not necessarily required, as in the case of optional variabilities. **Aspect-Orientation** (Sec.4.6) offers similar variability management possibilities than Module Replacement. It may be a viable alternative in cases when the same variations in functionality must augment or replace multiple different common functionalities. You may apply **Frame Technology** (Sec.4.7) instead of Module Replacement if you require more explicit variation points, wide variation granularity, Default support, or programming language independence.

4.5 Conditional Compilation

In order to avoid Cloning, especially when you use a programming language such as C or C++ which has a built-in preprocessor, you may embed variant product line infrastructure code elements of arbitrary meaning, such as (partial) modules, data structures, or algorithms, in conditional preprocessor statements, such as `#ifdef`s or `#if`s. You can optionally activate or deactivate the variants at construction time by providing appropriate preprocessor macros. This mechanism is called Conditional Compilation. It is simple to use, as existing code may usually remain in its original position in the module, and it is more versatile than its dual conditional mechanism, Conditional Execution. In particular, efficiency penalties do not exist for the resulting product line members.

Intent

Separate common from variant code in a product line infrastructure by extracting variant textual elements into cohesive elements which are conditionally enabled or removed at construction time, depending on preprocessor settings. Conditional Compilation allows you to manage predicted optional or alternative textual variants, without introducing new modules.

Motivation

Listing 6 shows how the wireless sensor node product line from the running example is realized using Conditional Compilation.

```
#include<string.h>
#include<stdio.h>
#include<stdbool.h>
#include<stdint.h>

// hardware initialization
void init();

// wireless transmission
// string to send
extern char send_buffer[61];
// sends send_buffer
void send();

// actuator abstractions
// switches led 2 on or off
void set_led_2(bool);
// toggles led 2: on <-> off
void toggle_led_2();

// clock abstraction
// clock value
extern int32_t the_clock;
// periodically set by ISR every sec
extern volatile bool period_elapsed;
// converts clock value to string
char* timetoa(int32_t);

// sensor values
#if HAS_X_POS_SENSOR
extern int16_t x_position;
#endif
#if HAS_SOUND_SENSOR
extern int8_t sound;
#endif

// sensor operations
// init_.. call before first use
// update_.. refeshes sensor value
#if HAS_X_POS_SENSOR
void init_x_position();
void update_x_position();
#endif
#if HAS_SOUND_SENSOR
void init_sound();
void update_sound();
#endif
```

```
bool event_happened=false;
int32_t event_time=0;
int16_t tilt_count=0;
int16_t tick=0;

void main() {
  init();
#if HAS_X_POS_SENSOR
  init_x_position();
#endif
#if HAS_SOUND_SENSOR
  init_sound();
#endif
  while(true) {
    if(period_elapsed) {
      period_elapsed=false;
#if HAS_X_POS_SENSOR
      update_x_position();
#endif
#if HAS_SOUND_SENSOR
      update_sound();
#endif
#if TILT_DETECTOR
      if((x_position>(-100+25) && !event_happened)
        ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        if(event_happened) { // a tilt has started
          event_time=the_clock; // start one-shot tmr
        }
        else { // a tilt has ended
          // has the device been tilted btw 1 and 5s?
          if(the_clock-event_time>0
             && the_clock-event_time<=5) {
            toggle_led_2();
            tilt_count++;
          }
        }
      }
      tick=tick+1;
      if(tick%5==0) {
        tick=0;
        sprintf(send_buffer,"drink=%d",tilt_count*25);
#elif DROP_DETECTOR
      if((x_position>(-100+25) && !event_happened)
        ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        // on change of tilt state, start a timer
        event_time=the_clock;
```

```
      }
      if(event_time>0 && the_clock-event_time>=1)
      { // tilted >1s
        set_led_2(event_happened);
        event_time=0;
        sprintf(send_buffer,"dropped=%d",
                event_happened ?1 :0);
#elif NOISE_DETECTOR
      if(sound>20) {
        event_time=the_clock; //start one-shot t.
        event_happened=true;
      }

      // forgetting
      if(the_clock-event_time>=10) {
        // forget when a presence was detected
        event_time=0;
        // forget about presence
        event_happened=false;
      }
      set_led_2(event_happened);
      tick=tick+1;
      if(tick%5==0) {
        tick=0;
        sprintf(send_buffer,"presence=%d",
                event_happened ?1 :0);
#endif
#if HAS_TIME_TRANSMISSION
        if(event_happened) {
          strcat(send_buffer,",time=");
          strcat(send_buffer,
                 timetoa(the_clock-event_time));
        }
#endif
        send();
      }
    }
  }
}
```

Listing 6: Sensor node realization with Conditional Compilation

The variant elements are shown in the same colors as in the requirements and architecture documents from Fig.14. The optional variation for extra wireless transmission is realized by the code marked in the cohesive yellow block, the alternative variation for the three different modes of detection is realized in the three cohesive green, brown and red blocks, and the two alternatives for realizing different sensors are shown in four non-cohesive bright and dark blue sections. Compared with each other, the optional variant is least complex because it only consists of a single cohesive section of code at one variation point (marked by the green arrow), and the alternative variant consisting of three cohesive variants at one variation point (blue arrow) is slightly more complex. The realization of the two alternatives is most complex, as it results in two pairs of variant elements at four variation points (red arrows). Assuming that the same number of variants is realized in each variability mechanism, there is no difference in complexity between the realization of the four Sensor variants (bright blue) in Conditional Compilation and, for example, Cloning (Fig.1): each time, there are four variation points (red arrows), and changing (e.g. adding) a variant requires changes at these four places. What is different, however, is the explicitness of these positions. So it would as well have been sufficient to discuss just the situation of a single variation point in order to clarify complexity issues among variability mechanisms.

Conditional Compilation leads to explicit variation points, as indicated by the gray code sections. If this mechanism is exclusively used for variability management purposes, as mostly seen in practice, variation points are also non-ambiguous.

Applicability

Use Conditional Compilation

- to consolidate common and variant code, especially if the end result must be a new optional variant and if refactorings of existing product line infrastructure code shall be minimized,
- in cases when extraction of functions as variant elements is infeasible or requires too much refactoring effort,
- if efficiency penalties due to variability management must be avoided,
- if variant elements of large and small sizes must be managed together, or
- if variation points shall be visible in core asset code.

Process

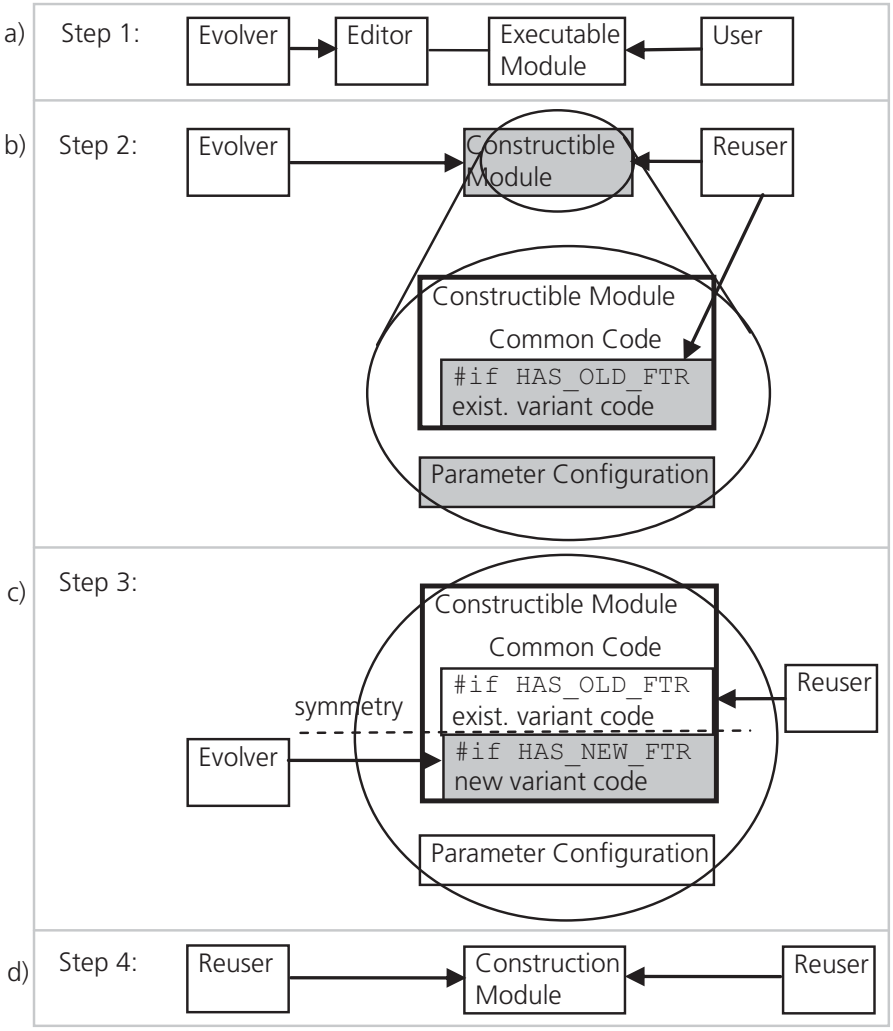


Figure 29: Snapshots of realizing a new alternative variability with Conditional Compilation

As shown in Figure 29, the development steps are very similar to those of Conditional Execution (Figure 26), and so is their result. First, the developer in the evolver role accesses the executable module, while the existing user (Def.8) also accesses it. In step 2, the executable module is changed most minimally, by embedding the existing variant elements in `#ifdef` statements, so that the module becomes a constructible module. A parameter configuration is also created. But in contrast to Conditional Execution, the module does not need to be coupled to this parameter configuration (comp. Fig.29b to Fig.26b). In step 3, a new variant element is created alongside the existing one. While the previous reuser is still free to modify his variant element, the evolver realizes the new element. In step 4, both developers act as reusers because they may adapt the constructible module to their needs.

Consequences

Conditional Compilation has the following main advantages: First, common elements are consolidated in one place, rather than duplicated, as in Cloning. Second, common code is somewhat decoupled from variant code, which is emphasized by the condition. Third, the mechanism is easy to introduce because it is only concerned with textual elements, independent of programming language semantics. In particular, variant code does not have to form a cohesive procedural element, which has been necessary in all variability mechanisms mentioned so far, except for Cloning. The fourth advantage is that Conditional Compilation does not lead to efficiency penalties in the resulting machine code. In fact, its code becomes indistinguishable from cloned code after construction time.

Conditional Compilation has one slightly negative characteristic. It can be used to express defaults, but these can only be overridden once, and not multiple times, as in Frame Technology (see the following Details section and the discussion on Default Addition in Sec.5.1).

Conditional Compilation has three drawbacks: First, it leads to core assets that contain both common and variant code elements, so the two cannot evolve independently. Common code details cannot be hidden from application engineers who shall not see them. Second, as a closed variability mechanism, Conditional Compilation does not support unpredicted changes that leave the existing module unchanged. Third, it becomes harder to ensure that the entire code – all common and variant (possibly nested) elements in meaningful combinations – is always compilable.

Details

Macro Definition. Conditional Compilation is realized in the C preprocessor by using macro parameters in the code, which are defined elsewhere. There are multiple ways to define or use conditional macros, which may lead to inconsistencies. Each preprocessor macro is realized as a key/value mapping. A macro can either be defined by specifying its name only, which sets the key to the macro name and the value to 1, or both the key and value may be set explicitly. Moreover, macros may either be set when invoking the preprocessor/compiler (`-D` option), either manually of from build scripts, such as a Makefile, or the macros may be defined within source code files using the `#define` statement. This results in four possibilities to set a macro, which already endangers consistency. Conversely, one can also specify a macro to be undefined (all but the built-in macros are undefined by default). This may be meaningful to override a previous macro definition. Again, undefining a macro can either be done as a command-line compiler option (`-U`), or in

a source code file (`#undef`). However, as opposed to defining a macro, undefining does not set the macro value to 0 (this must be done by defining it with the explicit 0 value), but it removes the key/value pair. This inconsistency is sometimes the reason for realization errors. Another problem arises when macros are defined in configuration header files rather than in a Makefile: All source code files tend to depend on the configuration header, and when that header is changed, the entire software system must be recompiled, which may drastically decrease compilation speed. A third problem with macro definitions is that feature interactions are often realized as explicit macro dependencies, nested configuration headers or nested Makefiles. In order to keep Conditional Compilation macro definitions simple, as few different possibilities for defining macros shall be used, and they must always be used consistently. For very simple products, it can make sense to omit the configuration module completely, so that configuration is manually done on direct compiler invocation.

Macro Usage. Conditional Compilation depends on macro definition and macro usage. As for macro definition, there are also various possibilities for conditional macro usage: Optional variabilities are either realized by `#ifdef`s or `#if`s, `#ifndef`s, nesting these clauses, or by combining several conditions using Boolean logic. Again, as few of these possibilities shall be used in the same product line infrastructure code in order to keep it simple. The `#ifdef` clause (or alternatively, `#if defined`), checks whether the macro has been defined, a Boolean choice. This may also be expressed explicitly with `#if MACRO_NAME==1`. Conversely, macro absence may be checked with `#ifndef` or `#if !defined`, but not with `#if MACRO_NAME==0`. Sometimes, the aliases YES or NO are defined for the values 1 and 0.

Alternative variabilities are realized by `#else` or `#elif` clauses, or by successive `#if MACRO_NAME==n` statements. The extreme options of unconditionally excluding source code elements (with `#if 0`) or including it (with `#if 1`) must be avoided, even for temporary code elimination or activation. Version management systems are a better choice to keep such code elements available.

Default Overriding. The goal of defaults (Def.55) is to simplify variability management. Frame Technology has good Default support, but it can also be realized with Conditional Compilation, as illustrated in Listing 7.

```
1  #ifndef HAS_ACKNOWLEDGE
   #define HAS_ACKNOWLEDGE 1
   #endif

    ...
5   void send(char* message) {
      initialize transmission
    #if HAS_ACKNOWLEDGE==1
      acknowledged=false;
    #endif
10  ...
      }
    }

   #undef HAS_ACKNOWLEDGE
```

Listing 7: Realizing defaults with Conditional Compilation

An optional variant has been realized in lines 7-9, so that line 8 is activated if the macro `HAS_ACKNOWLEDGE` is set. The macro is set in line 2, activating line 8 by default. The important point is that, as mentioned in Def.55, the configuration option, realized by the macro `HAS_ACKNOWLEDGE`, may be ignored in the default case, but if it is reset to 0, the default is overridden and line 8 vanishes. Note that this language-specific pattern I invented circumvents the C preprocessor inconsistency mentioned in the Macro Definition section above. Also note that, as opposed to Default Overriding in Frame Technology (Sec.4.7), Default Overriding in Conditional Compilation can only be applied once: overriding for a second time is not possible.

Evolution. Besides evolution in space, as supported by Default Overriding, short-term evolution in time can also be realized with Conditional Compilation, in a similar manner as in the versioning idiom known in Frame Technology [Bassett97, pp.182f.].

Tools. Besides the C preprocessor, several popular open-source tools support Conditional Compilation, for example *ifnames*, *diff, m4,* or *javapp*.

Ifnames scans all of the source code files named on the command line and emits a sorted list of all macro usages. It can either be used to detect which macros are used in a given set of source code modules, or to detect which modules are affected by a particular crosscutting feature. *Ifnames* is part of *GNU autotools*[5].

Diff is a program to compare file and directory contents and to output the differences in various ways. The output is usually a list of annotated differences. *GNU diff*[6] offers an option (-D) which results in an output

[5] www.gnu.org/software/autoconf (retrieved August 2009)

[6] www.gnu.org/software/diffutils (retrieved August 2009)

format in which differences are surrounded by #ifdef statements. This capability results in a simple automated refactoring possibility from Cloning to Conditional Compilation.

M4[7] is an advanced preprocessor which implements the traditional UNIX macro preprocessor, and which also realizes the *autotools* scripts. It shares many C preprocessor capabilities, such as file inclusion, defining macros and evaluating conditions on macros, so that *M4* can be also be used for Conditional Compilation. In addition, *M4* also supports shell command execution, string handling, integer arithmetic and iteration. The iteration capability is valuable for realizing variants of the range type, for example, when a variant source code element, possibly with minor sub-variations, has to be reused repeatedly, and the number of repetitions is known at construction-time.

Javapp[8] is a preprocessor for Java that supports Conditional Compilation.

Language-specific Alternatives. Some programming languages without a built-in preprocessor have alternative possibilities for realizing a restricted form of Conditional Compilation. For example, *if*- conditions on *final boolean* values in the Java programming language are optimized during compilation, as if they had been *#ifdef* statements in C. The same possibility is also offered by the *static if* statement in the D programming language[9].

Known Uses. Conditional Compilation is frequently used to manage optional variabilities in conventional reusable code. For example, the real-time operating system *µC/OS-II* [Labrosse02] makes extensive use of Conditional Compilation for various purposes. It uses a consistent naming convention for macros that enable optional code elements: these macros end with _EN (for enable; this is similar to our convention using the HAS_ prefix). For example, the macro OS_DEBUG_EN enables debugging code, and OS_EVENT_EN enables event code.

An idiom in many open-source C realizations, such as *Emacs*, *GCC*, *glibc* or *uClibc*, is to have an optional configuration header config.h which is used when the macro HAVE_CONFIG_H is defined. A large Japanese manufacturer uses Conditional Compilation for realizing optional features in their digital camera code, for example to provide anti-shake or GPS support.

[7] www.gnu.org/software/m4 (retrieved August 2009)

[8] www.slashdev.ca/javapp (retrieved August 2009)

[9] www.digitalmars.com/d (retrieved August 2009)

Related Patterns

You may use **Conditional Execution** (Sec.4.2), the dual runtime mechanism to Conditional Compilation, if runtime binding is required. However, this results in efficiency penalties and makes the code less comprehensible, as application logic and portfolio variation logic are mixed. In order to realize larger alternative variants, consider using **Module Replacement** (Sec.4.4), especially in cases when some alternative variants already exist and new ones are likely to be required soon. However, Module Replacement usually restricts variants to be functions. If you also need to have more granularity and programming language-independence, while also supporting open variation, consider using **Frame Technology** (Sec.4.7). Like Conditional Compilation, it has construction time binding, which avoids efficiency penalties, but in addition it has built-in Default and reuse hierarchy support.

4.6 Aspect-Orientation

As an alternative to Polymorphism or Module Replacement, when common and procedural variant elements are stored in separate modules, you may also organize common procedural elements in such a way that they may be augmented or overridden by variants. This approach is called Aspect-Orientation because the variants are stored in modules called Aspects.

Intent

Decouple common from variant algorithmic elements of product line infrastructure code by organizing the common elements in such a way that their functions become variation points at which behavior may be augmented or replaced, by extracting variant elements into one or more separate modules, and by letting the variant elements refer to their variation points. Aspect-Orientation allows you to consolidate common code and add new variants without changing existing common elements, potentially even for new unforeseen variants.

Motivation

Listing 8 shows a realization of the wireless sensor node from the running example using Aspect-Orientation.

Listing 8: Sensor node realization with Aspect-Orientation

Again, variant elements are color-coded as in Fig.14, arrows correspond to variation points, and gray code sections highlight non-explicit variation points. Orange-colored sections highlight other code elements that cannot be distinguished from variation points if one only sees the respective module, so that they may be mistaken for variation points. As in Module Replacement, these correspond to function calls (comp. Listing 5). In contrast to Module Replacement, variation optimization by Defaults (Def.55) is supported by Aspect-Orientation, which is why one of the alternative Detector variants (green code block in Listing 8) resides in the common module main.c as a default. Variants are cohesively stored in separate modules.

Again, there is one optional (yellow section) and three alternative variants (blue, brown, and red section) that refer to single variation points (green and blue arrows), plus another alternative variant that refers to several variation points (red arrows). With respect to variation point multiplicity, the same conclusion applies as mentioned above.

Applicability

Use Aspect-Orientation

- as an alternative to Polymorphism or Module Replacement, also in order to separate common and variant code in distinct modules, but possibly with defaults,
- to obtain a software system whose existing variants can be replaced by similar ones without changing common and existing variant elements,
- in order to be able to switch between execution time and runtime binding (depending on Aspect Weaver capability), or
- to facilitate the parallel evolution of alternative variant modules.

Process

Figure 30 illustrates how a new alternative variability is realized with Aspect-Orientation. The initial step is the same as in the other mechanisms: an executable module that is currently used by some users (Def.8) is selected for product line evolution by an evolver. The following steps are similar as in Module Replacement (compare to Figure 28). In step 2, variation points (Join Points in Aspect-oriented terminology) in the common module are made available (not shown in Figure 30b), and a new variant module is created. This module, called an aspect, refers to the common module and provides the existing variant algorithm. Alternatively, the existing variant element may also remain in the common module, as a default (Def.55), meant to be overridden. In step 3, a new variant module is created, potentially by cloning the existing one, and the new variant algorithm is introduced by the evolver. Finally,

different users access the same executable module, extending it by weaving in their individual aspect variants (Figure 30d).

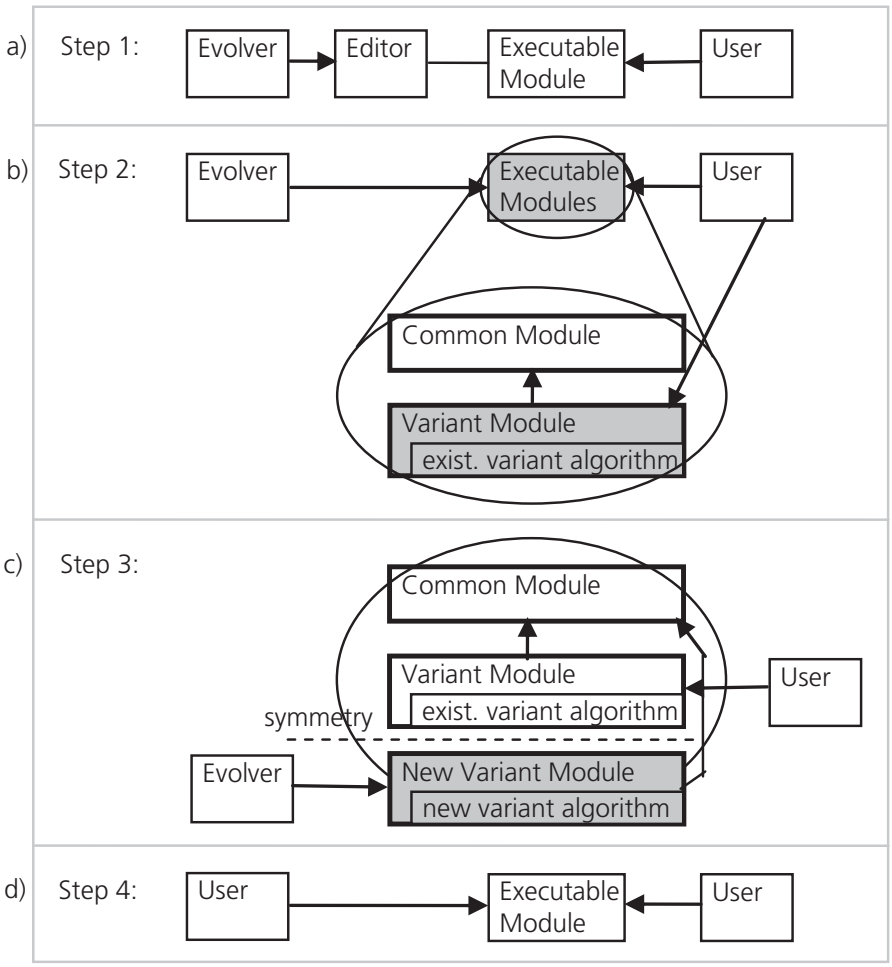


Figure 30: Snapshots of realizing a new alternative variability with Aspect-Orientation

Consequences

Aspect-Orientation has the following main advantages for product line realization: First, it consolidates common elements and decouples them from variants. Second, it decouples variants from each other, so that they may evolve in isolation, as long as the interface of the respective common elements does not change. Third, source code realizing alternative functionality becomes easy to exchange.

Four consequences of Aspect-Orientation are partially positive and negative: First, unlike the similar Module Replacement mechanism, it supports defaults as an optimization possibility for variability management. However, default elements are not clearly distinguishable from common ones because only function calls are available for distinction. Second, it sometimes supports unpredicted changes by its obliviousness characteristic [Filman+00], but only if appropriate variation points have already been realized in the common code. Otherwise,

common code must first be refactored accordingly, for example by classical refactorings such as Extract Method, Rename Method or Move Method [Fowler99], or by aspect-oriented refactorings [Monteiro05, Laddad08], with the risk of accidentally corrupting existing variation points. Third, Aspect-Orientation supports open variation at a similar quality than Module Replacement but with more effort (extra tooling, less explicit variation points). Fourth, there is the danger that, when Aspect-Orientation is used, common code is not cleanly decomposed according to its architecture, so that traceability suffers and the code becomes overly complex to evolve [Ali++10, Eaddy++08, Mens+07, Steimann06, Tourwe++03].

The following properties of Aspect-Orientation are disadvantageous for evolving product line infrastructure code: First, Aspect-Orientation is always dependent on a particular programming language and type of compiler, which is why it cannot be rapidly applied on special C compilers, as often needed in embedded systems development in practice. When performing the case study (Ch.6), we also encountered this problem that none of the available compilers for the given hardware was supported by any aspect-weaver. Third, it is only applicable at function call granularity, so that code may first have to be refactored before the mechanism can be applied. Fourth, depending on the aspect weaver, it may result in resource penalties when, for example, code of overridden defaults remains in the machine code, even though it is never executed in the particular product. Fifth, there is no standard aspect syntax, which increases learning effort and leads to inconsistent realizations.

Details

Data structure. Aspect-oriented mechanisms are usually concerned with variation in functionality, which is why we only discussed algorithmic code elements above. However, variation in state, for example in data structure, is sometimes supported as well.

Symmetries. Many dialects of Aspect-Orientation are asymmetric, that means, they distinguish between conventional code and Aspects. This means that only two-level reuse hierarchies (Def.40) are supported. In symmetric dialects of Aspect-Orientation however, Aspects may realize variation points (join points) and so may be adapted by other Aspects, so that n-level reuse hierarchies can be realized.

Tools. *AspectJ*[10] has become the reference tool for Aspect-Orientation. However, it depends on the Java programming language which is not commonly used for embedded systems development. *AspectC*, a subset

[10] eclipse.org/aspectj (retrieved August 2009)

of AspectJ for the C language, has been used for separating prefetching in operating systems code [Coady++01], but did not become publicly available. *AspectC++*[11] only supports the C++ programming language and requires a specific compiler. *XWeaver*[12] supports C++, but in a compiler-independent way. *ACC*[13] supports Standard C and is compiler independent, and thus is the only choice for general embedded systems development at the moment, but it is still in a research stage of development. A survey of Aspect-Orientation support for the C language is given in [Adams06]. In general, mature aspect weavers for industry-strength embedded systems development are still missing.

Known Uses. The literature on Aspect-Orientation [Elrad++01, Filman++05] often claims that certain tasks in software and product line development have been waiting for aspect-oriented solutions, for example to consolidate development tasks such as tracing. However, besides ongoing discussions of an aspect-oriented replacement for the Observer pattern [Hannemann+02] or toy examples of Java applications for mobile phones [Anastasopoulos+04, Alves07], not many real-world examples have been published with a single-systems embedded systems context [Coady++01, Lohmann++06]. Success stories in embedded systems product line development are missing, as recent reviews have shown [Rashid++10, Amin++10].

Related Patterns

If open variants are required, but AOP tool support is unavailable, use one of the conventional mechanisms **Polymorphism** (Sec.4.3) or **Module Replacement** (Sec.4.4). Like Aspect-Orientation, these depend on programming language syntax and allow you to consolidate common modules and separate them from variant modules (Tab.5). However, their variation points tend to be more visible than those of Aspect-Orientation. With the two conventional mechanisms, variation points may be automatically detected by the compiler. This is done by compiling common modules with variant modules missing, which results in compilation or link errors that hint at variation points. However, this approach does not work if Aspect-Orientation is used because it requires common modules that can be compiled although Aspects are missing. You may consider using **Frame Technology** (Sec.4.7) instead of Aspect-Orientation as another open-variant mechanism with Default support. Both mechanisms require additional tooling, but Frame Technology has deliberately been designed as a construction-time only mechanism, which guarantees that efficiency is not impacted, and which is programming language-independent. Frame Technology supports

[11] www.aspectc.org (retrieved August 2009)

[12] www.xweaver.org (retrieved August 2009)

[13] research.msrg.utoronto.ca/ACC (retrieved August 2009)

variability management at arbitrary levels of scale, from single textual tokens to entire subsystems. Frame Technology, with its more general mechanisms [Bassett07], covers a broader spectrum of variability than Aspect-Orientation. Defaults and Default Overriding are the primary mechanisms of Frame-Technology; this mechanism has been designed to offer them. On the other hand, they only appear as by-products in Aspect-Orientation; that mechanism has not deliberately designed around these properties.

4.7 Frame Technology

The mechanisms discussed in Sections 4.2 to 4.6 are used for one main reason: to avoid Cloning too much common code when evolving a product line infrastructure. In most of these cases, you are concerned with variant algorithms only, either in open or closed variation. Of these mechanisms, only Conditional Compilation allows you to always bind variation at construction time (see Fig.5), with the mentioned benefits, but with one major disadvantage: closed variation. If closed variation is enforced, the code becomes overly complex. Frame Technology, however, supports both open (and sometimes also closed) variation and construction time binding, combines the advantages of Conditional Compilation and Module Replacement without sharing their disadvantages. The mechanism is called Frame Technology because core assets are stored in modules called frames.

Intent

Decouple common from variant code in a product line infrastructure by extracting variant text elements of similar change rates into separate modules, while consolidating common modules. Frame Technology allows you to keep textual artifacts together that have a similar degree of variation in space and in time, without efficiency penalties.

Motivation

A Frame Technology realization of the running example of a wireless sensor node is shown in Listing 9.

Listing 9: Sensor node realization with Frame Technology

As before, the variant elements are marked in the same color as their requirements and architecture equivalents in Fig.14. Variation points are emphasized by colored arrows, and the corresponding textual elements are highlighted in gray. They can explicitly and non-ambiguously be identified in the core asset because Frame Technology has a dedicated syntactic for expressing variation points. Two variation points (blue and green arrows) are used by cohesive variants (green, brown and red sections), while four more variation points (red arrows) are used by variants that consist of more than a single variant element. Compared with each other, variation point referred by cohesive variants lead to less complexity in Frame Technology product line infrastructure code than several variation points referred to by multiple variant elements. But for the purpose of comparing mechanism complexity, variation point multiplicity is not the relevant factor, but variation point explicitness, as discussed above.

As Defaults are fundamental elements of Frame Technology, the code in Listing 9 makes use of this property by realizing one of the alternative Detector variants (green element) as a Default, as mirrored in the example of Aspect-Orientation (Listing 8). One of the benefits of the Default approach is that fewer variants are required. Another advantage of both Frame Technology and Aspect-Orientation is that each realized variation point in a core asset is a compact representation of actually three variation points because variants may refer to it in three ways (before, after, at), as indicated in Listing 8 and Listing 9 by the final red arrow and the blue arrow.

Applicability

Use Frame Technology

- as an alternative to Conditional Compilation, in order to physically separate common from variant code, and especially if the end result is a new alternative variant,
- in cases when extraction of functions as variant elements is infeasible or requires too much refactoring effort,
- if efficiency penalties due to variability management must be avoided,
- if variant elements of large and small sizes must be managed together,
- if variation points shall be visible in core asset code, or
- if it makes sense not only to have two levels of modules (two-level tree [Parnas08]), common and variable ones, but a deeper hierarchy of partially common and partially variant modules (multi-level tree or multiple partitioning [Parnas08]).

Process

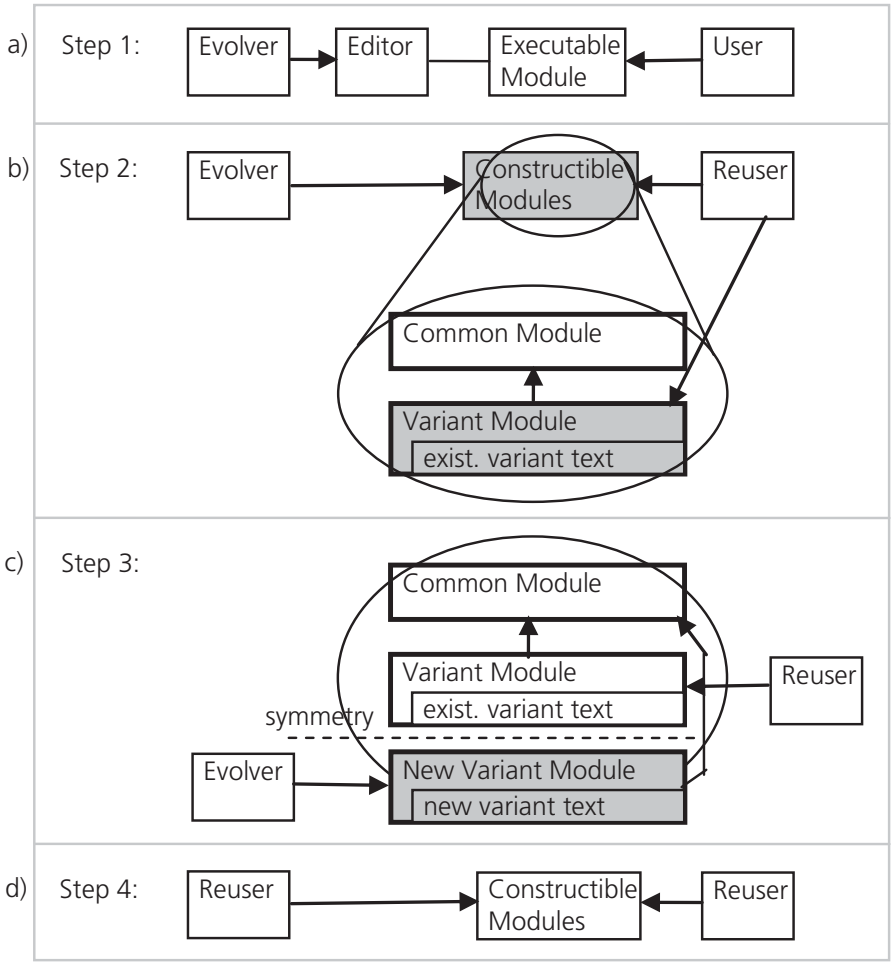


Figure 31: Snapshots of realizing a new alternative variability with Frame Technology

As shown in Figure 31, the development steps are similar to those of Aspect-Orientation (Figure 30), while the development artifacts resemble those of Conditional Compilation (Figure 29). First, the evolver-developer accesses the executable module, while the existing user (Def.8) also accesses it. In step 2, the module is split into a common and a variant module. The common module contains most of the existing code, enriched by Frame Technology annotations which make variation points explicit and look similar to those markers used in Templating, the Cloning variant (compare Listing 9 with Listing 2). The variant module contains the existing variant text, to be inserted at the variation points. As in Aspect-Orientation, the variant text element may alternatively reside in the common module as a Default if it is shared by most product line members, but not all. The modules become constructible modules because they are processed by a construction interpreter to produce the desired product line member. Frame commands represent actions that can be replayed to produce product line members from a product line infrastructure. In step 3, a new variant module is created that contains

new variant text. It resembles the existing variant module because their references to the common module are similar, usually indistinguishable (symmetry), while their variant text differs. In the last step, the constructible modules are reused by reusers to evolve them or produce product line members. The process is identical to the final process in Conditional Compilation (Figure 29d); only the Constructible Module details differ.

Consequences

Frame Technology has the following advantages as a variability mechanism in product line infrastructure code evolution: First, it decouples common from variant elements, either explicitly visible (if both are extracted into different modules), or less visible (if the variant text, as a Default, resides in the common module). Due to open variation, the mechanism supports unpredicted changes. Second, Frame Technology is programming language-independent because it is only concerned with textual elements, independent of programming language semantics. In particular, variant code does not have to form a cohesive procedural element. Due to this property, an additional degree of freedom is gained in realizing core assets, compared to language-dependent mechanisms. This makes it easier to extract variant elements, without extra refactoring effort. Third, Frame Technology does not lead to efficiency penalties in the resulting machine code. The code becomes indistinguishable from manually cloned code at execution time. Fourth, it facilitates variability management in space and time by organizing modules in reuse hierarchies according to their stability. Frame Technology provides Bounded Combinatorics [Krueger10] because adding a variant to an existing set of variants in a realization of an alternative variability only leads to a linear growth in modules. Fifth, it uses Defaults as first-class elements, instead of variants in most other variability mechanisms. Each Default leads to less complexity in variability management because the number of variant modules is reduced by one.

Frame Technology has these disadvantages: First, the mechanism is relatively unknown in practice, there is no standard frame syntax, and tool support is limited (see Details/Tools section). Second, some implementations of Frame Technology only offer open variation, which makes the mechanism less easy to introduce than its closed counter-mechanism, Conditional Compilation.

Details

Tools. Several commercial or academic tools with Frame Technology support (frame processors) exist. Their common characteristic is that common and Default elements can be distinguished in textual assets, that Defaults may be overridden and that reuse hierarchies can be set up. However, all tools differ in their syntax, and many add specific features, such as iteration or closed variation, which is either not needed in many situations, or which is already well-supported through other variability mechanisms.

Netron Fusion[14] is a commercial tool, containing the original frame processor and support tools. It has been used in a number of different IT development projects, especially in COBOL, for more than two decades [Bassett97]. Its COBOL-like syntax can be an obstacle if it is going to be introduced in non-COBOL projects. *XVCL* [Zhang++01, Bassett07] and its offsets *FPL* [Sauer02] and *LFP* [Loughran+04] are academic frame processors which use XML as their underlying text representation. *XVCL* is reported to have been used in different application domains, such as CAD systems [Zhang+03], C++ libraries [Basit++05], or cash desks [Schäfer++09]. I have developed *FP*[15] as a frame processor with deliberately limited features, and whose syntax is deliberately kept as human-readable and as compact as possible, which is why XML was not chosen. *FP* has been used in a number of projects, especially in the embedded systems domain, but it has also been used for variability management in non-source code artifacts, such as *TeX* files producing slide show documents. Its usage has been explained in [Patzke+03, Patzke08], and the *FP* source code used in the case study of this thesis is listed in Appendix B.1.

Evolution. Frame hierarchies [Bassett97] realize reuse hierarchies (Def.40) and thus explicitly separate modules with different evolution rates from each other. If a frame processor provides both open and closed variation, closed variation can be used for variation in time by marking default versions or deprecated code elements [Bassett97, pp.182f.]. Existing source code modules can easily be refactored into frames without changing source code by adding frame annotations (the source code must not contain annotation syntax). Adapting these modules facilitates parallel evolution without inconsistent co-evolution of common modules. If it turns out during evolution that certain temporal variants have higher or lower change rates than initially conceived, simple refactorings may be used to move them into their respective position in the frame hierarchy.

[14] www.netron.com/products (retrieved August 2009)

[15] frameprocessor.sf.net (retrieved August 2009)

Organization of Elements. The following rules help to decide where to place a code element within a frame hierarchy (also see Fig.11): When two elements have the same evolution rate, they are placed at the same level of the frame hierarchy. In addition, when these elements are always reused together, they are placed in the same frame. When the elements can be reused independently, e.g. as alternative variants, they belong into sibling frames. When they do not have the same change rate, i.e., one element implies a reuse of the other, but the other can be reused alone, then they belong into different levels.

Selection of Defaults. Frame Technology makes extensive use of default text. When a common element is framed, and variation points are defined, a meaningful default may also be provided in many cases, rather than leaving the default empty. The advantage is that the template provided by the frame becomes more comprehensible when it is adapted because the default serves as an integrated "best example", as in the Templating variant of Cloning (Sec.4.1). It can also make the ancestor frames smaller, which is always desirable because it minimizes product-specific code. Another observation concerning Defaults is that the set union of all defaults in a frame does not always need to result in a meaningful product line member: the frame may be reused more efficiently if each default alone does not require extensive overriding.

Variation Point Naming. Variation points must be named in such a way that the frame's context-freedom [Bassett97] is maintained. In other words, the variation point name shall not contain variant information, so that the core asset remains independent of particular variants. This means for example in the above case, that when a new variability is realized, the respective variation point shall have a name such as `more_init` instead of `init_sound_sensor`.

Optimizations. As mentioned above, frame technology reduces the number of variant modules through Defaults (each common module may contain some slightly variant code in the form of defaults, so that less variant modules are required). Also as mentioned above, the number of variation points is reduced because each variation point can be referred to in three different ways (e.g., for a variation point called "vp", two implicit ones called "before vp" and "after vp" exist). The number of required modules can drastically be reduced because a single frame may lead to the production of more than one file.

Known Uses. As mentioned in the Tools section above, three different frame processors have been reported to be used in several software development projects: Netron Fusion [Netron], *XVCL* [Zhang++01], and *FP* [Patzke+03]. *XVCL* has been used to eliminate redundancies in parts of the Standard Template Library of the C++ programming language [Basit++05]. *FP* has been used for framing product line realizations of

resource-constrained embedded systems in the automotive and consumer electronics application domain. In the context of the case study (Ch.6), I use it to realize a software product line of wireless sensor node applications and PC-based transceivers, where related variabilities are realized in two programming languages (C code for software running on the embedded devices, and Java code for the processing the received information in a PC).

Related Patterns

Although Frame Technology has been developed in order to fight the numerous liabilities of **Cloning** (Sec.4.1) [Bassett97, p.86], Cloning often provides quicker short-term results. As an intermediary between Frame Technology and Cloning, the manual Templating mechanism already provides explicit variation points, similar to those in Frame Technology. You may also consider using **Conditional Compilation** (Sec.4.5), as it is a more well-known programming language-independent variability mechanism. However, Conditional Compilation does not explicitly use Defaults and Default Overriding (although idioms may be used for that purpose). As opposed to Frame Technology, it employs closed variation, and for that reason it does not support unpredicted evolution. Although Conditional Compilation slightly decouples common from variant elements, they are still strongly coupled because they reside in the same module. A transition path from Conditional Compilation to Frame Technology has been shown in [Patzke07].

5 Product Line Evolution Method

The variability mechanisms presented in Chapter 4 are used by family engineers or automated agents to evolve existing product line infrastructure code into new product line infrastructure code, according to a new specification. Figure 32 shows the overall approach (cf. Fig.4) whose remaining elements will be presented in this chapter.

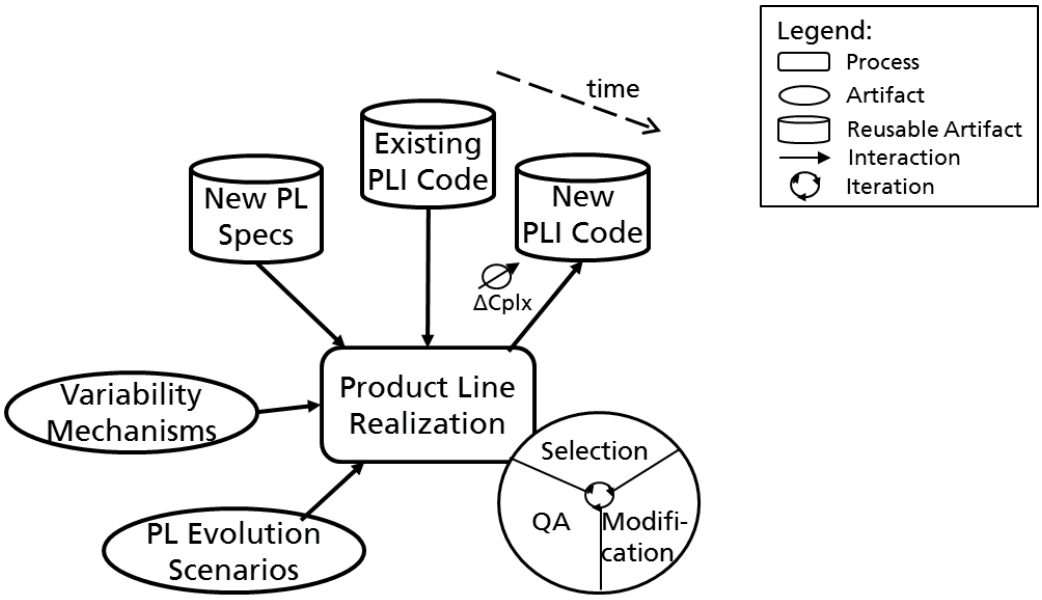


Figure 32: Product line evolution method

A first element that will be taken into account is which product line evolution possibilities, types of possible future next steps in product line-relevant evolution, are most likely to occur. These product line evolution scenarios generalize predominant types of new variability-related requirements that the family engineer must realize in the existing product line infrastructure code. For example, one type of product line evolution scenario is that a new optional feature is needed in the product line. This means that existing product line infrastructure code which has been ignorant of that variability will have to be changed so that it provides more variability. A classification of product line evolution scenarios is developed in Section 5.1. The goal is to capture types of future requirements changes (to predict unforeseen changes) that have an impact on variability in a product line infrastructure. The evolution scenarios consist of atomic generic and non-generic sub-processes which are instantiated and combined in a certain order so that evolution effort is kept as low as necessary.

The process phases, described in Section 5.2, are the third building block of the product line evolution method developed in the current thesis. In contrast to the evolution scenarios, the process phases are larger types of activities within the realization life cycle. They are applied iteratively and incrementally. The basic phases are Selection, Modification and Quality Assurance, all focused on variability management. Selection and Modification are associated with specific variability refactorings.

The fourth building block of the product line evolution method is variability complexity measurement, discussed in Section 5.3. It is applied as the second element of the Quality Assurance phase, besides Product Line Testing. Using the GQM approach, a customizable metrics suite is developed which describes to which degree the resulting product line infrastructure code becomes unnecessarily complex in variability management.

All method elements have been applied in the case study in Chapter 6 which evaluates the impact of variability mechanisms on evolvability. An existing set of single products is evolved into product lines using different types of evolution scenarios in different orders. Each evolution step (Def.68) is realized using all variability mechanisms mentioned in Chapter 4. Evolution traces of the involved code assets are presented. Variability complexity is measured and compared for product line infrastructure code in all stages.

5.1 Product Line Evolution Scenarios

Successful software systems evolve [Parnas94]. A software system evolves due to new requirements which are then realized in the code by a software engineer. When a product line evolves, the interplay of its commonalities and variabilities must typically change, which requires a family engineer to use variability mechanisms, as those presented in Chapter 4. However, the usage of mechanisms is often not disciplined in practice, which causes unnecessary complexities. The remedy suggested in this thesis is to capture development steps which result in well-behaved evolution, and to reapply them later if a similar development situation arises. For product line infrastructure evolution, this means to capture variability-related scenarios, as the presence of variability is the main distinction between product lines and single systems. In particular, product line evolution scenarios are concerned with (foreseen or unforeseen) changes in requirements. These changes have an impact on the interplay of existing common and variant elements in the product line infrastructure.

In this section, a set of basic product line evolution scenarios will be developed that covers the major types of variability. It will be shown that these scenarios consist of more atomic scenarios which can be mapped

to elementary realization activities. Moreover, it will be shown that it makes sense to apply them in a specific sequence most of the time. Suggestions will be given how to aggregate the scenarios into larger scenarios.

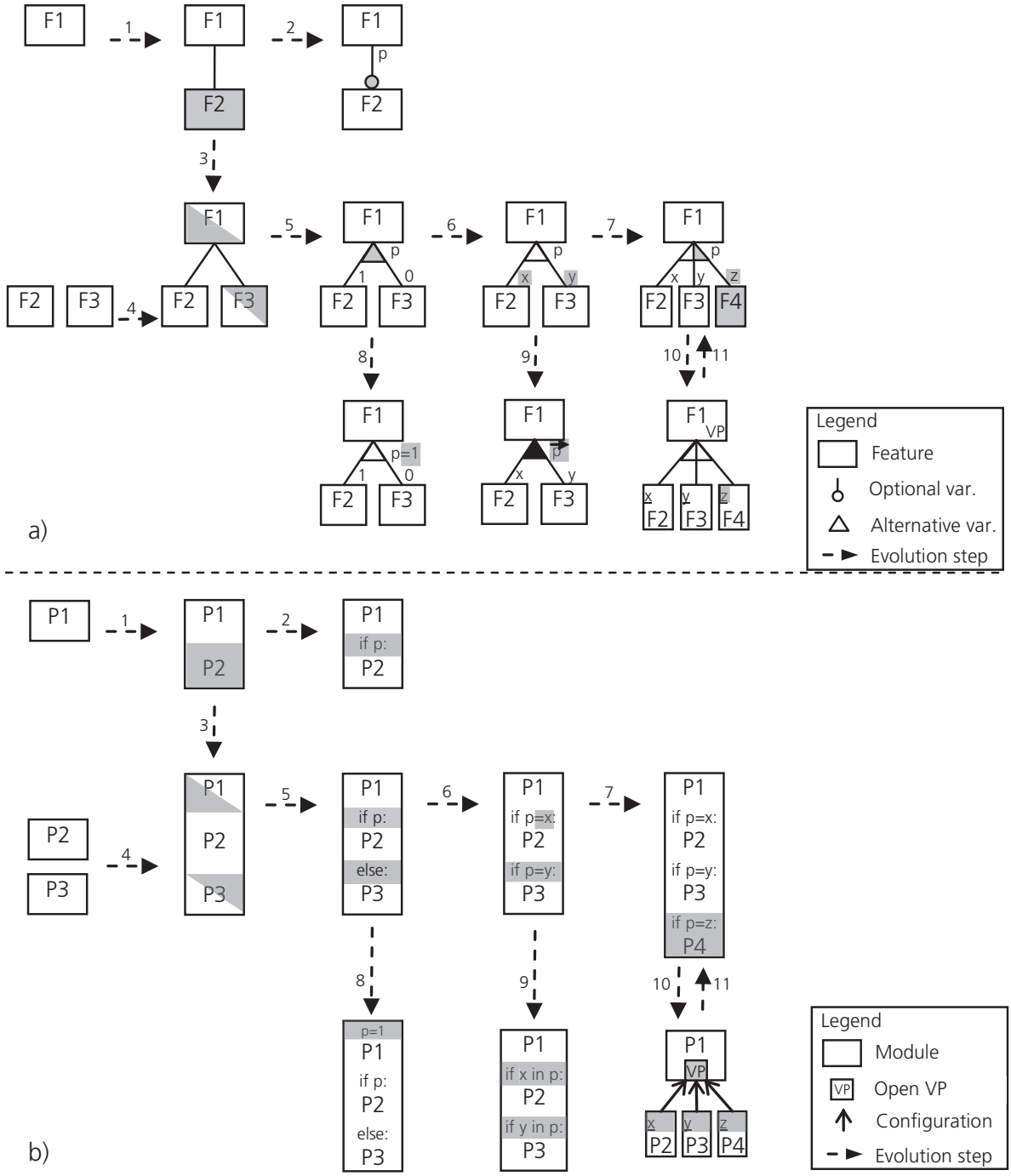


Figure 33: a) Elementary feature evolutions, b) corresponding pseudocode

Figure 33 summarizes the most atomic evolution possibilities that a product line asset can undergo. Figure 33a shows evolution steps in product line requirements (changes in features (Def.63)) as snapshots of annotated feature diagrams, Figure 33b depicts the corresponding

changes in snapshots of pseudocode artifacts. The pseudocode represents the family engineer's mental model on how to realize the new requirements. The goal of the split in Fig.33 is not to propose a certain mapping between requirements and code. It is assumed that family engineers in practice are capable of doing this. The goal is to illustrate that similar evolution sequences in requirements artifacts result in similar evolution sequences in code artifacts. Each step, shown as a dashed arrow, adds some primary common or variable element of interest. As in the process snapshots throughout Chapter 4, elements that have varied in time are shown in gray. In concatenation, all scenarios lead to end results in which variability in space has changed.

In all discussed product line evolution scenarios, the interplay of common and variant elements changes across time. Moreover, because I focus on reducing new complexity, only those scenarios are covered here that make core assets more complex with regard to variability. Scenarios which make them less complex (due to removal of variability) are not covered. They can be regarded as complexity-increasing scenarios in the backwards-time direction. For example, sub-step 2 in Fig.33 creates a variation point for an optional variant, whereas in the opposite time direction, the variation point is removed, which leads to less variability.

A product line evolution scenario may start in any of the shown states, comprising one or more of the basic evolution steps. In one starting state, the artifact does not realize any variability, or it realizes only variability that is irrelevant for the upcoming task. This corresponds to the first snapshot in Figure 33a, where only a single common feature F1 exists, realized as a single code artifact P1 that represents a construction-time constant. One new product line-specific requirement is to realize a new feature which shall optionally be available in the next version of the product line. This requires evolution steps 1 and 2 in Figure 33. The product line evolution scenario is called Optional Feature Creation. Figure 34 shows which aggregate product line evolution scenarios exist and how they can be obtained from Figure 33 by (re-)using atomic scenarios.

The following sub-sections present details of all listed product line evolution scenarios, with a consistent naming scheme: Creation describes a situation when a variability-related element is newly built which did not exist before. Addition means that an element is created where a similar element already existed before. Extraction happens if an element is made more visible within the larger system, and Inlining means the opposite. The names are predominantly chosen according to problem space (Def.50) artifacts because the scenarios are triggered by changes in requirements, although realizing them also depends on existing solution space artifacts.

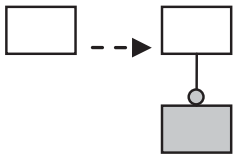
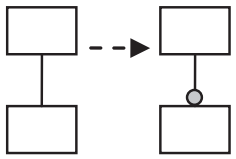
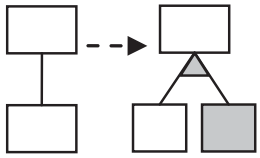
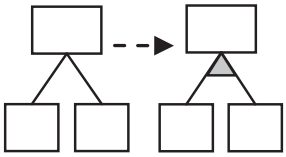
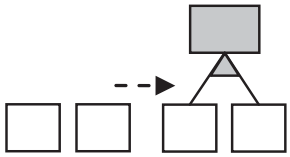
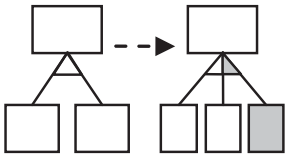
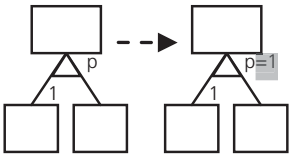
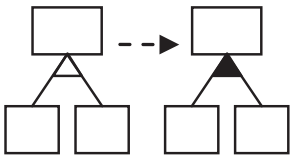
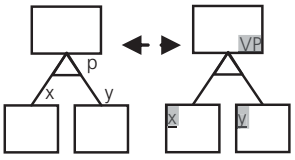
	Name	feature evolution	elementary steps (Figure 33)
a)	Optional feature creation		1, 2
b)	Optional variation point creation		2
c)	Alternative feature creation		3, 5
d)	Alternative variation point creation		5
e)	Common feature extraction		4, 5
f)	Alternative feature addition		(6), 7
g)	Default addition		8
h)	Addition of multiple coexisting possibilities		9
i)	Variable feature extraction / inlining		10 / 11

Figure 34: Basic product line evolution scenarios captured in Fig.33

Optional Feature Creation (Figure 34a)

This scenario means that a new feature needs to be realized which depends on an existing common (or quasi-common) feature, so that afterwards either a product can be produced that only realizes the pre-existing feature, or one that realizes the existing plus the new feature. In the first case, the resulting product is indistinguishable from the product that existed before the evolution step was taken. In the second case, the product offers the new characteristics in addition to the existing ones. The goal of the realization activity is to incrementally realize the new feature with minimal effort, adding just enough variability complexity as necessary to the product line infrastructure code.

Whichever approach is used, an unavoidable complexity is that the new feature must be realized, for example as new functionality. This complexity is unrelated to variability issues and is also encountered in single-systems development. The simplest realization is to clone the existing element, to modify the cloned element so that it realizes the new feature, and to select among the two when producing the respective product. However, as seen in Section 4.1, Cloning is least sustainable because it leads to a duplication of all common elements.

The simplest approach that keeps the common element consolidated is to augment the existing product line infrastructure code by new code in a least obtrusive way. This has three consequences. First, the new code is added as close to the existing code as possible, which means in the same modules that contain the existing code. This only requires realization by closed variation. Open variation is not necessary and would lead to unnecessary complexity. A second consequence of least obtrusiveness is that the existing product line infrastructure code is not changed more than necessary to offer variation points (it is assumed that in the general case of unpredicted change, appropriate variation points do not yet exist). This means that a possibility for enabling the new variant must be added. For variability management of text-based artifacts such as conventional source code, it is sufficient that this possibility is text-based. It is not necessary (and would lead to excess complexity) that the variation point is also related to the semantics of the source code [Bassett97, p.79]. This makes programming language agnostic mechanisms, such as Conditional Compilation, most appropriate in the given context (Def.20). A third consequence of least obtrusiveness is that only those added elements become variability managed that must be variability managed. In other words, at least one variation point must be realized, but not more variation points than necessary should be realized. More variation points make the variant less cohesive, but it should not become less cohesive than necessary. This does not mean that the variant must always consist of only a single variant element (complexity excess also means that fewer variant elements are realized than required

by the architecture). It only means that not more variation points should be realized than necessary.

As shown in Figure 35, Optional Feature Creation can be realized in two ways, depending on the order of its two sub-steps. This simplest example with only two steps already shows that complexity and productivity are influenced by the order in which process steps are executed, that this order can be planned according to certain criteria, and that random approaches which neglect such an order lead to unnecessary complexity. In a similar way, it also applies to all multi-step scenarios which follow.

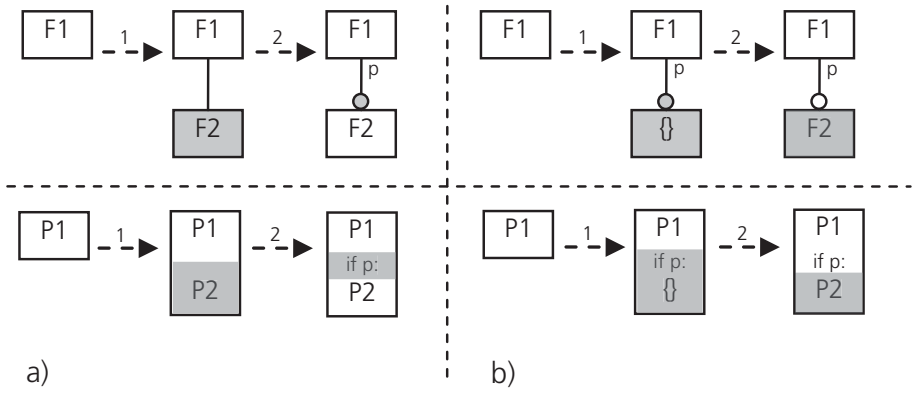


Figure 35: Optional Feature Creation sub-steps, starting with a) commonalities, b) variabilities

In the first approach, a new feature is first realized and added to the existing product line infrastructure code as if it were common, a construction-time constant (Figure 35a, step 1). Thereafter, this new construction-time constant is first converted into an equivalent construction-time variable with two indistinguishable values, and then one of these values is made void (step 2). The latter sub-step amounts to adding an optional variation point. In the second approach (Figure 35b), an optional variant without executable code (a null feature, illustrated by the symbol *{}*) is first realized next to the existing common element. This means that the variation point is realized first, and after that, executable code is written for the new variant.

Both approaches have the same end result, the realization of a common and an optional feature. However, a claim made in this thesis is that the order of performing the sub-scenarios should not be arbitrary because it results in excess complexity and decreases productivity. In the majority of cases, the first approach leads to a more evolvable product line infrastructure code, for at least two reasons: First, if an empty variant is added first, the family engineer must speculate where exactly to realize the variation point because its proper position can only be determined with certainty when both the common and the variant code exist. On the other hand, if new code is first added as if it were common code,

the family engineer can afterwards see the position of the newly required variation point, exactly at the necessary granularity, by comparing the old and new realizations. Second, if the family engineer first realizes an empty variant and his decision with regard to variation point position turns out to be overly complex, he will only notice this after the second development step has been completed. This will cause him to undo two previous steps, leading to productivity loss. Especially under time pressure, the introduced 'small' defect in the code tends to be neglected, which leads to code that is more complex than necessary. The first approach is more efficient because each step builds up on previous results. After completing the first sub-step, ignoring variability issues, the family engineer may already run and test the new product, as early as possible. If the family engineer makes an error at this stage, so that the new product does not execute as expected, only one step has to be undone. In contrast, if the same error is made in the alternative scenario, two steps will likely have to be undone, or the variation point remains overly complex.

The two approaches make a difference in productivity for another reason. In the first case (Figure 35a), the new feature F2 is realized by performing a large number of small steps first (line by line addition of new code, until the new product runs), and then the variation points are added by taking a small number of large steps (adding one or more variation points for the different variant elements). In the second case (Figure 35b), the large-step activities are performed first, followed by small-step activities. The first approach is favorable because a large-step activity (adding or re-adjusting variation points) is only performed once, whereas it may need to be performed several times in the second case.

Optional Variation Point Creation (Figure 34b)

The variability-related sub-scenario of Optional Feature Creation is called Optional Variation Point Creation. It is discussed as a separate product line evolution scenario because its precondition is different. Both scenarios share the precondition that the code has not realized an interesting variation that would influence the upcoming variability management task. In Optional Feature Creation, the feature to be realized as an optional variant has not existed before, whereas in Optional Variation Point Creation it exists already as an element of the common code. Detecting the common code elements tends to be easier in Optional Variation Point Creation because the feature has already been realized in the existing product line infrastructure code. In Optional Feature Creation, however, there is a higher probability to misjudge the common elements, as shown above.

In the general case, it cannot be assumed that all common code elements that shall be made optional variant elements in Optional Variation Point Creation form syntactically cohesive programming

language elements, such as subroutines. This means that generally, programming language-dependent variability mechanisms are not first choice because they require additional refactoring effort beforehand. Language-agnostic variability mechanisms tend to be the best choice for realizing this scenario in the simplest manner. Also, as discussed above, if there is no need for making the optional variant particularly visible, isolation is unnecessary, so that the simplest approach is to add the new variation point inline.

Alternative Feature Creation (Figure 34c)

This scenario denotes that an alternative feature is required as a substitute for an existing common feature element. This may happen, for example, when an existing element in a new product generation [Muthig02] shall become deprecated, but must yet be readily available.

As shown in Figure 34c, the precondition in this scenario is the same as in Optional Variation Point Creation: A common (or quasi-common) feature consists of two elements, one of which is invariant with regard to the upcoming evolution scenario, whereas the other becomes a variant. In Alternative Feature Creation, the family engineer's task is to introduce a new variant alongside existing code in the most productive way. The new variant must share variation points with its sibling, but must provide a different behavior in the resulting product line member.

If cloning is avoided, the simplest approach is a two-step process, similar as in Optional Feature Creation. First, a common element is created, and second, this element is made a variant. In the first step (step 3 in Figure 33), the reference common element is identified for which an alternative is to be provided. This happens as described in Optional Variation Point Creation. This step is applied first for the same reasons discussed in Optional Feature Creation. The resulting executable module should be compilable into machine code, but it might not provide the required functionality because it realizes two features, only one of which is valid in each product line member. Thus, in a second sub-step, an alternative variation point is introduced (step 5 in Figure 33). Similar as in Optional Variation Point Creation, variation may be closed, and a variant may be selected as result of a Boolean decision. Again, a programming language-agnostic variability mechanism, such as Conditional Compilation, is most appropriate in the majority of cases.

Alternative Variation Point Creation (Figure 34d)

Like Optional Variation Point Creation, this scenario corresponds to the variability-related sub-step of a Feature Creation scenario, but it is also a self-contained product line evolution scenario. Again, the artifacts in the initial state do not realize relevant variability. They can be considered common, but with latent elements that shall be made alternatives. In the

simplest case, only two of these elements exist. They are made more explicit by detecting their common variation point position and by realizing one or possibly few closed variation points there.

Common Feature Extraction (Figure 34e)

As in the scenarios mentioned before, the precondition of this scenario is that code exists which can be regarded common. In contrast to the aforementioned scenarios however, Common Feature Extraction starts with two realizations of similar independent features, possibly created by Cloning. The task is to consolidate the common elements so that they exist only once, converting the differing elements into alternatives. Thus, this scenario captures the important activity of clone removal and is a more general case of Consolidate Clones (Tab.8).

As shown in Figure 34e, the scenario consists of the two elementary sub-steps 4 and 5 from Figure 33. Again, the first sub-step is not concerned with variability, and the second sub-step is a variability-related scenario that is also used in other basic product line evolution scenarios: Alternative Variation Point Creation. As mentioned in the Details section of Conditional Compilation (Section 4.5), the *diff* tool may not only be used to visualize the differences of similar elements, but it can also automate the task of Common Feature Extraction by consolidating modules with the help of Conditional Compilation.

Alternative Feature Addition (Figure 34f)

In this scenario, the existing code realizes a common and two or more alternative features. Another alternative feature is to be realized alongside the existing ones. If the existing alternatives have been realized in a single module, as indicated in Figure 33b (following step 6), another element which realizes the new alternative is created by extending the module (Fig.33, step 7). This means that the existing variation point is extended to support a new value. On the other hand, if the existing alternatives have been realized in separate modules, the new alternative is also realized in a new sibling module.

Default Addition (Figure 34g)

Default Addition is an optimizing basic product line evolution scenario which also starts with code that realizes variability. Figure 34g illustrates the case of alternative features, but other types of variability will apply as well. As explained in Section 4.7, if one of variable feature dominates its siblings, it may be represented as a Default (Def.55), reducing the number of variant modules. This can be realized by deactivating the variation point in the default case, as shown in Fig.33, step 8, and by providing a mechanism to override that deactivation. Default Addition is orthogonal to the other mentioned product line evolution scenarios

because it may appear independently in all constellations in which variability has been realized.

Addition of Multiple Coexisting Possibilities (Figure 34h)

Sometimes, it becomes necessary in a new product that several features must be simultaneously available which have been alternatives in previous products. The realization in Figure 33 (before step 9) shows that the value for selecting variants can be seen as a scalar in case of alternative variations. By Addition of Multiple Coexisting Possibilities, this value is converted into a vector, often realized as a bit-field in embedded systems code.

Variable Feature Extraction / Variable Feature Inlining (Figure 34i)

These two complementary scenarios are usually applied to alternative or coexisting features.

Variable Feature Extraction decouples alternative variants that have previously existed in the same module. This usually becomes necessary if a growing number of alternative features are required. For example, in the realization of alternatives in Figure 33 (after step 7), the number of alternative features which are realized in the same module is three. It may now become necessary to extract them in order to evolve them in isolation. This requires converting them from closed variation to open variation.

Conversely, variability management in product line infrastructure code can be simplified by Variable Feature Inlining in the following cases: if the number of alternatives drops, if the code size of alternative variant elements becomes small, if alternatives realized in separate modules shall be evolved together, or if the number of alternatives is low and is not likely to increase. Existing open variation is converted into closed variation.

Variable Feature Extraction and Variable Feature Inlining are another set of scenarios that, like Default Addition, crosscut the other product line evolution scenarios. They are concerned with the variants' coupling.

5.2 Product Line Realization Process

All product line evolution scenarios discussed in Section 5.1 have in common that they proceed in a certain order. This iterative process, illustrated in Figure 36, consists of the three main phases Selection, Modification and Quality Assurance. As indicated in Figure 37, these phases represent the dynamic aspect of the product line evolution method developed in this thesis.

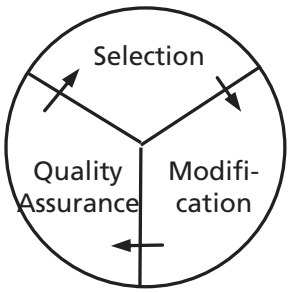


Figure 36: Basic realization phases

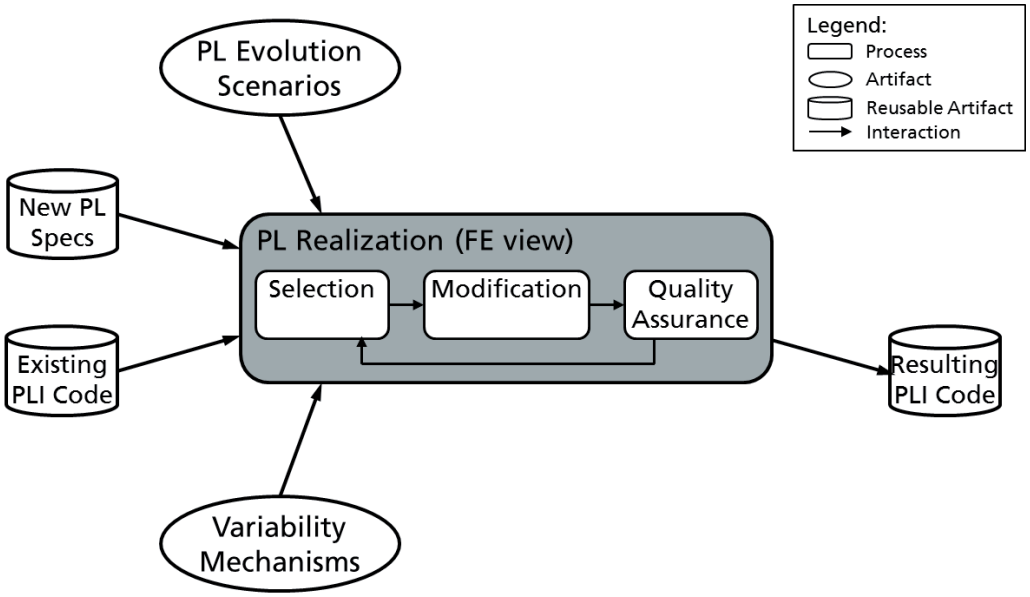


Figure 37: Product line evolution method discussed so far

As depicted in Figure 37, the family engineer first identifies and selects a specific detail of interest in the existing product line infrastructure code. This detail may or may not currently be concerned with variability issues, but it must be related to the current product line requirement which has to be realized in the code. The family engineer performs the first activity analytically: the goal is to comprehend and observe the code, not to change it. During this phase, excessive variability complexity (Def.65) may be observed that should be documented. This phase is called Selection.

In a second phase, the family engineer alters the selected product line infrastructure code elements, according to the type of product line evolution scenario, and according to the variability mechanisms that exist already in the code. Within the larger product line infrastructure code, this non-passive activity changes the selected elements to some degree, while the unselected elements remain completely invariant. Relevant variability complexity is removed by specific refactorings. This phase is called Modification.

The third phase, quality assurance, has not yet been discussed in the product line evolution scenarios. Through this phase, the family engineer tries to avoid mistakes that may have occurred in the previous steps, so that defects are detected and corrected as soon as possible. After this phase has been finished successfully, a new iteration starts by entering the selection phase again.

Similar phases of (software) evolution activities have partially been suggested elsewhere in the software engineering literature. For example, Jalote describes realization processes which contain the phases coding, refactoring and testing [Jalote05, pp.409ff.]. Somerville discusses an initial program understanding phase in software evolution processes during realization, followed by the phase of source code modification [Sommerville04, pp.499f.]. In classical refactorings, the common phases are to detect "code smells", perform the refactoring, and unit-test the results [Fowler99]. As mentioned in Sec.3.2, an incremental product line modeling approach exists which consists of the phases identification, modeling and quality assurance for common and variable artifacts and their relationships [Muthig02, pp.105ff.].

Selection

In Selection, the first phase of the product line realization process, the family engineer's immediate goal is to identify those elements in the existing product line infrastructure code which are most likely to be affected by the current product line requirement. For example, in a requirement to create a new optional feature the main goal is to detect which code elements in which modules will have to be altered in order to realize the newly required variation points. More generally, the goal is to focus on those few elements where evolution is going to happen, while suppressing the numerous other details that will remain unchanged. This allows the family engineer to concentrate on those areas in the code in which new code will later be created, areas where variation in time will occur, rather than being overwhelmed by the bulk of code which will remain invariant. This sub-phase is called the identification phase.

At the same time, the family engineer evaluates globally if variations of a similar type (e.g. optional variabilities) have already been realized in other

areas of the product line infrastructure code. If they exist, he verifies that these existing variability-related elements have been realized consistently. They will become reference elements whose construction process will be reused when the new elements are created later. If inconsistencies or defects in variability management are detected, these should be documented at this stage. However, correction shall be postponed until the primary goal, realizing the new variability, has been reached. This ensures that the main goal is reached as rapidly as possible, while detected defects are not neglected. This sub-phase is called comprehension.

In all selection activities, the family engineer only acts as a passive observer on the existing product line infrastructure code, so that the code represents a read-only artifact. Figure 38 summarizes the selection phase, its sub-activities, inputs and outputs. The figure shows another input element of the selection phase, which is used in the comprehension activity, called "product line infrastructure code smells". This element, named according to the "code smells" concept in conventional refactoring literature [Fowler99, Kerevinsky04, Wake04], describes particular types of defects in product line infrastructure code.

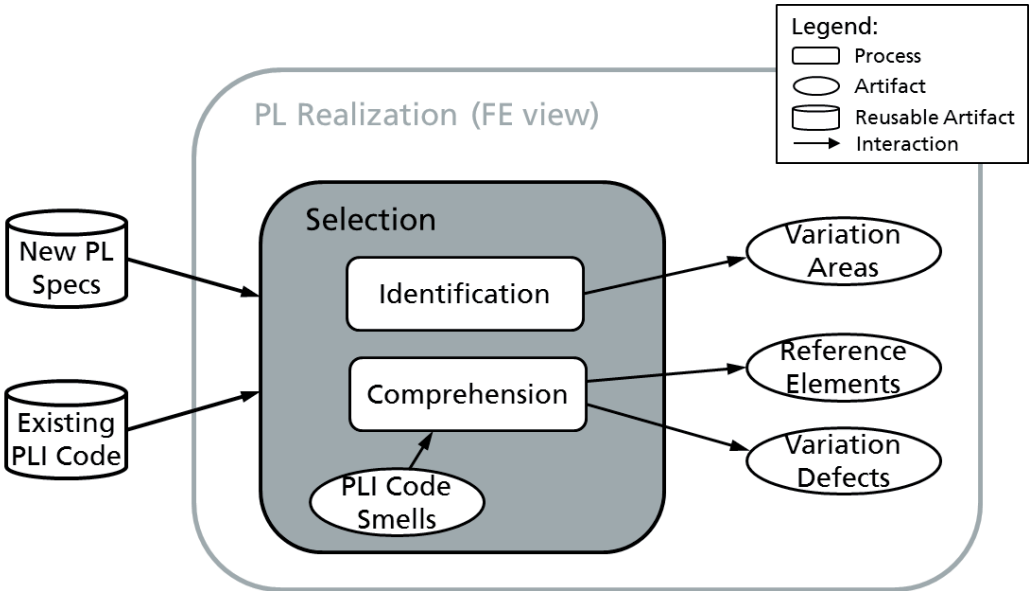


Figure 38: Details of the selection phase

Table 7 lists typical, mostly disjoint product line infrastructure code smells which I have frequently observed in reusable code in practice (and sometimes in other types of reusable artifacts) and which have repeatedly contributed to variability complexity excess.

No.	Name
1	Duplicated Code
2	Runtime Variation
3	Coupling of Application and Variation Logic
4	Startup Initialization
5	Ambiguous Variation Points
6	Nested Variation Points
7	Variation Point Excess
8	Crosscutting Excess
9	Coupling of Common Elements and Variants
10	Coupling of Alternative Variants
11	Excess of Variant Modules
12	Null Module
13	Non-Cohesive Configuration
14	Excess of Configuration Mechanisms
15	Explicit Product References
16	Unbounded Combinatorics
17	Lack of Variability
18	Lack of Defaults
19	Composition Excess
20	Restricted Variant Granularity
21	Speculative Variation Points
22	Excess of Variant Similarity
23	Excess of Variant Size

Table 7: Product line infrastructure code smells

Duplicated Code [Fowler99] denotes that existing larger common elements have not been consolidated at a single position, but that they have been cloned. As mentioned in Section 4.1, tools such as *DupLoc* [Ducasse++99] and *diff* can be used to get an overview of the cloning situation (also see [Demeyer++02, pp.173ff.]). Runtime Variation describes a situation when runtime binding has been used although execution or construction time binding would suffice. This leads to an unnecessary reduction of reusability (Def.19) and increases variability complexity (Def.65). As a consequence, there may be a Coupling of Application and Variation Logic, e.g. when a single conditional statement mixes two predicates, one for controlling application functionality and the other for distinguishing between product line members. Runtime Variation depends on setting runtime variables which are often initialized only once at startup time, but in the later execution always remain unchanged. Unless the production process demands it, this is a code smell I call Startup Initialization, as initialization could have been done earlier (see also [Bassett97, p.78]). Variability mechanisms such as Conditional Execution also make variation points less explicitly visible, because an 'if' statement could either act as a variation point, or it may control some functionality, or both. This code smell is called Ambiguous Variation Points.

The code smell Nested Variation Points means that dependencies among variants have been hard-coded, for example in long #ifdef statements with many AND/OR combinations of macros. Carelessly mapping every variant element from the architecture one-to-one to code may lead to Variation Point Excess in the code. For example, in many embedded systems it can be tolerated to initialize all available resources, although only particular subsets are used in any individual product line member, which makes variation of initialization unnecessary. Many variants are realized less cohesively than necessary, which leads to Crosscutting Excess. When variants are carelessly realized as closed variants, common and variant elements are forced to remain together in the same module, which results in Coupling of Common Elements and Variants. Coupling of Alternative Variants describes the situation when a module contains various large alternative variants that could have been kept separate. In the opposite case, too many individual modules may exist that realize alternatives with little code, which is an instance of Excess of Variant Modules. Null Module denotes that in case of an optional variability, the missing feature has been realized as a separate module without much executable code (similar to a Null Object [Woolf98]), which increases the number of small modules.

When the configuration settings for product line infrastructure code, for example macro `#define`s in C header files, are not located together for the entire product line infrastructure, but spread among several modules, a Non-Cohesive Configuration exists. Configuration Excess means that multiple configuration possibilities exist in the code of a single product line infrastructure, for example macro definitions and settings in non-volatile RAM, where a single configuration possibility would suffice. Explicit Product References, another frequently observed code smell in core assets [Krueger10], denotes that variant directories, source code file names, `#ifdef` macro names, or other variation point identifiers refer to specific products, rather than hiding product-specifics by only revealing product features[16].

Unbounded Combinatorics, the absence of Bounded Combinatorics [Krueger10], denotes that too many combinations exist for configuring the product line infrastructure code, so that more products can be produced than required. If core assets have Lack of Variability, they can typically not be configured enough internally, but must be composed externally, for example by abusing the Pipes and Filters style [Shaw+97] for reuse purposes, which leads to Composition Excess. Lack of Defaults arises if variability optimization possibilities (converting variants to Defaults) have not been used. Neither the common nor the variant elements in core asset code need to be syntactically complete for the programming language used. If, however, a mechanism needlessly

[16] We have repeatedly used the *ifnames* tool (Sec.4.5) to detect such inconsistencies in macro names [Patzke+04, Kolb++06]

enforces more syntactic completeness than necessary, this is a code smell I call Restricted Variant Granularity.

When variation points, for example as procedural interfaces, are only introduced because of a possible, but not yet required variability, they are called Speculative Variation Points. Excess of Variant Similarity means that variants belonging to the same variability have been realized with too many common elements and are thus too similar. Excess of Variant Size is related to the previous code smell and describes a situation when variant elements are needlessly large.

A common trait of all above-mentioned defects is that they make product line infrastructure code evolution more difficult by causing unnecessary complexities in variability management. Thus, they add to evolution difficulties that also exist in single system code – caused by conventional code smells. Not all product line infrastructure code smells are equally detrimental: sometimes, it may even be useful to deliberately commit them. Depending on the product line engineering context (Def.20), they may also be prioritized for successive removal during the following Modification activity (see below).

It is also unproductive if the family engineer tries to detect all kinds of product line infrastructure code smells, independent of their severity, in a single pass. A practical aim is to attack variability management defects incrementally, for example by time-boxing [Bassett97] each Selection step. Another tactic for incrementally reducing variability complexity in product line infrastructure code is to detect (and later remove) at least one larger product line infrastructure code smell per newly realized feature, so that the product line infrastructure code becomes less complex with each iteration.

The outputs of the Selection phase (Fig.38) are 1) a collection of variation areas (source code modules and areas within the source code where modification will occur, Fig.40), 2) potential reference elements that illustrate how a similar variability management task has already been performed in the existing code, and 3) variability management defects that have been detected in the current version of the product line infrastructure code. If variants are to be created, the variation areas will contain latent variation points where variation will newly occur, but which have not been distinguished from the existing code yet. If variants are to be added, similar reference elements will already exist that can serve as templates for realizing the new variabilities if these references do not have severe variation defects.

Modification

In the second phase of the product line realization process, the family engineer actively modifies those elements that he has passively focused on in the preceding selection phase. By including new variability-related elements, the current modification activity makes the resulting product line infrastructure code more asymmetric. In other words, common elements can be seen as symmetries in the product line infrastructure code (they do not change the product line member under the production transformation), variant elements add asymmetries (they change the product line member), and variation points represent symmetry axes.

The family engineer's main goal in this phase is to rapidly change the product line infrastructure code so that it satisfies the new product line requirement. At the same time, the goal is to improve variability-related code quality, so that the resulting code does not become more complex than necessary. In order to achieve the first goal in such a way that the existing system of commonalities and variabilities is least disturbed, the family engineer modifies the code in two successive orthogonal activities, as shown in Figure 39: commonality realization and variability realization.

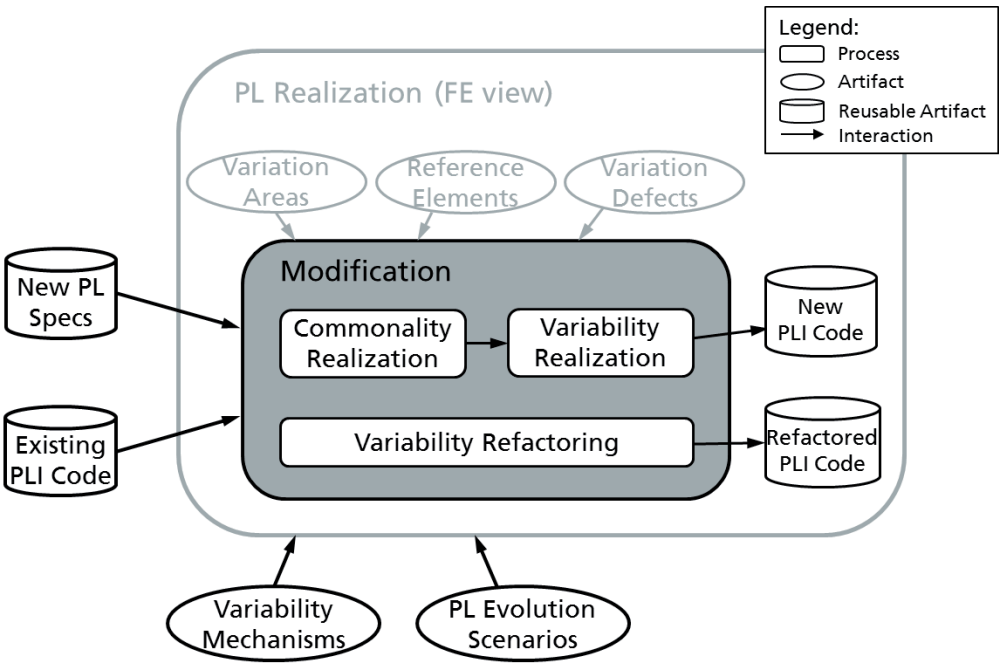


Figure 39: Details of the modification phase

As in the identification phase, the separation into two activities again serves to reduce the family engineer's work load. The separation of use and reuse activities also facilitates independent measurement, testing and optimization of the dual properties of use and reuse, as introduced in Section 2.1. Separating development activities into commonality-

related and successive variability-related ones has also been proposed in [Parnas76], and for the modeling phase in [Muthig02] (cf. Sec.3.2).

In the Commonality Realization phase, the new product line specification is realized within the variation areas of the existing code, as if it were completely common code, developed as part of a single system. I already gave some examples of Commonality Realization during the discussions of product line evolution scenarios in Section 5.1, in particular for Optional Feature Creation and Alternative Feature Creation (also cf. Figure 33). Even if the family engineer is aware of upcoming variability issues, he deliberately ignores them in this step and first focuses exclusively on realizing the new feature. This leads to the production of valuable functionality for the evolving product line as early as possible within the modification phase, which allows the new code to be evaluated for a maximal period. For example, if unit tests are created at the start[17] or at the end of the commonality realization phase, the execution behavior of the new feature can be tested early, so that functionality-related defects and complexities are less likely to propagate into later development phases. Another advantage of ignoring variability-related issues first is that these have yet been kept invariant throughout the product line realization process, analogous to keeping execution behavior invariant during classical refactorings in single system code.

During the Variability Realization phase, the realization of product line requirements is continued by introducing the missing required configuration possibilities in the newly added common code, in accordance with potential reference elements (Figure 39) that have previously been detected in other areas of the current product line infrastructure code. Whereas in Commonality Realization the family engineer has focused on executable properties only, he now places them in the background, mainly concentrating on reuse issues (a similar setup exists in test-driven development when the developer focuses on the testing and development activity in alternation). For example, he may now add an explicit variation point and the corresponding configuration facilities. Because he previously ignored variability issues, he might need to consult the previous version of the product line infrastructure code, available under configuration management, in order to identify code that realized previous features which have been eliminated during Commonality Realization. Note that not all newly introduced common code is likely to become a variant now: some new elements may remain common, while others in their neighborhood become variable. This is also the reason why I distinguish between "variation areas" and "variation points". In terms of symmetry considerations, the code outside the variation areas remains invariant, indistinguishable from the

[17] In case of test-driven development [Beck02, Meszaros07], tests are created first.

state before. Inside the variation areas, some code may have been created that becomes a common element of the product line infrastructure, indistinguishable among individual systems, while other new code is variable, asymmetric, both across space and time. But at the time the variation areas are realized, variation points are not completely determined. This is illustrated in Figure 40.

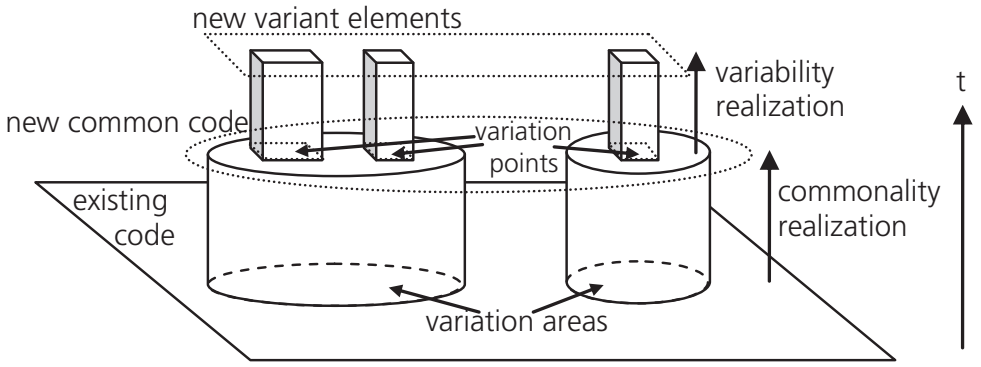


Figure 40: Commonality realization and variability realization over time

Other inputs to the Variability Realization phase are variability mechanisms and product line evolution scenarios, as shown in Figure 39. Variability mechanisms represent possibilities for realizing the variability management task at hand, which is an instance of a product line evolution scenario. Variability mechanisms are used and configured according to the reference elements, but also in the most easily applicable way, to keep complexity low. Variability optimization possibilities, for example likely defaults or variation points that need consolidation, may be documented in this step, but performing the optimization shall be deferred to a later process step.

In terms of mistake avoidance, another advantage of organizing commonality and variability realization in two separate steps besides the above mentioned avoidance of propagating behavioral errors is that possible errors in the variability realization phase only lead to backtracking within this phase (cf. the discussion on the sequence of steps in Fig.35). In other words, if an error is made during variability realization, at most the activities in this phase must be undone and repeated, but not those in the commonality realization phase. This leads to a mistake avoiding and more productive overall process, compared to a situation in which the two activities are intermixed. Performing commonality realization and variability realization in two successive steps is an example of a nearly backtrack-free sequence, one that is relatively stable across different development contexts: a reusable sequence.

A third Modification sub-activity is called variability refactoring (Figure 39). In this parallel activity to Commonality and Variability Realization, the family engineer repairs defects in the existing code which he has

detected during the Comprehension sub-activity in the previous Selection phase (Figure 38). As in the other sub-activities, the family engineer focuses only on a particular aspect, in this case on variability-related defects in the existing code. For example, he changes variability-related elements such as macro names to make them more consistent. Because this activity is performed in parallel to the other modification activities, the two may be distributed to several family engineers. However, it is important that the variability refactoring activity is only concerned with the previous code, not the newly developed one. This consolidation of the entire new product line infrastructure code will happen in the following larger quality assurance phase (Figure 36).

Variability refactorings have been defined in Section 2.3 (Def.69). They are transformations of the product line infrastructure code which serve to make the evolving artifacts less complex while the artifacts are evolved locally. They are similar to conventional refactorings [Fowler99] for single system code because both improve code quality while preserving an aspect of it, leaving it invariant, indistinguishable, symmetric. Conventional refactorings are concerned with properties of use (Def.6), for example by making the code easier to use for a developer, or by increasing its efficiency in time or space while maintaining its functionality. On the contrary, variability refactorings aim at making the product line infrastructure code more evolvable while it is growing, serving both the product line engineer as a user and reuser. Variability refactorings preserve those elements of the existing product line infrastructure code that lead to the formation of the existing products, which often means that not only the code's executable behavior appears to be unchanged to an end-user, but also that its structures remain invariant, as seen by an application engineer using the single product. However, the product line infrastructure code is enhanced while new features are added. The goal of variability refactorings is to counteract product line infrastructure code smells (Table 7).

Table 8 lists 37 variability refactorings, with increasing detail, which I have identified as different Modification sub-activities that let a family engineer change a product line infrastructure, especially its code, in order to make it cheaper to evolve or reuse (see Def.69), without changing the functionality of any product line member. This is one common invariant in all mentioned refactorings.

Replace Variant Element with Commonality (1) is used to decrease variability complexity when there is a particular variant element in the product line infrastructure code, such as an initialization algorithm for some embedded system sub-device which may be present in all product line members, without affecting the functionality of those product line members that do not need it. If this is the case, the sub-refactoring Remove Variation Point (24) may be applied for the particular variant

element. If the variant only consists of a single variant element (i.e., if the variant is a simple variant), the corresponding configuration option can also be removed from the respective variability assets (Def.57), so that the variant vanishes. In case of an optional variant, this amounts to removing the respective variability alltogether. The product line infrastructure code is less complex after this refactoring because one unneeded variation point has been eliminated, counteracting the product line infrastructure code smell Variation Point Excess.

No.	Name
1	Replace Variant Element with Commonality
2	Replace Commonality with Variant Element
3	Separate Variant from Commonality
4	Inline Commonality and Variant
5	Separate Variants from Each Other
6	Inline Variants
7	Replace Closed with Open Variant
8	Replace Open with Closed Variant
9	Extract Reuse Hierarchy
10	Flatten Reuse Hierarchy
11	Replace Product Reference with Feature Reference
12	Fork Core Asset
13	Consolidate Clones
14	Replace Variant Element with Default
15	Replace Default with Variant Element
16	Replace Commonality with Default
17	Limit Combinations
18	Decrease Variant Dependencies
19	Split Variant
20	Consolidate Variant Elements
21	Decrease Variant Element Size
22	Increase Variant Element Size
23	Create Variation Point
24	Remove Variation Point
25	Increase Variation Point Visibility
26	Make Variation Point Programming Language-Independent
27	Make Variation Point Programming Language-Dependent
28	Rename Variation Point
29	Replace Runtime Binding with Execution Time Binding
30	Replace Execution Time Binding with Runtime Binding
31	Replace Execution Time Binding with Construction Time Binding
32	Replace Construction Time Binding with Execution Time Binding
33	Replace Manual with Automated Binding
34	Replace Automated with Manual Binding
35	Separate Application from Variation Logic
36	Extract Variability Asset
37	Consolidate Configuration

Table 8: Variability refactorings

The opposite refactoring, Replace Commonality with Variant Element (2), is used if a common element in the product line infrastructure code contains elements such as a data structure or an algorithm that are not needed in all product line members, but only in a particular variant. In this case, the sub-refactoring Create Variation Point (23) is applied, and the respective variability assets are updated. The affected product line members become less complex, at the expense of a slight complexity increase in the product line infrastructure, caused by the code smell Lack of Variability.

Separate Variant from Commonality (3) is used if a core asset realizes both commonality and variability, but both of these have different change rates. For example, the code history may indicate that the common elements have been stable, while the variant elements in the same core asset have frequently changed during product line evolution, and this is expected to continue. Another example is that older and current core assets need to be evolved together, variant elements change, and these changes must be back-propagated more easily. In these cases, the sub-refactorings Replace Closed with Open Variant (7), Extract Reuse Hierarchy (9), or Increase Variation Point Visibility (25) are applied to consolidate common and variant elements according to their change frequency (see Fig.9c), which makes them easier and cheaper to evolve in the future and reduces the code smell Coupling of Common Elements and Variants.

The opposite refactoring, Inline Commonality and Variant (4), serves to reduce the number of modules, especially if there are small variant modules or if common and variant modules have been co-evolving in the past and are expected to co-evolve in the future. By applying the sub-refactorings Replace Open with Closed Variant (8) or Flatten Reuse Hierarchy (10), common and variant elements are grouped closer together (see Fig.10a). This trades off ease-of-change with decreased visibility of variants.

Separate Variants from Each Other (5) and Inline Variants (6) are a similar pair of refactorings. Separate Variants from Each Other is used if a core asset only contains related variants, such as all alternatives realizing an alternative variability, but if some of these variants have had a different change frequency than the others. The sub-refactorings Replace Closed with Open Variant (8) or Increase Variation Point Visibility (25) help to consolidate related variant elements (see Fig.10b). This makes them less complex to evolve, counteracting the code smell Coupling of Alternative Variants. As in Inline Commonality and Variant (4), the goal of Inline Variants is to reduce the number of modules, and another goal is to keep related variants closer together, especially if they are small (see Fig.9a). This reduces the code smell Excess of Alternative Modules.

The goals of Replace Closed with Open Variant (7) are to make variants easier to detect, and to isolate variants from each other (see Tab.2). The precondition of this refactoring is that variants have been realized with a closed variability mechanism such as Conditional Execution or Conditional Compilation, so that they form a fixed set, and a change to any variant may lead to a corruption of others because they share the same module. The postcondition is that all variants are separated from each other, so that any change to one of the variants is guaranteed not to affect others. The resulting variability mechanism is an open one, such as Polymorphism or Frame Technology, and it may prepare the code for a successive Extract Reuse Hierarchy (9) refactoring. Replace Closed with Open Variant typically leads to a variability mechanism change because most mechanisms are either closed or open (the general form of Frame Technology is an exception, supporting both open and closed variation). The code smells Coupling of Common Elements and Variants and Coupling of Alternative Variants are reduced by this refactoring.

Replace Open with Closed Variant (8) is the opposite refactoring, with reversed pre- and postconditions, serving to reduce the number of variant modules. The refactoring makes variants harder to detect because after refactoring they reside in the same module. On the other hand, this may be beneficial if the variants evolve together. This refactoring trades off coupling between common/variant elements with module quantity. It is associated with the Flatten Reuse Hierarchy (10) refactoring.

Extract Reuse Hierarchy serves to establish a hierarchy based on the reuse relation (Sec.2.2), which is beneficial because it may capture intermediate levels of reuse. The precondition of this refactoring is that the variants are closed, or if they are open no explicit attempt has been made to establish a deeper reuse hierarchy. In Frame Technology, and in symmetrical Aspect-Orientation, deeper reuse hierarchies can be established by reorganizing core assets according to the decisions presented in Fig.11. The postcondition is that the reuse hierarchy has become at least one level deeper. This refactoring counteracts the code smell Nested Variation Points. The opposite refactoring, Flatten Reuse Hierarchy (10), serves to reduce unnecessary reusability gradients.

As mentioned above, a typical code smell in practice is Explicit Product References. This means that core assets are not ignorant of specific product line members, but contain variation points, visible as macro names or module names that refer to particular products. In the variability refactoring Replace Product Reference with Feature Reference (11), these core assets are refactored in such a way that they only refer to features, which makes the product line easier to evolve as new products are added or old products are removed or change their name. There is no opposite refactoring for this one because having explicit product references is always undesirable.

Fork Core Asset (12) denotes that some elements of the product line infrastructure are deliberately cloned during evolution, in order to improve evolution speed for that element while simultaneously avoiding the risk to introduce defects into any existing product line member. The family engineer may use this refactoring in early realization phases of new larger features, in order to evaluate the consequences, and with the intention to perform the opposite refactoring, Consolidate Clones (13), soon after the necessary changes have been made. The precondition is that an appropriately small set of core assets to clone has been located, and that it is sufficiently uncertain how the required changes will affect the existing core assets. On the other hand, if the type and location of the variation point is clear in advance, there is less reason to use this refactoring. The main benefits of Fork Core Asset are short-term: the longer Consolidate Clones is postponed, the more effort tends to be required. However, as discussed in Section 3.3, consolidation may not be needed in any case, for example if the necessary deviation between the clone origin and the clone has become large, or if the clone origin has become obsolete. See also Section 4.1.

Besides deep reuse hierarchies, defaults serve to optimize reuse by providing a third element in between a common element and a variant. The refactoring Replace Variant Element with Default (14) can be used as a sub-refactoring of Replace Variant Element with Commonality (1), or as an alternative to Extract Reuse Hierarchy (9), in cases when a particular variant element is common for most product line members, but not for all. As in Replace Variant Element with Commonality, the corresponding configuration option can be removed from the corresponding variability assets if the variant is simple. Unlike in Replace Variant Element with Commonality, however, the variant does not vanish completely, which can be an advantage if this refactoring needs to be reverted later by the opposite refactoring Replace Default with Variant Element (15). The product line infrastructure becomes less complex because most of the time the respective variation point of the Default can be ignored when configuring products. Note that many variability mechanisms, such as Conditional Execution or Polymorphism, only have weak support for defaults, and Module Replacement is conceptually incapable of expressing defaults. The best support for defaults is provided by Frame Technology, but also Conditional Compilation and Aspect-Orientation provide it.

Replace Commonality with Default (16) is a useful refactoring in cases when the product line infrastructure code contains elements such as a data structure or an algorithm that are not needed in most product line members, but a minority requires them. Similar as in Replace Commonality with Variant Element (2), the sub-refactoring Create Variation Point (23) is applied, but the variation point remains deactivated by default.

Limit Combinations (17) is a product line-specific refactoring for reducing the Unbounded Combinatorics smell in many types of core assets, not just code. A recent publication explains how this refactoring is performed in a commercial product line tool [Krueger10].

A similar refactoring is Decrease Variant Dependencies (18) which counteracts the code smell Nested Variation Points. Many core assets in practice, especially those without explicit variability assets, contain hard-coded variant dependencies. The goal of this refactoring is to extract these dependencies into variability assets in the sub-refactoring Extract Variability Assets (36). The variability assets are then used to configure the core assets.

Split Variant (19) is used in situations when a core asset contains a large simple variant (i.e., a variant consisting of only one variant element) that consists of considerable commonality. In this case, the variant may be split into two or more variant elements, using the sub-refactoring Create Variation Point (23), in order to increase commonality, tolerating an increase of variation points. The opposite refactoring is called Consolidate Variant Elements (20).

In case a Restricted Variant Granularity code smell exists, a variant element may also cover more "space" than necessary in product line infrastructure code because the existing programming language-dependent variability mechanism enforces this. For example, Module Replacement may enforce a variant element to be realized as an entire function, even though only a partial algorithm varies among the products. The Decrease Variant Element Size (21) refactoring would change the code in such a way that the variant only uses as much "space" as necessary. Like the Replace Closed with Open Variant (7) and Replace Open with Closed Variant (8) refactorings, Decrease Variant Element Size usually leads to a variability mechanism change. The opposite refactoring, Increase Variant Element Size (22), may be appropriate in cases when this improves code comprehensibility, even if this leads to the code smell of Duplicated Code.

Create Variation Point (23) is an elementary refactoring used in many of the above mentioned refactorings, such as Replace Commonality with Variant Element (2), Consolidate Clones (13), or Replace Commonality with Default (16). The precondition is that in a core asset, a common element exists, and that new variant elements are due to evolve near that common element (see Fig.40). The postcondition is that a new variation point exists, both in the core asset and in related variability assets, such as configuration files. As a variability refactoring, Create Variation Point does not just preserve the functionality of all produced products (because functionality is not changed), but it also preserves the construction semantics of the product line assets. This means that the output of the construction interpreter (Def.25) before the refactoring

(and the input to the execution interpreter (Def.5)) is indistinguishable from the output of the construction interpreter after the refactoring. The opposite refactoring is Remove Variation Point (24). Both refactorings are not specific to source code, but may be applied to other types of product line assets as well.

An important refactoring is Increase Variation Point Visibility (25), counteracting the code smell Ambiguous Variation Points. In its precondition, the core asset contains variation points which the family engineer cannot easily see, for example due to a clone. In this very situation, the variation point is made more visible by providing tags in the cloned code, converting it to Templating (Sec.4.1), or by providing references in variability assets (Def.57). The refactoring may also lead to a variability mechanism change, for example to Conditional Compilation which provides clearly visible variation points, realized by `#ifdef`s. It may also just lead to semi-visible (ambiguous) variation points, as provided by Aspect-Orientation. But the postcondition in all cases is that variation point visibility increased. As in Replace Product Reference with Feature Reference (11), an opposite refactoring is not given, as a decrease in variation point visibility is not desirable.

The two elementary refactorings Make Variation Point Programming Language-Independent (26) and Make Variation Point Programming Language-Dependent (27) are sub-refactorings of Decrease/Increase Variant Size (21/22). Make Variation Point Programming Language-Independent gives the family engineer more degrees of freedom in realizing variants because after the refactoring has been performed, the variation point may still or may not be at boundaries imposed by the programming language, for example at procedural boundaries. On the contrary, Make Variation Point Programming Language-Dependent removes this degree of freedom.

Another elementary and code-independent refactoring of product line assets is Rename Variation Point (28). It is used, for example, as a sub-refactoring of Replace Product Reference with Feature Reference (11), and serves to make all variation points more consistent. It is simultaneously applied to core assets and related variability assets. It is a harmless refactoring that never alters construction semantics. As in similar conventional refactorings such as Rename Class [Fowler99], the postcondition is that renaming has taken place and the new name has not existed before.

A set of variability refactorings is concerned with changing the binding time, trading off usability (Def.7) with reusability (Def.19) because, as mentioned in the refactorings on programming language-dependence above, the later the binding happens, the less degrees of freedom exist for a family engineer to make product line infrastructure code reusable

(see also [Bassett97, p.13]). For an overview of binding issues in product line engineering, see Fig.7 and Fig.9.

The goal of Replace Runtime Binding with Execution Time Binding (29), from a family engineering perspective, is to be able to customize (Def.29) product line infrastructure code at all, in this case at least through composition (Def.9). As a precondition, the variant has been bound at runtime (including startup time), for example through Conditional Execution. While some development processes require such late binding, typically for example because of fine-grained calibration of range variations in embedded systems software after deployment in a physical environment, it makes the software overly rigid, as discussed for the code smells of Runtime Variation and Startup Initialization. In most cases, except for specific types of Aspect-Orientation that provide both runtime and execution time binding, this refactoring requires a variability mechanism change, for example towards Module Replacement. The postcondition of Replace Runtime Binding with Execution Time Binding is that composable variants exist which can be used (Def.6) by the product line engineer to customize product line infrastructure code. The opposite refactoring, Replace Execution Time Binding with Runtime Binding (30), is detrimental for a product line as a set of similar systems (Def.23) because it leads to a single, fixed and overly complex system realizing the union of all features, rather than providing just the required ones. This refactoring only makes sense if the development context changes in such a way that late binding becomes a must.

Replace Execution Time Binding with Construction Time Binding (31) serves to improve mass customization of product line infrastructure code because it replaces large-grained composition (Def.9) with fine-grained configuration (Def.28). In other words, this refactoring improves reusability by converting from use (Def.6) to reuse (Def.21). As a precondition, the respective variants are realized in fixed modules, as artifacts without variability. Besides counteracting this Lack of Variability code smell, the current refactoring also avoids Composition Excess and Ambiguous Variation Points. In many cases, this refactoring leads to a variability mechanism change. The postcondition is that the product line infrastructure code is easier to configure. Note that the two refactorings Replace Runtime Binding with Execution Time Binding and Replace Execution Time Binding with Construction Time Binding may be executed in succession, for example when replacing Conditional Execution with Conditional Compilation. The opposite refactoring, Replace Construction Time Binding with Execution Time Binding (32), leads to more variability complexity because it makes variation points dependent on programming language semantics. However, that refactoring may be required in a changing development context when source code access becomes restricted, for example when only binary modules shall be exchanged between providers and suppliers of COTS

(commercial-of-the-shelf) components, due to intellectual property rights issues.

Another pair of refactorings is also related to binding issues. Replace Manual with Automated Binding (33) aims at automatic production, reducing application engineering effort and avoiding the Duplicated Code smell, while Replace Automated with Manual Binding (34) may become useful as part of the Fork Core Asset (12) refactoring.

Separate Application from Variation Logic (35) counteracts the code smell Coupling of Application and Variation Logic, which may be present in a precondition where Runtime Variation exists. As a consequence of applying this refactoring, Nested Variation points may appear. Nonetheless, the refactoring is useful because it improves code comprehensibility.

Extract Variability Asset (36), another code-independent refactoring, describes a situation in which core assets (Def.56) have existed within a product line infrastructure (Def.62), and in particular in its product line assets (Def.59), but an explicit, consolidated variability asset (Def.57) has been missing for these core assets. For example, variability information may not have been captured outside the core asset, impeding traceability of variability, or multiple separate configuration possibilities may have obscured how to configure the asset. As a postcondition, this information has been captured in an orthogonal asset, consistent with other variability assets. As a special case, the variability asset serves as a configurator, which results in the refactoring Consolidate Configuration (37) which counteracts the code smell Excess of Configuration Mechanisms and leads to a clean configuration interface, reducing variability complexity.

As shown in Figure 39, the outputs of the modification phase are the new product line infrastructure code, created during commonality and variability realization, and refactored elements of the existing code in which variability management was simplified. These artifacts become inputs to the final phase of the product line realization process, quality assurance.

Quality Assurance

In the third phase of the product line realization process, the family engineer makes sure that the product line infrastructure code at hand that has been selected and modified in the previous steps is easy to understand, evolve, use, and reuse. For example, he evaluates if the newly introduced variability mechanisms are consistent with the previously existing ones, or he tests if all required product instances can be configured. The family engineer also performs code measurements to document how code reuse complexity is changing over time.

A standard definition of quality assurance is "a planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements" [IEEE610]. Typical software quality assurance activities comprise testing, inspections, and measurement of an artifact's internal quality characteristics. In the context of the product line evolution method, novel quality assurance issues arise because the code artifacts realize variability, which make them generic and reusable, adaptable to multiple contexts of use. The overall goal is keeping the code reusable and sustainable as it evolves. Thus, quality assurance activities are concerned with testing that the code is reusable as required, and measuring that the code's reuse complexity remains manageable. A precondition of the measurement activity is that the code is as reusable as required, which is why the testing activity precedes measurement.

Some sub-activities in previous process phases have already been concerned with quality assurance. Within the selection phase, the comprehension activity (Figure 38) identifies variability-related defects in existing code, which do not prevent each product to be configured at all, but which unnecessarily complicate this activity. Likewise, as part of the modification activity, the variability refactoring phase (Figure 39) serves to improve the existing product line infrastructure code by eliminating obvious "code smells". However, both activities exclusively focus on the previously existing code, in order to concentrate on one issue at a time. The present separate quality assurance phase is concerned with both the new and the refactored product line infrastructure code together, as illustrated in Figure 41. The product line testing sub-activity is discussed in this section, while Section 5.3 will elaborate on variability complexity measurement.

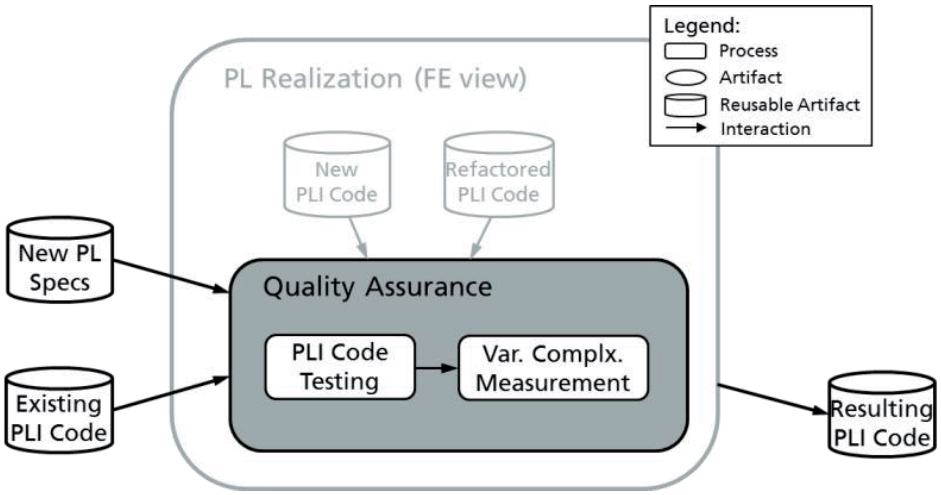


Figure 41: Details of the quality assurance phase

Besides the output artifacts of the modification phase, there are two more input elements to the quality assurance phase (Fig.41). The new

product line specifications are used in the testing sub-step to ensure that all required product line instances can be configured. The existing product line infrastructure code is used in the measurement sub-activity in order to capture to what extent variability complexity has changed in product line infrastructure code during the previous modification activity. The output of the quality assurance activity is the resulting code of the product line evolution method, which will become the existing code in the next iteration of the overall process (Figure 37).

Software testing has been defined as "1) the process of operating a system or component under specified conditions, observing or recording their results, and 2) the process of analyzing a software item to detect the differences between existing and required conditions, and to evaluate the features of the software items" [IEEE610]. Testing has two main goals [Northrop+07]: 1) helping to identify faults that lead to failures so they can be repaired and 2) determining whether the software under test can perform as specified by its requirements. Testing the required variability in the developed product line infrastructure code is the aspect of testing that is peculiar to product lines.

This means that the family engineer's main goals in product line testing are 1) to identify faults in managing the variant elements of a product line infrastructure, so that they can be repaired, and 2) to determine whether the product line infrastructure code under test can be composed and configured as required. Besides these construction issues, execution properties such as functionality or efficiency become secondary issues. Where conventional unit tests, such as those created in Test-Driven Development [Beck02, Meszaros07], execute the code under test, recording its conformance to expected behavior and measuring its execution time, product line infrastructure code tests must primarily construct, configure and compose individual systems from product line infrastructure code, recording if the product construction process succeeds as expected, and measuring which construction resources are needed. Another example of the different testing goals is that conventional single system testing often tries to cover all combinations of execution paths, whereas product line testing aims to cover all combinations of construction paths that result in the required product line members, irrespective of their execution behavior.

In order to account for both the construction and execution properties, product line infrastructure code testing consists of two separate orthogonal testing activities, called construction testing and execution testing. As Figure 42 illustrates, the inputs to the testing phase are the new and the refactored product line infrastructure code obtained in the previous modification phase (Figure 39).

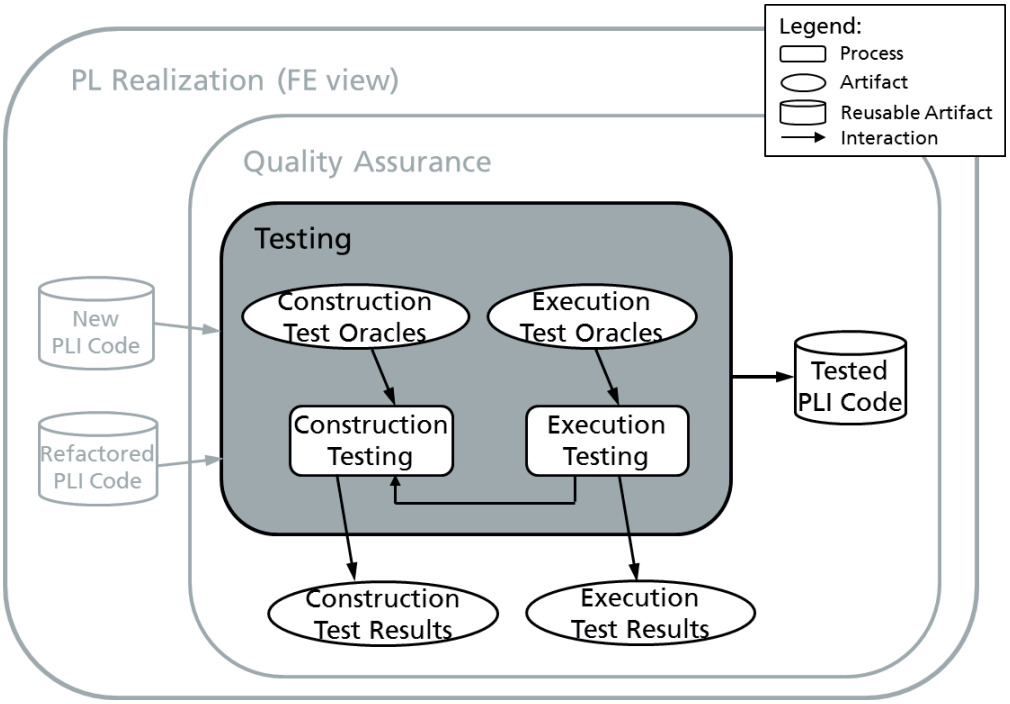


Figure 42: Product line infrastructure code testing phase

For a construction test to pass, its construction test oracle – the mechanism which determines if the test has passed or failed – needs to specify if the construction outputs are structurally identical (if irrelevant details are omitted) to the expected executable modules of the constructed product line instance. For example, it may suffice that the resulting code can be compiled or linked without errors for the construction test to pass. Other times, a construction test may compare the sizes of the constructed modules against expected values. Or it may compare the constructed modules against reference modules that have been obtained by Cloning, ensuring that all specified common and variant elements have been included, and that no undesired elements have been built. Construction testing is a novel concept that has not yet been considered in the product line testing community [Pohl++05, Geppert++04, Geppert++05, Knauber++06, Knauber++08, Neto++11]. Construction testing has been applied throughout the following case study, as documented by construction test results (for example, see Listing 27 in Appendix C).

The following execution test will ensure that the product instance behaves as expected at execution time using conventional execution test oracles. The execution of common elements can be tested with corresponding common tests, while the functionality of variant elements is tested by variant tests which are instantiated alongside their testees in the previous construction testing phase. If the code is organized in a reuse hierarchy, and if test code evolves at the same rate than the code of its testee, they should be organized together in the same constructible

module (cf. Figure 10a). This also supports traceability (Def.58) between the test code to the code being tested [Northrop+07].

Both testing activities may be automated, running in succession or interleaved. By separating the product line testing activities into two phases, the family engineer may evaluate and improve the construction and execution qualities of the code in isolation, as long as the applied variability mechanisms do not blur the distinction, which always happens when runtime mechanisms such as Conditional Execution have been used. More details of the interaction between Construction Testing and Execution Texting are shown in Fig.43 (comp. [Jalote05, Fig.10.1]).

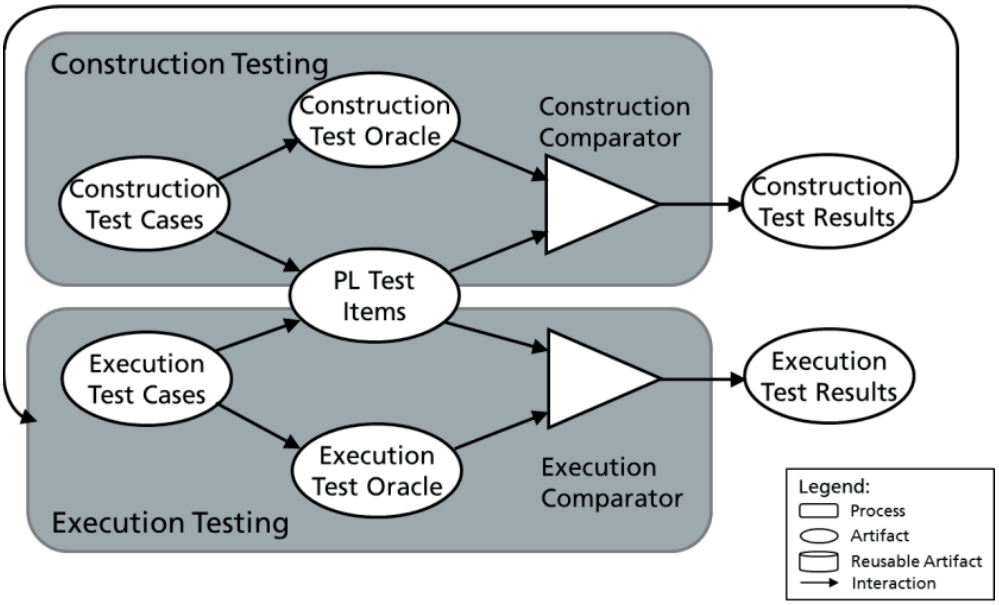


Figure 43: Interrelationship between Construction Testing and Execution Testing

The outputs of product line testing (Fig.42) are construction test results, focusing on the product line infrastructure code, execution test results dealing with individual product instances, and the tested product line infrastructure code. As part of the iterative product line realization process, the test results capture the quality trend of the evolving product line infrastructure code, both in space and in time. Depending on the severity of the detected defects, the product line infrastructure code is either repaired immediately, for example when a required variation is missing, or during the next larger iteration. Less serious defects may also be ignored. In this case the quality is regarded "good enough".

The ongoing simplification process is completed by a final quality assurance phase – complexity measurement.

5.3 Variability Complexity Measurement

The variability complexity measurement activity complements the sustainable product line evolution method (Figure 37), as summarized in Figure 32. This final quality assurance sub-activity is concerned with measurement in the resulting product line infrastructure code (Figure 41), more precisely with complexity measurement. In general, a quality metric is 1) a quantitative measure of the degree to which an item possesses a given quality attribute, and 2) a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute [IEEE610]. Until recently, product line research in this area did not exist [Knauber04, p.8]. Meanwhile, there are a few exceptions [Ajila+07, Lopez+08] which do not address the measurement goal (see Sec.3.5). This also applies to other metrics work discussed in Section 3.4 [Hall+00, Kelly06] which only addresses the evolution of conventional single system code over time, but not evolution in time and space.

In the variability complexity measurement phase under discussion the family engineer's main goal is to assess if the product line infrastructure code is simple enough for sustainable variability management, with regard to the current requirements, or if it is becoming significantly more complex than necessary. Two points are notable here: First, the goal is to make the code as simple as necessary, not to make it as simple as possible. The latter could result in excessive effort and would make the product line evolution method non-applicable for E-type software product lines in practice. Second, the goal is concerned with the currently existing requirements, excluding speculative future requirements. This makes the step sharply reactive: it fights all unnecessary speculative elements in the code that may have been caused by the family engineer being proactive, for example by adding variation points that are not needed yet.

The main goal is also not to measure product line-related properties free of context, for example the number of variation points or the percent of reuse. Instead, the purpose is to collect only those measures that make the product line simpler to use, evolve and reuse for the average developer. A similar point was addressed in the SEI's current Framework for Software Product Line Practice: "A higher level of software reuse is not, in itself, an end goal of a product line effort but merely a strategy for achieving goals such as shorter time to market" [Northrop+07]. This also means that it is not sufficient to perform the measurements once, but continually and in iterations (as in Continual Integration [Duvall++07]), in order to become aware of complexity trends. Figure 44 summarizes the details of the variability complexity measurement phase, its inputs, sub-activities and outputs.

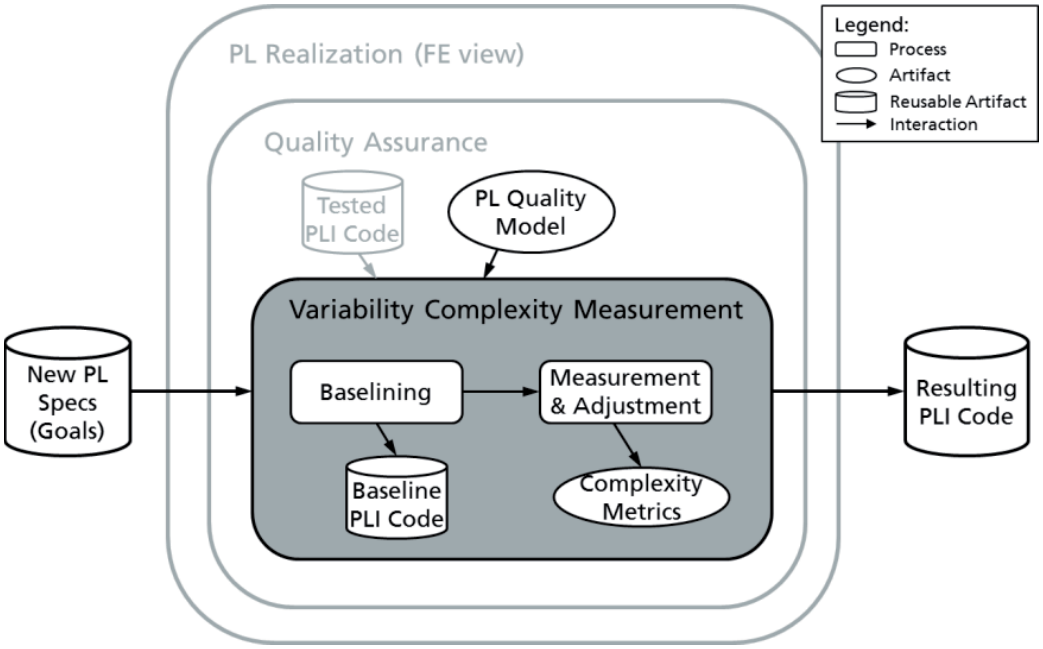


Figure 44: Variability complexity measurement phase

There are three input elements to this phase: the tested code obtained from the previous product line infrastructure code testing phase (Figure 42), a quality model for evolving product line infrastructure code, and the product line requirements for the current iteration, stating the quality goals. Three output artifacts result from the variability complexity measurement phase: the resulting code of the overall product line evolution method, plus baseline code and complexity metrics of the current iteration. The measurement activity is composed of two consecutive sub-activities. In the first, called Baselining, the family engineer identifies or sets up baseline product line infrastructure code which serves as a reference for evaluating the existing code. For example, a particular version of the code may be defined as a temporal reference for all following measurement activities in the evolving code, as indicated in Section 3.4 (Figure 22). The second sub-activity, Measurement and Adjustment, is concerned with executing the measurements and possibly adjusting the existing product line infrastructure code so that it becomes easier to evolve and reuse. For example, as a result of measurements that detect the code smell Runtime Variation (Tab.7), the respective refactoring Replace Runtime Binding with Execution Time Binding (Tab.8) may be applied. Both measurement sub-activities depend on a product line quality model and corresponding product line measurement goals which are provided as part of the product line specifications.

Product Line Quality Model

The quality model for evolving product line infrastructure code motivates which product line-specific quality attributes must be addressed in the

measurement phase. A reference quality model is presented next which is customizable to an organization's product line development needs. The quality model has been developed using the Goal-Question-Metric (GQM) approach [Solingen++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 45 shows the goals and sub-goals needed in the complexity-aware product line evolution method.

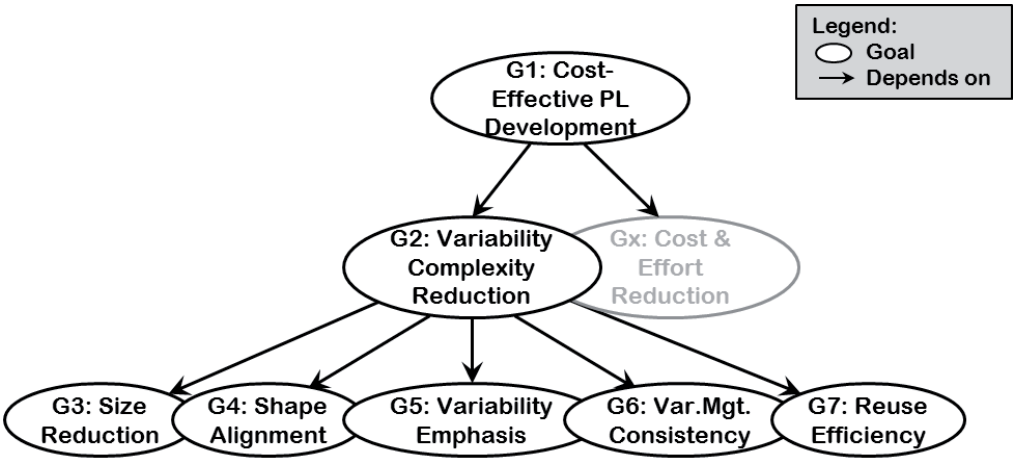


Figure 45: Goal hierarchy of the product line infrastructure code quality model

The overall goal is cost-effective product line development (G1). This goal ultimately aims at reducing unnecessary development costs and increasing development productivity from product line inception to retirement. It is related to the top tier of the product line methodology presented in Section 3.5 (Fig.19). In order to achieve the main goal, variability complexity reduction is proposed as a sub-goal (G2), related to the base tier in Fig.19. Complexity reduction means balanced reduction of variability complexity in product line infrastructure code, but only as far as necessary in the particular development context (Def.20). As the current thesis focuses on family engineering and variability complexity is produced there, application engineering issues are not further discussed. Software evolution research is aware of the fact that complexity results in development effort ("In the maintenance phase complexity determines […] how much effort will be required to modify program modules to incorporate specific changes [Curtis79]" [Eden+06]). As will be seen in a moment, this has been addressed in the current thesis by refining the Complexity Reduction goal accordingly (Q3 in Tab.10). This is also investigated in the case study (see the Effort Reduction sub-section of Sec.6.4).

There are five mostly orthogonal sub-goals for achieving complexity reduction. The first sub-goal is size reduction (G3), which means that the product line infrastructure code is constrained in size, as required for the in the current variability management context. The second sub-goal is code shape alignment (G4), which denotes that the storage and distribution of common and variant code elements sufficiently realize the required variability management tasks (see also [Pohl++05, Ch.4]). For example, different groups of variability mechanisms are sufficient for expressing optional variabilities than alternative variabilities. Or, depending on the evolution rates of common and variant code elements, it may be advisable to separate them into different modules, or to keep them together. The third sub-goal is concerned with emphasizing variant elements (G5). This means that family engineers can easily see and control those elements of product line infrastructure code which are different across space or time, while at the same time the common elements which always remain the same for all products are more suppressed [Bassett97, p.87]. The fourth sub-goal of complexity reduction addresses variability management consistency[18] (G6) because inconsistent realizations of variability are harder to evolve than necessary. The fifth 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 whose answers indicate to what extent the goal has been reached. The questions are then refined by concrete metrics. Tables 9 to 15 refine the goals from Figure 45 into corresponding questions.

Analyze the	product line realization process
for the purpose of	**reducing**
with respect to	**whole life cycle cost**
from the viewpoint of the	product line engineering manager

Q1: What is the cost of creating a product line?
Q2: What is the cost of sustaining a product line infrastructure?

Table 9: Goal G1 and questions: Product line development cost reduction

Analyze the	code of software product lines
for the purpose of	**reducing**
with respect to	**variability complexity**
from the viewpoint of the	product line engineer

Q3: What is the effort of adding, removing or changing a feature realization?
Q4: Are variation points harder to detect than necessary?
Q5: Are variant elements harder to add than necessary?
Q6: Are common elements harder to change than necessary?

Table 10: Goal G2 and questions: Variability complexity reduction

[18] Consistency is also known as Conceptual Integrity [Brooks10, p.70]

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

Q7: How large is the code? (Which code size is necessary?)
Q8: How much product line infr.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?)

Table 11: Goal G3 and questions: Product line infrastructure code size reduction

Analyze the	code of software product lines
for the purpose of	**balancing**
with respect to	**shape**
from the viewpoint of the	family engineer

Q11: How deep and wide is the code reuse hierarchy?
Q12: What is the runtime cyclomatic complexity? (What value is necessary?)
Q13: What is the construction time cyclomatic complexity? (What value is necessary?)

Table 12: Goal G4 and questions: Product line infrastructure code shape alignment

Analyze the	code of software product lines
for the purpose of	**emphasizing**
with respect to	**variant elements**
from the viewpoint of the	family engineer

Q14: How many variant elements are visible at the module level? (How many must be?)
Q15: How many variant elements are visible module-internally? (How many must be?)
Q16: How many variant elements are indistinguishable from common code?

Table 13: Goal G5 and questions: Variability emphasis

Analyze the	code of software product lines
for the purpose of	**keeping**
with respect to	**variability management consistency**
from the viewpoint of the	family engineer

Q17: How consistently is each variability mechanism used? (What is the trend?)
Q18: How consistently are all variability mechanisms used? (What is the trend?)
Q19: How consistent is the configuration? (What is the trend?)

Table 14: Goal G6 and questions: Variability management consistency

Analyze the	code of software product lines
for the purpose of	**improving**
with respect to	**reuse efficiency**
from the viewpoint of the	product line engineer

Q20: To which degree have reusable modules been reused?
Q21: How many defaults exist in the code? (How many must exist?)
Q22: How similar are variant "siblings"? (How similar must they be at least?)

Table 15: Goal G7 and questions: Reuse efficiency

In the next step of the GQM method, metrics are assigned to each question. Goals G1 and G2 are not further refined here, as they represent super-goals partially made by management which are fulfilled if their sub-goals are fulfilled. Table 16 lists the five most concrete sub-goals G3 to G7 from the bottom of the goal hierarchy (Figure 45), their questions and metrics. For some metrics, there is a second comparison value which captures the necessary metric. The distance between the actual and the necessary metric is an indicator of complexity (see Sec.3.4). The necessary kinds of metrics are estimated by using baselines or reference code, which will be explained in the following Baselining subsection. Each metric is explained next.

G	Q	Metric name	Description
3	Size reduction		
	7	LOC	Lines of code for entire product line infrastr. code
	8	$\nabla_{LOC,t}$	Temporal code churn in lines of code
	9	NOM	Number of modules
	10	NVP	Number of variation points
4	Shape alignment		
	11	DRH	Depth of reuse hierarchy
		WRH	Width of reuse hierarchy
	12	$v(G)_{rt,closed}$	Cyclomatic complexity of closed runtime conditions
		$v(G)_{rt,open}$	Cyclomatic complexity of open runtime conditions
	13	$v(G)_{ct,closed}$	Cycl. compl. of closed construction time conditions
		$v(G)_{ct,open}$	Cycl. compl. of open construction time conditions
	14	LOC_{ad}	Lines of code of adaptees
5	Variability emphasis		
	14	NV_e	Number of externally visible variant elements
	15	NV_i	Number of internally visible variant elements
	16	NV_a	Number of ambiguous variant elements
6	Variability management consistency		
	17	$NInc_{VM\text{-}usage}$	Number of inconsistent usages of a variability mech.
	18	$NInc_{VM}$	Number of inconsistent variability mechanisms
	19	$NInc_{cfg}$	Number of configuration inconsistencies
7	Reuse efficiency		
	20	RR	Reuse ratio
	21	NOD	Number of defaults
	22	$\nabla_{LOC,s}$	Spatial code churn among variant siblings
		K_{var}	Compression distance of variant siblings

Table 16: Metrics suite for sustainable product line infrastructure code evolution

The code size can be measured at different levels of scale, for example in lines of code (LOC) or in number of modules (NOM). Some variability mechanisms cause additional code or modules to be added, so that the numbers become larger than necessary, causing unnecessary construction complexity. As introduced in Sec.3.4, code churn [Hall+00] measures the amount of change in evolving source code. If the code churn over time ($\nabla_{LOC,t}$) for the entire product line infrastructure code exceeds its necessary value, then the code has become overly complex.

Variability management requires variation points, and ideally, there should be only one variation point per variability instance because in this case, the variant element is most consolidated. However, if the number of variation points (NVP) exceeds its necessary optimum, the module is crosscut more than necessary, another unnecessary complexity.

In Section 2.2 I have shown that reused code elements can be organized in a hierarchy based on their reuse relationships. The morphology of a hierarchy of modules can then be characterized by the depth of the reuse hierarchy (DRH) and by its width (WRH). If a reuse hierarchy is flatter or wider than necessary, less reuse opportunities have been taken than necessary, causing unnecessary complexities.

The shape of product line infrastructure code can also be measured by counting the number of conditional statements which are used in the product line infrastructure code to control all variants. For example, product-specific code may be selected by `if` statements in Conditional Execution, or by `#if` statements in Conditional Compilation. These conditions are anchored in the common code, so that their count characterizes the coupling of common and variant elements. A conventional metric which measures the shape of code depending on the number of conditions is Cyclomatic Complexity *v(G)* [McCabe76]. In a weighted sum of code size, cyclomatic complexity and other metrics, the maintainability index *MI* has been used to measure evolution effort in single system code [Coleman++94]. For product line infrastructure code measurement I propose to extend cyclomatic complexity into a two-dimensional metric $\vec{v}(G)$, for two reasons. The first reason is that the conventional metric only measures in terms of closed conditions, so that a module containing a single `if` or `case` statement has a cyclomatic complexity of two (the number of binary branches plus one). If the code is refactored by replacing the closed condition by an open one, which means applying the classical refactoring Replace Conditional with Polymorphism [Fowler99], the conventional cyclomatic complexity decreases [Tegarden++92], although the conditional situation has not changed. In order to express that conditional complexities still exist, I propose another type of metric for open conditions, $v(G)_{open}$, as a compensator. In other words, $v(G)_{open}$ corresponds to conventional cyclomatic complexity $v(G)_{closed}$, with closed conditions refactored to open ones, so that their sum will be invariant under that refactoring. The second reason for having a two-dimensional cyclomatic complexity is that the conventional metric usually covers conditions with runtime binding only. Similar as in the case above, refactoring product line infrastructure code that contains Conditional Execution into equivalent code with Conditional Compilation again leads to a decrease in conventional cyclomatic complexity, although the overall conditional situation is invariant. A cyclomatic complexity for construction time conditions $v(G)_{ct}$ is proposed as a dual of the conventional metric $v(G)_{rt}$

for runtime conditions. Because openness and binding time are orthogonal concepts, this results in four cyclomatic complexity measures, as depicted in Figure 46: $v(G)_{rt,closed}$ is the conventional cyclomatic complexity, $v(G)_{rt,open}$ is its open dual, $v(G)_{ct,closed}$ measures closed construction time conditions, and $v(G)_{ct,open}$ their open duals.

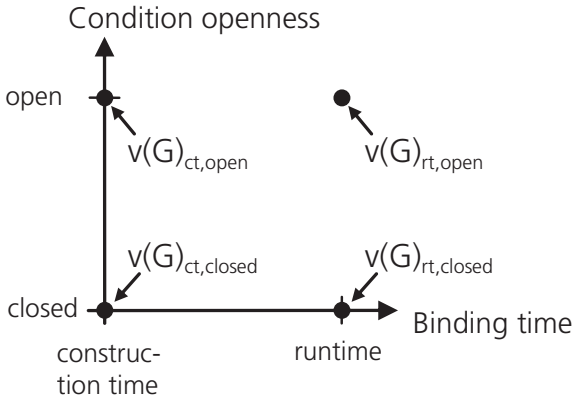


Figure 46: Two-dimensional cyclomatic complexity $\vec{v}(G)$

It has been suggested that the adaptability of a reusable module is inversely proportional to the amount of information needed to adapt it [Bassett97, p.128]. I propose a technology-independent metric to capture this, called LOC_{ad}, which measures the amount of code in all adaptees of a reusable module. If this number is larger than its required value, the common module is not adaptable enough, causing unnecessary adaptation complexities.

The visibility of variant elements also plays an important role in the evolution of reusable code ("Effective reuse highlights novelty – makes exceptions easy to see and control – while hiding what is routinely the same" [Bassett97, p.87]). This is why product line infrastructure code becomes less easy to evolve when variant elements are less visible than necessary, either externally (NV_e), that is, at module granularity, or internally within modules (NV_i). This is also true when a number of variant elements are ambiguous (NV_a), unnecessarily hard to distinguish from common elements.

Inconsistencies in variability management also cause unnecessary evolution difficulties. In this context, the number of inconsistent usages within a single mechanism $NInc_{VM\text{-}usage}$ is increased, for example, when Conditional Compilation uses both `#if` and `#ifdef` in the same product line infrastructure code. $NInc_{VM}$, the number of inconsistent variability mechanisms, is increased if the same type of variability problem is solved with different mechanisms, for example if two optional variabilities are realized with Conditional Execution and Conditional Compilation. There can also be inconsistencies in the way the code is configured, for example using a mixture of preprocessor flags and non-

volatile memory, as mentioned in [Krueger07], increasing $NInc_{cfg}$. The three aforementioned inconsistency measures may change over time, so that measuring their temporal delta ΔInc can hint at growing evolution problems.

A common metric for capturing reuse efficiency is reuse ratio [Poulin96, Bassett97], which is the size of reusable code in relation to total code size. When code size is measured in lines of code, the reuse ratio becomes $RR=1-LOC_{ad}/LOC$, with LOC and LOC_{ad} as described above. As shown in Sec.4.7, defaults also optimize reuse, so that the ratio between the number of defaults NOD and its optimal value indicates to what extent the family engineer has taken advantage of defaults in variability management.

Code churn has been used for measuring evolution across time [Hall+00, even in product line infrastructure code [Ajila+07]. However, to the best of my knowledge, it has not yet been explicitly used for measuring code distance across space at a fixed point in time. This measure, spatial code churn $\nabla_{LOC,s}$, can be applied to identify commonalities in sibling variant elements in a reuse hierarchy, for example in two realizations of different alternatives. If this measure becomes larger than required, for example when variants become too similar, the product line infrastructure code is too complex. Spatial code churn resembles edit distance (Levenshtein distance [Levenshtein66, Damerau64]). Edit distance has been applied to measure the similarity of character strings, for example in clone detection [Roy++09], error detection [Navarro01], or in data compression [Crochemore+96]. Edit distance measures the amount of difference between two sequences by considering the number of primitive evolution operations (addition, removal, change) that are required to transform one sequence to the other. As source code is usually represented in character string form, this metric can also be used to measure the (dis-)similarity of evolving product line realizations. In this case, edit distance denotes the number of additions, removals or changes of source code elements, for example lines of code, which corresponds to code churn. Across time, the metric characterizes how many code changes have been applied during each evolution step. Across space, edit distance (or spatial code churn) is the minimal number of code line additions, removals or changes to convert one realized product line member into another.

Another similarity metric is compression distance or Kolmogorov complexity *K* [Kolmogorov68, Cilibrasi+05] whose use in software development has been suggested as generative software complexity [Heering03]. In product line realization, the different variability mechanisms can be regarded as software generators. Compression distance is the length of the shortest of a set of generation possibilities to produce all product line members. It compares the compactness of the generators, the compression they achieve. As an alternative to spatial

code churn, compression distance K_{var} can be used to detect unnecessary similarities of variant elements, optimizing reuse. When two variant modules x and y are to be compared and C(x) and C(y) are the lengths of the compressed versions of x and y, and C(xy) is the length of the compressed version of the concatenated text of x and y, then K_{var} is their normalized compression distance [Cilibrasi+05]:

$$K_{\mathrm{var}}(x,y) = NCD(x,y) = \frac{C(x,y) - \min\{C(x),C(y)\}}{\max\{C(x),C(y)\}} \ .$$

Baselining

As shown in Figure 44, the family engineer sets up baseline product line infrastructure code in the Baselining phase. The need of this activity when measuring software evolution has already been motivated in [Hall+00]: to establish a fixed point against which all others can be compared. A fixed baseline reduces measurement effort, as shown in [Kelly06]. For a single system, this fixed point means a reference version of the code at some point in time, as the artifact at time $t=T_0$ from Figure 22 against which temporal stability has been measured. For variability complexity measurement, I suggest a novel baselining concept, in which not only one baseline is built up for product line infrastructure code, but two. The first is a temporal reference system as in the single system case. The second is a spatial reference, an "ideal" product line realization which exists at the same time as the existing code. Figure 47 illustrates the idea.

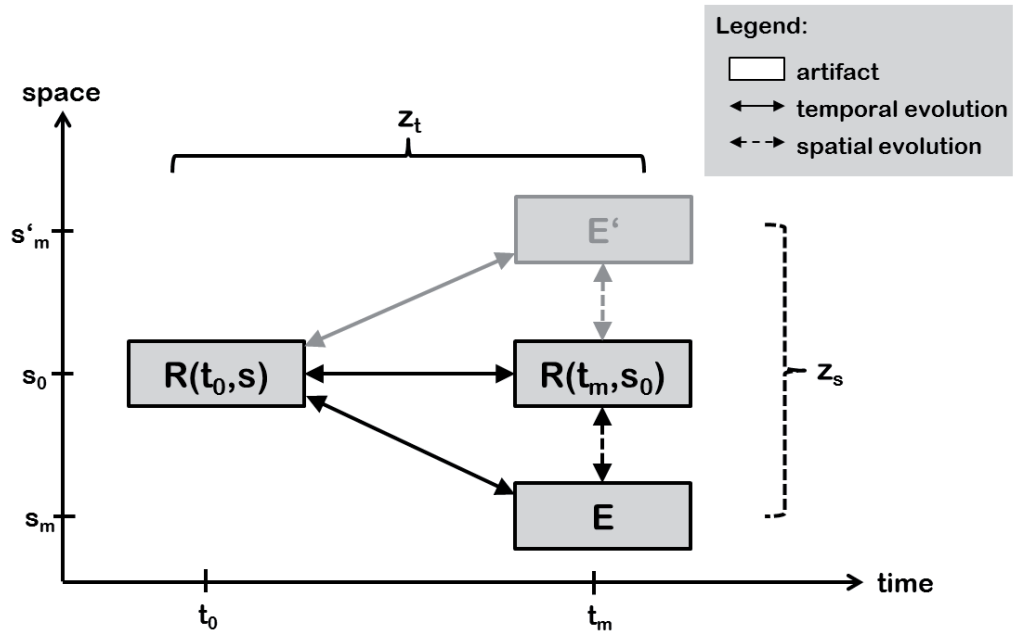


Figure 47: Two types of baselines for product line infrastructure code E: temporal ($R(t_0,s)$) and spatial ($R(t_m,s_0)$)

At some time t_m the existing product line infrastructure code E is to be measured for complexity. As one baseline, a reference system $R(t_0,s)$ is used, a code version from an earlier stage of the evolution at time $t_0<t_m$. The temporal evolution score z_t can be calculated for this dimension, as explained for the conventional standard score z in Section 3.4.

The second reference system is $R(t_m,s_0)$ which exists at the same time t_m as the existing system E, but has an "ideal" realization s_0 with regard to variability management. The engineer can then evaluate a realization alternative E' against the same two reference systems in order to decide if E or E' is less complex. The advantage of this approach over a single-reference approach, as shown in Figure 22, is that not all conceived variants E, E', E'', … at time t_m need to be compared against each other, but only against a single spatial reference. If the distance s_m-s_0 to the reference system is close, the product line infrastructure code has sufficient variability management quality. The distance corresponds to a spatial score z_s which can be used to evaluate different evolution alternatives. This approach will be demonstrated in the case study in Section 6.4.

After the variability complexity measurement phase has been finished, the next iteration of the product line realization process starts again in the selection phase, as shown in Figure 37.

6 Case Study

A case study has been conducted in order to compare the quality of sustainable product line infrastructure code evolution in various product line development contexts. The case study has monitored and evaluated the evolution of software product line generations of small and highly resource-constrained embedded systems. The product line realizations have been co-evolved by using each of the mechanisms presented in Chapter 4, and by applying the techniques suggested in Chapter 5. According to the classification in Section 5.1, different representative types of product line evolution scenarios have been realized as described in Section 5.2. The product line infrastructure code has been tested and its quality has been measured and compared using the approaches discussed in Section 5.3.

The results support the hypotheses that after a few initial iterations, Cloning leads to product line infrastructure code which takes more effort to evolve, with lower quality, than the other mechanisms whose complexity trend is more linear. Another observation is that there are groups of mechanisms that have some complexity characteristics in common, whereas others diverge. In these cases, the simplest variability mechanism is usually programming language-agnostic. A third result is that no variability mechanism is best in all contexts (no silver bullet [Brooks95]), so that applying a monoculture of variability mechanisms, as often seen in practice, leads to unnecessarily complex product line infrastructure code.

The following sections are organized as follows: Section 6.1 presents the background and objectives of the case study in more detail, and Section 6.2 introduces the technical context in which the case study was performed. In Section 6.3, the setup of the case study is shown, and Section 6.4 shows the results. Section 6.5 interprets the results per investigated hypothesis, and Section 6.6 discusses threats to validity.

6.1 Hypotheses

In order to evaluate the effects of variability mechanism characteristics on variability complexity in product line infrastructure code, and its resulting evolvability from a family engineering perspective, I have developed a hierarchy of hypotheses which matches the goal hierarchy from Figure 45, as presented in Figure 48.

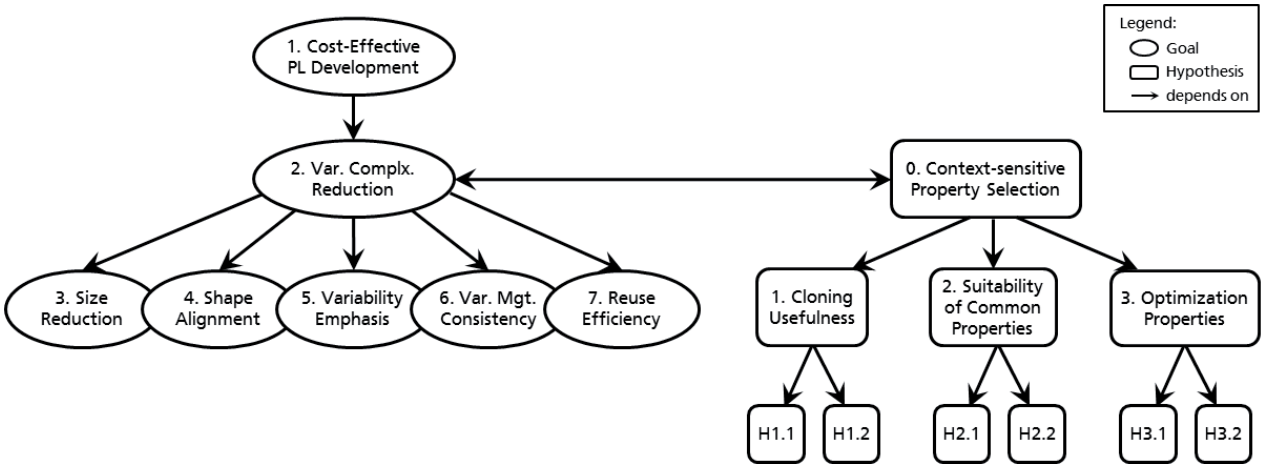


Figure 48: Investigated Goal and Hypothesis Hierarchies

The base hypothesis is that context-sensitive selection of variability mechanism properties leads to variability complexity reduction in the Variability Management base tier of reactive product line development (Fig.19). It is related to the main goal G2 within this tier, variability complexity reduction (see Sec.5.3). This base hypothesis is refined into three types of sub-hypotheses. The first investigates the usefulness of Cloning in family engineering. Sub-hypotheses 2 and 3 are concerned with properties of other types of mechanisms besides Cloning, and are formulated without particular reference to a single mechanism. The goal of sub-hypothesis 2 is to investigate the suitability of properties that exist in all variability mechanisms for reducing variability complexity. Sub-hypothesis 3 investigates the usefulness of optimization properties that are available in some, but not all mechanisms and which are also available for other types of artifacts than code. As for the goals (Sec.5.3), the base elements of the hypothesis hierarchy are subsumed by refined elements, and only these are investigated further. These six hypotheses H1.1 to H3.2 are listed in Table 17.

H	Description
1.1	In short term evolution, Cloning does not result in significantly higher variability complexity than most other mechanisms.
1.2	In long term evolution, Cloning results in variability complexity excess and increases evolution effort, compared to any other mechanism.
2.1	Late binding increases variability complexity significantly.
2.2	Programming language-dependence increases variability complexity significantly.
3.1	Defaults decrease variability complexity significantly.
3.2	Support for both closed and open variation decreases variability complexity significantly.

Table 17: Overview of investigated hypotheses

Hypothesis H1.1 investigates if short-term cloning, like conventional single systems development shown in the left bottom part of Fig.5b, does not lead to unacceptable complexity excess in product line

infrastructure code. In this case, it would be beneficial in the context of early evolution, as it can be performed rapidly, without risks (cf. Sec.3.3). This assumption is supported by various studies on Cloning in single systems (Sec.3.3). It also fits well to the product line realization process developed in this thesis (Sec.5.2), which suggests a Modification sub-process that contains the two successive activities Commonality Realization and Variability Realization (Fig.39). Cloning may be used in the early activity Commonality Realization, if it does not lead to complexity excess at this stage.

The next hypothesis, H1.2, expects that sticking with cloned code leads to complexity excess in the long term, and as a result makes the product line infrastructure code harder to evolve then, if this is required (cf. Sec.3.3). For this reason, the Variability Realization activity in the Modification sub-process (Sec.5.2) suggests some variability refactorings, in particular Consolidate Clones, to counteract the code smell of Duplicated Code.

Hypothesis H2.1 states that late binding, in particular runtime binding, is correlated with unnecessary increase of variability complexity. This assumption is based on the observation that the possibility for mass customization (Def.29), that is, composition (Def.9) and configuration (Def.28), is completely missing at runtime (Def.16), whereas only the possibility for configuration is missing at execution time (Def.15), and unlimited mass customization is supported at construction time (Def.31). This also motivates applying the variability refactorings Replace Runtime Binding with Execution Time Binding, and Replace Execution Time Binding with Construction Time Binding (Sec.5.2).

Hypothesis H2.2 investigates the correlation between a mechanism's programming language-dependence and variability complexity. The motivation for this assumption is that programming language-dependence forces variation points to be aligned with programming language constructs, such as function call boundaries, while programming language-independence offers the family engineer more degrees of freedom in setting variation points, with potentially less refactoring effort. Hypothesis H2.2 also refers to a special case contained in hypothesis H2.1, the relation between construction time and execution time binding.

As discussed in Def.55, defaults decrease the number of configuration options, and so hypothesis H3.1 examines if variability complexity is decreased in the presence of defaults. As shown in Chapter 4, some variability mechanisms support defaults, and if these showed lower variability complexity in the case study, this hypothesis would be supported. Defaults are also a target of the two variability refactorings Replace Variant Element with Default, and Replace Commonality with

Default (Sec.5.2), which counteract the minor code smell Lack of Defaults.

The final hypothesis H3.2 claims that variability complexity will be decreased if a single mechanism supports both closed and open variation. This assumption is justified in the typical situation when product line infrastructure code must realize both basic types of variabilities: optional and alternative variabilities. Although each of these can be expressed by the other [Synthesis93, Bayer++99, Pohl++05], the result is extra effort and complexity which may be avoided if both basic variability types are always supported.

6.2 Study Subject

As mentioned at the beginning of this chapter, the subject of the case study is code for small embedded systems. The systems are battery-powered wireless sensor nodes which are part of the Particle Computer[19] rapid prototyping platform for Ubiquitous and Pervasive Computing environments. The sensor nodes are able to communicate with internet gateways or with each other, forming an ad-hoc wireless sensor network (WSN). They can be equipped with various types of actuators and sensors. The sensors allow the node to register values of its physical environment such as the node's acceleration in two or three dimensions, temperature, light or noise. For reasons of energy efficiency and physical compactness, the sensors are equipped with extremely resource constrained hardware, such as an 8 bit microcontroller, 128kB of flash ROM and 4kB of RAM. Figure 49 depicts a wireless node and its sensor board.

Figure 49: Particle Computer wireless sensor node and sensor board

The sensor node software can be developed in the C programming language which is still among the most frequently used languages in practice for developing embedded systems code [Chen++05]. An open-

[19] particle.teco.edu (retirved August 2009)

source code library is available which offers basic functionalities, such as querying sensor values or transmitting data. The library has been written for the *SDCC*[20] compiler which supports a C dialect that closely matches ANSI C, with some C99 extensions. However, the library does not provide a consistent application programming interface. For example, different conventions exist for sensor initialization, refresh and value retrieval, depending if the sensor exists on the wireless main board or on an additional board, and the mechanism for obtaining timer functionality is completely different than the corresponding mechanism for obtaining sensor functionality. In order to avoid development complexities due to these inconsistencies, I developed a consistent hardware abstraction library as a façade [Gamma++95] of the original library. In other words, the hardware abstraction library changes the existing modules externally by composition (Def.9). For reasons of construction efficiency, both libraries are provided together as a binary module that the product line engineer can use (Def.6) for creating a sensor node application. The library provides simple and consistent interfaces (Listing 26) for building sensor node software that interacts with the physical environment (sensor, actuator and clock abstractions (Def.17)), and that is able to communicate with other wireless devices (transceiver abstraction). Figure 50 specifies these requirements in a problem frame diagram [Jackson01], as an instance of the Four Variable Model [Parnas+95].

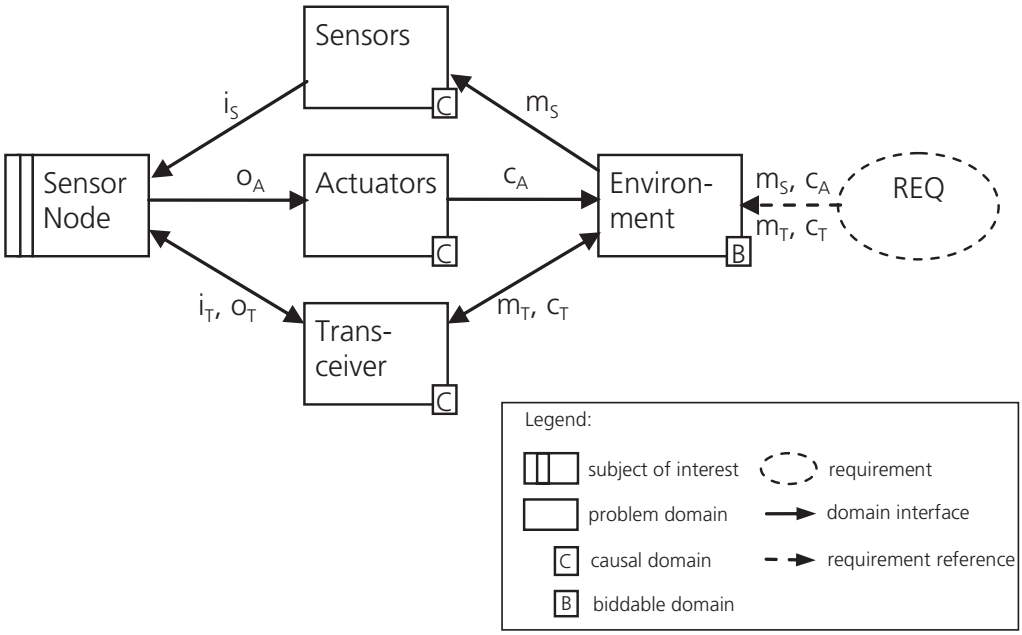


Figure 50: Sensor node problem frame

The sensor node is the system of interest. It receives input variables i_S from the sensors, which are related to the monitored variables m_S from the environment. The requirement REQ references the environment by referring to these monitored variables m_S. Likewise, the sensor node

[20] sdcc.sf.net (retrieved August 2009)

refers to the output phenomena o_A, which lead to phenomena c_A that control the environment. Similar sets of events exist as inputs i_T and outputs o_T of the sensor node to the transceiver, and as monitored and controlled variables m_T and c_T of the environment.

Not all sensor node applications are concerned with all development problems mentioned above. The decision model [Muthig02] in Table 18 illustrates that within a sensor node product line, the actuators and the possibility for wireless reception are optional variabilities, while sensors and wireless transmission are commonalities.

Decision	Question	Resolution
Actuators	Does the wireless sensor node use actuators?	no => remove o_A and c_A from Figure 50
Wireless reception	Does the wireless sensor node receive data wirelessly?	no => remove i_T and m_T from Figure 50

Table 18: Decision model for the sensor node product line specified in Figure 50

The set of sensor node applications which must be realized also have in common that they execute particular tasks at certain fixed time intervals. For example, the sensors must query the monitored variables [Parnas+95] of their physical environment at certain sampling rates, or the transceiver must send out the collected information periodically. The cooperative scheduler is an idiom [Coplien91] in the development of time-triggered embedded systems [Pont01, p.246] which solves the problem of scheduling periodic tasks in a simple, reliable and safe way. In order to make the sensor node applications particularly easy to develop, I provide a simple variant of a cooperative scheduler in the hardware abstraction layer, so that the product line engineer can focus on product line-related tasks rather than being concerned with functionality issues. A similar solution has also been described in the context of simplifying the software of an embedded system for controlling an autonomous helicopter [Wirth01, p.490], where the task was to periodically query sensors and compute aggregate values. The solution is to use a single interrupt service routine for periodically setting a Boolean variable, for example once every second. Sensor node applications use the interrupt service routine, and within their endless main loop, they constantly query if the variable has been set. In this case, they reset the variable and perform the periodic task.

6.3 Study Procedure

The case study simulates the evolution of a product line in six steps, as shown in Table 19. Initially, the source code of three different time-triggered sensor node applications is given, and these three systems shall be further evolved as a product line. This is a typical scenario seen in practice. The case study starts with three systems because in reported

experience, the pay-back point where it becomes cheaper to develop systems based on a product line infrastructure than without is typically three systems [Linden++07, Muthig02, Weiss+99].

The first application realizes a tilt detector whose functionality is to periodically query its orientation sensor values, to increment a counter if it measured a certain pattern of change (a tilt of the device), and to transmit the counter value in larger time intervals. By cloning and modifying the code of this system, a second product has been developed, a drop detector whose functionality is also to periodically query orientation sensors, but after detecting another pattern of change (a drop of the device), to transmit a warning message without delay. The third product, also developed from the first one by cloning, realizes a noise detector that periodically monitors the noise level of the environment, stores if a certain noise threshold has been exceeded, and transmits this information in larger intervals.

Step	Description
0	Similar alternative products: tilt detector, drop detector, noise detector
1	Addition of new product: movement detector
2	Adoption of optional time transmission feature for all products
3	Addition of new product: raw detector
4	Addition of new optional voltage detection feature for all products
5	Removal of delayed transmission feature from most products
6	Addition of new optional clock adjustment feature for all products

Table 19: Steps in the evolution of a sensor node product line

The three product realizations shall be evolved according to different typical evolution scenarios which in the majority of cases are instances of the product line evolution scenarios presented in Section 5.1. The reason for additionally having the 5th evolution scenario is to compare evolution effort (goal Q3 in Tab.10) in case the product line becomes simpler (see also the effort reduction sub-section in Sec.6.4). These evolutions are depicted in the feature diagram in Figure 51.

In the first evolution step, the existing set of similar products with three alternative features is to be extended by a fourth product. This product realizes a movement detector which periodically monitors its movement sensor, storing the time when a movement happened, and periodically transmits the collected information. The type of evolution is Alternative Feature Addition (Figure 34f).

In the second evolution step, the new time transmission capability introduced by the movement detector shall become an optional feature of all four products. This is an instance of the scenario Optional Variation Point Creation (Figure 34b), in which an existing functionality becomes a common product line feature. This results in a set of eight products.

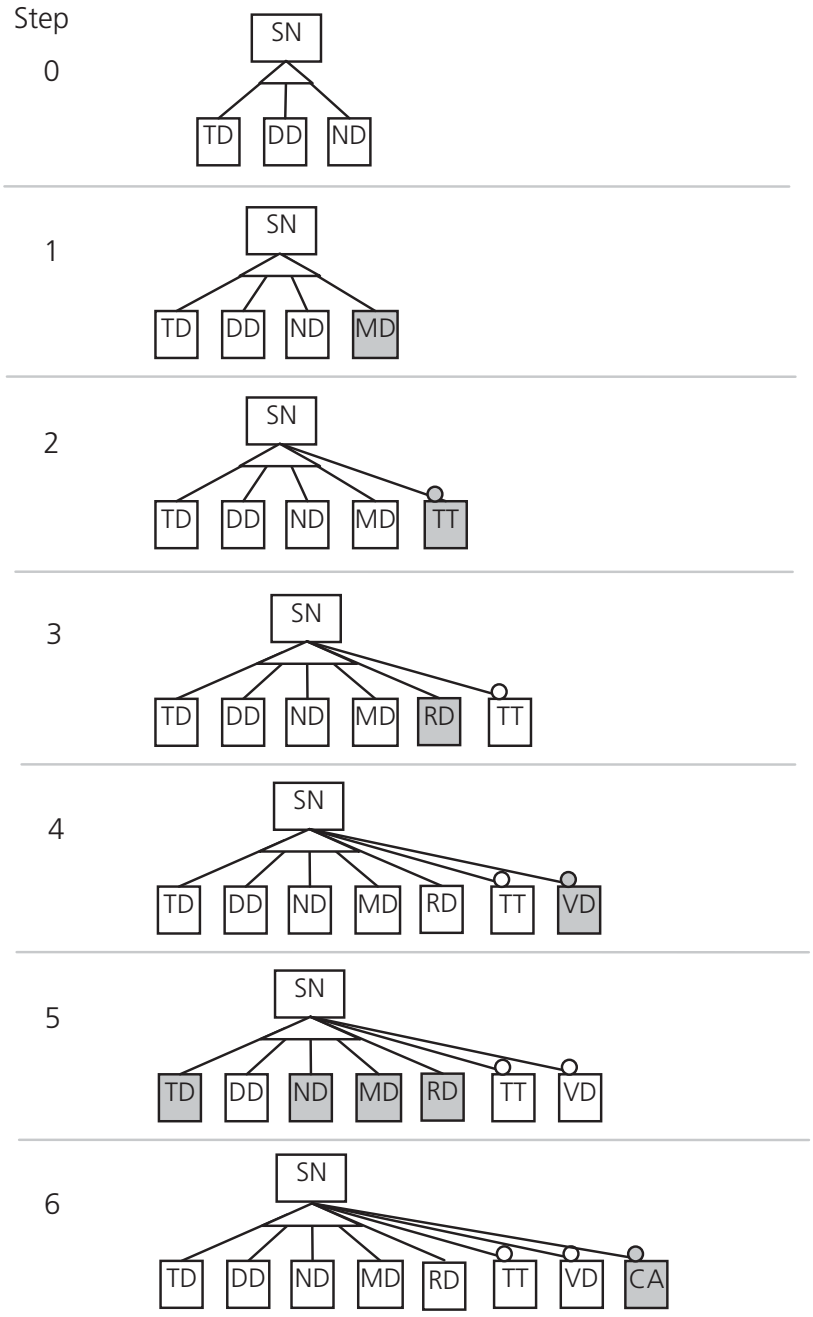


Figure 51: Feature diagram snapshots of the evolving sensor node product line (cf. Table 19)

The third evolution step requires another new product to be added, which means another alternative feature addition. The product functionality is to query all available raw sensor values and to transmit them. The cardinality of the set of products is ten.

In the fourth evolution step, all products must optionally monitor their battery voltage, transmitting it in large time intervals. The set of realized products to be managed and evolved together reaches twenty members. The type of evolution scenario is Optional Feature Creation (Figure 34a), where a new optional feature is made available to all product line members.

The fifth evolution step simplifies some products of the product line by removing a nearly common functionality. All products except for the drop detector perform their sensor sampling and their transmission at different intervals. This shall be removed now, so that all members of the product line obtain the same transmission behavior. There are still twenty products.

In the sixth and final evolution step, another new optional feature is to be added to all products. Its functionality is to synchronize the clocks of all sensor nodes. When a sensor node is put in operation, its clock value is zero. Periodically, the gateway software (or other sensors) may transmit the current time value if they are aware of the time. When other sensors are able to adjust their clocks accordingly, there will soon be a uniform notion of time within the entire wireless sensor network. This optional feature creation scenario increases the number of products to fourty.

As a result of realizing these scenarios, seven different versions of code are gained which capture a trace of the product line infrastructure code evolution, as shown in Figure 52 (compare Fig.23).

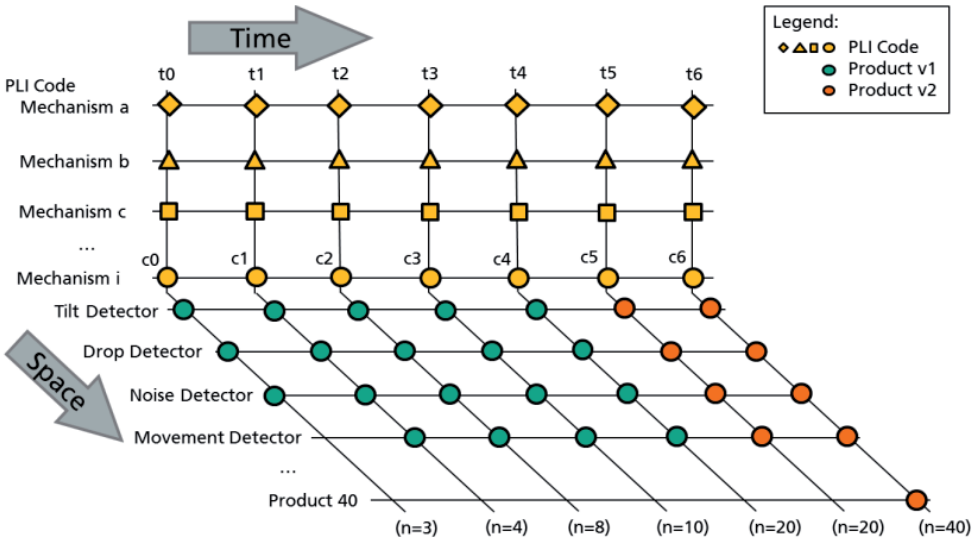


Figure 52: Evolution trace for product line infrastructure code

As described in Section 5.3 (Figure 47), measurements for the different code versions are compared with temporal baseline code at $t=t_0$ in order to evaluate the change in complexity over time. In order to compare the quality of variability mechanisms, each of the seven product line infrastructure code generations listed in Figure 52 has been realized nine times, which results in the sixty-three product line realizations depicted in Figure 53. Excerpts of these realizations are listed in Appendix C.

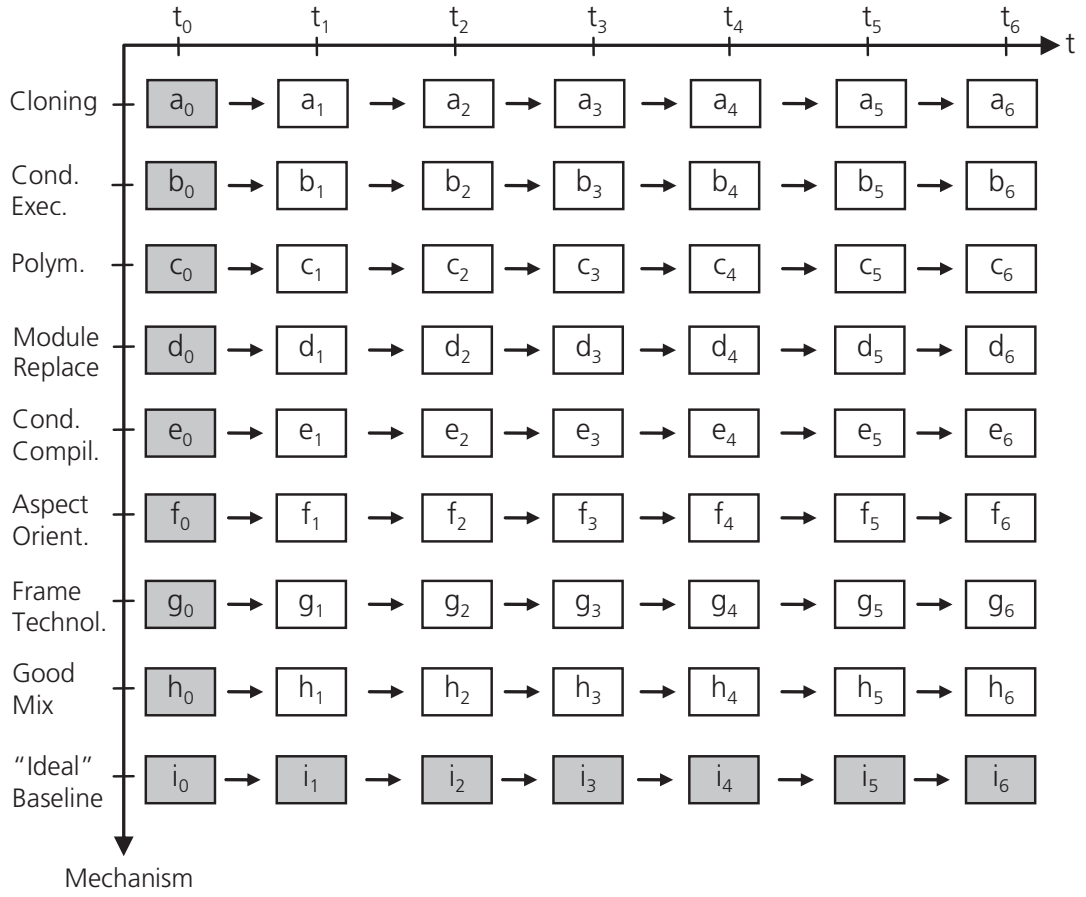


Figure 53: Evolution trace for product line infrastructure code, with baselines (gray)

Sequences "a" to "g" have each been realized using a monoculture of the seven variability mechanisms discussed in Chapter 4: Cloning, Conditional Execution, Polymorphism, Module Replacement, Conditional Compilation, Aspect-Orientation and Frame Technology. Sequence "i" represents the "ideal" realization at each point in time which best balances the tactics for effective family realization (Tab.2) and which serves as the spatial reference ($R(t_m,s_0)$ in Figure 47). It contains the same C code as in the other realizations, enriched with variability management pseudocode. The pseudocode expresses which activities a human software engineer or an automated construction interpreter must at least perform in order to create all required product instances from the C code elements, e.g. by changing the text at certain lines. It separates what must be performed for "good enough" variability management in the respective development situation from how to achieve these tasks. For example, a tactic is to realize a smaller optional element, or an alternative or coexisting variability with only two resolution possibilities, next to the common code. It makes less sense to extract the variant elements into separate modules because there is no reason for such a strong separation between the common and variant elements, especially if both are likely to evolve together. Conversely, an alternative variability with at least three choices shall usually be realized in one or more variant modules separate from the common one, for at least two reasons. The

first reason is that the more numerous variant elements belonging together tend to require more visibility than just one or two elements. The second reason is that a variation point for multiple alternatives tends to be stronger, more likely to be needed again in future variability management scenarios, than one that exists just due to an optional variability. The remaining sequence "h" uses a mix of variability mechanisms, staying as close as possible to the baseline.

In order to provide fair comparisons, the C code for each executable product is kept as consistent in temporal and spatial evolution as the mechanisms allow (comp. the different realizations in Appendix C). This means that all realizations start with the same precondition. In particular, latent variation points are not provided to give all mechanisms the same chance in unpredicted evolution. This means, for example, that global variables are deliberately used throughout the code, rather than making them static or extracting them into functions that emerge due to some variability mechanisms. Functions are not extracted and modules are not split, unless the variability mechanism demands it. Wherever possible, variability management is kept consistent and comparable. Possible optimizations in variability management are deliberately avoided if they are unrelated to the primary mechanism. For example, variable code could have been extracted into a separate module in the Conditional Execution and Conditional Compilation, but this extra step has intentionally been omitted because it would introduce Module Replacement as a secondary mechanism. Variation points are used sparingly. For example, extra variation points for initializing or updating individual sensors have not been realized because these tasks are not central to successful product configuration – all products can tolerate the small additional overhead caused by keeping these features common. Each variability mechanism is also realized as consistently as possible, for example by configuring the runtime mechanisms Conditional Execution and Polymorphism in the same way as Conditional Compilation, or by organizing the common and variant elements similarly for the pairs Conditional Execution / Conditional Compilation, Polymorphism / Module Replacement and Aspect-Orientation / Frame Technology.

Seven out of the nine sequences (49 product lines with 735 products) result in machine code which can be executed on the hardware of the wireless sensor node. For that task, Makefiles are provided which can either configure and compile a single product, or which configure and compile all products of a certain product line in succession. Some of the software products have been used in Ambient Intelligence prototype systems [Patzke++08], for example the tilt detector was embedded in a cup, and the drop detector was part of a stick. The other two sequences for Aspect-Orientation and the baseline have been realized in pseudocode. The baseline has been realized in pseudocode because it is used to represent how the family engineer intends to realize the current evolution step, independent of any concrete realization mechanism. The

Aspect-oriented sequence "f" has been realized in pseudocode and does not result in executable machine code because the required *SDCC* compiler, with its necessary hardware-specific C extensions for interrupt service routines and embedded assembler code, is not supported by any of the few available C aspect weavers. Nonetheless, the provided aspect pseudocode can at least be processed by the *ACC* weaver, although the resulting intermediate code cannot be further compiled with *SDCC*.

Frame Technology is used in the sequences "g" and partially in "h", supported by the frame processor *FP* [Patzke+03] which I have been developing since 2002. The tool deliberately offers only those frame technology-specific variability management capabilities which are missing in other variability mechanisms: open construction time variation. In contrast to other frame processors, closed variation has deliberately been omitted because it is already offered by Conditional Compilation.

6.4 Results

Depending on the applied mechanism, the following issues have been observed in the respective realization sequences: When Cloning is applied and an optional feature must be introduced, the existing number of modules is doubled. The names of the new modules have consistently been extended by the new feature name. In Conditional Execution and Polymorphism, global runtime variables of integer type have been used for configuration (Listing 18, l.10-13; Listing 20, main.c, l.11-14). They are initialized before execution and remain unchanged thereafter which wastes memory resources. Due to the programming language-dependency of Conditional Execution, the alternative feature for time transmission has been duplicated for each alternative behavior (step b2 and b3). It has not been extracted into a separate function because of consistency with the other mechanisms (Listing 18, l.49-52, l.68-71, l.91-94, l.107-110, l.126-129). Initialization effort is particularly high in Polymorphism because all function pointers have to be set individually, according to the values of the configuration variables (Listing 20, main.c, l.23-55). A function pointer table could have simplified the configuration, at the expense of consistency with other mechanisms. Another characteristic of both conventional open mechanisms Polymorphism and Module Replacement is that their common module main.c (Listing 20 and 21) contains a particularly small amount of common code, and they both depend on forward function declarations, which requires the presence of header files in consistent realizations (Listing 20, main.c, l.2-5; Listing 21, main.c, l.2-4). Moreover, Module Replacement requires a separate null module for each optional variant (for reasons of consistency, the three resulting null modules no_time_transmission, no_voltage_check and no_clock_sync (Listing 21) have not been extracted into a single module). In both the programming

language agnostic mechanisms of Conditional Compilation and Frame Technology, the optimization possibilities for managing incomplete C code elements have deliberately not been used initially, so that all alternative detector elements end with two redundant lines. The elements have only been extracted after this became necessary because of the new time transmission feature in step 2. During the later evolution steps it became increasingly necessary to manage small or nesting code elements in order to account for small differences among variant elements (Listing 19, l.92-100, l.102-104; Listing 23, drop_detector, l.14-16).

Defaults have not been used in Conditional Compilation because they are not a primary element of this mechanism. This is not the case for Frame Technology, and for reasons of consistency, Defaults have also been used in Aspect-Oriented code. In any case, Aspect-Orientation requires an extraction of a function (main_loop), in order to override its contents (Listing 22, main.c, l.11-28). The extraction is also necessary in Module Replacement which results in a simpler and more direct realization (Listing 21, main.c, l.23). In step f2, the optional time transmission feature has been realized in a time_transmission aspect which composes (Def.9) the `send()` function of the hardware abstraction library. This is only possible because that function is only used once for each product. However, with the advent of the new raw detection feature in step f3, the send function is called twice in a product, which invalidates the aspect's assumption and requires the new wrapper function `send2()` to be introduced – an inelegant, but necessary step caused by the limits of Aspect-Orientation (Listing 22, main.c, l.30-32). At the same time, the new raw detector must always set the event_happend variable, so that it can use the time_transmission aspect as-is (Listing 22, raw_detector.acc, l.10, l.12). Again, as in Module Replacement (step d3), it becomes necessary to abuse C code statements for variability management purposes because of the limited configuration (Def.28) possibilities of the applied variability mechanism: The raw detector requires a restricted use of the existing time transmission feature, but the available mechanism is not able to configure this internally, but must use an external ad-hoc solution.

In contrast to Aspect-Orientation, the default code for tilt detection can remain inline in Frame Technology because there is no necessity to extract it (Listing 23, main, l.29-45). In step g2, the new time transmission feature necessitates the Default to be changed: it is split into two elements instead of shrinking it to only one smaller element (VP more_loop) because this way the existing alternative detectors can still only refer to the first variation point, whereas the new time transmission feature is only associated with the new second variation point (more_loop2). The frame technology sequence uses the parameterized adaptation feature of *FP* to avoid using wrapper frames [Bassett97,

pp.178f.] for combining optional features, which reduces the number of needed modules (Listing 23, drop/noise/movement/raw_detector, voltage_check, clock_sync, l.1). A combination of only two mechanisms is necessary to realize sequence "h" which closely resembles the ideal sequence "i". The mechanisms are Conditional Compilation and Frame Technology, as provided by FP. Optional features are realized with Conditional Compilation (Listing 24, main, l.47-55, l.59-66, l.67-75), while Frame Technology is used for realizing alternative features. In contrast to the monoculture of Conditional Compilation in sequence "e", the conditional elements in sequence "h" do not require nested variation.

As mentioned in Sec.5.3, the super-goals G1 and G2 have not been refined further. For most of the other identified goal categories of the product line quality model, the corresponding metrics from Table 16 have been captured for all 63 product line realizations, either manually or semi-automatically, for example using the scripts listed in Appendix B.2. As explained above, the code has been developed as consistent as possible, both in terms of its executable and variability management properties. For that reason, variability management inconsistencies have deliberately been avoided, so that their metrics (questions 17 to 19 in Table 16) can be assumed to be low in all realizations and are thus ignored. Another metric that has deliberately been kept at comparable low values for all sequences is the number of variation points which is also ignored in the following discussion. The following sub-sections list aggregated results found for the goals G3, G4, G5, and G7. Detailed results for each metric are shown in Appendix D.

Size Reduction

The first metric captured for the size reduction goal (G3) are the lines of product line infrastructure code. Because a consistent coding style has been used throughout, the code does not contain differences in indentation, commenting or spacing, so that its comparable size could simply be obtained using the Unix line count command *wc –l*. Table 20 shows the results. The gray title row and column in this and the following tables indicate that the table contains values that have been measured in the code, while tables which have white title rows and columns contain aggregated values.

In these and the following measurements, the values in all successive evolution steps always increase monotonically, except for step 5. The reason for the overall trend is that the product line infrastructure code becomes more sophisticated, with increasing variability management complexity. In step 5, however, a product line-wide simplification happens. Some quasi-common functionality is removed (transmission at a lower frequency than sampling), without changing the distribution of common and variant elements, as depicted in Figure 51. For that reason,

this step does not represent a typical product line evolution step, which is why it is not contained in the basic product line evolution scenarios in Section 5.1 (Figure 34). The step has been introduced in the case study to compare evolution effort, as will be described in the Sustainable Evolution sub-section below. In order to highlight only product line-specific complexity trends step 5 is omitted in the following graphs.

LOC	0	1	2	3	4	5	6
a (Cloning)	98	129	268	349	788	708	1606
b (Cond. Exec.)	76	95	116	143	154	137	149
c (Polymorph.)	110	139	181	218	252	232	268
d (Module Replac.)	99	125	152	185	211	191	219
e (Cond. Compil.)	72	91	89	112	122	109	120
f (Aspect-Orient.)	73	92	94	122	133	120	131
g (Frame Technol.)	76	95	93	118	128	117	128
h (Good Mix)	76	95	93	118	128	117	128
i (Ideal Impl.)	64	80	82	104	114	102	113

Table 20: Code size evolution in all realization sequences (cf. Fig.53)

As explained in Sections 3.4 (Fig.22) and 5.3 (Fig.47), the temporal code size delta expresses how much the product line infrastructure code has changed within a sequence, compared to the temporal baseline at $t=t_0$. This is shown in Table 21a and Figure 54 which in the upper part shows the overall trend, whereas in the lower part, the values for Cloning have been stripped, for reasons of legibility. After step 6, the values for sequence e to i remain close together, while Polymorphism and Module Replacement result in at least 100% higher values.

z_t	0	1	2	3	4	5	6
a	0	31	170	251	690	610	1508
b	0	19	40	67	78	61	73
c	0	29	71	108	142	122	158
d	0	26	53	86	112	92	120
e	0	19	17	40	50	37	48
f	0	19	21	49	60	47	58
g	0	19	17	42	52	41	52
h	0	19	17	42	52	41	52
i	0	16	18	40	50	38	49

a)

z_s	0	1	2	3	4	5	6
a	34	49	186	245	674	606	1493
b	12	15	34	39	40	35	36
c	46	59	99	114	138	130	155
d	35	45	70	81	97	89	106
e	8	11	7	8	8	7	7
f	9	12	12	18	19	18	18
g	12	15	11	14	14	15	15
h	12	15	11	14	14	15	15
i	0	0	0	0	0	0	0

b)

Table 21: Code size deltas: a) in time, b) in space

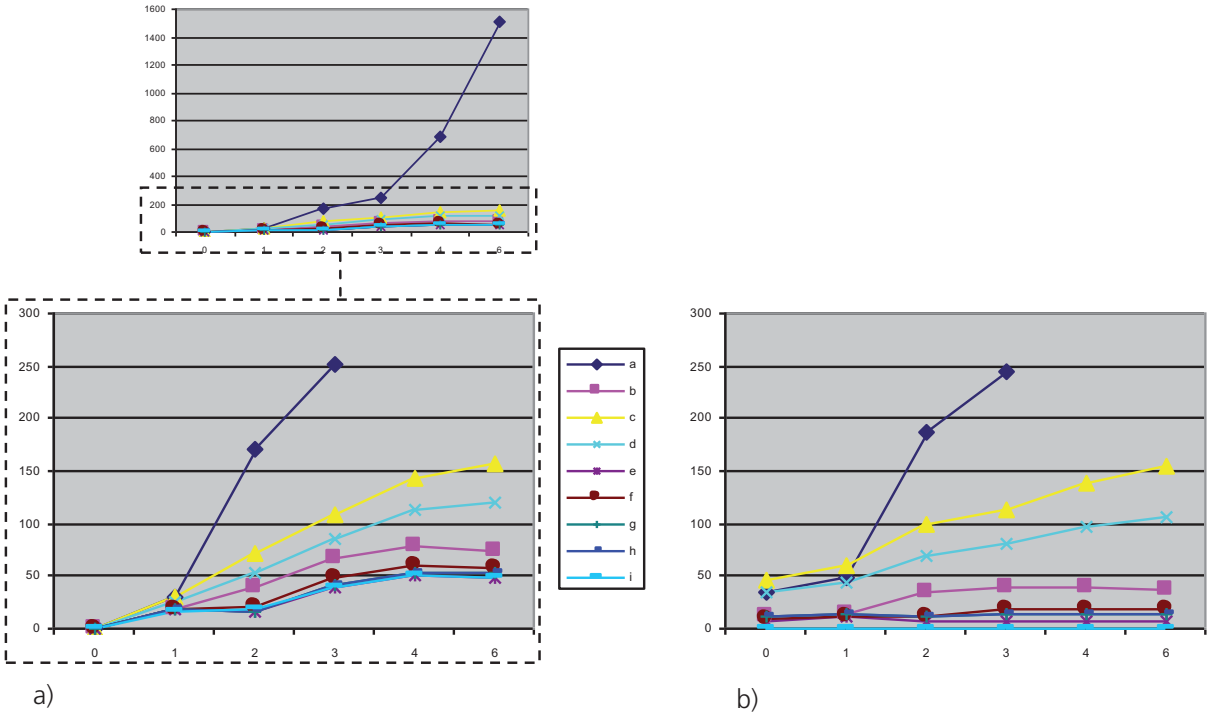


Figure 54: Trends for code size deltas a) in time, b) in space

The values for spatial code size delta are comparable, as shown in Table 21b and Figure 54. They express the difference in code size to the ideal realization $s=s_0$. The overall trend for both kinds of metrics is roughly identical: At most until step 3, the metrics for all sequences have a similar order of magnitude because their variability management complexity is still low. At least after step 4, when two optional variabilities are introduced, the values for Cloning become considerably higher than those of all other mechanisms, with an exponential trend. Compared to Cloning, the metrics for the other mechanisms remain closer together, with a more linear trend.

Similar results have been obtained for temporal code churn (Table 22 and Figure 55), the amount of code needed to transfer the initial version to each successor. These measurements were performed semi-automatically using a custom script (Appendix B.2) which calculates the line-wise edit distance between two files or between all files in two directories. The slopes for Polymorphism and Module Replacement are higher though, which means that over time, these realizations deviate more from the baseline code. This happens because each new feature is realized in a new module, with corresponding overheads for function extraction and file inclusion.

$\nabla_{LOC,t}$	0	1	2	3	4	5	6
a	0	31	170	251	690	610	1508
b	0	19	40	67	78	69	81
c	0	29	79	115	149	147	183
d	0	26	53	86	112	112	140
e	0	18	24	47	57	53	64
f	0	19	21	49	60	63	74
g	0	19	25	52	62	60	71
h	0	19	25	50	60	55	66
i	0	18	20	42	52	49	60

Table 22: Code churn in time $\nabla_{LOC,t}$ for all sequences

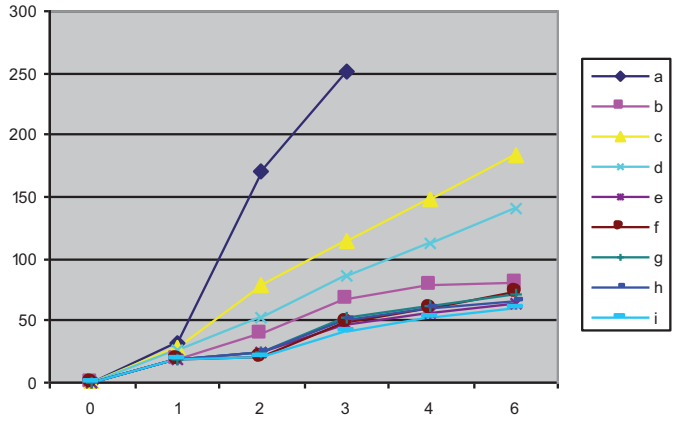


Figure 55: Trends for code churn in lines of code, compared to baselines at $t=t_0$

A similar trend exists in terms of the number of modules in each product line realization (Tab.23a) and their spatial code churn (Tab.23b and Fig.56). Although Cloning starts at an ideal number of modules, the deviation from the ideal realization after step 6 is more than 200% higher than for any other mechanism. Polymorphism and Module Replacement have comparatively high values because they unconditionally spawn new modules in each step. Conversely, Conditional Execution and Conditional Compilation always result in fewer modules than desirable, with a negative trend because they use a single module only.

NOM	0	1	2	3	4	5	6
a	3	4	8	10	20	20	40
b	1	1	1	1	1	1	1
c	5	6	8	9	11	11	13
d	5	6	9	10	13	13	16
e	1	1	1	1	1	1	1
f	3	4	5	6	7	7	8
g	3	4	5	6	7	7	8
h	3	4	4	5	5	5	5
i	3	4	4	5	5	5	5

a)

z_s	0	1	2	3	4	5	6
a	0	0	4	5	15	15	35
b	-2	-3	-3	-4	-4	-4	-4
c	2	2	4	4	6	6	8
d	2	2	5	5	8	8	11
e	-2	-3	-3	-4	-4	-4	-4
f	0	0	1	1	2	2	3
g	0	0	1	1	2	2	3
h	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0

b)

Table 23: a) Evolution in number of modules; b) comparison to spatial baseline

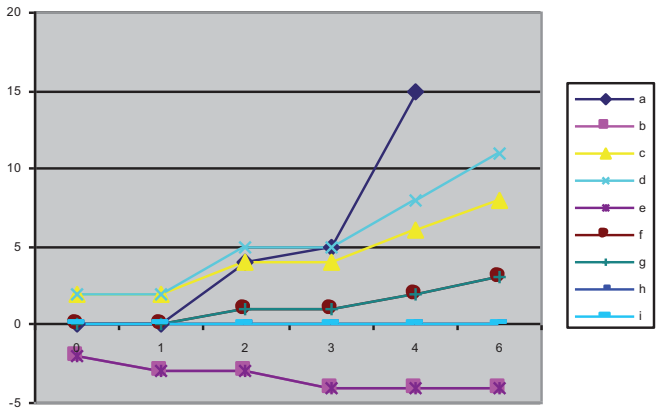


Figure 56: Trends for number of module delta, compared to ideal realization

Effort Reduction

As mentioned above, step 5 has been has been introduced to compare evolution effort in the different sequences when some variation exists in the product line. After step 4, five different alternative behaviors exist in the product line infrastructure code for capturing and transmitting monitored variables. Most behaviors (except for drop detection) transmit the information at a lower frequency than sampling it. This requirement becomes obsolete in step 5, which means that the respective code elements must be removed from the code. In particular, the task is to manually eliminate all code elements concerned with the `tick` variable, to re-indent the code correspondingly and to rename all `tick2` variables to `tick`. This task has been performed on product line infrastructure code from step a4 (Cloning), d4 (Module Replacement), e4 (Conditional Compilation) and g4 (Frame Technology). Two subjects performed these tasks, measuring the time from starting each development task until successful compilation of all twenty products. Table 24 lists the results. Although the values differ among the subjects, due to their different product line realization experience, both subjects on average performed the tasks in about half the time when a proper variability mechanism existed, compared to a situation in which all previous code had been cloned. As in the size measurements, this is another strong indicator that cloning leads to less evolvable code.

$time_{4->5}$/min	**subject 1**	**t/t_a**	**subject 2**	**t/t_a**
a (Cloning)	6:35		21:46	
d (Module Repl.)	3:05	47%	4:50	22%
e (Cond. Comp.)	5:10	78%	14:20	66%
g (Frame Techn.)	3:10	48%	10:26	48%

Table 24: Effort for realizing scenario 5, compared to Cloning

Shape Alignment

Tables 25 and 26 show the values for the depth and width of the reuse hierarchies in the different realizations (goal G4). Whereas the ideal cases only require a depth of two in all scenarios (as more or less reuse levels are not required in this situation, see Fig.11), they constantly remain at their minimal value 1 for Cloning and the closed mechanisms. There is a slight increase for the open mechanisms, due to interdependencies of variant elements, and a stronger increase for Frame Technology because it performs all configuration activities by its frame hierarchy, whereas the other mechanisms additionally require the Makefile for that purpose.

DRH	0	1	2	3	4	5	6
a	1	1	1	1	1	1	1
b	1	1	1	1	1	1	1
c	2	2	3	3	3	3	3
d	2	2	3	3	3	3	3
e	1	1	1	1	1	1	1
f	2	2	3	3	3	3	3
g	2	2	3	3	4	4	5
h	2	2	2	2	2	2	2
i	2	2	2	2	2	2	2

Table 25: Evolution in depth of reuse hierarchy

WRH	0	1	2	3	4	5	6
a	3	4	8	10	20	20	40
b	1	1	1	1	1	1	1
c	3	4	5	6	7	7	8
d	3	4	6	7	9	9	11
e	1	1	1	1	1	1	1
f	2	3	4	5	6	6	7
g	2	3	3	4	4	4	4
h	2	3	3	4	4	4	4
i	2	3	3	4	4	4	4

a)

z_s	0	1	2	3	4	5	6
a	1	1	5	6	16	16	36
b	-1	-2	-2	-3	-3	-3	-3
c	1	1	2	2	3	3	4
d	1	1	3	3	5	5	7
e	-1	-2	-2	-3	-3	-3	-3
f	0	0	1	1	2	2	3
g	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0

b)

Table 26: a) Evolution in width of reuse hierarchy; b) comparison to spatial baseline

As depicted in Figure 57, the trend for the delta in width of the reuse hierarchy is similar as that for the number of modules (Figure 56), except that in this case, frame technology matches the ideal shape, while Aspect-Orientation still diverges slightly.

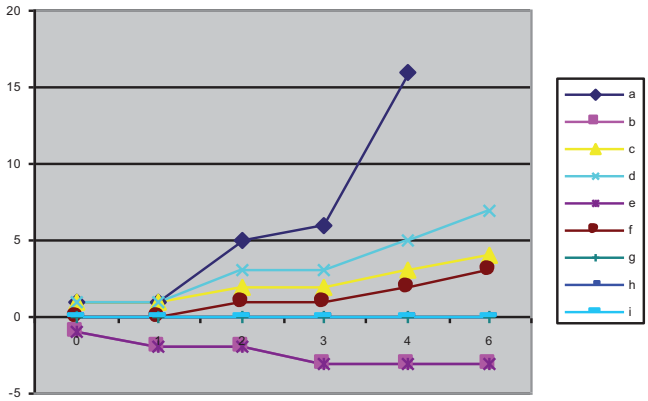


Figure 57: Trends for width of reuse hierarchy delta, compared to ideal realization

Tables 27 and 28 show the evolution of closed and open runtime and construction time cyclomatic complexity, as invented in Section 5.3.

$v(G)_{rt,closed}$	0	1	2	3	4	5	6
a	26	33	68	81	180	164	388
b	23	28	35	40	43	39	43
c	26	32	35	40	45	41	47
d	23	28	29	33	36	32	36
e	20	24	24	27	29	25	28
f	23	28	29	34	37	33	37
g	22	27	28	32	35	31	35
h	22	27	27	31	33	29	31
i	18	23	23	27	29	25	28

a)

$v(G)_{rt,open}$	0	1	2	3	4	5	6
a	3	4	8	10	20	20	40
b	1	1	1	1	1	1	1
c	7	9	14	17	20	20	23
d	4	5	7	8	10	10	12
e	1	1	1	1	1	1	1
f	3	4	5	6	7	7	8
g	3	4	5	6	7	7	8
h	3	4	4	5	5	5	5
i	3	4	4	5	5	5	5

b)

Table 27: Evolution of a) closed, and b) open runtime cyclomatic complexity

$v(G)_{ct,closed}$	0	1	2	3	4	5	6
a	3	4	8	10	20	20	40
b	1	1	1	1	1	1	1
c	5	6	8	9	11	11	13
d	5	6	9	10	13	13	16
e	4	5	10	13	14	10	11
f	3	4	5	6	7	7	8
g	3	4	5	6	7	7	8
h	3	4	5	6	7	7	8
i	3	4	5	6	7	7	8

a)

$v(G)_{ct,open}$	0	1	2	3	4	5	6
a	3	4	8	10	20	20	40
b	1	1	1	1	1	1	1
c	5	6	8	9	11	11	13
d	8	10	21	25	30	30	35
e	1	1	1	1	1	1	1
f	5	7	12	16	18	18	20
g	5	7	9	13	15	15	17
h	5	7	7	11	11	11	11
i	5	7	7	11	11	11	11

b)

Table 28: Evolution of a) closed, and b) open construction time cyclomatic complexity

For conventional closed runtime cyclomatic complexity, Cloning results in at least 800% higher values after step 6 than any other mechanism, while the values for these remain close together. The open runtime cyclomatic complexity corresponds to the number of modules for Cloning and the closed mechanisms because these do not have open variation. The constant value 1 for the latter denotes that there is not

enough open variation. Closed construction time cyclomatic complexity is the number of modules in the case of all monocultures except for Conditional Compilation because the metric counts the number of construction time conditions. The values for Cloning, Polymorphism, Module Replacement and Conditional Compilation end up too high, while Conditional Execution results in the undesirable minimal value. For open construction time cyclomatic complexity, the values for Cloning, the runtime mechanisms and Conditional Execution are the number of modules because either no construction time mechanisms exist or because variation is not open. Module Replacement has a particularly high open construction time complexity, exceeding the value for Cloning until step 6. Both Aspect-Orientation and the employed Frame Technology dialect have values above the ideal case because they unconditionally employ open variation.

sum_{rt}	0	1	2	3	4	5	6
a	29	37	76	91	200	184	428
b	24	29	36	41	44	40	44
c	33	41	49	57	65	61	70
d	27	33	36	41	46	42	48
e	21	25	25	28	30	26	29
f	26	32	34	40	44	40	45
g	25	31	33	38	42	38	43
h	25	31	31	36	38	34	36
i	21	27	27	32	34	30	33

a)

sum_{ct}	0	1	2	3	4	5	6
a	6	8	16	20	40	40	80
b	2	2	2	2	2	2	2
c	10	12	16	18	22	22	26
d	13	16	30	35	43	43	51
e	5	6	11	14	15	11	12
f	8	11	17	22	25	25	28
g	8	11	14	19	22	22	25
h	8	11	12	17	18	18	19
i	8	11	12	17	18	18	19

b)

Table 29: Evolution in a) runtime, and b) construction time cyclomatic complexity

sum_{closed}	0	1	2	3	4	5	6
a	29	37	76	91	200	184	428
b	24	29	36	41	44	40	44
c	31	38	43	49	56	52	60
d	28	34	38	43	49	45	52
e	24	29	34	40	43	35	39
f	26	32	34	40	44	40	45
g	25	31	33	38	42	38	43
h	25	31	32	37	40	36	39
i	21	27	28	33	36	32	36

a)

sum_{open}	0	1	2	3	4	5	6
a	6	8	16	20	40	40	80
b	2	2	2	2	2	2	2
c	12	15	22	26	31	31	36
d	12	15	28	33	40	40	47
e	2	2	2	2	2	2	2
f	8	11	17	22	25	25	28
g	8	11	14	19	22	22	25
h	8	11	11	16	16	16	16
i	8	11	11	16	16	16	16

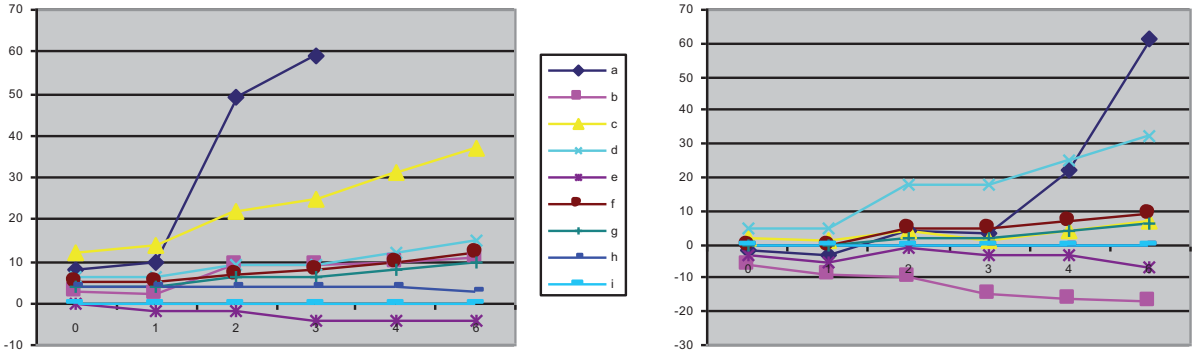
b)

Table 30: Evolution in a) closed, and b) open cyclomatic complexity

Table 29 lists the sums of the runtime and construction time complexities. Table 30 shows the sums of closed vs. open-variant complexities.

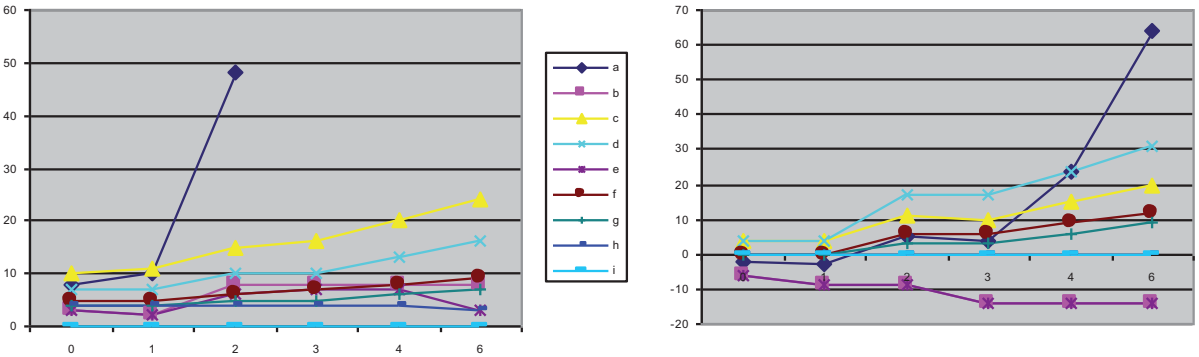
Figures 58 and 59 depict the corresponding deltas, in relation to an ideal realization. Cloning and Polymorphism lead to an excess in runtime complexity (Figure 58a), whereas the other mechanisms remain in a close range. A similar trend exists for closed-variant complexity (Figure 59a).

Cloning and Module Replacement lead to an excess in construction time complexity (Figure 58b), Conditional Execution has a clear lack thereof. Monocultures of mechanisms never match an ideal closed-variant complexity (Figure 59b). Cloning and all four closed mechanisms lead to an excess, the two open mechanisms lead to a shortage in closed-variant complexity.



a) b)

Figure 58: Trends for a) runtime, and b) construction time complexity delta, compared to ideal code



a) b)

Figure 59: Trends for a) closed, and b) open complexity delta, compared to ideal code

Whereas the open-variant metrics (Tab.29b and Tab.30b) express how many modules have been adapting the open variation points, the lines of adaptee code metric expresses the same phenomenon at a finer level of granularity, by counting the lines of code within the adapting modules. Table 31a lists lines of adaptee code in each product line realization. As mentioned in Section 5.3, the metric is inversely proportional to adaptability. For that reason, adaptability in relation to the ideal realization corresponds to the adapted lines of code for the ideal code divided by the adapted lines of the corresponding code (Table 31b). The corresponding graph in Figure 60 illustrates that all conventional mechanisms (a-e) are always less adaptable than the more advanced ones (f-i), with extremely low values for Cloning.

LOC_{ad}	0	1	2	3	4	5	6
a	98	129	268	349	788	708	1606
b	76	95	116	143	154	137	149
c	75	100	126	151	168	148	166
d	77	102	123	150	168	148	167
e	72	91	89	112	122	109	120
f	30	47	49	67	77	69	77
g	34	51	47	66	75	69	77
h	34	51	41	57	57	51	51
i	25	39	35	50	50	43	43

a)

ad	0	1	2	3	4	5	6
a	0,26	0,3	0,13	0,14	0,06	0,06	0,03
b	0,33	0,41	0,3	0,35	0,32	0,31	0,29
c	0,33	0,39	0,28	0,33	0,3	0,29	0,26
d	0,32	0,38	0,28	0,33	0,3	0,29	0,26
e	0,35	0,43	0,39	0,45	0,41	0,39	0,36
f	0,83	0,83	0,71	0,75	0,65	0,62	0,56
g	0,74	0,76	0,74	0,76	0,67	0,62	0,56
h	0,74	0,76	0,85	0,88	0,88	0,84	0,84
i	1	1	1	1	1	1	1

b)

Table 31: Evolution in a) LOC of adaptees, and b) adaptability, compared to ideal realization

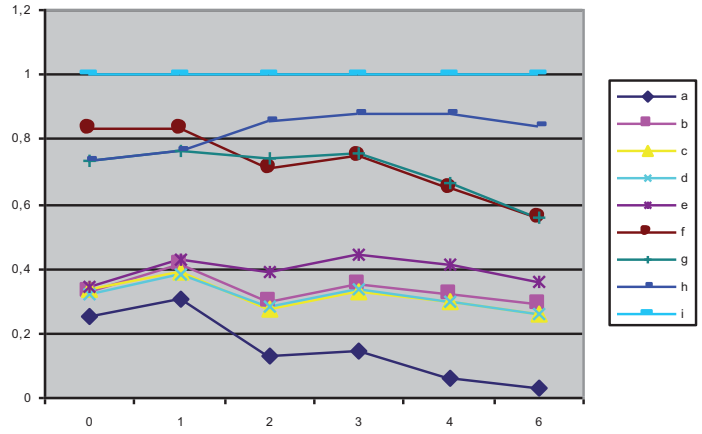


Figure 60: Trends for adaptability, compared to ideal realization

Variability Emphasis

Another goal (G5) in sustainable product line evolution is to emphasize variant elements, so that the family engineer can easily see and control them. Variant elements are most visible when they can be seen at module granularity, that is, externally. However, not all variant elements require external visibility. For example, it is often sufficient for optional variants to be internally visible. In any case, it is undesirable that variant elements are ambiguous, hard to distinguish from common elements. Table 32 lists the numbers of externally visible, internally visible and ambiguous variant elements in the case study code. The corresponding deltas compared to the spatial baseline are listed in Table 33, illustrated graphically in Figure 61. Cloning and the closed mechanisms result in a lack of externally visible variant elements, and the open mechanisms in an excess thereof. Cloning, Conditional Execution, Polymorphism and Aspect-Orientation cause a clear lack of internally visible variant elements. For the runtime mechanisms, this is due to the ambiguity of those elements, as revealed in Figure 61c which shows them as having the highest ambiguous values. For the same reason, this also holds for Aspect-Orientation: it scores worst for internally visible variant elements, and at the same time results in a clear excess of ambiguous variant

elements. This is mainly due to an inherent feature of Aspect-Orientation that many of its proponents claim to be a virtue: obliviousness [Filman+00, Steimann06]. Obliviousness means that the realization of the common elements makes no assumptions on the variant elements which adapt them. While this is a good tactic for isolating variants (Tab.2), it is not well-realized in language-dependent component-based mechanisms such as Aspect-Orientation or Module Replacement because the common elements must usually be functions which the family engineer cannot easily detect as true variation points by looking at core asset code alone, so that these become ambiguous (see Listing 5 and Listing 8).

NV_e	0	1	2	3	4	5	6
a	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0
c	3	4	5	6	7	7	8
d	3	4	5	6	7	7	8
e	0	0	0	0	0	0	0
f	2	3	4	5	6	6	7
g	2	3	4	5	6	6	7
h	2	3	3	4	4	4	4
i	2	3	3	4	4	4	4

a)

NV_i	0	1	2	3	4	5	6
a	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0
d	0	0	0	0	0	0	0
e	3	4	5	6	7	7	8
f	0	0	0	0	0	0	0
g	1	1	2	3	3	3	3
h	1	1	3	4	5	5	6
i	1	1	2	3	4	4	5

b)

NV_a	0	1	2	3	4	5	6
a	0	0	0	0	0	0	0
b	3	4	8	10	11	11	12
c	1	1	5	6	7	7	8
d	0	0	0	0	0	0	0
e	0	0	0	0	0	0	0
f	1	1	5	6	6	6	6
g	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0

c)

Table 32: Evolution in a) externally visible, b) internally visible, and c) ambiguous variant elements

z_s	0	1	2	3	4	5	6
a	-2	-3	-3	-4	-4	-4	-4
b	-2	-3	-3	-4	-4	-4	-4
c	1	1	2	2	3	3	4
d	1	1	2	2	3	3	4
e	-2	-3	-3	-4	-4	-4	-4
f	0	0	1	1	2	2	3
g	0	0	1	1	2	2	3
h	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0

a)

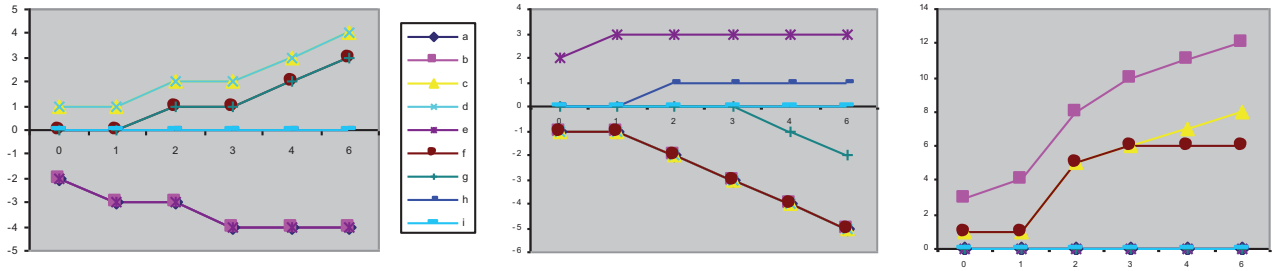
z_s	0	1	2	3	4	5	6
a	-1	-1	-2	-3	-4	-4	-5
b	-1	-1	-2	-3	-4	-4	-5
c	-1	-1	-2	-3	-4	-4	-5
d	-1	-1	-2	-3	-4	-4	-5
e	2	3	3	3	3	3	3
f	-1	-1	-2	-3	-4	-4	-5
g	0	0	0	0	-1	-1	-2
h	0	0	1	1	1	1	1
i	0	0	0	0	0	0	0

b)

z_s	0	1	2	3	4	5	6
a	0	0	0	0	0	0	0
b	3	4	8	10	11	11	12
c	1	1	5	6	7	7	8
d	0	0	0	0	0	0	0
e	0	0	0	0	0	0	0
f	1	1	5	6	6	6	6
g	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0

c)

Table 33: Deltas to baseline for a) externally visible, b) internally visible, and c) ambiguous variant elements



a) b) c)

Figure 61: Trends for a) externally visible, b) internally visible, and c) ambiguous variant element deltas, compared to ideal code

The applied Frame Technology dialect also results in less internally visible variant elements than necessary because it only supports open variation. It compensates this by its ability to make Defaults internally visible. This is why it also does not lead to ambiguities in variant elements. Unoptimized Conditional Compilation, on the other hand, results in an excess of internally visible variant elements because it lacks support of open variability, which the simple Frame Technology dialect has in excess. For that reason, a combination of the two mechanisms is desirable, and this has been performed in the near-ideal sequence "h".

Reuse Efficiency

The efficiency of code reuse (goal G7) can be measured by reuse ratio, using two of the already collected metrics, total and adaptee lines of code. Table 34 shows the aggregated values, and Figure 62 depicts the corresponding trends.

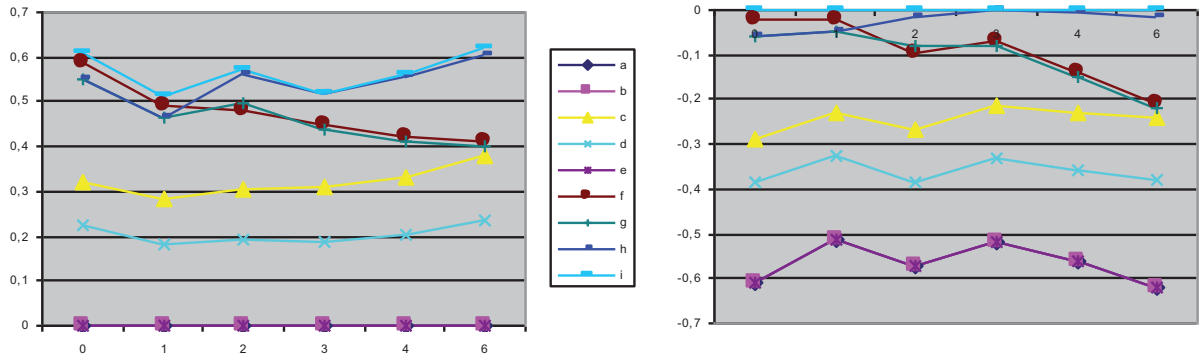
RR	0	1	2	3	4	5	6
a	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0
c	0,32	0,28	0,3	0,31	0,33	0,36	0,38
d	0,22	0,18	0,19	0,19	0,2	0,23	0,24
e	0	0	0	0	0	0	0
f	0,59	0,49	0,48	0,45	0,42	0,43	0,41
g	0,55	0,46	0,49	0,44	0,41	0,41	0,4
h	0,55	0,46	0,56	0,52	0,55	0,56	0,6
i	0,61	0,51	0,57	0,52	0,56	0,58	0,62

a)

z_s	0	1	2	3	4	5	6
a	-0,61	-0,51	-0,57	-0,52	-0,56	-0,58	-0,62
b	-0,61	-0,51	-0,57	-0,52	-0,56	-0,58	-0,62
c	-0,29	-0,23	-0,27	-0,21	-0,23	-0,22	-0,24
d	-0,39	-0,33	-0,38	-0,33	-0,36	-0,35	-0,38
e	-0,61	-0,51	-0,57	-0,52	-0,56	-0,58	-0,62
f	-0,02	-0,02	-0,09	-0,07	-0,14	-0,15	-0,21
g	-0,06	-0,05	-0,08	-0,08	-0,15	-0,17	-0,22
h	-0,06	-0,05	-0,01	-0	-0,01	-0,01	-0,02
i	0	0	0	0	0	0	0

b)

Table 34: a) Evolution in reuse ratio; b) comparison to spatial baseline



a) b)

Figure 62: a) Trends for reuse ratio; b) comparison to spatial baseline

For Cloning, reuse ratio is always 0 because no module is used for building more than a single product. The values for the closed mechanisms are also 0 because variable code was only measured at module granularity, and for these mechanisms, no module exists that contains variant elements only. The highest reuse ratios after step 6 are

obtained in the ideal and well-realized sequences "i" and "h". Aspect-Orientation and the employed Frame Technology dialect result in 20% lower reuse ratios after step 6 because as closed mechanisms they result in more adaptation code at module granularity, compared to mixed-mode mechanisms. The value for Polymorphism is relatively high, due to its large amount of code for both the variable and the common elements. Due to its lack of common initialization code, compared to Polymorphism, the reuse ratio for Module Replacement is constantly lower.

The goal of defaults (Def.55) is to optimize reuse efficiency. Table 35 shows the amount of defaults in the different case study sequences. The conventional mechanisms, applied consistently, do not lead to defaults. Frame Technology, the good mechanism mix and the ideal realization result in a maximum of defaults. When applying Aspect-Orientation in a comparable way, only a single default is obtained.

NOD	0	1	2	3	4	5	6
a	0	0	0	0	0	0	0
b	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0
d	0	0	0	0	0	0	0
e	0	0	0	0	0	0	0
f	1	1	1	1	1	1	1
g	1	1	2	3	3	3	3
h	1	1	2	3	3	3	3
i	1	1	2	3	3	3	3

Table 35: Evolution in number of defaults

Reuse efficiency also suffers if alternative variant modules become too similar. As explained in Section 5.3, the similarity of two or more modules in lines of code can be measured by spatial code churn (edit distance). Tab.36a lists the corresponding values which were obtained using a custom script (Appendix B.2). Table 36b presents the deviation from an ideal case, which is also depicted in Figure 63. The closed mechanisms do not apply here because they only offer a single module. For that reason, they have a strong negative deviation from the ideal case. Like for the other code line-related metrics, Cloning leads to extremely unfavorable metrics after few evolution steps. Polymorphism and Module Replacement also lead to alternative siblings which are more similar than necessary, while Aspect-Orientation, Frame Technology and the mixed realization lead to metrics close to the ideal case.

$\nabla_{LOC,s}$	0	1	2	3	4	5	6
a	32	53	116	163	335	302	604
b							
c	32	52	51	71	71	65	65
d	30	49	55	72	82	76	87
e							
f	12	25	21	35	35	34	34
g	15	28	25	38	38	36	36
h	15	28	25	38	38	36	36
i	15	27	23	37	37	30	30

a)

z_s	0	1	2	3	4	5	6
a	17	26	93	126	298	272	574
b	-15	-27	-23	-37	-37	-30	-30
c	17	25	28	34	34	35	35
d	15	22	32	35	45	46	57
e	-15	-27	-23	-37	-37	-30	-30
f	-3	-2	-2	-2	-2	4	4
g	0	1	2	1	1	6	6
h	0	1	2	1	1	6	6
i	0	0	0	0	0	0	0

b)

Table 36: a) Evolution in spatial code churn among variable siblings; b) comparison to baseline

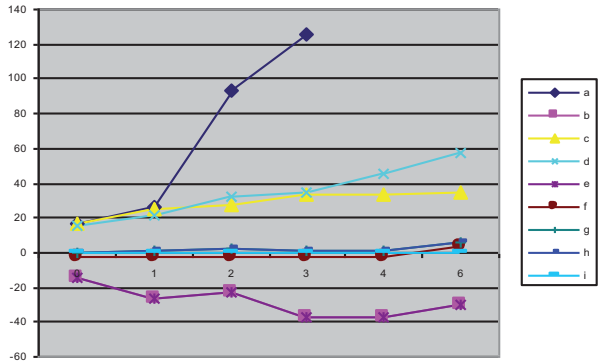


Figure 63: Trends for spatial code churn among variable siblings, compared to ideal code

At a finer level of granularity, the similarity between alternative variable siblings has been measured as compression distance K, as introduced in Section 5.3, again partially automated. The values and deviations are listed in Table 37, and the deviation trend is shown in Figure 64. Again, the closed mechanisms do not apply. The conventional open mechanisms lead to a positive deviation, which means that their variant elements have more commonalities than necessary. The cloned modules are too similar, compared to the ideal case, while the values for the remaining mechanisms converge to the ideal value.

K_{var}	0	1	2	3	4	5	6
a	0,45	0,44	0,41	0,44	0,43	0,45	0,46
b							
c	0,72	0,67	0,74	0,72	0,76	0,77	0,78
d	0,73	0,69	0,74	0,73	0,76	0,76	0,77
e							
f	0,59	0,54	0,58	0,63	0,65	0,68	0,68
g	0,61	0,58	0,64	0,65	0,68	0,69	0,69
h	0,61	0,58	0,6	0,61	0,62	0,66	0,67
i	0,66	0,6	0,61	0,62	0,63	0,65	0,66

a)

z_s	0	1	2	3	4	5	6
a	-0,21	-0,16	-0,2	-0,18	-0,2	-0,2	-0,2
b							
c	0,06	0,07	0,13	0,1	0,13	0,12	0,12
d	0,07	0,09	0,13	0,11	0,13	0,11	0,11
e							
f	-0,07	-0,06	-0,03	0,01	0,02	0,03	0,02
g	-0,05	-0,02	0,03	0,03	0,05	0,04	0,03
h	-0,05	-0,02	-0,01	-0,01	-0,01	0,01	0,01
i	0	0	0	0	0	0	0

b)

Table 37: a) Evolution in compression distance among variable siblings; b) comparison to baseline

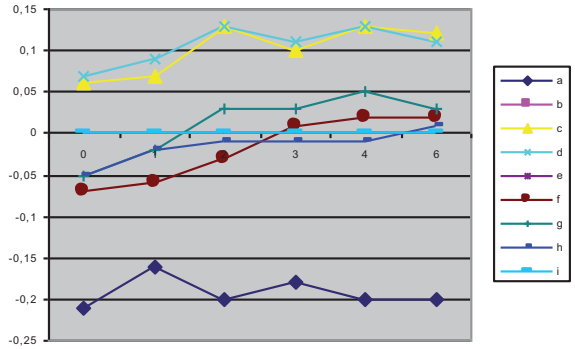


Figure 64: Trends for compression distance among variable siblings, compared to ideal code

Summary

In order to compare complexity trends associated with each mechanism from the case study, the metrics for single complexity factors presented so far were aggregated in two ways: as temporal snapshots and as goal-specific aggregations.

For the first type of aggregation, snapshots of complexity values for the different mechanisms were made that existed at a particular moment in time (i.e. after a certain evolution step). For example, Table 38 lists the measured 17 values for all investigated 9 variability mechanisms after performing the final evolution step 6. The corresponding tables for the other evolution steps 0 to 5 can be found in Appendix D. This representation helps to compare the absolute values caused by all mechanisms at a fixed point in time during evolution.

	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}	NV_e	NV_i	NV_a	RR	NOD	$\nabla_{LOC,s}$	K_{var}
a	1606	1508	40	1	40	388	40	40	40	1606	0	0	0	0	0	604	0,46
b	149	81	1	1	1	43	1	1	1	149	0	0	12	0	0	0	0
c	268	183	13	3	8	47	23	13	13	166	8	0	8	0,38	0	65	0,78
d	219	140	16	3	11	36	12	16	35	167	8	0	0	0,24	0	87	0,77
e	120	64	1	1	1	28	1	11	1	120	0	8	0	0	0	0	0
f	131	74	8	3	7	37	8	8	20	77	7	0	6	0,41	1	34	0,68
g	128	71	8	5	4	35	8	8	17	77	7	3	0	0,4	3	36	0,69
h	128	66	5	2	4	31	5	8	11	51	4	6	0	0,6	3	36	0,67
i	113	60	5	2	4	28	5	8	11	43	4	5	0	0,62	3	30	0,66

Table 38: Seventeen measured values for all mechanisms after evolution step 6

In order to make these values easier to compare, each metric type was normalized to an interval between 0 and 1, where lower values denote less complexity. Because Cloning often resulted in values that exceeded the others considerably, normalization happened against the worst of

the remaining values in these cases, so that the values ranged between 0 for the ideal realization "i", and 1 for the worst case. For example, Table 41 shows that the highest metric for LOC excluding the Cloning case "a" is 268, obtained for Polymorphism ("c"). As a result, the normalized complexity values for LOC are in the range between 0 for mechanism "i" and 1 for mechanism "c", as shown in Table 39.

	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}	NV_e	NV_i	NV_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

Table 39: Normalized metrics from Table 38

Thereafter, average values have been computed for the corresponding four goal categories. For example, the average value for size reduction in Cloning is (9,63+11,8+3,18)/3=8,2. These values, after evolution step 6, are listed in Table 40, together with an unweighted average value for each mechanism, which corresponds to the primary goal G1 (see Fig.45).

	size	shape	emph.	optim.	avg.
a	8,2	6,31	0,67	3,09	4,57
b	0,26	0,56	1	0,88	0,67
c	0,91	0,66	0,89	0,55	0,75
d	0,78	0,73	0,67	0,7	0,72
e	0,15	0,34	0,53	0,88	0,48
f	0,17	0,29	0,75	0,28	0,37
g	0,15	0,29	0,38	0,13	0,24
h	0,05	0,03	0,07	0,04	0,05
i	0	0	0	0	0

Table 40: Aggregated normalized complexities after evolution step 6

The corresponding Kiviat diagram in Figure 65a illustrates that the values for Cloning exceed the others considerably at this late evolution phase. The excerpt in Figure 65b highlights the differences of the remaining mechanisms, for example, that the mechanism "h" actually results in near-ideal values (small deviations from zero), that Polymorphism and Module Replacement result in a large excess in size, that Conditional Compilation performs poor with regard to variability optimization, and that Conditional Execution, Polymorphism and Aspect-Orientation result in lack of variability emphasis.

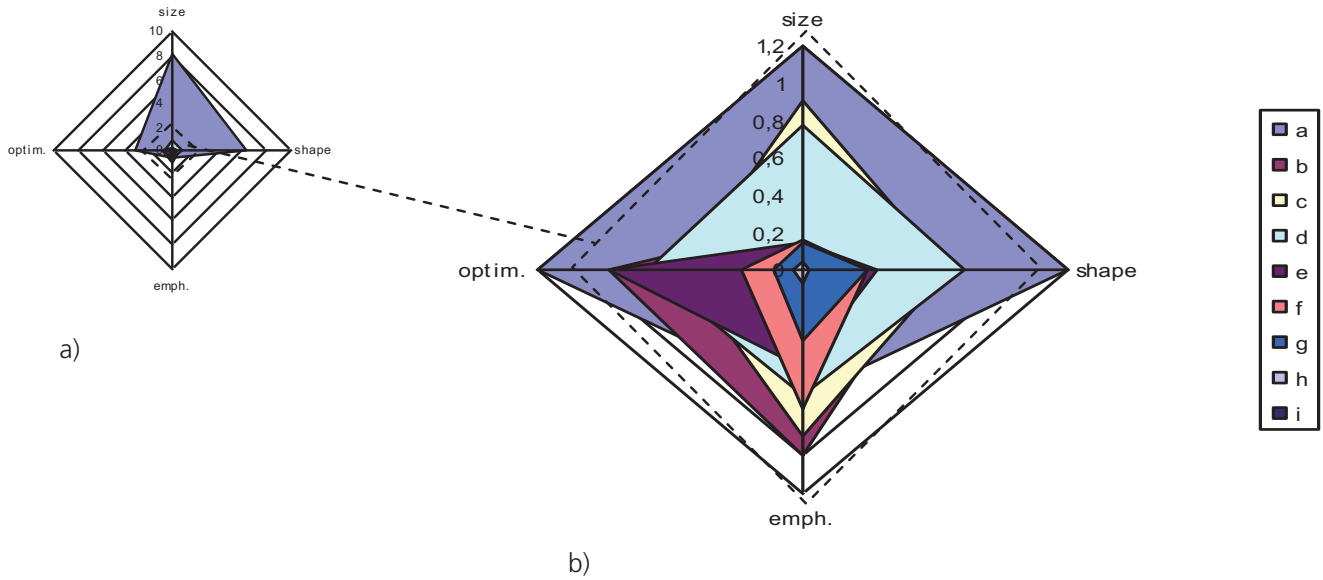


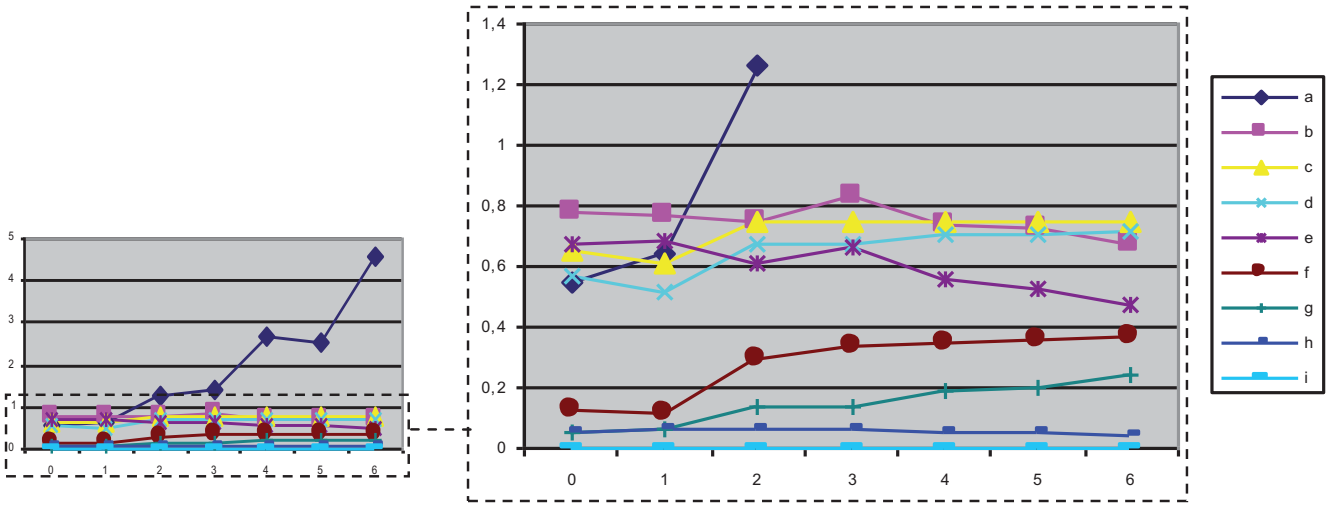
Figure 65: a) Kiviat diagram according to Table 40; b) excerpt for automated approaches

Table 41 summarizes all average values, and Figure 66 shows the corresponding complexity trends per mechanism.

cplx	0	1	2	3	4	5	6
a	0,52	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

Table 41: Evolution in complexity, compared to ideal realization

The figure illustrates that after few product line evolution scenarios in the case study, Cloning leads to a large excess of complexity. The three conventional variability mechanisms Conditional Execution, Polymorphism and Module Replacement lead to a comparable level of overall complexity, while Conditional Compilation results in complexity decrease in later evolution phases. From all mechanisms used in monocultures, the unoptimized Frame Technology dialect used in the case study always results in lowest complexity, while the idealized Aspect-oriented pseudocode leads to nearly 90% more complexity on average, but still to lower complexity than other monocultures. Using a combination of Conditional Compilation and Frame Technology, the lowest complexity was achieved, which also remained constantly low across all evolution steps.



a) b)

Figure 66: a) Complexity trends according to Table 41; b) excerpt for automated approaches

A second type of aggregation was used to illustrate complexity trends at a finer level of granularity, that is, for each goal category. Using the same aggregation process as explained above for average complexity (Tab.40, Tab.41), complexity trends have been investigated for the four goals G3, G4, G5, and G7. As an example, Table 42 shows how complexities according to goal G3 (size reduction) have increased for the different mechanisms. The tables for the other goals are shown in Appendix D.

size	0	1	2	3	4	5	6
a	0,25	0,67	1,74	2	4,45	4,09	8,2
b	0,42	0,45	0,43	0,49	0,35	0,32	0,26
c	0,67	0,89	0,93	0,93	0,92	0,92	0,91
d	0,59	0,72	0,76	0,77	0,77	0,78	0,78
e	0,39	0,4	0,25	0,31	0,2	0,2	0,15
f	0,07	0,1	0,11	0,15	0,16	0,18	0,17
g	0,09	0,12	0,13	0,15	0,15	0,16	0,15
h	0,09	0,12	0,07	0,08	0,06	0,06	0,05
i	0	0	0	0	0	0	0

Table 42: Evolution in size complexity, compared to ideal realization

It was found that the complexity trends for the goals G3, G4, and G7 were similar than the overall trend, with Cloning complexity rising excessively in later evolution phases, while it remained relatively moderate in the complexity trends for variability emphasis (goal G5), shown in Figure 67.

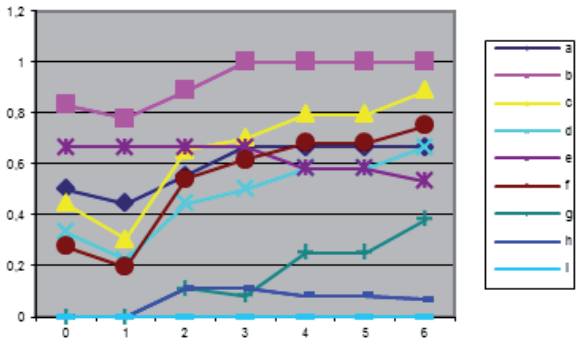


Figure 67: Complexity trends for variability emphasis

6.5 Interpretation

The aggregated results obtained as described above were used to validate the hypotheses for the case study. The following sub-sections list the findings for each group of hypotheses. The corresponding detailed measurement results are shown in Appendix E.

Hypothesis 1

The goal of hypothesis 1 (H1.1 and H1.2) is to investigate the short-term and long-term effects of Cloning on variability complexity. To this end, the complexity values obtained by Cloning in the case study (e.g. row "a" in Tab.41) were compared to the average values of all other monocultures (e.g. rows "b" to "g" in Tab.41). This was both done for the aggregated values per goal (e.g. for goal G3), and for the unweighted complexity of all values, corresponding to goals G1 or G2. As an example, according to Tab.41, the complexity value for Cloning in step 0 is 0.52, and the average value for the other monocultures at this step is (0.77+0.65+0.57+0.67+0.13+0.06+0.06)/7= 0.48 (rounded). This means that Cloning is 0.52/0.48-1= 8% more complex than the others. The corresponding values are shown in Tab.43.

	0	1	2	3	4	5	6
G1	0,08	0,4	1,35	1,52	3,84	3,6	7,48
G3	-0,33	0,51	3,01	3,27	9,44	8,61	19,4
G4	-0,11	0,36	1,48	1,67	5,23	5,14	12,1
G5	0,17	0,23	0,01	0,12	0,03	0,03	-0,1
G7	0,48	0,44	1,26	1,38	2,77	2,54	4,45

Table 43: Cloning complexity excess, compared to other mechanism monocultures

This calculation was also repeated without taking the more advanced mechanisms of Aspect-Orientation and Frame Technology into account, in order to compare Cloning just with the plain mechanisms of

Conditional Execution, Polymorphism, Module Replacement, and Conditional Compilation, which are always available to a family engineer in real-world embedded systems development with C/C++. The results are summarized in Table 44, which is illustrated in Figure 68.

	0	1	2	3	4	5	6
G1	-0,23	-0,01	0,82	0,95	2,86	2,7	5,97
G3	-0,52	0,09	1,95	2,19	6,92	6,38	14,7
G4	-0,37	-0,03	1	1,14	4,1	4,05	10
G5	-0,12	-0,1	-0,2	-0,07	-0,1	-0,1	-0,1
G7	0,02	-0,01	0,63	0,72	1,78	1,65	3,12

Table 44: Cloning complexity excess, compared to other conventional mechanisms

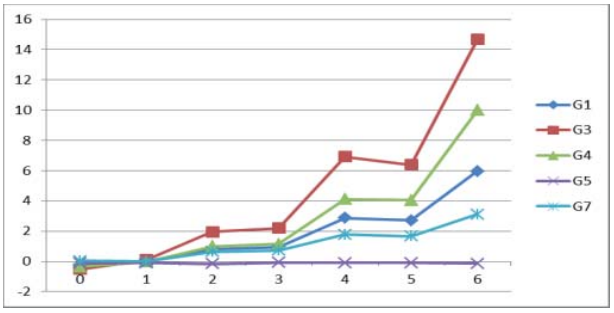


Figure 68: Complexity trends according to Tab.44

The results show that in early evolution phases, especially within the first three evolution steps, Cloning has a very similar, sometimes even lower overall complexity than the other mechanisms (12% average complexity decrease after 2nd step, 19% complexity increase after 3rd step). This supports hypothesis H1.1. Cloning may even be tolerated to a certain degree during the next evolution step (38% complexity increase). But in the long term, as for the steps 4, 5, and 6 in Fig.67, Cloning results in significant complexity increase (88% to 187%), rising more than linearly compared to other mechanisms, which supports hypothesis H1.2.

Hypothesis 2

In order to validate hypothesis H2.1 which investigates the negative impact of late binding on variability complexity, complexity values measured for the two runtime variability mechanisms in the case study, Conditional Execution and Polymorphism, were compared to the complexity values of the other mechanisms, except for Cloning. This was done by comparing their average values, again for the main goal and the three sub-goals. For example, according to Tab.41, the average complexity values for Conditional Execution (row "b") and Polymorphism (row "c") in step 0 is (0.77+0.65)/2= 0.71, whereas the average value for the other four mechanisms (rows "d" to "g") is (0.57+0.67+0.13+0.06)/4= 0.36, so that the complexity of the runtime

mechanisms is 0.71/0.36-1= 99% higher. The values are shown in Tab.45, which is illustrated in Figure 69.

	0	1	2	3	4	5	6
G1	0,99	1	0,73	0,74	0,64	0,64	0,58
G3	0,92	1,01	1,18	1,06	0,98	0,89	0,87
G4	1,21	1,15	0,5	0,54	0,45	0,52	0,46
G5	1	1	0,75	0,82	0,71	0,71	0,62
G7	0,85	0,86	0,65	0,65	0,54	0,5	0,44

Table 45: Runtime mechanism complexity excess

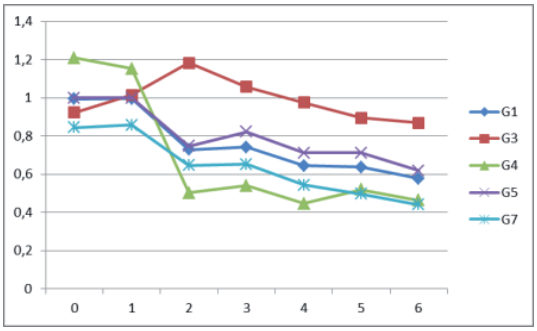


Figure 69: Complexity trends according to Tab.45

The results show that for all goals, runtime mechanisms lead to more complexity than mechanisms with earlier binding times, although with a decreasing trend over time. On average, the complexity of runtime mechanisms is 76% higher than for execution time or construction time mechanisms (Tab.59), which supports hypothesis H2.1.

The goal of hypothesis H2.2 is to investigate the effect of a mechanism's programming language dependence on variability complexity. This was investigated by comparing the average complexity values of the four programming language-dependent mechanisms Conditional Execution, Polymorphism, Module Replacement, and Aspect-Orientation with the average complexity values of the programming language-independent mechanisms Conditional Compilation and Frame Technology in the same way as mentioned above. The results are shown in Table 46 and Fig.70.

	0	1	2	3	4	5	6
G1	0,46	0,34	0,65	0,62	0,68	0,72	0,76
G3	0,82	1,11	1,95	1,52	2,1	2,07	2,52
G4	0,66	0,31	0,57	0,45	0,51	0,69	0,76
G5	0,42	0,13	0,63	0,88	0,84	0,84	0,8
G7	0,16	0,12	0,25	0,2	0,2	0,19	0,19

Table 46: Complexity excess due to programming language-dependence

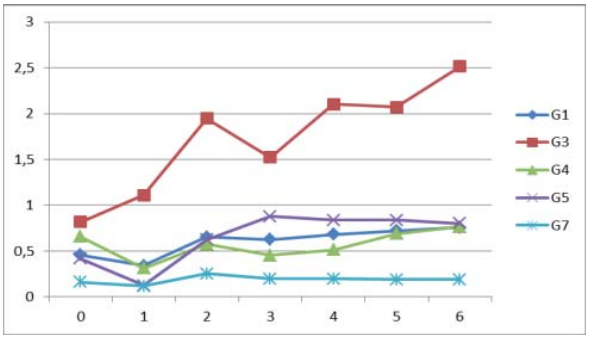


Figure 70: Complexity excess according to Tab.46

The results show that programming language dependent mechanisms have higher complexity values than programming language-independent mechanisms, mostly independent of evolution time. On average, they have been 60% more complex in the case study (Tab.60), which is lower than the average complexity increase due to runtime binding (76%). The results support hypothesis H2.2.

Hypothesis 3

Whereas hypothesis 2 dealt with the influence of code characteristics on complexity, hypothesis 3 contains two code-independent sub-hypotheses which are investigated on source code in the current case study.

The goal of hypothesis H3.1 is to investigate the effect of defaults (Def.55) on variability complexity. As shown in the variability mechanism pattern language, Polymorphism and Module Replacement definitely do not support defaults, while they may be realized, for example, with Aspect-Orientation. Frame Technology has built-in support for defaults. Although not strictly necessary, the aspect-oriented pseudocode in the case study was also realized using defaults, that is, function bodies contained default code that was overridden by *around* advices. This was done in order not to cause complexity disadvantages compared to Frame Technology, where the Default idiom is commonly used. In order to validate the current hypothesis, a second set of Frame Technology realizations has been developed, without making use of defaults which lead to identical code after construction. With all other conditions equal, it should have higher complexity than the original realization with defaults. As Frame Technology may be used for artifacts beyond code, this investigation also applies to other types of artifacts, such as architecture or requirement documents. The results are shown in Table 47 and Figure 71.

The results indicate that, at least for Frame Technology as used in the case study, defaults lead to complexity reduction, especially noticeable in early evolution phases, with a decreasing impact over time. The average complexity reduction over all evolution scenarios was 58% (Tab.61),

which is comparable to the complexity increase caused by language-dependence (60%). The results support hypothesis H3.1.

	0	1	2	3	4	5	6
G1	-0,85	-0,78	-0,59	-0,55	-0,47	-0,45	-0,39
G3	-0,68	-0,52	-0,46	-0,47	-0,39	-0,34	-0,29
G4	-0,78	-0,65	-0,38	-0,37	-0,33	-0,33	-0,29
G5	-1	-1	-0,5	-0,5	-0,25	-0,25	-0,18
G7	-0,93	-0,92	-0,87	-0,84	-0,8	-0,75	-0,73

Table 47: Complexity decrease due to defaults

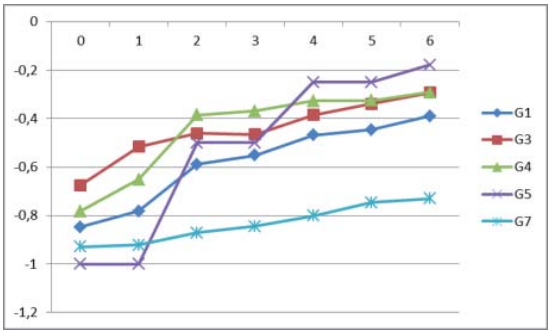


Figure 71: Complexity reduction according to Tab.47

Hypothesis H3.2 investigates if a single variability mechanism that supports both closed and open variation leads to less complex product line infrastructures, and in particular code. In order to validate this, one set of product line infrastructure code in the case study has been realized with two mechanisms (mechanism "h" in Fig.52), one of which supports only closed variation (Conditional Compilation), and the other only open variation (restricted Frame Technology). The resulting complexity is now compared against the average value for the single mechanisms. If that complexity is lower, it would hint at the validity of hypothesis H3.2. The results were obtained in a similar way as described for the two sub-hypotheses H2.1 and H2.2. For example, according to Tab.41, the complexity for mechanism "h" in evolution step 0 is 0.06, while the average complexity of the mechanisms "e" (Conditional Compilation) and "g" (Frame Technology) is (0.67+0.06)/2=0.36, which means a complexity decrease of 85%. All results are shown in Tab.48 and Fig.72.

	0	1	2	3	4	5	6
G1	-0,85	-0,84	-0,83	-0,84	-0,87	-0,84	-0,87
G3	-0,64	-0,55	-0,65	-0,67	-0,65	-0,67	-0,68
G4	-0,75	-0,77	-0,86	-0,88	-0,89	-0,87	-0,9
G5	-1	-1	-0,71	-0,7	-0,8	-0,8	-0,85
G7	-0,92	-0,92	-0,95	-0,98	-0,98	-0,92	-0,93

Table 48: Complexity decrease due to both open and closed variation support

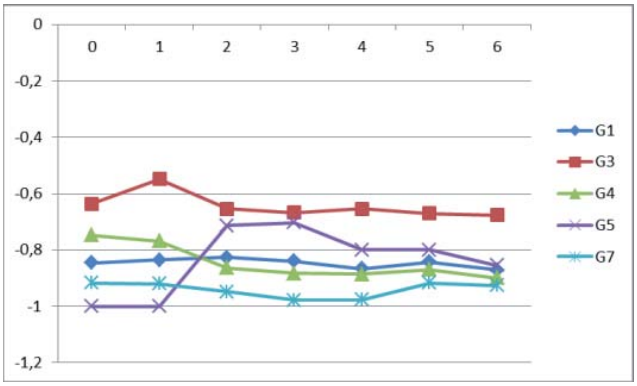


Figure 72: Complexity reduction according to Tab.48

The results show a strong complexity decrease for all goals and across all evolution steps, with a constant trend. The average complexity decrease over all investigated steps is -85% (Tab.62), which is more than the increases for runtime mechanisms (76%) and programming language-dependence (60%). Hypothesis H3.2 is supported by the case study.

Summary and Recommendations

As summarized in Table 49, all six investigated hypotheses have been supported by the case study.

H	Description	Supported
1.1	In short term evolution, Cloning does not result in significantly higher variability complexity than most other mechanisms.	yes (for 2-3 iterations)
1.2	In long term evolution, Cloning results in variability complexity excess and increases evolution effort, compared to any other mechanism.	yes (super-linear growth)
2.1	Late binding increases variability complexity significantly.	yes (76% on avg.)
2.2	Programming language-dependence increases variability complexity significantly.	yes (60% on avg.)
3.1	Defaults decrease variability complexity significantly.	yes (-58% on avg.)
3.2	Support for both closed and open variation decreases variability complexity significantly.	yes (-85% on avg.)

Table 49: Validation summary

The following consequences result for the investigated mechanisms, or for other mechanisms that possess or lack the investigated properties:

Cloning should not be banned from the outset as a product line infrastructure evolution possibility in all contexts of family engineering. Particularly in early evolution stages, when new features are realized in the code, Cloning may be a cost-effective and riskless alternative to more involved variability mechanisms, especially when variation points

remain detectable. Cloning may also be viable if the change frequency of the branched code will be low during future evolution. However, Cloning leads by far to highest complexity excess in long-term evolution, compared to other mechanisms, if the cloned code elements undergo frequent changes.

Mechanisms that result in runtime binding, such as Conditional Execution or Subtype Polymorphism, should not be used for variability management in product line infrastructure code, unless the production process does requires it. This also applies to startup initialization. If the family engineer is uncertain about this issue, he should prefer early binding techniques. In cases when late binding is currently needed, but when the possibility exists that production process changes may make late binding obsolete, and if these changes must then be realized instantly, Aspect-Orientation may be selected if its particular tooling supports both late and earlier binding times.

Family engineers should prefer language-independent mechanisms over language-dependent ones if new features are to be realized and the existing product line infrastructure code is not yet organized in such a way that easily usable variation points exist. The rationale behind this heuristic is that programming language-independent mechanisms may both be used in situations when programming language-dependent variation points exist or if they are missing, whereas programming language-dependent mechanisms require additional refactoring effort in the latter case and potentially make the code harder to evolve in the future. Conditional Compilation, Frame Technology and Cloning are language-independent.

Variability mechanisms that provide Defaults (Def.55) should be preferred over those without Default support, even if this property is not yet needed because this strategy makes it easier to later refactor to defaults if needed. Defaults are especially valuable for realizing optional variabilities or alternatives with few variants which are unlikely to grow to more variants in mid-term evolution. Defaults are valuable in that context because they reduce the number of variant modules in case of open variability, and lead to less configuration effort. Defaults can be realized with many mechanisms, but are unsupported in Module Replacement and Subtype Polymorphism.

When developing support for product line infrastructure evolution, both closed and open variation should be supported, as this reduces the complexity of mechanism excess often observed in family engineering in practice [Krueger07].

6.6 Threats to Validity

The validity of the case study results can be threatened along at least five dimensions [Wohlin00, Yin03]: internal, external, construct, conclusion, and reliability validity, which are discussed in this section. Threats to internal validity are internal issues that may affect that the conclusions drawn from the case study are true, for example in case of false positives or false negatives. External validity denotes to what degree the results may be generalized, in this case for example to real-world product line infrastructures. Construct validity ensures that the construction of the case study is related to the investigated research problem. Conclusion validity is concerned with the statistical significance of the results. Threats to reliability validity are associated with reproducibility issues.

Internal Validity

Internal validity is threatened in two respects. First, the realization of the product line infrastructures may contain defects that lead to wrong measurement results, and second, the measurements may be erroneous.

Concerning the first issue, the product line infrastructure code in the case study was created manually by three different subjects, which may increase inconsistencies, but also reduced the risk of undetected errors. In order to avoid deviations in functionality, the underlying hardware abstraction layer code has been unit tested on the target hardware, and representative product samples have been deployed at fixed intervals. In order to avoid construction errors, continual construction tests (Sec.5.2) were made which documented the consumed resources of all produced products. The code was not developed at one point in time, but has been evolved over more than one year, in several dozens of stable versions, which improves confidence in its low defect rate. On the other hand, during this time there were several changes in the applied technology (one hardware update and two compiler updates), which could have caused functionality defects. However, this was counteracted by developing a stable hardware abstraction layer, and issues of functionality were not the primary measurement concern. Moreover, efforts have been made to keep the code as simple and consistent as possible, for example by deliberately using a minimum of comments in the code, or by using Unix symbolic links for realizing identical modules in successive stages of evolution, instead of cloning them.

The issue of incorrect measurement results was counteracted by partially automating the measurement procedure. The obtained measurement tools were also continually improved, and they were unit-tested as well. However, some measurements were still performed manually, as well as documenting the results in spreadsheets and aggregating them there, which may have caused certain irregularities. To reduce the probability of

manual measurement errors, they were always performed within a short interval, uninterrupted, while cross-checking trends. Inconsistencies among aggregations were avoided by spreadsheet automation.

External Validity

External validity can be threatened by the development environment used in the case study, the studied system itself, and the development process.

To ensure external validity of the development environment, standard development tools were used (the open-source C compiler GCC, the build system GNU make, standard development environments such as Eclipse and Emacs, and the host operating systems Windows, Linux and MacOS), which in our experience represent in these combinations the state of the practice in small to medium-sized embedded systems development. The employed wireless sensor network hardware is part of a commercial system that has also been used in various other academic and industrial settings[21].

The studied system itself is a small-scale embedded system, with comparable functionality and sophistication than I have seen in various industrial settings in the automotive and related industries. Although only a single product line infrastructure was studied, it resembles reuse infrastructures used in these environments, which often consist of a mix of custom and 3rd party code. In a similar way, the C99 standard libraries and PIC controller-specific libraries provided by the SDCC compiler and Particle-specific code accompanying the hardware were 3rd party code in the case study. Yet, more case studies should be performed on systems of different application types and sizes, in order to generalize my findings in other application domains beyond embedded systems and to evaluate scalability issues (see also the outlook in Sec.7).

Another external threat is if the results for Aspect-Orientation may be generalized to industrial practice, and if they may be compared to the others, as only the Aspect-oriented pseudocode in the case study is less realistic than the code in all other cases, which can be run on real hardware. On the one hand, it is difficult to judge industrial usage of Aspect-Orientation in real-world projects at all, especially in embedded systems development and in product lines, as reports and standard tool support are missing. On the other hand, the study could be repeated if an aspect weaver is developed that supports the SDCC compiler.

A threat to external validity is also if the results for Cloning may not apply in product line development in practice. As shown in Section 3.3, various recent empirical studies have shown that Cloning is not generally

[21] particle.teco.edu/publications (retrieved August 2009)

harmful in single systems development in practice, especially for variability in time. As this is the first study to investigate these issues for both variability in space and time, further work should investigate this issue.

As product line infrastructures are just entering mainstream industrial development, a typical development process, especially in product line realization, has not yet been established. This makes it hard to judge typical product line evolution steps. However, from my experience, the predominating types of variability are optional and alternative, and these have been covered in the case study in representative scenarios. Numeric variabilities which play a greater role in other product line realization examples [Weiss+99] have deliberately been omitted, as they typically do not pose serious realization problems. In my experience, either proactive approaches as documented in [Pohl++05] are applied for new projects, or more reactive approaches are used for product line adoption in existing projects [Buhrdorf++04]. The former represent more an initial development situation for P- or S-type systems (Sec.3.4) which are not the focus of this thesis. For the latter, more studies should be conducted to confirm the results found in this thesis.

Another process-related external threat is only a minority of the practically relevant types of product line infrastructure evolution has been covered. It is true that the current thesis deliberately focuses only on those evolution scenarios in which each version invalidates its predecessor, so that evolution only happens on the most current version. This is because the focus is on new development. I am aware that there is another type of evolution situation in practice, in which not (only) the current version must evolve, but also previous ones, which becomes a topic for configuration management [Anastasopoulos++09]. Follow-up studies are necessary to evaluate synergies of the two approaches.

Construct Validity

There may be threats to construct validity concerning my quality model. A threat to both construct and external validity is if the invented metrics are indicators for complexity growth in product line infrastructures in practice. On the one hand, as comparable types of metrics have been missing, and as they must have practical relevance, the most popular mechanism for goal-oriented software measurement in practice, the GQM method, has been applied for metrics identification. On the other hand, the criteria for using the described goals were based on decades of practical reuse research [Bassett97], which increases validity. Both of these measures were taken to increase construct validity. The construct validity threat of redundant metrics is mitigated because the metrics are based on a disjoint set of goals. Another threat in this category is if measurements are comparable for all mechanisms, for example in case of Compression Distance K_{var} for variant sibling modules, when some

mechanisms inevitably had more such modules than others, increasing their compression distance. Such measurements were balanced by using comparable modules in both cases. Construct validity for each goal is also increased because multiple sources of evidence (i.e., multiple metrics) are provided for each goal.

A related threat is that realizations using different mechanisms at the same evolution stage are not comparable because of mutual inconsistencies that are not related to product line issues, for example different code formatting or coding conventions. While such inconsistencies due to manual realization cannot be completely avoided, effort has been spent to reduce them as much as possible, for example by using a single editor with identical formatting rules (C mode in GNU Emacs) for developing all code, or by semi-automatically comparing the respective realizations with *diff* to keep them as consistent as possible.

Another threat for both external and construct validity is that all goals and metrics were given equal weighting during aggregation, which makes the results unrealistic. While it is true that not all goals and metrics are equally important in each family engineering context, they must be customized (prioritized and extended) in an industrial context, using the given goals as a starting point.

Construct validity is also threatened by mechanism or experimenter bias. While it is true that the developed complexity model was influenced by ideas that led to the development of Frame Technology, these ideas were conceptual ones that did not penalize a particular mechanism (except for Cloning) in advance.

Conclusion Validity

Two conclusion validity threats exist for the current case study. First, the data set is relatively small, so that further studies with larger evolution scenarios, for example, would be necessary to increase statistical significance. Second, the smaller long-term differences in average complexity among many mechanisms could also mean that they may arbitrarily be selected in the long term. However, finer-grained investigations lead to the opposite results, that significant complexity reduction can in fact be achieved.

Reliability Validity

Reliability validity could be threatened if the results are not reproducible. However, various artifacts such as code, build scripts, measurement scripts and aggregation tables are available so that all automatic results can instantly be reproduced. Documented hints in the aggregation tables also make it possible to reproduce the manual measurements.

7 Summary and Outlook

This thesis presented a reactive product line evolution method that helps family engineers in practice to keep product line infrastructure code reusable in the long run. Like conventional single-system development, the development of product lines does not end with initial construction. In practice, product line infrastructure code becomes less reusable over time. The reason is not only that the scope changes and that the product line infrastructure is not changed, but the cause is also that the code becomes increasingly complex because it is changed inappropriately.

To avoid this situation, this thesis offers a practical guide, aimed at family engineers, for well-behaved product line infrastructure code evolution so that its decay is avoided. The focus is on product line realization techniques and variability management in the code, which provides the foundation for a software product line practice [Krueger07]. In contrast to the single system case, product line infrastructure code evolution is more difficult because it is developed to be reused. This requires the family engineer to make additional trade-offs among variability, reuse efficiency, or ease-of configuration, which causes additional complexity, but much of this variability-related complexity is non-essential.

Variability mechanisms allow the family engineer to intentionally realize variability in core assets. A set of five orthogonal tactics for effective family realization is developed, and a set of seven types of plain variability mechanisms is presented which cover all combinations of the mentioned tactics. The set of mechanisms comprises Cloning, Conditional Execution, Polymorphism, Module Replacement, Conditional Compilation, Aspect-Orientation and Frame Technology. For didactic purposes, the mechanisms are presented as interconnected elements of a pattern language, in a format known to software engineers in practice which highlights each mechanism's intent, motivation, applicability, process, consequences, details and related patterns, and which addresses the family engineer directly. Except for Cloning, all mechanisms aim to make the resulting code easier to use in the long term by consolidating common core asset elements and separating them from variant elements in different ways. By selecting those tactics and combinations of tactics which are at least required, the family engineer is guided towards variability mechanisms with low complexity.

Within the larger product line evolution method developed in this thesis, the mentioned variability mechanisms are one input to a product line realization process. Product line evolution scenarios are a second input. They serve to describe recurring product line requirements that can result

in more complex product line infrastructure code. Nine different scenarios are presented, and it is shown that these scenarios consist of combinations of atomic evolution steps. It is also shown that the order of these sub-steps has an impact on complexity qualities of the overall evolution process.

The product line realization process developed in this thesis consists of the three phases Selection, Modification, and Quality Assurance, performed in an incremental and iterative way. The goal of the Selection phase is that the family engineer understands how variability is currently managed in the existing code, which particular core assets are probably affected by the upcoming change, and which defects exist in the current code that may have a negative impact on realizing these changes. A classification of 23 typical defects (product line infrastructure code smells) is given.

The goal of the Modification phase is for the family engineer to realize the required changes in the given product line infrastructure code. It is explained that a core asset can be regarded as symmetrical if it only contains a common element, whereas it becomes more asymmetric when variation points or variants are added. A catalog of 37 variability refactorings is presented whose purpose is to counteract the above-mentioned product line infrastructure code smells.

The Quality Assurance phase consists of product line testing and measurement sub-activities. It is shown that product line testing is not only different from conventional testing because its artifacts contain common and variant elements. The testing process is split into two parts: Construction Testing and Execution Testing. The novel Construction Testing process only tests whether the product can be produced. The following Execution Testing process then performs conventional tests at runtime.

For variability complexity measurement, a quality model according to the GQM approach is proposed, consisting of seven goals, organized in a 3-level goal hierarchy. The goals are cost effective product line development, variability complexity reduction, and five basic sub-goals. The sub-goals are size reduction, shape alignment, variability emphasis, variability management consistency, and reuse efficiency. Questions are given for all goals, and all basic sub-goals are refined to 22 concrete, mostly newly invented metrics. For size reduction, the metrics are lines of product line infrastructure code, temporal code churn, number of modules and number of variation points. Shape alignment metrics are the depth and width of the reuse hierarchy, lines of adaptee code and four different cyclomatic complexity types, three of which have newly been developed, and two of which are product line-specific. Variability emphasis metrics are the numbers of externally and internally visible variant elements, and the number of ambiguous variant elements.

Proposed metrics for variability management consistency are the number of inconsistent usages of a mechanism, the number of inconsistent variability mechanisms and the number of configuration inconsistencies. Reuse efficiency is measured by reuse ratio, number of defaults, spatial code churn among variant siblings and compression distance of spatial variant siblings.

Because the underlying complexity concept used in this thesis always requires a reference, a product line-specific baselining approach is developed. It extends the single system approach which uses a temporal reference by a second dimension that applies a spatial reference simultaneously. The spatial baseline corresponds to an ideal product line realization which expresses the family engineer's variability management intentions, possibly with variability management pseudocode. This is illustrated in the case study.

The case study simulates the evolution of an embedded systems product line as seen in practice. The goal is to compare what impact the presented variability mechanisms have on different dimensions of variability complexity and on sustainable evolution. Six different hypotheses are developed, two of which investigate the impact of Cloning on variability complexity, and four are mechanism-independent, investigating properties suggested in the family realization tactics.

The code, developed in the C programming language, runs on real hardware: wireless sensor nodes that form part of an ambient intelligence system. All product line members capture product-specific monitored variables of their physical environment through different types of sensors and in different ways. Some product line members collect this information. All product line members wirelessly transmit the results to a receiver in certain intervals. The product line initially consists of three different products. In six evolution steps, the product line is gradually changed, mostly enhanced by new products or features, as instances of the product line evolution scenarios described above. This means that seven different product lines have been realized as the product line's evolution trace. Each of these are realized using a monoculture of all the seven variability mechanisms under discussion, plus pseudocode corresponding to ideal product line realizations, according to a fixed set of family engineering tactics, and a realization of this ideal baseline using a best mix of the mechanisms. All 63 product lines were realized as consistently as possible, to eliminate complexities due to inconsistencies. In order not to favor single mechanisms in advance, the source code was deliberately kept as simple as necessary, which meant omitting premature or arbitrary separation of variants or function extractions. For all 63 product line realizations, measurements have been performed for five different goals of the quality model, using 17 of the above mentioned metrics.

The results show that all hypotheses are supported. In accordance with recent results in single-system clone research, Cloning was shown not to be universally harmful, even in a product line context. In the short-term, it resulted in similar complexity than the other mechanisms, and only in the long term it performed considerably worse. The language-independent hypotheses which expected significant results in complexity reduction were all supported, with average reduction rates between 58% and 80%.

Outlook

The presented approach of sustainable product line infrastructure code evolution is extensible in various respects. First, the complexity ideas should be extended to cover single system development as well, which means evaluating which semantic source code elements are essential and which cause arbitrary complexities in specific contexts, and to deliberately omit the latter elements when there is no reason to use them. For example, if a multi-paradigm programming language such as C++ is used, deliberate simplification would mean not to apply object-orientation unless its presence becomes essential for the software development task at hand. Extension to single system development also means tailoring the presented product line realization process, with its selection, modification and quality assurance phases, to one-of-a-kind systems, omitting the spatial dimension and just focusing on temporal evolution. This also involves continuous feedback on architectural compliance, both for during realization and architecting, as well as prediction of likely system evolution.

A second extension of the approach is to broaden its dynamic aspect as a process description from realization activities to other software engineering activities, for example architecting, so that all product line infrastructure artifacts are systematically simplified and become more evolvable. This point is especially concerned with finding an appropriate overall software development sequence in a particular context, balancing proactive and reactive approaches in such a way that the resulting product line generations become as sustainable as required. A related issue would be to apply formal algebraic approaches to describe and prescribe the product line development process. This will result in a group theoretic method for passively and actively controlling the product line realization process, where changes in common and variable user needs are characterized by operations on symmetries and asymmetries in the structure or structures[22] of product line assets, covering both variability in space and variability in time. This means that alternative approaches could be researched for describing product line ecosystems,

[22] This term refers to Clements' definition of software architecture as "the structure or structures of the system", according to [Clements01] a tribute to Parnas' early contributions to this field [Parnas74].

where the focus is not to characterize their common and variable elements, but to elaborate the common and variable activities which lead to their development and evolution.

Third, the scalability of the product line evolution method proposed in this thesis should be empirically validated in the development of larger and long-lived product line systems, for example in prominent open-source systems or heavily reused systems in industry, in order to better evaluate its usefulness under real-world constraints. This could also mean to apply it in other development contexts besides embedded systems development, for example in IT systems development, where different trade-offs between usability and reusability are made. Scaling up the method will also require more tool automation, for example by extending the measurement or variability management tools developed in this thesis, as a complement to existing tools for checking architectural compliance in single systems [Knodel10], alongside tools for variant comparison [Duszynski++09]. More tool support could comprise Recommendation Systems [Robillard++10] for variability management. Other practically relevant issues to investigate in this context are integrated solutions for handling the co-existence of multiple product line infrastructures at a fixed point in time (see the discussion on [Elsner+10] and Fig.23 in Sec.3.5), and solutions to foster bounded combinatorics (see the discussion in Sec.3.2).

Fourth, synergies with other dimensions of product line realization technologies should be researched, in particular how a unified variability management approach across space and time can be facilitated by product line-specific construction and build environments (more modern build environments than Makefiles, such as the *scons*[23] build system already offer some variability support), configuration management [Anastasopoulos++09] (for expressing product line snapshots), and variability management in the source code (by synergies with commercial product line tools such as *Gears*[24] or PureVariants[25]). This also means to refine some of the evaluated methods and techniques, for example to invent lightweight variability assets (Def.57) for core asset code, or to automate code smell detection and refactoring activities.

Finally, method and tool support could also be improved in other areas, for example in quality assurance or agile development. To this end, the applicability of novel Cloning approaches, such as Clone Region Descriptors [Ekoko+10], should be investigated. In product line quality assurance, synergies of Construction Testing and other published product line testing approaches should be investigated, as well as

[23] www.scons.org (retrieved August 2009)

[24] www.biglever.com (retrieved August 2009)

[25] www.pure-systems.com (retrieved August 2009)

variability inspection approaches, or the usefulness of the Alpha Complexity metric (Sec.3.4) for variability complexity detection and reduction.

References

[Adams06] B. Adams: AOP on the C-Side. LATE-2, AOSD 2006

[Ajila+07] S.A. Ajila, R.T. Dumitrescu: Experimental Use of Code Delta, Code Churn, and Rate of Change to Understand Software Product Line Evolution. Journal of Systems and Software 80(1), 74–91, 2007

[Ajila+08] S.A. Ajila, A.B. Kaba: Evolution support mechanisms for software product line process. Journal of Systems and Software 81: 1784-1801, 2008

[Alexander++77] C. Alexander, S. Ishikawa, M. Silverstein: A Pattern Language. Oxford University Press, 1977

[Alexander02] C. Alexander: The Nature of Order, Book 2. CES Publishing, 2002

[Alexandrescu01] A. Alexandrescu. Modern C++ Design. Generic Programming and Design Patterns Applied. Addison-Wesley, 2001

[Ali++10] M.S. Ali, M. Ali Babar, L. Chen, K.-J. Stol: A systematic review of comparative evidence of aspect-oriented programming. Information and Software Technology 52: 871-887, 2010

[Alves++05] V., P. Matos Jr., L. Cole, P. Borba, G. Ramalho: Extracting and Evolving Mobile Games Product Lines. SPLC-9: 70-81, Springer LNCS 3714, 2005

[Alves++06] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, C. Lucena: Refactoring Product Lines. GPCE'06: 201-210, 2006

[Alves07] V. Alves: Implementing Software Product Line Adoption Strategies. PhD Thesis, Universidade Federal de Pernambuco, 2007

[Amin++10] F. Amin, A.K. Mahmood, A. Oxley: A Review on Aspect-Oriented Implementation on Product Line Components. Information Technology Journal 9(6): 1262-1269, 2010

[Anastasopoulos+01] M. Anastasopoulos, C. Gacek: Implementing Product Line Variabilities. ACM Software Engineering Notes 26(3): 109-117, 2001

[Anastasopoulos++04] M. Anastasopoulos, D. Muthig, I. John: Model-driven and Efficient Development (of Embedded Systems). A Case Study from the Mobile Phone Domain. Embedded World 2004: 615-624, 2004

[Anastasopoulos++09] M. Anastasopoulos, D. Muthig, T.H. Burgos de Oliveira, E.S. Almeida, S.R. de Lemos Meira: Evolving a Software Product Line Reuse Infrastructure: A Configuration Management Solution. VaMoS-3: 19-28, 2009

[Apel07] S. Apel: The Role of Features and Aspects in Software Development. PhD Thesis, University of Magdeburg, 2007

[Apel++07] S. Apel, C. Lengauer, D. Batory, B. Möller, C. Kästner: An Algebra for Feature-Oriented Software Development. Technical Report MIP-0706, Department of Informatics and Mathematics, University of Passau, 2007

[Apel+09] S. Apel, C. Kästner: An Overview of Feature-Oriented Software Development. Journal of Object Technology 8(5), 2009

[Atkinson++01] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, J. Zettel. Component-Based Product-Line Engineering with UML. Addison-Wesley, 2001

[Aversano++07] L. Aversano, L. Cerulo, M. Di Penta: How Clones are Maintained: An Empirical Study. CSMR-11: 81-90, 2007

[Bachmann++04] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, A. Vilbig: A Meta-model for Representing Variability in Product Family Development. PFE-5: 66-80, Springer LNCS 3014, 2004

[Bachmann+05] F. Bachmann, P. C. Clements: Variability in Software Product Lines. Technical Report CMU/SEI-2005-TR-12, Software Engineering Institute, 2005

[Bakota++07] T. Bakota, R. Ferenc, T. Gyimothy: Clone Smells in Software Evolution. ICSM-23: 24-33, 2007

[Basit++05] H.A. Basit, D.C. Rajapakse, S. Jarzabek: Beyond Templates: A Study of Clones in the STL and Some General Implications. ICSE-27: 451-459, 2005

[Bass++97] L. Bass, P.Clements, S. Cohen, L. Northrop, J. Withey: Product Line Practice Workshop Report. Technical Report CMU/SEI-97-TR-003, Software Engineering Institute, 1997

[Bass++98] L. Bass, P. Clements, R. Kazman: Software Architecture in Practice. Addison-Wesley, 1998

[Bass++03] L. Bass, P. Clements, R. Kazman: Software Architecture in Practice (2nd ed.). Addison-Wesley, 2003

[Bass++04] L. Bass, F. Bachmann, M. Klein: Making Variability Decisions during Architecture Design. PFE-5: 454-465, Springer LNCS 3014, 2004

[Bassett84] P.G. Bassett: Design Principles for Software Manufacturing Tools. ACM'84 Annual Conference: 85-93, 1984

[Bassett87] P.G. Bassett: Frame-Based Software Engineering. IEEE Software 44(4): 9-16, 1987

[Bassett97] P.G. Bassett: Framing Software Reuse. Lessons From The Real World. Prentice Hall, 1997

[Bassett02] P.G. Bassett: Does (Software + Engineering) = Software Engineering? Position Paper, Canada's Association of I.T. Professionals (CIPS), 2002

[Bassett07] P.G. Bassett: The Case for Frame-Based Software Engineering. IEEE Software 24(4): 90-99, 2007

[Batory++04] D. Batory, J.N. Sarvela, A. Rauschmayer: Scaling Step-Wise Refinement. IEEE Trans. Software Engineering 30(6): 355-371, 2004

[Batory05] D. Batory: Feature Models, Grammars, and Propositional Formulas. SPLC-9: 7-20, Springer LNCS 3714, 2005

[Batory++06] D. Batory, David Benavides, Antonio Ruiz-Cortés: Automated Analysis of Feature Models: Challenges Ahead. Communications of the ACM 49(12): 45-47, 2006

[Bayer++99] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, J.-M. DeBaud: PuLSE: A Methodology to Develop Software Product Lines. SSR'99: 122-131, 1999

[Bayer+02] J. Bayer, T. Widen: Introducing Traceability to Product Lines. PFE-4: 399-406, Springer LNCS 2290, 2002

[Bayer04] J. Bayer: View-Based Software Documentation. PhD Theses in Experimental Software Engineering Vol.15, Fraunhofer-Verlag, 2004

[Bayer++06] J. Bayer, S. Gerard, Ø. Haugen, J. Mansell, B. Møller-Pedersen, J. Oldevik, P. Tessier, J.-P. Thibault, T. Widen: Consolidated Product Line Variability Modeling. In [Käkölä+06]: 195-241

[Beck96] K. Beck: Smalltalk Best Practice Patterns. Prentice Hall, 1996

[Beck02] K. Beck: Test-Driven Development: By Example. Addison-Wesley, 2002

[Becker00] M. Becker: Generic components: a symbiosis of paradigms. GPCE'00, 100-113, 2000

[Becker04] M.Becker: Anpassungsunterstützung in Software-Produktfamilien. PhD Thesis, University of Kaiserslautern, 2004

[Berg++05] K. Berg, J. Bishop, D. Muthig: Tracing Software Product Line Variability – From Problem to Solution Space. SAICSIT 2005: 182-192

[Berger++10] C. Berger, H. Rendel, B. Rumpe: Measuring the Ability to Form a Product Line from Existing Products. VaMoS-4: 151-154, 2010

[Bettenburg++10] N. Bettenburg, W. Shang, W.M. Ibrahim, B. Adams, Y. Zou, A.E. Hassan: An empirical study on inconsistent changes to code clones at the release level. Science of Computer Programming, 2010 (in print)

[Boehm10] B. Boehm: The Changing Nature of Software Evolution. IEEE Software 27(4): 26-29, 2010

[Booch91] G. Booch: Object-Oriented Design with Applications. Benjamin/Cummings Publishing, 1991

[Bosch00] J. Bosch: Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley, 2000

[Bosch02] J. Bosch: Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. SPLC-2: 247-262, Springer LNCS 2379, 2002

[Bosch++02] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J.H. Obbink, K. Pohl: Variability Issues in Software Product Lines. PFE-4: 13-21, Springer LNCS 2290, 2002

[Bosch09] J. Bosch: From Software Product Lines to Software Ecosystems. SPLC-13: 111-119, 2009

[Brcina++09] R. Brcina, S. Bode, M. Riebisch: Optimisation Process for Maintaining Evolvability during Software Evolution. ECBS-16: 109-118, 2009

[Breivold++08] H.P. Breivold, I. Crnkovic, P.J. Eriksson: Analyzing Software Evolvability. COMPSAC-32: 327-330, 2008

[Breivold09] H.P. Breivold: Software Architecture Evolution and Software Evolvability. PhD Thesis, University of Mälardalen, 2009

[Brooks95] F.P. Brooks: The Mythical Man-Month: Essays on Software Engineering (20th Anniversary Edition). Addison-Wesley, 1995

[Brooks10] F.P. Brooks: The Design of Design. Addison-Wesley, 2010

[Brownsword+96] L. Brownsword, P. Clements: A Case Study in Successful Product Line Development. Technical Report CMU/SEI-96-TR-016, Software Engineering Institute, 1996

[Bruntink++04] M. Bruntink, A. v.Deursen, R. v.Engelen, T. Tourwe: An Evaluation of Clone Detection Techniques for Identifying Cross-Cutting Concerns. ICSM-20: 200-209, 2004

[Bühne++04] S. Bühne, K. Lauenroth, K. Pohl: Why is it not Sufficient to Model Requirements Variability with Feature Models? AURE04: 5-12, 2004

[Buhrdorf++04] R. Buhrdorf, D. Churchett, C.W. Krueger: Salion's Experience with a Reactive Software Product Line Approach. PFE-5: 317-322, Springer LNCS 3014, 2004

[Buschmann++96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: Pattern-Oriented Software Architecture: A System of Patterns. Wiley, 1996

[Campbell++90] G.H. Campbell, S.R. Faulk, D.M. Weiss: Introduction to Synthesis. Software Productivity Consortium, 1990

[Campbell07] G. Campbell: Software-Intensive Systems Producibility: A Vision and Roadmap (v 0.1), Technical Note CMU/SEI-2007-TN-017, Software Engineering Institute, 2007

[Chen++05] Y. Chen, R. Dios, A. Mili, L. Wu, K. Wang: An Empirical Study of Programming Language Trends. IEEE Software 22(3): 72-79, 2005

[Cilibrasi+05] R. Cilibrasi, P. Vitanyi: Clustering by Compression. IEEE Trans. Information Theory 51(4): 1523-1545, 2005

[Classen++08] A. Classen, P. Heymans, P. Schobbens: What's in a Feature: A Requirements Engineering Perspective. FASE 2008: 16-30, 2008

[Clements01] Introduction to [Parnas74]. In [Hoffman+01]: 157-159

[Clements+01] P. Clements, L.M. Northrop: Software Product Lines: Practices and Patterns. Addison-Wesley, 2001

[Clements++03] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford: Documenting Software Architectures. Views and Beyond. Addison-Wesley, 2003

[Clements+06] P. Clements, D. Muthig (Eds.): Variability Management – Working with Variability Mechanisms (SPLC-10 Workshop). Fraunhofer IESE Report 152.06/E, 2006

[Coady++01] Y. Coady, G. Kiczales, M. Feely, G. Smolyn: Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. ESEC/SIGSOFT FSE 2001: 88-98

[Codenie++10] W. Codenie, N. Gonzalez-Deleito, J. Deleu, V. Blagojevic, P. Kuvaja, J. Similä: Managing Flexibility and Variability: A Road to Competitive Advantage. In [Kang++10]: 269-313

[Coleman++94] D. Coleman, D. Ash, B. Lowther, P. Oman: Using Metrics to Evaluate Software System Maintainability. IEEE Computer 27(8): 44-49, 1994

[Coplien91] J.O. Coplien: Advanced C++ Programming Styles and Idioms. Addison-Wesley, 1991

[Coplien99] J.O. Coplien: Multi-Paradigm Design for C++. Addison-Wesley, 1999

[Cordy03] J.R. Cordy: Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation. IWPC-11: pp.196, 2003

[Crochemore+96] M. Crochemore, T. Lecroq: Pattern-Matching and Text-Compression Algorithms. ACM Computing Surveys 28(1): 39-41, 1996

[Curtis79] B. Curtis. In search of software complexity. Workshop on Qualitative Software Models for Reliability, Complexity and Cost: 95-106, 1979

[Czarnecki++00] C. Czarnecki, U.W. Eisenecker: Generative Programming: Methods, Tools and Applications. Addison-Wesley, 2000

[Damerau64] F.J. Damerau: A Technique for Computer Detection and Correction of Spelling Errors. Communications of the ACM 7(3): 171-176, 1964

[Davis87] S.M. Davis; Future Perfect. Addison-Wesley, 1987

[Deelstra03] S. Deelstra: Evolution of Variability in Software Product Families. Master Thesis, University of Groningen, 2003

[Deelstra++05] S. Deelstra, M. Sinnema, J. Bosch: Product derivation in software product families: a case study. Journal of Systems and Software 74(2): 173-194, 2005

[Deelstra+08] S.K. Deelstra, M. Sinnema: Managing the Complexity of Variability in Software Product Families. PhD Thesis, University of Groningen, 2008

[Demeyer++02] S. Demeyer, S. Ducasse, O. Nierstrasz: Object-Oriented Reengineering Patterns. Dpunkt Verlag, 2002

[Dijkstra68] E.W. Dijkstra: Complexity controlled by hierarchical ordering of function and variability. In [Naur+69]: 181-185

[Dijkstra69] E.W. Dijkstra: Notes on Structured Programming. In O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare: Structured Programming. Academic Press Inc., 1972

[Ducasse++99] S. Ducasse, M. Rieger, S. Demeyer: A Language-Independent Approach for Detecting Duplicated Code. ICSM-15: 109-118, 1999

[Duret++01] A. Duret-Lutz, T. Geraud, A. Demaille: Design Patterns for Generic Programming in C++. COOTS-6: 189-202, 2001

[Duszynski++09] S. Duszynski, J. Knodel, M. Naab: Analyzing Variability in Software Variants with the Variant Comparison Technique, IESE Report 005.09/E, 2009

[Duvall++07] P.M. Duvall, S. Matyas, A. Glover: Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley, 2007

[Eaddy++08] M. Eaddy, T. Zimmermann, K.D. Sherwood, V. Garg, G.C. Murphy, N. Nagappan, A.V. Aho: Do Crosscutting Concerns Cause Defects? IEEE Trans. Software Engineering, 34(4): 497-515, 2008

[Ebraert09] P. Ebraert: A bottom-up approach to program variation. PhD Thesis, University of Brussels, 2009

[Eden+06] A.H. Eden, T. Mens: Measuring Software Flexibility. IEE Software 153 (3), 2006

[Ekoko+10] E. Duala-Ekoko, M.P. Robillard: Clone Region Descriptors: Representing and Tracking Duplication in Source Code. ACM Trans. Softw. Eng. Methodol. 20(1): 1-31. 2010

[Elrad++01] T. Elrad, R.E. Filman, A. Bader (Guest Eds.): Aspect-Oriented Programming. Communications of the ACM 44(10): 28-97, October 2001

[Elsner++10] C. Elsner, G. Botterweck, D. Lohmann, W. Schröder-Preikschat: Variability in Time – Product Line Variability and Evolution Revisited. VaMoS-4: 131-137, 2010

[Endres+03] A. Endres, H.D. Rombach: A Handbook of Software and Systems Engineering. Empirical Observations, Laws and Theories. Addison-Wesley, 2003

[Estublier++10] J. Estublier, I.A. Dieng, T. Leveque: Software Product Line Evolution: The Selecta System. PLEASE'10: 32-39, 2010

[Fenton96] N.E. Fenton: Software Metrics. A Rigorous and Practical Approach (2nd ed.). International Thomson Computer Press, 1996

[Filman+00] R.E. Filman, D.P. Friedman: Aspect-Oriented Programming is Quantification and Obliviousness. OOPSLA'00 Workshop on Advanced Separation of Concerns, 2000

[Filman++05] R.E. Filman, T. Elrad, S. Clarke, M. Askit: Aspect-Oriented Software Development. Addison-Wesley, 2005

[Fowler99] M. Fowler: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999

[Gabriel96] R.P. Gabriel: Patterns of Software. Tales From The Software Community. Oxford University Press, 1996

[Gamma++95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

[Ganesan++06] D. Ganesan, D. Muthig, K. Yoshimura: Predicting Return-on-Investment for Product Line Generations. SPLC-10: 13-24, 2006

[Garlan++95] D. Garlan, R. Allen, J. Ockerbloom: Architectural Mismatch or Why it's hard to build systems out of existing parts. ICSE-17: 179-185, 1995

[Geppert++04] B. Geppert, C. Krueger, J. Jenny Li (Eds.): SPLiT-1, 2004

[Geppert++05] B. Geppert, C. Krueger, T. Trew (Eds.): SPLiT-2, Avaya Labs Technical Report ALR-2005-17, 2005

[Godfrey+08] M.W. Godfrey, D.M. German: The Past, Present and Future of Software Evolution. ICSM-24: 129-138, 2008

[Godfrey+10] M. Godfrey, C. Kapser: Copy-Paste as a Principled Engineering Tool. In G. Wilson, A. Oram (Eds.): Making Software: What Really Works, and Why We Believe It. O'Reilly, 2010

[Gomaa04] H. Gomaa: Designing Software Product Lines with UML. From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, 2004

[Greenwald+59] I.D. Greenwald, M. Kane: The Share 709 System: Programming and Modification. Journal of the ACM 6(2): 128-133, 1959

[Gurp++01] G. van Gurp, J. Bosch, M. Svahnberg: On the Notion of Variability in Software Product Lines. Working IEEE/IFIP Conference on Software Architecture, 2001

[Hall+00] G.A. Hall, J.C. Munson: Software Evolution: Code Delta and Code Churn. Journal of Systems and Software 54(2): 111-118, 2000

[Hannemann+02] J. Hannemann, G. Kiczales: Design Pattern Implementation in Java and AspectJ. OOPSLA'02: 161-173, 2002

[Hanssen+08] G.K. Hanssen, T.E. Faegri: Process fusion: An industrial case study on agile software product line engineering. Journal of Systems and Software 81(6): 843-854, 2008

[Heering03] J. Heering: Quantification of Structural Information: On a Question Raised by Brooks. ACM Sigsoft Software Engineering Notes 28(3): 6, 2003

[Hoffman+01] D.M. Hoffman, D.M. Weiss (Eds.): Software Fundamentals: Collected Papers by David L. Parnas. Addison-Wesley, 2001

[Hordijk++09] W. Hordijk, M.L. Ponisio, R. Wieringa: Harmfulness of Code Duplication - A Structured Review of the Evidence. EASE-13, 2009

[Hotta++10] K. Hotta, Y. Sano, Y. Higo, S. Kusumoto: Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software. IWPSE-EVOL'10: 73-82, 2010

[Hotz++06] L. Hotz, K. Wolter, T. Krebs, S. Deelstra, M. Sinnema, J. Nijhuis, J. MacGregor: Configuration in Industrial Product Families: The ConIPF Methodology. Akademische Verlagsgesellschaft, 2006

[Hunt06] J.M. Hunt: Managing Product Line Asset Bases. PhD Thesis, Clemson University, 2006

[IEEE610] IEEE Standard Glossary of Software Engineering Terminology. IEEE-Std 610.12-1990

[IEEE1517] IEEE Standard for Information Technology – System and Software Lifecycle Processes – Reuse Processes. IEEE-Std 1517-2010

[IEEE12207] IEEE Standard for Systems and Software Engineering- Software Lifecycle Processes. IEEE Std 12207-2008

[ISO24765] ISO Standard for Systems and Software Engineering - Vocabulary. ISO/IEC/IEEE 24765:2009

[Jackson01] M. Jackson: Problem Frames: Analysing and Structuring Software Development Problems. Addison-Wesley, 2001

[Jacobson++97] I. Jacobson, M. Griss, P. Jonsson: Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley, 1997

[Jalote05] P. Jalote: An Integrated Approach to Software Engineering (3rd ed.), Springer-Verlag, 2005

[Jarzabek+10] S. Jarzabek, Y. Xue: Are Clones Harmful for Maintenance? IWSC'10: 73-74, 2010

[John++07] I. John, J. Lee, D. Muthig: Separation of Variability Dimension and Development Dimension. VaMoS-1: 45-49, 2007

[John10] I. John: Pattern-based Documentation Analysis for Software Product Lines. PhD Theses in Experimental Software Engineering Vol. 30, Fraunhofer-Verlag, 2010

[Jürgens++09] E. Jürgens, F. Deissenboeck, B. Hummel, S. Wagner: Do code clones matter? ICSE-31: 485-495, 2009

[Jürgens+10] E. Jürgens, F. Deissenböck: How Much is a Clone? SCM-4, 2010

[Käkölä+06] T. Käkölä, J. C. Dueñas: Software Product Lines. Research Issues in Engineering and Management. Springer-Verlag, 2006

[Kästner10] C. Kästner: Virtual Separation of Concerns: Toward Preprocessors 2.0, PhD Thesis, University of Magdeburg, 2010

[Kang++90] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson: Feature-Oriented Domain Analysis (FODA) Feasibility Study. SEI Technical Report CMU/SEI-90-TR-21, Software Engineering Institute ,1990

[Kang+98] K.C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, M. Huh: FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5(1): 143-168, 1998

[Kang++10] K.C. Kang, V. Sugumaran, S. Park (Eds.): Applied Software Product Line Engineering. Auerbach Publications, 2010

[Kapser+06] C.J. Kapser, M.W. Godfrey: Supporting the analysis of clones in software systems: a case study. Journal of Software Maintenance and Evolution: Research and Practice 18: 61-82, 2006

[Kapser+08] C. Kapser, M.W. Godfrey: "Cloning Considered Harmful" Considered Harmful. Empirical Software Engineering 13(6): 645-692, 2008

[Kapser09] C.J. Kapser: Toward an Understanding of Software Code Cloning as a Development Practice. PhD Thesis, University of Waterloo, 2009

[Karlsson95] E.-A. Karlsson: Software Reuse. A Holistic Approach. Wiley, 1995

[Kelly06] D. Kelly: A Study of Design Characteristics in Evolving Software Using Stability as a Criterion. IEEE Trans. Software Engineering 32(5): 315-329, 2006

[Kerevinsky04] J. Kerevinsky: Refactoring to Patterns. Addison-Wesley, 2004

[Kiczales++97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin: Aspect-Oriented Programming. ECOOP'97: 220-242, Springer LNCS 1241, 1997

[Kiczales+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold: Getting Started with AspectJ. In [Elrad++01]: 59-65

[Kim++04] M. Kim, L. Bergman, T. Lau, D. Notkin: An Ethnographic Study of Copy and Paste Programming Practices in OOPL. ISESE'04: 83-92, 2004

[Kim++05a] S.D. Kim, J.S. Her, S.H. Chang: A theoretical foundation of variability in component-based development. Information and Software Technology 47: 663-673, 2004

[Kim++05b] M. Kim, V. Sazawal, D. Notkin, G.C. Murphy: An Empirical Study of Code Clone Genealogies. ESEC/SIGSOFT FSE 2005: 187-196, 2005

[Kim08] M. Kim: Analyszing and Inferring the Structure of Code Changes. PhD Thesis, University of Washington, 2008

[Kirby++10] J. Kirby, D.M. Weiss, R.R. Lutz: Evidence-based Software Production. FSE-FoSER 2010: 191-194, 2010

[Knauber04] P. Knauber: Managing the Evolution of Software Product Lines. Poster Summary from ICSR-8, 2004

[Knauber++06] P. Knauber, C. Krueger, T. Trew (Eds.): SPLiT-3, 2006

[Knauber++08] P. Knauber, A. Metzger, J. McGregor (Eds): SPLiT-5, 2008

[Knodel10] J. Knodel: Sustainable Structures in Software Implementations by Live Compliance Checking. PhD Theses in Experimental Software Engineering, Fraunhofer-Verlag, 2010 (to appear)

[Kokol++99] P. Kokol, V. Podgorelec, M. Zorman, M. Pidgin: Alpha – A Generic Software Complexity Metric. ESCOM-10: 397-405, 1999

[Kolb++06] R. Kolb, D. Muthig, T. Patzke, K. Yamauchi: Refactoring a legacy component for reuse in a software product line: a case study. Journal of Software Maintenance and Evolution: Research and Practice 18(2):109–132, 2006

[Kolb+10] R. Kolb, F. van der Linden: The Need for Speed / Why Do We Do Product Lines (Point - Counterpoint Column). IEEE Software 27(3): 56-59, 2010

[Kolmogorov68] A.L. Kolmogorov: Logical Basis for Information Theory and Probability Theory. IEEE Trans. Information Theory 14(5): 662-664,1968

[Krinke07] J. Krinke: A Study of Consistent and Inconsistent Changes to Code Clones. WCRE'07: 170-178, 2007

[Krinke08] J. Krinke: Is Cloned Code more stable than Non-Cloned Code? SCAM-8: 57-66, 2008

[Kruchten95] P. Kruchten: Architectural Blueprints—The "4+1" View Model of Software Architecture. IEEE Software 12(6): 42-50, 1995

[Krueger92] C.W. Krueger: Software Reuse. ACM Computing Surveys 24(2): 131-183, 1992

[Krueger02a] C.W. Krueger: Easing the Transition to Software Mass Customization. PFE-4: 282-293, Springer LNCS 2290, 2002

[Krueger02b] C.W. Krueger: Variation Management for Software Production Lines. SPLC-2: 37-48, Springer LNCS 2379, 2002

[Krueger04] C.W. Krueger: Towards a Taxonomy for Software Product Lines. PFE-5: 323-331, Springer LNCS 3014, 2004

[Krueger07] C.W. Krueger: The 3-Tiered Methodology: Pragmatic Insights from New Generation Software Product Lines. SPLC-11: 97-106, 2007

[Krueger10] C.W. Krueger: New Methods behind a New Generation of Software Product Line Successes. In [Kang++10]: 39-60

[Labrosse02] J.J. Labrosse: MicroC/OS-II: The Real-Time Kernel. Butterworth Heinemann, 2002

[Laddad06] R. Laddad: Aspect Oriented Refactoring. Addison-Wesley, 2008

[Laird+06] L.M. Laird, C.M. Brennan: Software Measurement and Estimation: A Practical Approach. Wiley & Sons, 2006

[Lapham06] M.A. Lapham: Sustaining Software-Intensive Systems. Technical Note CMU/SEI-2006-TN-007, Software Engineering Institute, 2006

[Lee+04] J. Lee, K.C. Kang: Feature Binding Analysis for Product Line Component Development. PFE-5: 250-260, Springer LNCS 3014, 2004

[Lee+10] J. Lee, G. Kotonya: Combining Service Orientation with Product Line Engineering. IEEE Software 27(3): 35-41, 2010

[Lehman80] M.M. Lehman: Programs, Life Cycles, and Laws of Software Evolution. Proceedings of the IEEE 68(5): 1060-1076, 1980

[Lehman02] M.M. Lehman: Software Evolution. In [Marciniak02]: 1507-1513

[Lehman+06a] M.M. Lehmann, J.C. Fernandez-Ramil: Rules and Tools for Software Evolution Planning and Management. In [Madhavij++06]: 539-563

[Lehman+06b] M.M. Lehman, J.C. Fernandez-Ramil: Software Evolution. In [Madhavij++06]: 7-40

[Levenshtein66] V.I. Levenshtein: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10(8): 707-710, 1966

[Linden++07] F. van der Linden, K. Schmid, E. Rommes: Software Product Lines in Action. The Best Industrial Practice in Product Line Engineering. Springer-Verlag, 2007

[Liskov94] B.H. Liskov, J.M. Wing: A Behavioral Notion of Subtyping. ACM Trans. Programming Languages and Systems 16(6): 1811-1841,1994

[Lösch+07] F. Loesch, E. Ploedereder: Restructuring Variability in Software Product Lines using Concept Analysis of Product Configurations. CSMR-11: 159-170, 2007

[Lohmann++06] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, W. Schröder-Preikschat: A Quantitative Analysis of Aspects in the eCos Kernel. EuroSys'06: 191-204, 2006

[Lopez+08] R.E. Lopez-Herrejon, S. Trujillo: How complex is my Product Line? The case for Variation Point Metrics. VaMoS-2: 97-100, 2008

[Loughran+04] N. Loughran, A. Rashid: Framed Aspects: Supporting Variability and Configurability for AOP. ICSR-8: 127-140, 2004

[Lozano++07] A. Lozano, M. Wermelinger, B. Nuseibeh: Evaluating the harmfulness of cloning: a change based experiment. MSR'07 at ICSE-29, 2007

[Lozano++08] A. Lozano, M. Wermelinger: Assessing the effect of clones on changeability. ICSM-24: 227-236, 2008

[Lozano09] A. Lozano Roduigez: Assessing the effect of source code characteristics on changeability. PhD Thesis, The Open University, United Kingdom, 2009

[Lutz++10] R. Lutz, D.M. Weiss, S. Krishnan, J. Yang: Software Product Line Engineering for Long-Lived, Sustainable Systems. SPLC-14: 430-434, Springer LNCS 6287, 2010

[Madhavij++06] N.M. Madhavij, J. Fernandez-Ramil, E.E. Perry (Eds.): Software Evolution and Feedback - Theory and Practice. Wiley & Sons, 2006

[Mäntylä09] M. Mäntylä: Software Evolvability – Empirically Discovered Evolvability Issues and Human Evaluations. PhD Thesis, Helsinki University of Technology, 2009

[Marciniak02] J.J. Marciniak (Ed.): Encyclopedia of Software Engineering (2nd Ed.), John Wiley & Sons, 2002

[Martin02] R.C. Martin: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, 2002

[McCabe76] T.J. McCabe: A Complexity Measure. IEEE Trans. Software Engineering 2(4): 308-320, December 1976

[McGregor03] J.D. McGregor: The Evolution of Product Line Assets. Technical Report CMU/SEI-2003-TR005, Software Engineering Institute, 2003

[McGregor++10] J.D. McGregor, D. Muthig, K. Yoshimura, P. Jansen: Successful Software Product Line Practices. IEEE Software 27(3): 16-21, 2010

[McIlroy68] M.D. McIlroy: Mass Produced Software Components. In [Naur+69]: 88-98

[Mens+07] K. Mens, T. Tourwe: Evolutionary Problems in Aspect-Oriented Software Development. 3rd ECRIM Workshop on Software Evolution, 2007

[Mens+08] T. Mens, S. Demeyer (Eds.): Software Evolution. Springer-Verlag, 2008

[Meszaros07] G. Meszaros: xUnit Patterns: Refactoring Test Code. Addison-Wesley, 2007

[Metzger++07] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, G. Saval: Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. RE-15, 2007

[Meyer97] B. Meyer: Object-Oriented Software Construction. Prentice Hall, 1997

[Meyer+97] M. Meyer and A. Lehnerd; The Power of Product Platforms, Free Press, 1997

[Mohan++10] K. Mohan, B. Ramesh, V. Sugumaran: Integrating Software Product Line Engineering and Agile Development. IEEE Software 27(3): 48-55, 2010

[Monteiro05] M.J.T.P. Monteiro: Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts. PhD Thesis, University do Minho, Portugal, 2005

[Munson96] J.C. Munson: Software Faults, Software Failures and Software Reliability Modeling. Information and Software Technology, 1996

[Murphy++10] E. Murphy-Hill, G.C. Murphy, W.G. Griswold: Understanding Context: Creating a Lasting Impact in Experimental Software Engineering Research. FSE-FoSER'10: 255-258, 2010

[Muthig02] D. Muthig: A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines. PhD Theses in Experimental Software Engineering Vol. 11, Fraunhofer-Verlag, 2002

[Muthig++02] D. Muthig, M. Anastasopoulos, R. Laqua, S. Kettemann, T. Patzke: Technology Dimensions of Product Line Implementation Approaches. Fraunhofer IESE Report 051.02/E, 2002

[Muthig+03] D. Muthig, T. Patzke: Generic Implementation of Product Line Components. NetObjectDays 2002: 313-329, 2003

[Muthig09] D. Muthig. Software Product Line Engineering for Embedded Systems. Postgraduate Distance Studies Textbook E-M.3, Fraunhofer IESE & Technical University of Kaiserslautern (DIST), 2009

[Myllymäki01] T. Myllymäki: Variability Management in Software Product Lines. PhD Thesis, University of Tampere University of Technology, 2001

[Nagrappan+05] N. Nagrappan, T. Ball: Use of Relative Code Churn Measures to Predict System Defect Density, ICSE-27: 284-292, 2005

[Naur+69] P. Naur, B. Randell (Eds.): Software Engineering: Report on a Conference sponsored by the NATO Science Committee. NATO, 1969

[Navarro01] G. Navarro: A Guided Tour to Approximate String Matching. ACM Computing Surveys 33(1): 31-88, 2001

[Neighbors80] J.M. Neighbors: Software Construction Using Components. PhD thesis, University of California, Irvine, 1980

[Neto++11] P.A. Neto, I. Machado, J.D. McGregor, E. Almeida, S.R. Meira: A systematic mapping study of software product lines testing. Information and Software Technology 53: 407-423, 2011

[Northrop++06] L. Northrop et al: Ultra-Large-Scale Systems: The Software Challenge of the Future. Software Engineering Institute, 2006

[Northrop+07] L.M. Northrop, P.C. Clements: A Framework for Software Product Line Practice, Version 5.0. Available at www.sei.cmu.edu/product-lines/framework.html (Retrieved August 2009)

[O'Connor++94] J. O'Connor, C. Mansour, J. Turner-Harris, G.H. Campbell: Reuse in Command-and-Control Systems. IEEE Software 11(5): 70-79, 1994

[Olbrich++10] S.M. Olbrich, D.S. Cruzes, D.I.K. Sjøberg: Are all Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of three Open Source Systems. ICSM-26, 2010

[Ommering04] R.C. van Ommering: Building Product Populations with Software Components. PhD Thesis, Groningen, 2004

[Opdyke92] W.F. Opdyke: Refactoring Object-Oriented Frameworks. PhD Thesis, University of Illinois, 1992

[Parnas72] D.L. Parnas: On the Criteria To Be Used in Decomposing Systems into Modules. In [Hoffman+01]: 145-155

[Parnas74] D.L. Parnas: On A Buzzword: Hierarchical Structure. In [Hoffman+01]: 161-170

[Parnas76] D.L. Parnas: On the Design and Development of Program Families. In [Hoffman+01]: 193-213

[Parnas78] D.L. Parnas: Some Software Engineering Principles. In [Hoffman+01]: 257-265

[Parnas79] D.L. Parnas: Designing Software for Ease of Extension and Contraction. In [Hoffman+01]: 269-290

[Parnas94] D.L. Parnas: Software Aging. In [Hoffman+01]: 551-567

[Parnas+95] D.L. Parnas, J. Madey: Functional Documents for Computer Systems. Science of Computer Programming 25(1): 41-61, 1995

[Parnas07] D.L. Parnas: Software Product-Lines: What To Do When Enumeration Won't Work. VaMoS-1: 7-14, 2007

[Parnas08] D.L. Parnas: Multi-Dimensional Software Families: Document Defined Partitions of a Set of Products. Keynote at SPLC-12, 2008

[Patzke+03] T. Patzke, D. Muthig: Product Line Implementation with Frame Technology: A Case Study. Fraunhofer IESE Report 018.03/E, 2003

[Patzke+04] T. Patzke, D. Muthig: Improving Variability Management in a Product Line of Embedded Systems – A Case Study from Industry. SET'04: 45-48, 2004

[Patzke07] T. Patzke: An Incremental Approach for Improving Variability Management in Embedded Systems Code. Fraunhofer IESE Report 045.07/E, 2007

[Patzke08] T. Patzke: A Method for Reducing Arbitrary Source Code Complexity in Reusable Embedded Systems Code. SVPP'08, 2008

[Patzke++08] T. Patzke, L. Vajda, A. Török: Evolving Heterogeneous Wireless Sensor Networks - An Assisted Living Case Study. RCEAS'07: 89-93, 2008

[Patzke10a] T. Patzke: The Impact of Variability Mechanisms on Sustainable Product Line Code Evolution. SE 2010: 189-200, GI-Edition Lecture Notes in Informatics (LNI) P-159, 2010

[Patzke10b] T. Patzke: Product Line Engineering Terminology: A Survey. Fraunhofer IESE Report (to appear)

[Pine93] B.J. Pine II: Mass Customization. The New Frontier in Business Competition. Harvard Business School Press, 1993

[Pohl++05] K. Pohl, G. Böckle, F. van der Linden: Software Product Line Engineering. Foundations, Principles, and Techniques. Springer-Verlag, 2005

[Pont01] M.J. Pont: Patterns for Time-Triggered Embedded Systems. Addison-Wesley, 2001

[Poulin96] J.S. Poulin: Measuring Software Reuse. Principles, Patterns, and Economic Models. Addison-Wesley, 1996

[Pree94] Meta Patterns—A Means For Capturing the Essentials of Reusable Object-Oriented Design. ECOOP'94, 1994

[Pressman10] R.S. Pressman: Software Engineering. A Practitioner's Approach (7th Ed.). McGraw-Hill, 2010

[Prieto94] R. Prieto-Diaz: The Disappearance of Software Reuse. ICSR-3: 225,1994

[Pussinen02] M. Pussinen: A survey on software product-line evolution. Institute of Software Systems, Tampere University of Technology, 2002

[Rahman++10] F. Rahman, C. Bird, P. Devanbu: Clones: What is that Smell? MSR-7: 72-81, 2010

[Rajapakse+07] D.C. Rajapakse, S. Jarzabek: Using Server Pages to Unify Clones in Web Applications: A Trade-off Analysis. ICSE-29, 2007

[Rajlich+00] V.T. Rajlich, K.H. Bennett: A Staged Model for the Software Life Cycle. IEEE Computer 33(7): 66-71, 2000

[Ramasubbo+10] N. Ramasubbu, R.K. Balan: Evolution of a Bluetooth Test Application Product Line: A Case Study. FSE-18: 107-116, 2010

[Ran99] A. Ran: Software Isn't Build from Lego Blocks. SSR'99: 164-169, 1999

[Rashid++10] A. Rashid, T. Cottenier, P. Greenwood, R. Chitchyan, R. Meunier, R. Coelho, M. Südholt, W. Joosen: Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe, IEEE Computer 43(2): 19-26, 2010

[Reiser08] M.-O. Reiser: Managing Complex Variability in Automotive Product Lines with Subscoping and Configuration Links. PhD Thesis, University of Berlin, 2008

[Riebisch03] M. Riebisch. Towards a More Precise Definition of Feature Models. Position Paper in: M. Riebisch, J. 0. Coplien, D. Streitferdt (Eds.): Modelling Variability for Object-Oriented Product Lines, 2003

[Roberts99] D.B. Roberts: Practical Analysis for Refactoring. PhD Thesis, University of Illinois, 1999

[Robillard++10] M.P. Robillard, R. J. Walker, T. Zimmermann: Recommendation Systems for Software Engineering. IEEE Software 27(4): 80-86, 2010

[Roy++09] C.K. Roy, J.R. Cordy, R. Koschke: Comparison and evaluation of clone detection techniques and tools: A quantitative approach. Science of Computer Programming 74(7): 470-495, 2009

[Saha++10] R.K. Saha, M.Asaduzzaman, M.F. Zibran, C.K. Roy, K.A. Schneider: Evaluating Code Clone Genealogies at Release Level: An Empirical Study. SCAM-10: 87-96, 2010

[Sauer02] F. Sauer: Metadata-Driven Multi-Artifact Code Generation Using Frame Oriented Programming. OOPSLA'02 Workshop on Generative Techniques in the Context of MDA, 2002

[Savolainen+01] J. Savolainen, J. Kuusela: Volatility Analysis Framework for Product Lines. SSR'01: 133-141, 2001

[Savolainen++09] J. Savolainen, J. Bosch, J. Kuusela, T. Männistö: Default Values for Improved Product Line Management. SPLC-13: 51-60, 2009

[Schäfer++09] I. Schäfer, A. Worret, A. Poetzsch-Heffter: A Model-Based Framework for Automated Product Derivation. MAPLE'09, 2009

[Schmid03] K. Schmid: Planning Software Reuse - A Disciplined Scoping Approach for Software Product Lines. PhD Theses in Experimental Software Engineering Vol. 12, Fraunhofer-Verlag, 2003

[Schmid+04] K. Schmid, I. John: A customizable approach to full lifecycle variability management. Science of Computer Programming 53(3): 259-284, 2004

[Selim++10] G.M.K. Selim, L. Barbour, W. Shang, B. Adams, A.E. Hassan, Y. Zou: Studying the Impact of Clones on Software Defects. WCRE-17: 13-21, 2010

[Shaw+97] M. Shaw, P. Clements: A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. COMPSAC-21: 6-13, 1997

[Shaw05] Mary Shaw (Ed.). Software Engineering for the 21st Century: A basis for rethinking the curriculum, Technical Report CMU-ISRI-05-108, Carnegie Mellon University, 2005

[Sinnema++04] M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch: COVAMOF: A Framework for Modeling Variability in Software Product Families. SPLC-3: 197-213, Springer LNCS 3154, 2004

[Smaragdakis+02] Y. Smaragdakis, D. Batory: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Trans. Software Engineering and Methodology 11(2): 215-255, 2002

[Sneed++10] H.M. Sneed, R. Seidl, M. Baumgartner: Software in Zahlen. Carl Hanser Verlag, 2010

[Solingen++02] R. van Solingen, V.R. Basili, G. Caldiera, H.D. Rombach: Goal Question Metric (GQM) Approach. In [Marciniak02]: 578-583

[Sommerville04] I. Sommerville: Software Engineering (7th Ed.). Pearson Education, 2004

[Steimann06] F. Steimann: The Paradoxical Success of Aspect-Oriented Programming. OOPSLA'06: 481-497, 2006

[Svahnberg+99] M. Svahnberg, J. Bosch: Evolution in Software Product Lines. Journal of Software Maintenance 11(6): 391-422, 1999

[Svahnberg03] M. Svahnberg: Supporting Software Architecture Evolution. PhD Thesis, Blekinge Institute of Technology, 2003

[Svahnberg++05] M. Svahnberg, J: van Gurp, J. Bosch: A Taxonomy of Variability Realization Techniques. Software - Practice and Experience 35: 705-754, Wiley & Sons, 2005

[Synthesis93] Reuse-Driven Software Process Guidebook. Software Productivity Consortium, 1993

[Szyperski98] C. Szyperski: Component Software. Addison-Wesley, 1998

[Tarr++99] P. Tarr, H. Ossher, W. Harrison: N Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE-21: 107-119, 1999

[Tegarden++92] D.P. Tegarden, S.D. Sheetz, D.E. Monarchi: Effectiveness of Traditional Software Metrics for Object-Oriented Systems. System Sciences: 359-368, 1992

[Thörn10] C. Thörn: Current state and potential of variability management practices in software-intensive SMEs: Results from a regional industrial survey. Information and Software Technology 52: 411-421, 2010

[Thummalapenta++10] S. Thummalapenta, L. Cerulo, L. Aversano, M. Di Penta: An empirical study on the maintenance of source code clones. Empirical Software Engineering 15: 1-34, 2010

[Tourwe03] T. Tourwe, J. Brichau, K. Gybels. On the Existence of the AOSD Evolution Paradox. AOSD'03 SPLAT Workshop, 2003

[Trujillo07] S. Trujillo Gonzalez: Feature Oriented Model Driven Product Lines. PhD Thesis, Universidad del País Vasco, 2007

[Vlissides98] J. Vlissides: Pattern Hatching: Design Patterns Applied. Addison-Wesley, 1998

[Wake04] W.C. Wake: Refactoring Workbook. Addison-Wesley, 2004

[Weiss+99] D.M. Weiss, C.T.R. Lai: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley, 1999

[Whithey96] J. Withey: Investment Analysis of Software Assets for Product Lines. Technical Report CMU/SEI-96-TR-010, Software Engineering Institute, 1996

[Wilkes++51] M.V. Wilkes, D.J. Wheeler, S. Gill: The Preparation of Programs for an Electronic Digital Computer. Addison-Wesley, 1951

[Wirfs09] R. J. Wirfs-Brock: Creating Sustainable Designs. IEEE Software 26(3): 5-7, 2009

[Wirth01] N. Wirth: Embedded Systems and Real-Time Programming. EMSOFT 2001: 486-492, Springer LNCS 221, 2001

[Wohlin00] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslen: Experimentation in Software Engineering. An Introduction. Kluwer Academic Publishers, 2000

[Woolf98] B. Woolf: The Null Object Pattern. In R.C. Martin, D. Riehle, F. Buschmann (Eds.): Pattern Language of Program Design 3. Addison-Wesley, 1998

[Wong++01] T.W. Wong, S. Jarzabek, S.M. Swe, R. Shen, H. Zhang: XML Implementation of Frame Processor. SSR'01: 164-172, 2001

[Yin03] R.K. Yin: Case Study Research. Design and Methods (3rd ed.). Sage Publications, 2003

[Zave99] P. Zave, "FAQ Sheet on Feature Interactions", www.research.att.com/~pamela/faq.html, 1999 (retrieved August 2009)

[Zhang++01] H. Zhang, S. Jarzabek, S. Myat Swe: XVCL Approach to Separating Concerns in Product Line Assets. GCSE-3: 36-47, Springer LNCS 2186, 2001

[Zhang++03] H. Zhang., S. Jarzabek: An XVCL-Based Approach to Software Product Line Development. SEKE-15, 2003

[Zhang++08] T. Zhang, L. Deng, J. Wu, Q. Zhou, C. Ma: Some Metrics for Accessing Quality of Product Line Architecture. International Conference on Computer Science and Software Engineering: 500-503, 2008

[Zhang+10] H. Zhang, S. Jarzabek: A Hybrid Approach to Feature-Oriented Programming in XVCL. SPLC-14: 440-445, Springer LNCS 6287, 2010

[Zhang++11] M. Zhang, T. Hall, N. Baddoo: Code Bad Smells: a review of current knowledge. Journal of Software Maintenance and Evolution: Research and Practice 23: 179-202, 2011

Appendix A Glossary

Abstraction (17): A succinct description which suppresses details that are unimportant for the purpose at hand, while emphasizing properties that are important to this purpose.

Activity (10): "A set of cohesive tasks of a process" [IEEE12207].

Application Engineering (AE, 61): The process of PLE in which a particular PL member is produced by consuming elements from the PLI. The goal is to efficiently produce all required PL members. Contrast with: Family Engineering.

Artifact (44): The output of an engineering process. An artifact may be a requirements specification, an architecture, a source code module, a test case, or any other useful process result.

Binding (13): "The act of assigning a value to a variable in a module" [Bassett97].

Binding Time (14): The moment when binding happens.

Commonality (45): Prescribes what needs to be identical among a set of PL members. The goal is to facilitate rapid, cost-effective development. Contrast with: Variability.

Complexity (43): The absence of simplicity in an artifact or process. This defect makes the artifact more difficult to develop than necessary. It arises when elements have been realized in engineering that are not immediately required by stakeholders. Complexity reduction aims at making the artifact easier to understand and change. See also: Variability Complexity.

Composition (9): a) The activity of a user who combines executable modules without modifying them internally, or b) the result of the activity in a). Contrast with: Configuration.

Configuration (28): The activity of a reuser adapting constructible modules to modify them internally via manual techniques or automated mechanisms. Contrast with: Composition.

Constructible Module (26): A module that is interpreted by a construction interpreter. Contrast with: Executable Module.

Construction (27): The interpretation of a constructible module by a construction interpreter. Contrast with: Execution.

Construction Interpreter (25): An interpreter whose input consists of constructible modules. Contrast with: Execution Interpreter.

Construction Time (31): The binding time during which a construction interpreter interprets a constructible module, emitting executable modules. Contrast with: Execution Time.

Context (20): "The setting in which software engineering is practiced" [Murphy++10].

Core Asset (56): A reusable artifact that is developed for reuse in more than one PL member. Core assets explicitly capture the PL's commonality and predicted variability. The goal is to support the efficient production of all PL members. Contrast with: Variability Asset.

Default (55): A variant that is automatically chosen if no other variant is selected in its place. The goal is to simplify production, decreasing the number of configuration options.

Development (12): All the activities associated with a software product, from conception through client negotiation, design, realization, validation, operation, and evolution.

Domain Engineering: The process (11) of software product line engineering in which the commonality and the variability of the PL are defined and realized [Pohl++05].

Encapsulation (18): Hides the elements of an executable abstraction that its users do not need to know.

Engineering (41): A process "governing the total technical and managerial effort required to transform a set of [stakeholders'] needs into a solution and to support that solution throughout its life" [ISO24765]. The goal is to support "practical, cost-effective solutions to problems [in system development] in a timely and predictable manner" [Shaw05]. Syn.: Systems Engineering. See also: Software Engineering.

Evolution (66): The sub-activity of development during which changes occur in the problem space over an extended period of time which lead to changes in real-world artifacts in the solution space. The goals are to explicitly address long-term issues and to "aim at long-term progressive quality change trends, possibly tolerating short-term degradation" [Lehmann+06]. Syn.: Sustainment, Variability in Time. See also: Variability Evolution.

Evolution Step (68): A smaller sequence of changes during the larger evolution of a system. The goal is to break down the evolution activity into more manageable sub-activities that keep the evolving artifact maximally stable.

Executable Module (4): a) A binary module that can run on computer hardware, or b) a module that can be compiled and linked to run on computer hardware. Contrast with: Constructible Module.

Execution (3): The interpretation a) of a binary module by computer hardware, or b) of a module "by a compiler-linker-computer trio, or by

any functionally equivalent interpreter" [Bassett97]. Contrast with: Construction.

Execution Interpreter (5): An interpreter whose input consists of executable modules. Contrast with: Construction Interpreter.

Execution Time (15): The binding time during which an execution interpreter interprets an executable module, emitting machine code. Contrast with: Construction Time, Runtime.

Family Engineering (FE, 60): The process of PLE in which PL assets are developed for a given scope. Domain Engineering consists of family engineering and scoping. The goal is to reduce PL complexity by developing just the required PL assets. Contrast with: Application Engineering. Note: Family engineering and product line engineering are different concepts.

Feature (63): In PLE, a feature is an end-user visible functional or non-functional characteristic of a PL member. The goals are a) to communicate variable characteristics between stakeholders and software engineers, and b) to document variability in the form of abstract requirements.

Interpreter (2): "An agent capable of interacting with a module" [Bassett97].

Mass Customization (29): Focuses on the means of efficiently producing and evolving multiple similar products, "exploiting what they have in common and managing what varies among them" [Krueger02a].

Method: Guidance and criteria that prescribe a systematic, repeatable technique for performing an activity [Synthesis93].

Module (1): An artifact containing a group of symbols that can be consistently referenced as a unit.

Needs (24): "The considerations that customers identify as desired capabilities, perceived weaknesses, or desired improvements in a system of interest" [Campbell07]. See also: Requirements.

Problem (36): "The gap between a system as it exists and the system as would better enable a customer in achieving objectives" [Campbell07]. Contrast with: Solution.

Problem Space (50): Early activities in PLE where PL members are specified. Contrast with: Solution Space.

Process (11): Defines, in a repeatable and consistent way, how "development is - or should be - performed, i.e. the specific activities that need to be conducted" [Linden++07].

Product Line (PL, 23): A set of similar systems that "share a common, managed set of features satisfying the needs of a particular market

segment […], and that are developed from a common set of core assets in a prescribed way" [Clements+01].

Product Line Asset (59): An artifact that consists of a set of core assets and the corresponding variability assets. The goal is to capture the output of FE in an integrated form.

Product Line Engineering (PLE, 49): "An engineering approach that subsumes all processes […] supporting the development […] of a PL" [Muthig09].

Product Line Infrastructure (PLI, 62): A repository of all PL assets of an organization, including common methods and tools for developing these assets in FE, and for reusing them in AE. The main goals are to capture all types of elements relevant in the PLE life cycle, and to provide an explicit interface between FE and AE. Syn.: Core Asset Base.

Product Line Member (48): "A deployed software-intensive system or software" [Northrop+07] "that has been defined [by stakeholders] to be built [from a PLI]" [Metzger++07].

Production (30): "The process used for building all products in a PL" [Northrop+07]. Syn.: Instantiation, Product Derivation.

Production Plan (38): A guide to show how products in the PL will be composed and constructed from modules.

Realization (34): a) The lower, more detailed level of an abstraction, or b) the process of developing the artifact in a). Contrast with: Specification.

Requirements (35): "The criteria, consistent with needs and constraints, that determine whether a product is acceptable as a solution to a problem" [Campbell07].

Reusability (19): The capability of a module to be adapted in order to become usable in a specific context. Reusability depends on usability, variability and adaptability. See also: Usability.

Reuse (21): "The process of adapting" a module "in order to make it usable" (adapted from [Bassett97]). See also: Use.

Reuse Hierarchy (40): In the reuse hierarchy which is formed when a constructible module A reuses a constructible module B, there exist reuse levels with the following properties: 1) Level 0 is the set of all constructible modules A such that there does not exist a constructible module B for which R(A,B); 2) Level n is the set of all constructible modules A such that a) there exists a constructible module B at level n-1 such that R(A,B), and b) if R(A,C) then C is at level n-1 or lower (adapted from [Parnas74]).

Reuse Relation (39): We can say of two modules A and B that A reuses B if correct construction of B may be necessary for A to complete the

production process described in its specification (adapted from [Parnas79]). See also: Use Relation.

Reuser (22): An agent capable of reusing a module. See also: User.

Runtime (16): The binding time during which machine code runs on computer hardware. Contrast with: Execution Time.

(Development) Scenario: Describes a certain arrangement of software development activities that lead to the development of a piece of software (or even a whole PL) [Schmid03].

Scoping: The process of determining the boundaries of the PLE activity [Linden++07]. Syn.: Product Management [Pohl++05].

Software Engineering: An engineering approach to the practical, cost-effective multi-person development of multi-version software systems, with a primary focus on processes (adapted from [Jalote05, Shaw05, Parnas78]).

Solution (37): "A means of transforming a system to resolve an identified problem" [Campbell07]. Contrast with: Problem.

Solution Space (51): Later activities in PLE where PL members are realized. Contrast with: Problem Space.

Specification (33): Serves to state requirements, and represents the higher of the two levels of abstraction. Contrast with: Realization.

Stakeholder (42): Someone who has a vested interest in a system and who is entitled to contribute to requirements.

Traceability (58): The ability to establish a relationship between two artifacts developed in different engineering phases. The goal in PLE is to efficiently identify dependencies between core assets that exist due to variability.

Usability (7): The capability of an executable module to be used again. Usability depends on functionality, efficiency and ease-of-change. See also: Reusability.

Use (6): The process of reapplying an executable module in unmodified form. Syn.: Unmodified Reuse. See also: Reuse.

Use Relation (32): We can say of two modules A and B that A uses B if correct execution of B may be necessary for A to complete the task described in its specification (adapted from [Parnas79]). See also: Reuse Relation.

User (8): An agent capable of using an executable module. See also: Reuser.

Variability (46): Prescribes what may differ among a set of PL members. "The goal […] is to maximize ROI [for developing products] over a specified period of time or number of products" [Bachmann+05]. The

major types of variability are optional and alternative variability. Contrast with: Commonality.

Variability Asset (57): An artifact, such as a Decision Model, a Variability Diagram, or a Product Model, that captures the relationships, constraints and resolutions of variability in core assets in an integrated form. The goal is to facilitate traceability of variability throughout the engineering life cycle. Contrast with: Core Asset.

Variability Complexity (65): The absence of simplicity in a PLI or PLE process. This defect makes the PLI more difficult to evolve than necessary. It arises when variability-related elements have been realized in FE that are not immediately required by AE. PL complexity reduction aims at making the PLI easier to evolve, especially the variants within the core assets. See also: Complexity.

Variability Evolution (67): The sub-activity of PLE during which changes occur in the problem space over an extended period of time, which lead to changes in solution-space artifacts of the PLI. The goals are to explicitly address long-term issues, such as unpredicted changes in the variability of the PL. See also: Evolution.

Variability Management: Encompasses the activities of explicitly representing variability in software artifacts throughout the lifecycle, managing dependences among different variabilities, and supporting the instantiation of the variabilities [Schmid+04]. Syn.: Variation Management [Krueger02b].

Variability Mechanism (64): A particular way of intentionally realizing variability in core assets. The goal is to balance reuse effort and evolution effort by efficiently organizing common elements and variants, as appropriate in the particular context of PLE.

Variability Refactoring (69): A specific FE activity by which a PLI is changed in order to evolve or reuse it in a more cost-effective way.

Variant (53): A realization of variation within PL assets, at a particular VP. A variant consists of one or more variant elements. The goal is to realize how PL members differ from each other.

Variant Element (54): A cohesive part of a variant.

Variation (47): A particular instance of variability. The goal is to define how PL members have to differ conceptually from each other.

Variation Point (VP, 52): A particular realization of variability within PL assets. The main goal is to highlight where variability occurs within the realized commonality, making the realized variations easy to see and control.

Appendix B Scripts

The following listings show the executable code of the Python scripts used in the case study. The frame processor scripts are accompanied by unit tests (not shown), and the *lineio* scripts are shared by the frame processor and the metrics scripts. The metrics scripts have been used to measure edit and compression distances, code delta and code churn.

B.1 **Frame Processor** (version 1.8.3)

```
#!/usr/bin/env python
'''
    fp - a variability management tool based on frame technology
    Copyright (C) 2002-2009 Thomas Patzke (thomas.patzke@web.de)

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
'''

__author__ ='Thomas Patzke'
__version__='1.8.3'

import sys,os
from os.path import dirname,basename
from lineio import readlines,writelines
from frame import *

frameStore={}
adaptPairs=[]

def createFrame(fn):
  if not fn in frameStore:
    frameStore[fn]=Frame(readlines(fn))
  return frameStore[fn]

def getAdaptees(fn):
  ret=[]
  f=createFrame(fn)
  for el in getChildren(f):
    ret.append((fn,el))
  return ret
def makeAdaptPairs(a):
```

```
    for el in a:
45    adaptPairs.append(el)
      d=getAdaptees(el[1])
      makeAdaptPairs(d)

  def buildFrames():
50  for el in adaptPairs:
      f0=createFrame(el[0]).getFramePiece(el[1])
      f1=createFrame(el[1])
      f0.adapt(f1)
    processed=[]
55  for el in adaptPairs:
      if not el[1] in processed:
        processed.append(el[1])
        f=createFrame(el[1])
        f.doChange()
60
  def createOutput():
    for f in frameStore.values():
      o=getOutput(f)
      f={}
65    for el in o:
        if hasKeyword(el,'OUTFILE'):
          p=getParameter(el)
          f[p]=[]
        elif not hasKeyword(el,'VP') and not hasKeyword(el,'END'):
70        f[p].append(el)
      for el in f:
        print('=> '+el+' ('+str(len(f[el]))+' l.)')
        writelines(el,f[el])

75 def run(f,fn='#'):
    frameStore[fn]=Frame(f)
    a=getAdaptees(fn)
    makeAdaptPairs(a)
    buildFrames()
80  createOutput()

  usage='Frame Processor '+__version__+'\n'+\
  '''Usage: fp [options] spc

85 Argument:
  spc                 Specification frame

  Option:
  -h                  Print this help
90
  Commands:
  OUTFILE <file>      Create output in <file>
  ADAPT <adaptee>     Adapt the frame <adaptee>
  VP <vp>             Define a variation point <vp>
95 END                 End of a variation point (mandatory)
  INSERT <vp>         Override the text of variation point(s) <vp>
  INSERT_BEFORE <vp>  Insert before the variation point
  INSERT_AFTER <vp>   Insert after the variation point
  '''
100
  def main():
    if len(sys.argv)==1 or sys.argv[1]=='-h':
      print(usage)
      sys.exit(-1)
105 p=dirname(sys.argv[1])
```

```
    if p!='':
      os.chdir(p)
    frameStore.clear()
    spc=[]
    for el in sys.argv[1:]:
      b=basename(el)
      spc.append('ADAPT '+b)
    run(spc)

if __name__=='__main__':
    main()
```

Listing 10: Frame processor driver: fp.py

```
import sys
from frameparser import *

def getAdaptText(f,a):
  ret=[]
  fnd=0
  for el in f:
    if hasAdaptee(el,a):
      fnd=1
      continue
    if fnd:
      if hasKeyword(el,'ADAPT') or hasKeyword(el,'OUTFILE'):
        return ret
      else:
        ret.append(el)
  return ret

def getChildren(f):
  ret=[]
  for el in f:
    if hasKeyword(el,'ADAPT'):
      c=getParameter2(el)
      if not c in ret:
        ret.append(c)
  return ret

def getModifyText(f,i):
  ret=[]
  for j in range(i+1,len(f)):
    if hasCommand(f[j]):
      return ret
    ret.append(f[j])
  return ret

def getOutput(f):
  ret=[]
  doOutput=0
  for el in f:
    if hasKeyword(el,'OUTFILE'):
      doOutput=1
    if hasKeyword(el,'ADAPT'):
      doOutput=0
    if doOutput:
      ret.append(el)
  return ret
```

```
def appendUnique(lst,d):
  if not d in lst:
    lst.append(d)
def extendUnique(lst,l):
  for el in l:
    appendUnique(lst,el)

def getVpEndPos(f,i):
  pos=i
  indent=0
  while pos<len(f):
    if hasKeyword(f[pos],'VP'):
      indent=indent+1
    if hasKeyword(f[pos],'END'):
      indent=indent-1
      if indent==0:
        return pos
    pos=pos+1
  print("error: VP '"+getParameter(f[i])+"' without END")
  sys.exit(-1)

class Frame(list):
  def __init__(self,d=[]):
    super(Frame,self).__init__(d)
    self.commands=[]
  def adapt(self,f):
    for i in range(len(self)):
      if hasKeyword(self[i],'INSERT_BEFORE') or hasKeyword(self[i],
         'INSERT_AFTER') or hasKeyword(self[i],'INSERT'):
        modifier=(self[i],getModifyText(self,i))
        appendUnique(f.commands,modifier)
    extendUnique(f.commands,self.commands)

  def doChange(self):
    for i in range(len(self.commands)):
      # this iteration must come first,
      #   because the commands must only be traversed once!
      vp=getParameter(self.commands[i][0])
      j=0
      while j<len(self):
        if matchesVp(self[j],vp):
          m=self.commands[i][1]
          if hasKeyword(self.commands[i][0],'INSERT_BEFORE'):
            self[j:j]=m
            j=j+len(m)
          if hasKeyword(self.commands[i][0],'INSERT_AFTER'):
            p=getVpEndPos(self,j)
            self[p+1:p+1]=m
            j=j+p+len(m)
          if hasKeyword(self.commands[i][0],'INSERT'):
            p=getVpEndPos(self,j)
            self[j+1:p]=m
            j=j+p+len(m)
        j=j+1

  def getFramePiece(self,a):
    ret=Frame(getAdaptText(self,a))
    ret.commands=self.commands
    return ret
```

Listing 11: Logic for processing a single frame: frame.py

```
import os

def hasKeyword(str,kw):
  lst=str.split()
  if lst!=[]:
    return lst[0]==kw

commands=('INSERT_BEFORE','INSERT_AFTER','ADAPT','OUTFILE','INSERT')
def hasCommand(str):
  lst=str.split()
  if len(lst)>0:
    return lst[0] in commands

def getParameter(str):
  pos=str.find(' ')
  if pos!=-1:
    return str[pos+1:].rstrip()
def getParameter2(str):
  c=getParameter(str)
  if c in os.environ:
    return os.environ[c]
  return c

def matchesVp(str,vp):
  if hasKeyword(str,'VP'):
    return getParameter(str)==vp
def hasAdaptee(str,ad):
  if hasKeyword(str,'ADAPT'):
    return getParameter2(str)==ad
```

Listing 12: Logic for parsing a single line: frameparser.py

```
from os import sep

def readlines(fn):
  f=open(fn)
  ret=f.readlines()
  f.close()
  for pos in range(len(ret)):
    if ret[pos].endswith('\n'):
      if sep=='/': # on Unix systems, handle Dos leaves correctly
        if len(ret[pos])>1 and ret[pos][-2]=='\r':
          ret[pos]=ret[pos][:-2]+ret[pos][-1]
      ret[pos]=ret[pos][:-1]
  return ret
def writelines(fn,data):
  f=open(fn,'w')
  # use tmp because data has to remain const!
  tmp=[]
  for el in data:
    if el.endswith('\n'):
      tmp.append(el)
    else:
      tmp.append(el+'\n')
  f.writelines(tmp)
  f.close()
```

Listing 13: Logic for input and output of text lines: lineio.py

B.2 **Measurement Scripts** (version 0.1.7)

```
#!/usr/bin/env python

import sys,zlib,math
from optparse import OptionParser
from os.path import basename
from lineio import readlines

def readlines_noindent(f):
  d=f.readlines()
  ret=[]
  for el in d:
    el2=el.strip()
    ret.append(el2)
  return ret
def readData(modules,chars):
  ret=[]
  for m in modules:
    f=open(m)
    if chars==1:
      d=f.read()
    else:
      d=readlines_noindent(f)
    f.close()
    ret.append(d)
  return ret

def ncd(s1,s2):
  sz1=len(zlib.compress(s1))
  sz2=len(zlib.compress(s2))
  c_all=len(zlib.compress(s1+s2))
  c_min=min((sz1,sz2))
  c_max=max((sz1,sz2))
  return float(c_all-c_min)/c_max
def ncdx(s1,s2):
  if s1==s2:
    return 0.0
  ret=(ncd(s1,s2)+ncd(s2,s1))/2
  if ret>1:
    return 1.0
  return ret

def levenshtein(s,t):
  m=len(s)
  n=len(t)
  d=[list(range(n+1))]
  for i in range(1,m+1):
    d=d+[[i]]
  for i in range(m):
    for j in range(n):
      rm=d[i][j+1]+1
      add=d[i+1][j]+1
      c=0
      if s[i]!=t[j]:
        c=1
      chg=d[i][j]+c
      val=min(rm,add,chg)
      d[i+1].append(val)
  return d[m][n]
def levenshtein2(s,t):
```

```
  m=len(s)
  n=len(t)
  d=[[]]
  for i in range(n+1):
    d[0].append((i,0,0,0)) # dist,remove,add,change
  for i in range(1,m+1):
    d=d+[[(i,0,0,0)]]
  for i in range(m):
    for j in range(n):
      rm=d[i][j+1][0]+1
      add=d[i+1][j][0]+1
      c=0
      if s[i]!=t[j]:
        c=1
      chg=d[i][j][0]+c
      if rm<=add:
        if rm<=chg:
          d[i+1].append((rm,d[i][j+1][1]+1,d[i][j+1][2],d[i][j+1][3]))
        else:
          d[i+1].append((chg,d[i][j][1],d[i][j][2],d[i][j][3]+c))
      elif chg<=add:
        d[i+1].append((chg,d[i][j][1],d[i][j][2],d[i][j][3]+c))
      else:
        d[i+1].append((add,d[i+1][j][1],d[i+1][j][2]+1,d[i+1][j][3]))
  return d[m][n]

def tri(m):
  ret=[]
  dim=int(math.sqrt(len(m)))
  for i in range(len(m)):
    # for dimension n, element x*(n+1) is dropped (main diagonal)
    if i%(dim+1)!=0:
      ret.append(m[i])
  return ret
def avg(v):
  sum=0.0
  for el in v:
    sum=sum+el
  return sum/len(v)
def stdev(v):
  a=avg(v)
  sum=0.0
  for el in v:
    sum=sum+(el-a)*(el-a)
  x=sum/len(v)
  return math.sqrt(x)

def dist(modules,type='compression',agg='none'):
  if type.startswith('edit'):
    chars=0
  else:
    chars=1
  d=readData(modules,chars)

  v=[]
  for el in d:
    for el2 in d:
      if type=='compression':
        val=ncdx(el,el2)
      elif type=='edit':
        val=levenshtein(el,el2)
      elif type=='edit2':
```

```
          val=levenshtein2(el,el2)
        elif type=='ncd':
          val=ncd(el,el2)
130     v.append(val)

    if agg=='none':
      return v
    t=tri(v)
135 if agg=='avg':
      return avg(t)
    elif agg=='stdev':
      return stdev(t)
    elif agg=='min':
140   return min(t)
    elif agg=='max':
      return max(t)

  if __name__=='__main__':
145 usage='%prog [options] module1...'
    desc='Calculate module distance statistics.'
    parser=OptionParser(usage,description=desc)
    parser.set_defaults(agg='none',
                        type='compression',
150                     quiet=False,
                        file='',
                        list=False)
    parser.add_option('-a',metavar='MODE',dest='agg',
                      help='aggregation MODE: none*,avg,stdev,min or max')
155 parser.add_option('-f',dest='file',
                      help='input FILE')
    parser.add_option('-t',metavar='TYPE',dest='type',
                      help='distance TYPE: compression* , edit or ncd')
    parser.add_option('-q',action='store_true',dest='quiet',
160                   help='quiet (only display numbers)')
    parser.add_option('-l',action='store_true',dest='list',
                      help='output as list')
    opts,args=parser.parse_args()

165 if opts.agg not in ['avg','stdev','min','max','none']:
      parser.error('-a option needs argument none, avg, stdev, min or max')
    if opts.type not in ['compression','edit','edit2','ncd']:
      parser.error('-t option needs argument compression, edit or ncd')
    if opts.file!='':
170   args=readlines(opts.file)
    else:
      if len(args)==0:
        parser.error('at least 1 argument required')
      if len(args)==1:
175     args.append(basename(args[0]))

    v=dist(args,type=opts.type,agg=opts.agg)
    if opts.agg!='none':
      print v,
180   if not opts.quiet:
        for el in args:
          print el,
      print
    else:
185   dim=len(args)
      if opts.list:
        for i in range(dim):
          for j in range(0,i):
```

```
            print args[i]+' '+args[j],
190         print v[i*dim+j]
      else:
        for i in range(dim):
          for j in range(dim):
            print v[i*dim+j],
195       if not opts.quiet:
            print args[i],
          print
```

Listing 14: Calculation of edit and compression distance: dist.py

```
  1 #!/usr/bin/env python

    import os,sys
    from os.path import isfile
  5 from lineio import readlines

    def readlines_noindent(f):
      d=f.readlines()
      ret=[]
 10   for el in d:
        el2=el.strip()
        ret.append(el2)
      return ret

 15 def readData(modules):
      ret=[]
      for m in modules:
        f=open(m)
        d=readlines_noindent(f)
 20     f.close()
        ret.append(len(d))
      return ret

    def delta(modules):
 25   d=readData(modules)
      return d[0]-d[1]

    if __name__=='__main__':
      p=sys.argv[1]
 30   if isfile(p):
        print(delta(sys.argv[1:]))
        sys.exit(0)
      f=[]
      for el in os.walk(p):
 35     f=el[2]
      p2='.'
      if len(sys.argv)>2:
        p2=sys.argv[2]
      f2=[]
 40   for el in os.walk(p2):
        f2=el[2]

      ret=0
      for el in f:
 45     if el=='Makefile':
          continue
        if el not in f2:
          fi=open(p+os.sep+el)
          d=fi.readlines()
 50       ret=ret+len(d)
```

```
      fi.close()
    for el in f2:
      if el=='Makefile':
        continue
55    if el in f:
        pn=[p+os.sep+el,p2+os.sep+el]
        ret=ret+delta(pn)
      else:
        fi=open(p2+os.sep+el)
65      d=fi.readlines()
        ret=ret-len(d)
        fi.close()
    print(ret)
```

Listing 15: Calculation of code delta: delta.py

```
 1  #!/usr/bin/env python

    import os,sys
    from os.path import isfile
 5  from lineio import readlines

    def readlines_noindent(f):
      d=f.readlines()
      ret=[]
10    for el in d:
        el2=el.strip()
        ret.append(el2)
      return ret

15  def readData(modules):
      ret=[]
      for m in modules:
        f=open(m)
        d=readlines_noindent(f)
20      f.close()
        ret.append(d)
      return ret

    def levenshtein(s,t):
25    m=len(s)
      n=len(t)
      d=[list(range(n+1))]
      for i in range(1,m+1):
        d=d+[[i]]
30    for i in range(m):
        for j in range(n):
          rm=d[i][j+1]+1
          add=d[i+1][j]+1
          c=0
35        if s[i]!=t[j]:
            c=1
          chg=d[i][j]+c
          val=min(rm,add,chg)
          d[i+1].append(val)
40    return d[m][n]

    def churn(modules):
      d=readData(modules)
      return levenshtein(d[0],d[1])
45
    if __name__=='__main__':
```

```
  p=sys.argv[1]
  if isfile(p):
    print(churn(sys.argv[1:]))
    sys.exit(0)
  f=[]
  for el in os.walk(p):
    f=el[2]
  p2='.'
  if len(sys.argv)>2:
    p2=sys.argv[2]
  f2=[]
  for el in os.walk(p2):
    f2=el[2]

  ret=0
  for el in f:
    if el=='Makefile':
      continue
    if el not in f2:
      fi=open(p+os.sep+el)
      d=fi.readlines()
      ret=ret+len(d)
      fi.close()
  for el in f2:
    if el=='Makefile':
      continue
    if el in f:
      pn=[p+os.sep+el,p2+os.sep+el]
      ret=ret+churn(pn)
    else:
      fi=open(p2+os.sep+el)
      d=fi.readlines()
      ret=ret+len(d)
      fi.close()
  print(ret)
```

Listing 16: Calculation of code churn: churn.py

Appendix C Code Excerpts from the Case Study

The following listings show representative C code, Makefile and pseudocode excerpts from different evolution stages of the sensor node product line, using different variability mechanisms (Figure 53), followed by the realized HAL interfaces and construction test output.

Listings 17-25 show that the code which realizes functionality is consistent, to which degree the realizations differ for the respective variability mechanisms, and how the code is build. Section 6.4 refers to some of these listings.

```
#include “hal.h”

bool event_happened=false;
int32_t event_time=0;
int16_t tick=0;
int16_t tilt_count=0;

void main() {
  init();
  init_x_position();
  while(true) {
    if(period_elapsed) {
      period_elapsed=false;
      update_x_position();
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        if(event_happened) { // a tilt has started
          event_time=the_clock; // start one-shot timer
        }
        else { // a tilt has ended
          // has the device been tilted for a period between 1 and 5s?
          if(the_clock-event_time>0 && the_clock-event_time<=5) {
            toggle_led_2();
            tilt_count++;
          }
        }
      }
      tick=tick+1;
      if(tick%5==0) {
        tick=0;
        sprintf(send_buffer,"drink=%d",tilt_count*25);
        send();
      }
    }
  }
}
```

```
CC=   sdcc -mpic16 -p18f6720
CFLAGS=-I ../../hal
LDFLAGS=-Wl,-ms../../teco/system/"app\#229.lkr" ../../hal/hal.lib

all:
```

```
      export BEHAVIOR=tilt_detector; make clean product
      export BEHAVIOR=noise_detector; make clean product
      export BEHAVIOR=drop_detector; make clean product
      export BEHAVIOR=movement_detector; make clean product
      export BEHAVIOR=tilt_detector_time; make clean product
      export BEHAVIOR=noise_detector_time; make clean product
      export BEHAVIOR=drop_detector_time; make clean product
      export BEHAVIOR=movement_detector_time; make clean product
      export BEHAVIOR=raw_detector; make clean product
      export BEHAVIOR=raw_detector_time; make clean product
      export BEHAVIOR=tilt_detector_voltage; make clean product
      export BEHAVIOR=noise_detector_voltage; make clean product
      export BEHAVIOR=drop_detector_voltage; make clean product
      export BEHAVIOR=movement_detector_voltage; make clean product
      export BEHAVIOR=tilt_detector_voltage_time; make clean product
      export BEHAVIOR=noise_detector_voltage_time; make clean product
      export BEHAVIOR=drop_detector_voltage_time; make clean product
      export BEHAVIOR=movement_detector_voltage_time; make clean product
      export BEHAVIOR=raw_detector_voltage; make clean product
      export BEHAVIOR=raw_detector_voltage_time; make clean product
      export BEHAVIOR=tilt_detector_sync; make clean product
      export BEHAVIOR=noise_detector_sync; make clean product
      export BEHAVIOR=drop_detector_sync; make clean product
      export BEHAVIOR=movement_detector_sync; make clean product
      export BEHAVIOR=tilt_detector_sync_time; make clean product
      export BEHAVIOR=noise_detector_sync_time; make clean product
      export BEHAVIOR=drop_detector_sync_time; make clean product
      export BEHAVIOR=movement_detector_sync_time; make clean product
      export BEHAVIOR=raw_detector_sync; make clean product
      export BEHAVIOR=raw_detector_sync_time; make clean product
      export BEHAVIOR=tilt_detector_sync_voltage; make clean product
      export BEHAVIOR=noise_detector_sync_voltage; make clean product
      export BEHAVIOR=drop_detector_sync_voltage; make clean product
      export BEHAVIOR=movement_detector_sync_voltage; make clean product
      export BEHAVIOR=tilt_detector_sync_voltage_time; make clean product
      export BEHAVIOR=noise_detector_sync_voltage_time; make clean product
      export BEHAVIOR=drop_detector_sync_voltage_time; make clean product
      export BEHAVIOR=movement_detector_sync_voltage_time; \
             make clean product
      export BEHAVIOR=raw_detector_sync_voltage; make clean product
      export BEHAVIOR=raw_detector_sync_voltage_time; make clean product

product: $(BEHAVIOR).hex

$(BEHAVIOR).hex:
      $(CC) $(CFLAGS) $(LDFLAGS) $(BEHAVIOR).c

clean:
      rm -f *.hex *.cod *.map *.asm *.lst *.o
```

Listing 17: Original tilt detector from product line (a6): tilt_detector.c, Makefile

```
#include<string.h>
#include<stdlib.h>
#include "hal.h"

bool event_happened=false;
int32_t event_time=0;
int16_t tilt_count=0;
int16_t tick=0;
bool time_set=false;
int behavior=BEHAVIOR;
```

```
bool has_time=HAS_TIME_TX;
bool has_voltage_check=HAS_VOLTAGE_CHECK;
bool has_clock_sync=HAS_CLOCK_SYNC;

void main() {
  init();
  init_x_position();
  init_y_position();
  init_sound();
  init_light();
  init_temperature();
  while(true) {
    update_receive();
    if(period_elapsed) {
      period_elapsed=false;
      update_x_position();
      update_y_position();
      update_sound();
      update_light();
      update_temperature();
      update_voltage();
      if(behavior==0) {
        if((x_position>(-100+25) && !event_happened)
         ||(x_position<(-100-25) && event_happened)) {
          event_happened=x_position>-100;
          if(event_happened) { // a tilt has started
            event_time=the_clock; // start one-shot timer
          }
          else { // a tilt has ended
            // has the device been tilted for a period between 1 and 5s?
            if(the_clock-event_time>0 && the_clock-event_time<=5) {
              toggle_led_2();
              tilt_count++;
            }
          }
        }
        sprintf(send_buffer,"drink=%d",tilt_count*25);
        if(has_time) {
          if(event_happened) {
            strcat(send_buffer,",time=");
            strcat(send_buffer,timetoa(the_clock-event_time));
          }
        }
        send();
      }
      if(behavior==1) {
        if((x_position>(-100+25) && !event_happened)
         ||(x_position<(-100-25) && event_happened)) {
          event_happened=x_position>-100;
          // on change of tilt state, start a timer
          event_time=the_clock;
        }
        if(event_time>0 && the_clock-event_time>=1) { // tilted > 1s
          set_led_2(event_happened);
          event_time=0;
          sprintf(send_buffer,"dropped=%d",event_happened ?1 :0);
          if(has_time) {
            if(event_happened) {
              strcat(send_buffer,",time=");
              strcat(send_buffer,timetoa(the_clock-event_time));
            }
          }
```

```
          send();
        }
      }
      if(behavior==2) {
        if(sound>20) {
          event_time=the_clock; // start one-shot timer
          event_happened=true;
        }
        // forgetting
        if(the_clock-event_time>=10) {
          // forget when a presence was detected
          event_time=0;
          // forget about presence
          event_happened=false;
        }
        set_led_2(event_happened);
        sprintf(send_buffer,"presence=%d",event_happened ?1 :0);
        if(has_time) {
          if(event_happened) {
            strcat(send_buffer,",time=");
            strcat(send_buffer,timetoa(the_clock-event_time));
          }
        }
        send();
      }
      if(behavior==3) {
        update_movement();
        if(!event_happened && movement) {
          event_time=the_clock;
          event_happened=true;
        }
        set_led_2(event_happened);
        sprintf(send_buffer,"movement=%d",event_happened ?1 :0);
        if(has_time) {
          if(event_happened) {
            strcat(send_buffer,",time=");
            strcat(send_buffer,timetoa(the_clock-event_time));
          }
        }
        send();
      }
      if(behavior==4) {
        if(!movement) {
          update_movement();
        }
        sprintf(send_buffer,"x=%d,y=%d,mv=%d,snd=%d",
                            x_position,y_position,movement,sound);
        if(movement) {
          movement=false;
        }
        send();
        sprintf(send_buffer,"li=%d,tmp=%d,vlt=%d",
                            light,temperature,voltage);
        if(has_time) {
          strcat(send_buffer,",time=");
          strcat(send_buffer,timetoa(the_clock-event_time));
        }
        send();
      }
      if(has_voltage_check) {
        tick=tick+1;
        if(tick%60==0) {
```

```
          tick=0;
          if(voltage<1200) {
            set_led_1(true);
          }
        }
      }
      if(has_clock_sync) {
        if(has_received) {
          if(!time_set && strncmp(receive_buffer,"set time=",9)==0) {
            the_clock=atol(receive_buffer+9);
            time_set=true;
          }
        }
      }
    }
  }
}
```

```
CC=   sdcc -mpic16 -p18f6720
CFLAGS=-I ../../hal
LDFLAGS=-Wl,-ms../../teco/system/"app\#229.lkr" ../../hal/hal.lib

CFLAGS+=-DBEHAVIOR=$(BEHAVIOR)
CFLAGS+=-DHAS_TIME_TX=$(HAS_TIME_TX)
CFLAGS+=-DHAS_VOLTAGE_CHECK=$(HAS_VOLTAGE_CHECK)
CFLAGS+=-DHAS_CLOCK_SYNC=$(HAS_CLOCK_SYNC)

BEHAVIORS=0 1 2
BEHAVIORS+=3
BEHAVIORS+=4
all:
      for el in $(BEHAVIORS); do \
      for el2 in 0 1; do \
      for el3 in 0 1; do \
      for el4 in 0 1; do \
        export BEHAVIOR=$$el HAS_TIME_TX=$$el2 HAS_VOLTAGE_CHECK=$$el3
HAS_CLOCK_SYNC=$$el4; \
        make clean product; \
      done; \
      done; \
      done; \
      done;

product: main.hex

main.hex:
      $(CC) $(CFLAGS) $(LDFLAGS) main.c

clean:
      rm -f *.hex *.cod *.map *.asm *.lst *.o
```

Listing 18: Conditional execution code after 6th evolution step (b6): main.c, Makefile

```
#include<string.h>
#include<stdlib.h>
#include "hal.h"

bool event_happened=false;
int32_t event_time=0;
int16_t tilt_count=0;
int16_t tick=0;
bool time_set=false;

```

```
void main() {
  init();
  init_x_position();
  init_y_position();
  init_sound();
  init_light();
  init_temperature();
  while(true) {
    update_receive();
    if(period_elapsed) {
      period_elapsed=false;
      update_x_position();
      update_y_position();
      update_sound();
      update_light();
      update_temperature();
      update_voltage();
#if TILT_DETECTOR
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        if(event_happened) { // a tilt has started
          event_time=the_clock; // start one-shot timer
        }
        else { // a tilt has ended
          // has the device been tilted for a period between 1
          // and 5s?
          if(the_clock-event_time>0 && the_clock-event_time<=5) {
            toggle_led_2();
            tilt_count++;
          }
        }
      }
      sprintf(send_buffer,"drink=%d",tilt_count*25);
#elif DROP_DETECTOR
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        // on change of tilt state, start a timer
        event_time=the_clock;
      }
      if(event_time>0 && the_clock-event_time>=1) { // tilted longer
                                                    // than 1s
        set_led_2(event_happened);
        event_time=0;
        sprintf(send_buffer,"dropped=%d",event_happened ?1 :0);
#elif NOISE_DETECTOR
      if(sound>20) {
        event_time=the_clock; // start one-shot timer
        event_happened=true;
      }
      // forgetting
      if(the_clock-event_time>=10) {
        // forget when a presence was detected
        event_time=0;
        // forget about presence
        event_happened=false;
      }
      set_led_2(event_happened);
      sprintf(send_buffer,"presence=%d",event_happened ?1 :0);
#elif MOVEMENT_DETECTOR
      update_movement();
```

```
      if(!event_happened && movement) {
        event_time=the_clock;
        event_happened=true;
      }
      set_led_2(event_happened);
      sprintf(send_buffer,"movement=%d",event_happened ?1 :0);
#elif RAW_DETECTOR
      if(!movement) {
        update_movement();
      }
      sprintf(send_buffer,"x=%d,y=%d,mv=%d,snd=%d",
                          x_position,y_position,movement,sound);
      if(movement) {
        movement=false;
      }
      send();
      sprintf(send_buffer,"li=%d,tmp=%d,vlt=%d",
                          light,temperature,voltage);
#endif
#if HAS_TIME_TX
#if !RAW_DETECTOR
        if(event_happened)
#endif
        {
          strcat(send_buffer,",time=");
          strcat(send_buffer,timetoa(the_clock-event_time));
        }
#endif
      send();
#if DROP_DETECTOR
      }
#endif
#if HAS_CLOCK_SYNC
      if(has_received) {
        if(!time_set && strncmp(receive_buffer,"set time=",9)==0) {
          the_clock=atol(receive_buffer+9);
          time_set=true;
        }
      }
#endif
#if HAS_VOLTAGE_CHECK
      tick=tick+1;
      if(tick%60==0) {
        tick=0;
        if(voltage<1200) {
          set_led_1(true);
        }
      }
#endif
    }
  }
}
```

```
CC=   sdcc -mpic16 -p18f6720
CFLAGS=-I ../../hal
LDFLAGS=-Wl,-ms../../teco/system/"app\#229.lkr" ../../hal/hal.lib

CFLAGS+=-D$(BEHAVIOR)
CFLAGS+=-DHAS_TIME_TX=$(HAS_TIME_TX)
CFLAGS+=-DHAS_VOLTAGE_CHECK=$(HAS_VOLTAGE_CHECK)
CFLAGS+=-DHAS_CLOCK_SYNC=$(HAS_CLOCK_SYNC)
```

```
BEHAVIORS=TILT_DETECTOR NOISE_DETECTOR DROP_DETECTOR
BEHAVIORS+=MOVEMENT_DETECTOR
BEHAVIORS+=RAW_DETECTOR
all:
      for el in $(BEHAVIORS); do \
      for el2 in 0 1; do \
      for el3 in 0 1; do \
      for el4 in 0 1; do \
        export BEHAVIOR=$$el HAS_TIME_TX=$$el2
HAS_VOLTAGE_CHECK=$$el3 HAS_CLOCK_SYNC=$$el4; \
        make clean product; \
      done; \
      done; \
      done; \
      done;

product: main.hex

main.hex:
      $(CC) $(CFLAGS) $(LDFLAGS) main.c

clean:
      rm -f *.hex *.cod *.map *.asm *.lst *.o
```

Listing 19: Conditional compilation code after 6th evolution step (e6): main.c, Makefile

```
#include "hal.h"
#include "detectors.h"
#include "time_transmission.h"
#include "voltage_check.h"
#include "clock_sync.h"

void (*more_loop)()=0;
void (*more_loop_2)()=0;
void (*before_send)()=0;
void (*do_sync)()=0;
int behavior=BEHAVIOR;
bool has_time_tx=HAS_TIME_TX;
bool has_volt_chk=HAS_VOLT_CHK;
bool has_clock_sync=HAS_CLOCK_SYNC;

void main() {
  init();
  init_x_position();
  init_y_position();
  init_sound();
  init_light();
  init_temperature();
  if(behavior==0) {
    more_loop=&tilt_more_loop;
  }
  else if(behavior==1) {
    more_loop=&drop_more_loop;
  }
  else if(behavior==2) {
    more_loop=&noise_more_loop;
  }
  else if(behavior==3) {
    more_loop=&movement_more_loop;
  }
  else if(behavior==4) {
    more_loop=&raw_more_loop;
```

```
  }
  if(has_time_tx) {
    before_send=&transmit_time;
  }
  else {
    before_send=&no_transmit_time;
  }
  if(has_volt_chk) {
    more_loop_2=&check_voltage;
  }
  else {
    more_loop_2=&no_check_voltage;
  }
  if(has_clock_sync) {
    do_sync=&sync_clock;
  }
  else {
    do_sync=&no_sync_clock;
  }
  while(true) {
    update_receive();
    if(period_elapsed) {
      period_elapsed=false;
      update_x_position();
      update_y_position();
      update_sound();
      update_light();
      update_temperature();
      update_voltage();
      (*more_loop)();
      (*do_sync)();
      (*more_loop_2)();
    }
  }
}
```

```
#include "detectors.h"
#include "hal.h"

extern void (*before_send)();
extern bool event_happened;
extern int32_t event_time;

void movement_more_loop() {
  update_movement();
  if(!event_happened && movement) {
    event_time=the_clock;
    event_happened=true;
  }
  set_led_2(event_happened);
  sprintf(send_buffer,"movement=%d",event_happened ?1 :0);
  (*before_send)();
  send();
}
```

```
#include "detectors.h"
#include "hal.h"

extern void (*before_send)();
static int16_t tilt_count=0;
extern bool event_happened;
extern int32_t event_time;

void tilt_more_loop() {
```

```
  update_x_position();
  if((x_position>(-100+25) && !event_happened)
   ||(x_position<(-100-25) && event_happened)) {
    event_happened=x_position>-100;
    if(event_happened) { // a tilt has started
      event_time=the_clock; // start one-shot timer
    }
    else { // a tilt has ended
      // has the device been tilted for a period between 1 and 5s?
      if(the_clock-event_time>0 && the_clock-event_time<=5) {
        toggle_led_2();
        tilt_count++;
      }
    }
  }
  sprintf(send_buffer,"drink=%d",tilt_count*25);
  (*before_send)();
  send();
}
```

```
#include "detectors.h"
#include "hal.h"

extern void (*before_send)();
extern bool event_happened;
extern int32_t event_time;

void drop_more_loop() {
  if((x_position>(-100+25) && !event_happened)
   ||(x_position<(-100-25) && event_happened)) {
    event_happened=x_position>-100;
    // on change of tilt state, start a timer
    event_time=the_clock;
  }
  if(event_time>0 && the_clock-event_time>=1) { // tilted longer than 1s
    set_led_2(event_happened);
    event_time=0;
    sprintf(send_buffer,"dropped=%d",event_happened ?1 :0);
    (*before_send)();
    send();
  }
}
```

```
#include "detectors.h"
#include "hal.h"

extern void (*before_send)();
extern bool event_happened;
extern int32_t event_time;

void noise_more_loop() {
  update_sound();
  if(sound>20) {
    event_time=the_clock; // start one-shot timer
    event_happened=true;
  }
  // forgetting
  if(the_clock-event_time>=10) {
    // forget when noise was detected
    event_time=0;
    // forget about noise
    event_happened=false;
  }
  set_led_2(event_happened);
```

```
  sprintf(send_buffer,"presence=%d",event_happened ?1 :0);
  (*before_send)();
  send();
}
```

```
#include "detectors.h"
#include "hal.h"

extern void (*before_send)();
extern bool event_happened;

void raw_more_loop() {
  if(!movement) {
    update_movement();
  }
  sprintf(send_buffer,"x=%d,y=%d,mv=%d,snd=%d",
                      x_position,y_position,movement,sound);
  if(movement) {
    movement=false;
  }
  send();
  sprintf(send_buffer,"li=%d,tmp=%d,vlt=%d",light,temperature,voltage);
  event_happened=true; // just to use time_transmitter as-is
  (*before_send)();
  send();
}
```

```
#include "time_transmission.h"
#include<string.h>
#include "hal.h"

// these variables had to become non-static because they
// need to be accessed by all detectors & time_transmitter
bool event_happened=false;
int32_t event_time=0;

void transmit_time() {
  if(event_happened) {
    strcat(send_buffer,",time=");
    strcat(send_buffer,timetoa(the_clock-event_time));
  }
}

void no_transmit_time() {
}
```

```
#include "voltage_check.h"
#include "hal.h"

int16_t tick=0;

void check_voltage() {
  tick=tick+1;
  if(tick%60==0) {
    tick=0;
    if(voltage<1200) {
      set_led_1(true);
    }
  }
}

void no_check_voltage() {
}
```

```
#include "clock_sync.h"
#include<string.h>
```

```
#include<stdlib.h>
#include "hal.h"

bool time_set=false;

void sync_clock() {
  if(has_received) {
    if(!time_set && strncmp(receive_buffer,"set time=",9)==0) {
      the_clock=atol(receive_buffer+9);
      time_set=true;
    }
  }
}

void no_sync_clock() {
}
```

```
CC=   sdcc -mpic16 -p18f6720
CFLAGS=-I ../../hal
LDFLAGS=-Wl,-ms../../teco/system/"app\#229.lkr" ../../hal/hal.lib

OBJ=  tilt_detector.o noise_detector.o drop_detector.o
OBJ+= movement_detector.o
OBJ+= raw_detector.o
OBJ+= time_transmission.o
OBJ+= voltage_check.o
OBJ+= clock_sync.o
CFLAGS+=-DBEHAVIOR=$(BEHAVIOR)
CFLAGS+=-DHAS_TIME_TX=$(HAS_TIME_TX)
CFLAGS+=-DHAS_VOLT_CHK=$(HAS_VOLT_CHK)
CFLAGS+=-DHAS_CLOCK_SYNC=$(HAS_CLOCK_SYNC)
LDFLAGS+=libc18f.lib

BEHAVIORS=0 1 2
BEHAVIORS+=3
BEHAVIORS+=4
all:
      for el in $(BEHAVIORS); do \
      for el2 in 0 1; do \
      for el3 in 0 1; do \
      for el4 in 0 1; do \
        export BEHAVIOR=$$el HAS_TIME_TX=$$el2 HAS_VOLT_CHK=$$el3 \
               HAS_CLOCK_SYNC=$$el4; \
        make clean product; \
      done; \
      done; \
      done; \
      done;

product: main.hex

main.hex: $(OBJ)
      $(CC) $(CFLAGS) $(LDFLAGS) main.c $(OBJ)

clean:
      rm -f *.hex *.cod *.map *.asm *.lst *.o
```

Listing 20: Polymorphism code after 6th evolution step (c6): main.c, tilt/drop/noise/movement/raw_detector.c, time_transmission.c, voltage_check.c, clock_sync.c, Makefile

```
#include "hal.h"
#include "detector.h"
#include "voltage_check.h"
```

```
#include "clock_sync.h"

void main() {
  init();
  init_x_position();
  init_y_position();
  init_sound();
  init_light();
  init_temperature();
  while(true) {
    update_receive();
    if(period_elapsed) {
      period_elapsed=false;
      update_x_position();
      update_y_position();
      update_sound();
      update_light();
      update_temperature();
      update_voltage();
      more_loop();
      sync_clock();
      check_voltage();
    }
  }
}
```

```
#include "detector.h"
#include "hal.h"
#include "time_transmission.h"

bool event_happened=false;
int32_t event_time=0;
int16_t tilt_count=0;

void more_loop() {
  if((x_position>(-100+25) && !event_happened)
   ||(x_position<(-100-25) && event_happened)) {
    event_happened=x_position>-100;
    if(event_happened) { // a tilt has started
      event_time=the_clock; // start one-shot timer
    }
    else { // a tilt has ended
      // has the device been tilted for a period between 1 and 5s?
      if(the_clock-event_time>0 && the_clock-event_time<=5) {
        toggle_led_2();
        tilt_count++;
      }
    }
  }
  sprintf(send_buffer,"drink=%d",tilt_count*25);
  transmit_time();
  send();
}
```

```
#include "detector.h"
#include "hal.h"
#include "time_transmission.h"

bool event_happened=false;
int32_t event_time=0;

void more_loop() {
  if((x_position>(-100+25) && !event_happened)
   ||(x_position<(-100-25) && event_happened)) {
```

```
    event_happened=x_position>-100;
    // on change of tilt state, start a timer
    event_time=the_clock;
  }
  if(event_time>0 && the_clock-event_time>=1) { // tilted longer than 1s
    set_led_2(event_happened);
    event_time=0;
    sprintf(send_buffer,"dropped=%d",event_happened ?1 :0);
    transmit_time();
    send();
  }
}
```

```
#include "detector.h"
#include "hal.h"
#include "time_transmission.h"

bool event_happened=false;
int32_t event_time=0;

void more_loop() {
  if(sound>20) {
    event_time=the_clock; // start one-shot timer
    event_happened=true;
  }
  // forgetting
  if(the_clock-event_time>=10) {
    // forget when a presence was detected
    event_time=0;
    // forget about presence
    event_happened=false;
  }
  set_led_2(event_happened);
  sprintf(send_buffer,"presence=%d",event_happened ?1 :0);
  transmit_time();
  send();
}
```

```
#include "detector.h"
#include<string.h>
#include "hal.h"
#include "time_transmission.h"

bool event_happened=false;
int32_t event_time=0;

void more_loop() {
  update_movement();
  if(!event_happened && movement) {
    event_time=the_clock;
    event_happened=true;
  }
  set_led_2(event_happened);
  sprintf(send_buffer,"movement=%d",event_happened ?1 :0);
  transmit_time();
  send();
}
```

```
#include "detector.h"
#include "hal.h"
#include "time_transmission.h"

bool event_happened=false;
int32_t event_time=0;
```

```
void more_loop() {
  if(!movement) {
    update_movement();
  }
  sprintf(send_buffer,"x=%d,y=%d,mv=%d,snd=%d",
                      x_position,y_position,movement,sound);
  if(movement) {
    movement=false;
  }
  send();
  sprintf(send_buffer,"li=%d,tmp=%d,vlt=%d",light,temperature,voltage);
  event_happened=true;
  transmit_time();
  send();
}
```

```
#include "time_transmission.h"

void transmit_time() {
}
```

```
#include "time_transmission.h"
#include<string.h>
#include "hal.h"

extern bool event_happened;
extern int32_t event_time;

void transmit_time() {
  if(event_happened) {
    strcat(send_buffer,",time=");
    strcat(send_buffer,timetoa(the_clock-event_time));
  }
}
```

```
#include "voltage_check.h"

void check_voltage() {
}
```

```
#include "voltage_check.h"
#include "hal.h"

int16_t tick=0;

void check_voltage() {
  tick=tick+1;
  if(tick%60==0) {
    tick=0;
    if(voltage<1200) {
      set_led_1(true);
    }
  }
}
```

```
#include "clock_sync.h"

void sync_clock() {
}
```

```
#include "clock_sync.h"
#include<string.h>
#include<stdlib.h>
#include "hal.h"

bool time_set=false;

void sync_clock() {
```

```
  if(has_received) {
    if(!time_set && strncmp(receive_buffer,"set time=",9)==0) {
      the_clock=atol(receive_buffer+9);
      time_set=true;
    }
  }
}
```

```
CC=   sdcc -mpic16 -p18f6720
CFLAGS=-I ../../hal
LDFLAGS=-Wl,-ms../../teco/system/"app\#229.lkr" ../../hal/hal.lib

OBJ=  $(BEHAVIOR).o
OBJ+= $(FEATURE).o
OBJ+= $(FEATURE2).o
OBJ+= $(FEATURE3).o
LDFLAGS+=libc18f.lib

BEHAVIORS=tilt_detector noise_detector drop_detector
BEHAVIORS+=movement_detector
BEHAVIORS+=raw_detector
FEATURES=no_time_transmission time_transmission
FEATURES2=no_voltage_check voltage_check
FEATURES3=no_clock_sync clock_sync
all:
      for el in $(BEHAVIORS); do \
      for el2 in $(FEATURES); do \
      for el3 in $(FEATURES2); do \
      for el4 in $(FEATURES3); do \
        export BEHAVIOR=$$el FEATURE=$$el2 FEATURE2=$$el3 \
               FEATURE3=$$el4; \
        make clean product; \
      done; \
      done; \
      done; \
      done;

product: main.hex

main.hex: $(OBJ)
      $(CC) $(CFLAGS) $(LDFLAGS) main.c $(OBJ)

clean:
      rm -f *.hex *.cod *.map *.asm *.lst *.o
```

Listing 21: Module replacement code after 6th evolution step (d6): main.c, tilt/drop/noise/movement/raw_detector.c, no_/time_transmission.c,no_/ voltage_check.c, no_/clock_sync.c, Makefile

```
#include<string.h>
#include<stdlib.h>
#include "hal.h"

bool event_happened=false;
int32_t event_time=0;
int16_t tick=0;
int16_t tilt_count=0;
bool time_set=false;

void more_loop() {
  if((x_position>(-100+25) && !event_happened)
   ||(x_position<(-100-25) && event_happened)) {
    event_happened=x_position>-100;
```

```
    if(event_happened) { // a tilt has started
      event_time=the_clock; // start one-shot timer
    }
    else { // a tilt has ended
      // has the device been tilted for a period between 1 and 5s?
      if(the_clock-event_time>0 && the_clock-event_time<=5) {
        toggle_led_2();
        tilt_count++;
      }
    }
  }
  sprintf(send_buffer,"drink=%d",tilt_count*25);
  send();
}

void send2() {
  send(); // only for supporting the new raw_detector aspect
}

void main() {
  init();
  init_x_position();
  init_y_position();
  init_sound();
  init_light();
  init_temperature();
  while(true) {
    update_receive();
    if(period_elapsed) {
      period_elapsed=false;
      update_x_position();
      update_y_position();
      update_sound();
      update_light();
      update_temperature();
      update_voltage();
      more_loop();
    }
  }
}
```

```
around():execution(void more_loop()) {
  if((x_position>(-100+25) && !event_happened)
   ||(x_position<(-100-25) && event_happened)) {
    event_happened=x_position>-100;
    // on change of tilt state, start a timer
    event_time=the_clock;
  }
  if(event_time>0 && the_clock-event_time>=1) { // tilted longer than 1s
    set_led_2(event_happened);
    event_time=0;
    sprintf(send_buffer,"dropped=%d",event_happened ?1 :0);
    send();
  }
}

```

```
around():execution(void more_loop()) {
  if(sound>20) {
    event_time=the_clock; // start one-shot timer
    event_happened=true;
  }
  // forgetting
  if(the_clock-event_time>=10) {
```

```
    // forget when a presence was detected
    event_time=0;
    // forget about presence
    event_happened=false;
  }
  set_led_2(event_happened);
  sprintf(send_buffer,"presence=%d",event_happened ?1 :0);
  send();
}
```

```
around():execution(void more_loop()) {
  update_movement();
  if(!event_happened && movement) {
    event_time=the_clock;
    event_happened=true;
  }
  set_led_2(event_happened);
  sprintf(send_buffer,"movement=%d",event_happened ?1 :0);
  send();
}
```

```
around():execution(void more_loop)() {
  if(!movement) {
    update_movement();
  }
  sprintf(send_buffer,"x=%d,y=%d,mv=%d,snd=%d",
                      x_position,y_position,movement,sound);
  if(movement) {
    movement=false;
  }
  send2(); // this shall not be affected by the existing time aspect!
  sprintf(send_buffer,"li=%d,tmp=%d,vlt=%d",light,temperature,voltage);
  event_happened=true; // just to use transmit_time as-is
  send();
}

```

```
before():execution(void send()) {
  if(event_happened) {
    strcat(send_buffer,",time=");
    strcat(send_buffer,timetoa(the_clock-event_time));
  }
}
```

```
after():execution(void more_loop()) {
  tick=tick+1;
  if(tick%60==0) {
    tick=0;
    update_voltage();
    if(voltage<1200) {
      set_led_1(true);
    }
  }
}
```

```
after():execution(void more_loop()) {
  if(has_received) {
    if(!time_set && strncmp(receive_buffer,"set time=",9)==0) {
      the_clock=atol(receive_buffer+9);
      time_set=true;
    }
  }
}
```

Listing 22: AOP pseudocode after 6th evolution step (f6): main.c, drop/noise/movement/raw_detector.acc, time_transmission.acc, voltage_check.acc, clock_sync.acc

```
OUTFILE main.c
#include<string.h>
#include<stdlib.h>
#include "hal.h"

bool event_happened=false;
int32_t event_time=0;
int16_t tick=0;
int16_t tilt_count=0;
bool time_set=false;

void main() {
  init();
  init_x_position();
  init_y_position();
  init_sound();
  init_light();
  init_temperature();
  while(true) {
    update_receive();
    if(period_elapsed) {
      period_elapsed=false;
      update_x_position();
      update_y_position();
      update_sound();
      update_light();
      update_temperature();
      update_voltage();
VP more_loop
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        if(event_happened) { // a tilt has started
          event_time=the_clock; // start one-shot timer
        }
        else { // a tilt has ended
          // has the device been tilted for a period between 1 and 5s?
          if(the_clock-event_time>0 && the_clock-event_time<=5) {
            toggle_led_2();
            tilt_count++;
          }
        }
      }
      sprintf(send_buffer,"drink=%d",tilt_count*25);
END
VP more_loop2
      send();
END
    }
  }
}
```

```
ADAPT ADAPTEE
INSERT more_loop
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        // on change of tilt state, start a timer
        event_time=the_clock;
      }
      if(event_time>0 && the_clock-event_time>=1) { // tilted longer
                                                    // than 1s
        set_led_2(event_happened);
```

```
          event_time=0;
          sprintf(send_buffer,"dropped=%d",event_happened ?1 :0);
INSERT more_loop2
          send();
        }
ADAPT ADAPTEE
INSERT more_loop
        if(sound>20) {
          event_time=the_clock; // start one-shot timer
          event_happened=true;
        }
        // forgetting
        if(the_clock-event_time>=10) {
          // forget when a presence was detected
          event_time=0;
          // forget about presence
          event_happened=false;
        }
        set_led_2(event_happened);
        sprintf(send_buffer,"presence=%d",event_happened ?1 :0);
ADAPT ADAPTEE
INSERT more_loop
        update_movement();
        if(!event_happened && movement) {
          event_time=the_clock;
          event_happened=true;
        }
        set_led_2(event_happened);
        sprintf(send_buffer,"movement=%d",event_happened ?1 :0);
ADAPT ADAPTEE
INSERT more_query
INSERT more_loop
        if(!movement) {
          update_movement();
        }
        sprintf(send_buffer,"x=%d,y=%d,mv=%d,snd=%d",
                            x_position,y_position,movement,sound);
        if(movement) {
          movement=false;
        }
        send();
        sprintf(send_buffer,"li=%d,tmp=%d,vlt=%d",
                            light,temperature,voltage);
ADAPT main
INSERT_BEFORE more_loop2
VP more_query
        if(event_happened)
END
        {
          strcat(send_buffer,",time=");
          strcat(send_buffer,timetoa(the_clock-event_time));
        }
ADAPT ADAPTEE2
INSERT_AFTER more_loop2
        tick=tick+1;
        if(tick%60==0) {
          tick=0;
          if(voltage<1200) {
            set_led_1(true);
          }
        }
ADAPT ADAPTEE3
```

```
INSERT_AFTER more_loop2
      if(has_received) {
        if(!time_set && strncmp(receive_buffer,"set time=",9)==0) {
          the_clock=atol(receive_buffer+9);
          time_set=true;
        }
      }
```

```
CC=   sdcc -mpic16 -p18f6720
CFLAGS=-I ../../hal
LDFLAGS=-Wl,-ms../../teco/system/"app\#229.lkr" ../../hal/hal.lib

BEHAVIORS=drop_detector noise_detector
BEHAVIORS+=movement_detector
BEHAVIORS+=raw_detector
FEATURES=main time_transmission
all:
      make clean
      fp main
      make product
      make clean
      fp time_transmission
      make product
      make clean
      export ADAPTEE2=main;fp voltage_check
      make product
      make clean
      export ADAPTEE2=time_transmission;fp voltage_check
      make product
      make clean
      export ADAPTEE3=main;fp clock_sync
      make product
      make clean
      export ADAPTEE3=time_transmission;fp clock_sync
      make product
      make clean
      export ADAPTEE3=voltage_check ADAPTEE2=main;fp clock_sync
      make product
      make clean
      export ADAPTEE3=voltage_check ADAPTEE2=time_transmission;fp
clock_sync
      make product
      for el in $(BEHAVIORS); do \
      for el2 in $(FEATURES); do \
        make clean; \
        export ADAPTEE=$$el2; \
        fp $$el; \
        make product; \
        make clean; \
        export ADAPTEE=voltage_check ADAPTEE2=$$el2; \
        fp $$el; \
        make product; \
        make clean; \
        export ADAPTEE=clock_sync ADAPTEE3=$$el2; \
        fp $$el; \
        make product; \
        make clean; \
        export ADAPTEE=voltage_check ADAPTEE2=clock_sync ADAPTEE3=$$el2; \
        fp $$el; \
        make product; \
      done; \
      done;

```

```
product: main.hex

main.hex:
      $(CC) $(CFLAGS) $(LDFLAGS) main.c

clean:
      rm -f *.hex *.cod *.map *.asm *.lst *.o *.c
```

Listing 23: Frame technology code after 6th evolution step (g6): main, drop/noise/movement/raw_detector, time_transmission, voltage_check, clock_sync, Makefile

```
OUTFILE main.c
#include<string.h>
#include<stdlib.h>
#include "hal.h"

bool event_happened=false;
int32_t event_time=0;
int16_t tick=0;
int16_t tilt_count=0;
bool time_set=false;

void main() {
  init();
  init_x_position();
  init_y_position();
  init_sound();
  init_light();
  init_temperature();
  while(true) {
    update_receive();
    if(period_elapsed) {
      period_elapsed=false;
      update_x_position();
      update_y_position();
      update_sound();
      update_light();
      update_temperature();
      update_voltage();
VP more_loop
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        if(event_happened) { // a tilt has started
          event_time=the_clock; // start one-shot timer
        }
        else { // a tilt has ended
          // has the device been tilted for a period between 1
          // and 5s?
          if(the_clock-event_time>0 && the_clock-event_time<=5) {
            toggle_led_2();
            tilt_count++;
          }
        }
      }
      sprintf(send_buffer,"drink=%d",tilt_count*25);
END
#if HAS_TIME_TX
VP more_query
      if(event_happened)
END
      {
```

```
        strcat(send_buffer,",time=");
        strcat(send_buffer,timetoa(the_clock-event_time));
      }
#endif
VP more_loop2
      send();
END
#if HAS_CLOCK_SYNC
      if(has_received) {
        if(!time_set && strncmp(receive_buffer,"set time=",9)==0) {
          the_clock=atol(receive_buffer+9);
          time_set=true;
        }
      }
#endif
#if HAS_VOLTAGE_CHECK
      tick=tick+1;
      if(tick%60==0) {
        tick=0;
        if(voltage<1200) {
          set_led_1(true);
        }
      }
#endif
    }
  }
}
```

```
ADAPT main
INSERT more_loop
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        // on change of tilt state, start a timer
        event_time=the_clock;
      }
      if(event_time>0 && the_clock-event_time>=1) { // tilted
                                                   // longer than 1s
        set_led_2(event_happened);
        event_time=0;
        sprintf(send_buffer,"dropped=%d",event_happened ?1 :0);
INSERT more_loop2
        send();
      }
```

```
ADAPT main
INSERT more_loop
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        // on change of tilt state, start a timer
        event_time=the_clock;
      }
      if(event_time>0 && the_clock-event_time>=1) { // tilted
                                                   // longer than 1s
        set_led_2(event_happened);
        event_time=0;
        sprintf(send_buffer,"dropped=%d",event_happened ?1 :0);
INSERT more_loop2
        send();
      }
```

```
ADAPT main
INSERT more_loop
      if(sound>20) {
```

```
        event_time=the_clock; // start one-shot timer
        event_happened=true;
      }
      // forgetting
      if(the_clock-event_time>=10) {
        // forget when a presence was detected
        event_time=0;
        // forget about presence
        event_happened=false;
      }
      set_led_2(event_happened);
      sprintf(send_buffer,"presence=%d",event_happened ?1 :0);
```

```
ADAPT main
INSERT more_loop
      update_movement();
      if(!event_happened && movement) {
        event_time=the_clock;
        event_happened=true;
      }
      set_led_2(event_happened);
      sprintf(send_buffer,"movement=%d",event_happened ?1 :0);

```

```
ADAPT main
INSERT more_query
INSERT more_loop
      if(!movement) {
        update_movement();
      }
      sprintf(send_buffer,"x=%d,y=%d,mv=%d,snd=%d",
                          x_position,y_position,movement,sound);
      if(movement) {
        movement=false;
      }
      send();
      sprintf(send_buffer,"li=%d,tmp=%d,vlt=%d",
                          light,temperature,voltage);

```

```
CC=   sdcc -mpic16 -p18f6720
CFLAGS=-I ../../hal
LDFLAGS=-Wl,-ms../../teco/system/"app\#229.lkr" ../../hal/hal.lib

CFLAGS+=-DHAS_TIME_TX=$(HAS_TIME_TX)
CFLAGS+=-DHAS_VOLTAGE_CHECK=$(HAS_VOLTAGE_CHECK)
CFLAGS+=-DHAS_CLOCK_SYNC=$(HAS_CLOCK_SYNC)

BEHAVIORS=main noise_detector drop_detector
BEHAVIORS+=movement_detector
BEHAVIORS+=raw_detector
all:
      for el in $(BEHAVIORS); do \
      for el2 in 0 1; do \
      for el3 in 0 1; do \
      for el4 in 0 1; do \
        export HAS_TIME_TX=$$el2 HAS_VOLTAGE_CHECK=$$el3 \
               HAS_CLOCK_SYNC=$$el4; \
        make clean; \
        fp $$el; \
        make product; \
      done; \
      done; \
      done; \
      done;
```

```
product: main.hex

main.hex:
      $(CC) $(CFLAGS) $(LDFLAGS) main.c

clean:
      rm -f *.hex *.cod *.map *.asm *.lst *.o *.c
```

Listing 24: Ideal compilable code code after 6th evolution step (h6): main, drop/noise/movement/raw_detector, Makefile

```
#include<string.h>
#include<stdlib.h>
#include "hal.h"

bool event_happened=false;
int32_t event_time=0;
int16_t tick=0;
int16_t tilt_count=0;
bool time_set=false;

void main() {
  init();
  init_x_position();
  init_y_position();
  init_sound();
  init_light();
  init_temperature();
  while(true) {
    update_receive();
    if(period_elapsed) {
      period_elapsed=false;
      update_x_position();
      update_y_position();
      update_sound();
      update_light();
      update_temperature();
      update_voltage();
      if((x_position>(-100+25) && !event_happened)
       ||(x_position<(-100-25) && event_happened)) {
        event_happened=x_position>-100;
        if(event_happened) { // a tilt has started
          event_time=the_clock; // start one-shot timer
        }
        else { // a tilt has ended
          // has the device been tilted for a period between 1 and 5s?
          if(the_clock-event_time>0 && the_clock-event_time<=5) {
            toggle_led_2();
            tilt_count++;
          }
        }
      }
      sprintf(send_buffer,"drink=%d",tilt_count*25);
option time_transmission
      if(event_happened)
      {
        strcat(send_buffer,",time=");
        strcat(send_buffer,timetoa(the_clock-event_time));
      }
end
      send();
```

```
option sync_voltage
      if(has_received) {
        if(!time_set && strncmp(receive_buffer,"set time=",9)==0) {
          the_clock=atol(receive_buffer+9);
          time_set=true;
        }
      }
end
option voltage_check
      tick=tick+1;
      if(tick%60==0) {
        tick=0;
        if(voltage<1200) {
          set_led_1(true);
        }
      }
end
    }
  }
}
```

```
change lines 31 to 43 of main
        // on change of tilt state, start a timer
        event_time=the_clock;
      }
      if(event_time>0 && the_clock-event_time>=1) { // tilted longer
                                                    // than 1s
        set_led_2(event_happened);
        event_time=0;
        sprintf(send_buffer,"dropped=%d",event_happened ?1 :0);
add after line 50 of main
      }
```

```
change lines 25 to 40 of main
      if(sound>20) {
        event_time=the_clock; // start one-shot timer
        event_happened=true;
      }
      // forgetting
      if(the_clock-event_time>=10) {
        // forget when a presence was detected
        event_time=0;
        // forget about presence
        event_happened=false;
      }
      set_led_2(event_happened);
      sprintf(send_buffer,"presence=%d",event_happened ?1 :0);
```

```
change lines 25 to 40 of main
      update_movement();
      if(!event_happened && movement) {
        event_time=the_clock;
        event_happened=true;
      }
      set_led_2(event_happened);
      sprintf(send_buffer,"movement=%d",event_happened ?1 :0);
```

```
change lines 25 to 40 of main
      if(!movement) {
        update_movement();
      }
      sprintf(send_buffer,"x=%d,y=%d,mv=%d,snd=%d",
                          x_position,y_position,movement,sound);
      if(movement) {
        movement=false;
      }
```

```
10          send();
            sprintf(send_buffer,"li=%d,tmp=%d,vlt=%d",
                                light,temperature,voltage);
    delete line line 41 of main
```

Listing 25: Ideal pseudocode after 6th evolution step (i6): main, drop/noise/movement/raw_detector

Listing 26 shows the four realized simple interfaces of the hardware abstraction library for sensors, actuators, transceiver and clock.

```
 1  #ifndef SENSORS_H
    #define SENSORS_H

    // sensor abstractions
 5
    #include<stdbool.h>
    #include<stdint.h>

    // sensor values
10  extern int16_t x_position;
    extern int16_t y_position;
    extern int16_t z_position;
    extern bool movement;
    extern int8_t sound;
15  extern int16_t light;
    extern int16_t infrared;
    extern int8_t temperature;
    extern int16_t voltage;
    extern int16_t force;
20
    // sensor operations
    // init_.. must be called before first sensor use
    // update_.. is called to refesh the sensor value

25  void init_x_position();
    void update_x_position();

    void init_y_position();
    void update_y_position();
30
    void init_z_position();
    void update_z_position();

    void update_movement();
35
    void init_sound();
    void update_sound();

    void init_light();
40  void update_light();

    void init_infrared();
    void update_infrared();

45  void init_temperature();
    void update_temperature();

    void update_voltage();

50  void init_force();
    void update_force();
```

```
#endif
```

```
#ifndef ACTUATORS_H
#define ACTUATORS_H

// actuator abstractions

#include<stdint.h>
#include<stdbool.h>

void beep(uint16_t frequency,uint16_t duration);

// switches led 1 on or off
void set_led_1(bool);
// toggles led 1: on <-> off
void toggle_led_1();

// switches led 2 on or off
void set_led_2(bool);
// toggles led 2: on <-> off
void toggle_led_2();

#endif
```

```
#ifndef TRANSCEIVER_H
#define TRANSCEIVER_H

// wireless transmission and reception

#include<stdbool.h>

// becomes true when sth. has been received
extern bool has_received;
// string to send
extern char send_buffer[61];
// received string
extern char receive_buffer[61];

// sends send_buffer
void send();
// updates receive_buffer
void update_receive();

#endif
```

```
#ifndef CLOCK_H
#define CLOCK_H

// clock abstraction

#include<stdint.h>
#include<stdbool.h>

// clock value (seconds since startup; implicitly updated)
extern int32_t the_clock;

// periodically set to true by ISR every second
extern volatile bool period_elapsed;

// converts the clock value to a string
char* timetoa(int32_t);

#endif
```

Listing 26: HAL interface realizations (init.h, sensors.h, actuators.h, clock.h)

Listing 27 shows construction test output obtained after constructing, compiling Frame Technology code for evolution scenarios 3 and 4. These tests have been made for all 63 product line realizations (mechanisms "a" to "e", "g" and "h" for evolution steps 0 to 6. The first column shows the configured product, the second the code size, and the third the stack size. The same results were obtained for Conditional Compilation (scenarios "e3 and e4). It can be seen that all produced product line members have different sizes and that the same variants cause the same size increases. This indicates that the product line members have been constructed correctly. It can also be seen that product size in successive scenarios nearly does not change.

```
tilt_det,normal      59586   1083
tilt_det,time_tr     59764   1083
drop_det,normal      59112   1083
drop_det,time_tr     59290   1083
noise_det,normal     59494   1083
noise_det,time_tr    59672   1083
movmt_det,normal     59414   1083
movmt_det,time_tr    59592   1083
raw_det,normal       59020   1080
raw_det,time_tr      59190   1080
```

```
tilt_det,normal,no_voltg     59588   1085
tilt_det,time_tr,no_voltg    59766   1085
tilt_det,normal,voltg        59676   1085
tilt_det,time_tr,voltg       59856   1085
drop_det,normal,no_voltg     59114   1085
drop_det,time_tr,no_voltg    59292   1085
drop_det,normal,voltg        59614   1085
drop_det,time_tr,voltg       59794   1085
noise_det,normal,no_voltg    59496   1085
noise_det,time_tr,no_voltg   59674   1085
noise_det,normal,voltg       59584   1085
noise_det,time_tr,voltg      59764   1085
movmt_det,normal,no_voltg    59416   1085
movmt_det,time_tr,no_voltg   59594   1085
movmt_det,normal,voltg       59504   1085
movmt_det,time_tr,voltg      59684   1085
raw_det,normal,no_voltg      59022   1082
raw_det,time_tr,no_voltg     59192   1082
raw_det,normal,voltg         59640   1082
raw_det,time_tr,voltg        59810   1082
```

Listing 27: Construction test output in successive scenarios (g3 and g4)

Appendix D Detailed Results

The following tables list the results discussed in Sec.6.4.

	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}	NV_e	NV_i	NV_a	RR	NOD	$\nabla_{LOC,s}$	K_{var}
a	98	0	3	1	3	26	3	3	3	98	0	0	0	0	0	32	0,45
b	76	0	1	1	1	23	1	1	1	76	0	0	3	0	0	0	0
c	110	0	5	2	3	26	7	5	5	75	3	0	1	0,32	0	32	0,72
d	99	0	5	2	3	23	4	5	5	77	3	0	0	0,22	0	30	0,73
e	72	0	1	1	1	20	1	4	4	72	0	3	0	0	0	0	0
f	73	0	3	2	2	23	3	3	3	30	2	0	1	0,59	1	12	0,59
g	76	0	3	2	2	22	3	3	3	34	2	1	0	0,55	1	15	0,61
h	76	0	3	2	2	22	3	3	3	34	2	1	0	0,55	1	15	0,61
i	64	0	3	2	2	18	3	3	3	25	2	1	0	0,61	1	15	0,66
y	78	0	4	2	3	22	3	4	4	56	3	1	0	0,28	0	29	0,63
z_s																	
a	34	0	0		1	8	0	0	0	73	2	1	0	0,61	1	17	0,21
b	12	0	2	1	1	5	2	2	2	51	2	1	3	0,61	1	15	0,66
c	46	0	2	0	1	8	4	2	2	50	1	1	1	0,29	1	17	0,06
d	35	0	2	0	1	5	1	2	2	52	1	1	0	0,39	1	15	0,07
e	8	0	2	1	1	2	2	1	1	47	2	2	0	0,61	1	15	0,66
f	9	0	0	0	0	5	0	0	0	5	0	1	1	0,02	0	3	0,07
g	12	0	0	0	0	4	0	0	0	9	0	0	0	0,06	0	0	0,05
h	12	0	0	0	0	4	0	0	0	9	0	0	0	0,06	0	0	0,05
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	14	0	1	0	1	4	0	1	1	31	1	0	0	0,33	1	14	0,03
max	46	0	2	1	1	8	4	2	2	52	2	2	3	0,61	1	17	0,66
z_s/max																	
a	0,74	0	0	0	1	1	0	0	0	1,4	1	0,5	0	1	1	1	0,32
b	0,26	0	1	1	1	0,63	0,5	1	1	0,98	1	0,5	1	1	1	0,88	1
c	1	0	1	0	1	1	1	1	1	0,96	0,5	0,5	0,33	0,48	1	1	0,09
d	0,76	0	1	0	1	0,63	0,25	1	1	1	0,5	0,5	0	0,64	1	0,88	0,11
e	0,17	0	1	1	1	0,25	0,5	0,5	0,5	0,9	1	1	0	1	1	0,88	1
f	0,2	0	0	0	0	0,63	0	0	0	0,1	0	0,5	0,33	0,03	0	0,18	0,11
g	0,26	0	0	0	0	0,5	0	0	0	0,17	0	0	0	0,09	0	0	0,08
h	0,26	0	0	0	0	0,5	0	0	0	0,17	0	0	0	0,09	0	0	0,08
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	0,3	0	0,5	0	1	0,5	0	0,5	0,5	0,6	0,5	0	0	0,54	1	0,82	0,05

avg_{goal}					avg_{all}
a	0,25	0,49	0,5	0,83	0,52
b	0,42	0,87	0,83	0,97	0,77
c	0,67	0,85	0,44	0,64	0,65
d	0,59	0,7	0,33	0,66	0,57
e	0,39	0,66	0,67	0,97	0,67
f	0,07	0,1	0,28	0,08	0,13
g	0,09	0,1	0	0,04	0,06
h	0,09	0,1	0	0,04	0,06
i	0	0	0	0	0
y	0,27	0,44	0,17	0,6	0,37

Table 50: Measurements for initial versions (directly measured values in gray rows)

	LOC	$\nabla_{LOC,t}$	NOM	DRH	WRH	$v(G)_{t,closed}$	$v(G)_{t,open}$	$v(G)_{ct,closed}$	$v(G)_{ct,open}$	LOC_{ad}	NV_e	NV_i	NV_a	RR	NOD	$\nabla_{LOC,s}$	K_{var}
a	129	31	4	1	4	33	4	4	4	129	0	0	0	0	0	53	0,44
b	95	19	1	1	1	28	1	1	1	95	0	0	4	0	0	0	0
c	139	29	6	2	4	32	9	6	6	100	4	0	1	0,28	0	52	0,67
d	125	26	6	2	4	28	5	6	10	102	4	0	0	0,18	0	49	0,69
e	91	18	1	1	1	24	1	5	1	91	0	4	0	0	0	0	0
f	92	19	4	2	3	28	4	4	7	47	3	0	1	0,49	1	25	0,54
g	95	19	4	2	3	27	4	4	7	51	3	1	0	0,46	1	28	0,58
h	95	19	4	2	3	27	4	4	7	51	3	1	0	0,46	1	28	0,58
i	80	18	4	2	3	23	4	4	7	39	3	1	0	0,51	1	27	0,6
y	97	19	5	2	4	27	4	5	7	73	4	1	0	0,25	0	42	0,63
z_s																	
a	49	13	0	1	1	10	0	0	3	90	3	1	0	0,51	1	26	0,16
b	15	1	3	1	2	5	3	3	6	56	3	1	4	0,51	1	27	0,6
c	59	11	2	0	1	9	5	2	1	61	1	1	1	0,23	1	25	0,07
d	45	8	2	0	1	5	1	2	3	63	1	1	0	0,33	1	22	0,09
e	11	0	3	1	2	1	3	1	6	52	3	3	0	0,51	1	27	0,6
f	12	1	0	0	0	5	0	0	0	8	0	1	1	0,02	0	2	0,06
g	15	1	0	0	0	4	0	0	0	12	0	0	0	0,05	0	1	0,02
h	15	1	0	0	0	4	0	0	0	12	0	0	0	0,05	0	1	0,02
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	17	1	1	0	1	4	0	1	0	34	1	0	0	0,27	1	15	0,03
max	59	11	3	1	2	9	5	3	6	63	3	3	4	0,51	1	27	0,6
z_s/max																	
a	0,83	1,18	0	1	1	1,11	0	0	0,5	1,43	1	0,33	0	1	1	0,96	0,27
b	0,25	0,09	1	1	1	0,56	1	1	1	0,89	1	0,33	1	1	1	1	1
c	1	1	0,67	0	1	1	1	0,67	0,17	0,97	0,33	0,33	0,25	0,45	1	0,93	0,12
d	0,76	0,73	0,67	0	1	0,56	0	0,67	0,5	1	0,33	0,33	0	0,64	1	0,81	0,15
e	0,19	0	1	1	1	0,11	1	0,33	1	0,83	1	1	0	1	1	1	1
f	0,2	0,09	0	0	0	0,56	0	0	0	0,13	0	0,33	0,25	0,05	0	0,07	0,1
g	0,25	0,09	0	0	0	0,44	0	0	0	0,19	0	0	0	0,1	0	0,04	0,03
h	0,25	0,09	0	0	0	0,44	0	0	0	0,19	0	0	0	0,1	0	0,04	0,03
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	0,29	0,09	0,33	0	1	0,44	0	0,33	0	0,54	0,33	0	0	0,52	1	0,56	0,05

avg_{goal}				avg_{all}
a	0,67	0,65	0,44	0,81	0,64
b	0,45	0,86	0,78	1	0,77
c	0,89	0,61	0,31	0,62	0,61
d	0,72	0,49	0,22	0,65	0,52
e	0,4	0,7	0,67	1	0,69
f	0,1	0,1	0,19	0,05	0,11
g	0,12	0,09	0	0,04	0,06
h	0,12	0,09	0	0,04	0,06
i	0	0	0	0	0
y	0,24	0,26	0,11	0,53	0,28

Table 51: Measurements after evolution step 1

	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}	NV_e	NV_i	NV_a	RR	NOD	$\nabla_{LOC,s}$	K_{var}
a	268	170	8	1	8	68	8	8	8	268	0	0	0	0	0	116	0,41
b	116	40	1	1	1	35	1	1	1	116	0	0	8	0	0	0	0
c	181	79	8	3	5	35	14	8	8	126	5	0	5	0,3	0	51	0,74
d	152	53	9	3	6	29	7	9	21	123	5	0	0	0,19	0	55	0,74
e	89	24	1	1	1	24	1	10	1	89	0	5	0	0	0	0	0
f	94	21	5	3	4	29	5	5	12	49	4	0	5	0,48	1	21	0,58
g	93	25	5	3	3	28	5	5	9	47	4	2	0	0,49	2	25	0,64
h	93	25	4	2	3	27	4	5	7	41	3	3	0	0,56	2	25	0,6
i	82	20	4	2	3	23	4	5	7	35	3	2	0	0,57	2	23	0,61
y	105	26	6	3	5	28	5	6	9	70	5	2	0	0,33	0	39	0,6
z_s																	
a	186	150	4	1	5	45	4	3	1	233	3	2	0	0,57	2	93	0,2
b	34	20	3	1	2	12	3	4	6	81	3	2	8	0,57	2	23	0,61
c	99	59	4	1	2	12	10	3	1	91	2	2	5	0,27	2	28	0,13
d	70	33	5	1	3	6	3	4	14	88	2	2	0	0,38	2	32	0,13
e	7	4	3	1	2	1	3	5	6	54	3	3	0	0,57	2	23	0,61
f	12	1	1	1	1	6	1	0	5	14	1	2	5	0,09	1	2	0,03
g	11	5	1	1	0	5	1	0	2	12	1	0	0	0,08	0	2	0,03
h	11	5	0	0	0	4	0	0	0	6	0	1	0	0,01	0	2	0,01
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	23	6	2	1	2	5	1	1	2	35	2	0	0	0,24	2	16	0,01
max	99	59	5	1	3	12	10	5	14	91	3	3	8	0,57	2	32	0,61
z_s/max																	
a	1,88	2,54	0,8	1	1,67	3,75	0,4	0,6	0,07	2,56	1	0,67	0	1	1	2,91	0,33
b	0,34	0,34	0,6	1	0,67	1	0,3	0,8	0,43	0,89	1	0,67	1	1	1	0,72	1
c	1	1	0,8	1	0,67	1	1	0,6	0,07	1	0,67	0,67	0,63	0,47	1	0,88	0,21
d	0,71	0,56	1	1	1	0,5	0,3	0,8	1	0,97	0,67	0,67	0	0,67	1	1	0,21
e	0,07	0,07	0,6	1	0,67	0,08	0,3	1	0,43	0,59	1	1	0	1	1	0,72	1
f	0,12	0,02	0,2	1	0,33	0,5	0,1	0	0,36	0,15	0,33	0,67	0,63	0,16	0,5	0,06	0,05
g	0,11	0,08	0,2	1	0	0,42	0,1	0	0,14	0,13	0,33	0	0	0,14	0	0,06	0,05
h	0,11	0,08	0	0	0	0,33	0	0	0	0,07	0	0,33	0	0,02	0	0,06	0,02
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	0,23	0,1	0,4	1	0,67	0,42	0,1	0,2	0,14	0,38	0,67	0	0	0,42	1	0,5	0,02

avg_{goal}				avg_{all}
a	1,74	1,44	0,56	1,31	1,26
b	0,43	0,73	0,89	0,93	0,74
c	0,93	0,76	0,65	0,64	0,75
d	0,76	0,8	0,44	0,72	0,68
e	0,25	0,58	0,67	0,93	0,61
f	0,11	0,35	0,54	0,19	0,3
g	0,13	0,26	0,11	0,06	0,14
h	0,07	0,06	0,11	0,03	0,06
i	0	0	0	0	0
y	0,24	0,42	0,22	0,48	0,34

Table 52: Measurements after evolution step 2

	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}	NV_e	NV_i	NV_a	RR	NOD	$\nabla_{LOC,s}$	K_{var}
a	349	251	10	1	10	81	10	10	10	349	0	0	0	0	0	163	0,44
b	143	67	1	1	1	40	1	1	1	143	0	0	10	0	0	0	0
c	218	115	9	3	6	40	17	9	9	151	6	0	6	0,31	0	71	0,72
d	185	86	10	3	7	33	8	10	25	150	6	0	0	0,19	0	72	0,73
e	112	47	1	1	1	27	1	13	1	112	0	6	0	0	0	0	0
f	122	49	6	3	5	34	6	6	16	67	5	0	6	0,45	1	35	0,63
g	118	52	6	3	4	32	6	6	13	66	5	3	0	0,44	3	38	0,65
h	118	50	5	2	4	31	5	6	11	57	4	4	0	0,52	3	38	0,61
i	104	42	5	2	4	27	5	6	11	50	4	3	0	0,52	3	37	0,62
y	133	57	7	3	6	32	6	7	13	89	6	3	0	0,33	0	40	0,61
z_s																	
a	245	209	5	1	6	54	5	4	1	299	4	3	0	0,52	3	126	0,18
b	39	25	4	1	3	13	4	5	10	93	4	3	10	0,52	3	37	0,62
c	114	73	4	1	2	13	12	3	2	101	2	3	6	0,21	3	34	0,1
d	81	44	5	1	3	6	3	4	14	100	2	3	0	0,33	3	35	0,11
e	8	5	4	1	3	0	4	7	10	62	4	3	0	0,52	3	37	0,62
f	18	7	1	1	1	7	1	0	5	17	1	3	6	0,07	2	2	0,01
g	14	10	1	1	0	5	1	0	2	16	1	0	0	0,08	0	1	0,03
h	14	8	0	0	0	4	0	0	0	7	0	1	0	0	0	1	0,01
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	29	15	2	1	2	5	1	1	2	39	2	0	0	0,19	3	3	0,01
max	114	73	5	1	3	13	12	7	14	101	4	3	10	0,52	3	37	0,62
z_s/max																	
a	2,15	2,86	1	1	2	4,15	0,42	0,57	0,07	2,96	1	1	0	1	1	3,41	0,29
b	0,34	0,34	0,8	1	1	1	0,33	0,71	0,71	0,92	1	1	1	1	1	1	1
c	1	1	0,8	1	0,67	1	1	0,43	0,14	1	0,5	1	0,6	0,41	1	0,92	0,16
d	0,71	0,6	1	1	1	0,46	0,25	0,57	1	0,99	0,5	1	0	0,64	1	0,95	0,18
e	0,07	0,07	0,8	1	1	0	0,33	1	0,71	0,61	1	1	0	1	1	1	1
f	0,16	0,1	0,2	1	0,33	0,54	0,08	0	0,36	0,17	0,25	1	0,6	0,13	0,67	0,05	0,02
g	0,12	0,14	0,2	1	0	0,38	0,08	0	0,14	0,16	0,25	0	0	0,15	0	0,03	0,05
h	0,12	0,11	0	0	0	0,31	0	0	0	0,07	0	0,33	0	0	0	0,03	0,02
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	0,25	0,21	0,4	1	0,67	0,38	0,08	0,14	0,14	0,39	0,5	0	0	0,36	1	0,08	0,02

avg_{goal}				avg_{all}
a	2	1,6	0,67	1,42	1,42
b	0,49	0,81	1	1	0,83
c	0,93	0,75	0,7	0,62	0,75
d	0,77	0,75	0,5	0,69	0,68
e	0,31	0,67	0,67	1	0,66
f	0,15	0,35	0,62	0,22	0,33
g	0,15	0,25	0,08	0,06	0,14
h	0,08	0,05	0,11	0,01	0,06
i	0	0	0	0	0
y	0,29	0,4	0,17	0,37	0,3

Table 53: Measurements after evolution step 3

	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}	NV_e	NV_i	NV_a	RR	NOD	$\nabla_{LOC,s}$	K_{var}
a	788	690	20	1	20	180	20	20	20	788	0	0	0	0	0	335	0,43
b	154	78	1	1	1	43	1	1	1	154	0	0	11	0	0	0	0
c	252	149	11	3	7	45	20	11	11	168	7	0	7	0,33	0	71	0,76
d	211	112	13	3	9	36	10	13	30	168	7	0	0	0,2	0	82	0,76
e	122	57	1	1	1	29	1	14	1	122	0	7	0	0	0	0	0
f	133	60	7	3	6	37	7	7	18	77	6	0	6	0,42	1	35	0,65
g	128	62	7	4	4	35	7	7	15	75	6	3	0	0,41	3	38	0,68
h	128	60	5	2	4	33	5	7	11	57	4	5	0	0,55	3	38	0,62
i	114	52	5	2	4	29	5	7	11	50	4	4	0	0,56	3	37	0,63
y	143	67	8	4	7	35	7	8	15	98	7	3	0	0,31	0	53	0,61
z_s																	
a	674	638	15	1	16	151	15	13	9	738	4	4	0	0,56	3	298	0,2
b	40	26	4	1	3	14	4	6	10	104	4	4	11	0,56	3	37	0,63
c	138	97	6	1	3	16	15	4	0	118	3	4	7	0,23	3	34	0,13
d	97	60	8	1	5	7	5	6	19	118	3	4	0	0,36	3	45	0,13
e	8	5	4	1	3	0	4	7	10	72	4	3	0	0,56	3	37	0,63
f	19	8	2	1	2	8	2	0	7	27	2	4	6	0,14	2	2	0,02
g	14	10	2	2	0	6	2	0	4	25	2	1	0	0,15	0	1	0,05
h	14	8	0	0	0	4	0	0	0	7	0	1	0	0,01	0	1	0,01
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	29	15	3	2	3	6	2	1	4	48	3	1	0	0,25	3	16	0,02
max	138	97	8	2	5	16	15	7	19	118	4	4	11	0,56	3	45	0,63
z_s/max																	
a	4,88	6,58	1,88	0,5	3,2	9,44	1	1,86	0,47	6,25	1	1	0	1	1	6,62	0,32
b	0,29	0,27	0,5	0,5	0,6	0,88	0,27	0,86	0,53	0,88	1	1	1	1	1	0,82	1
c	1	1	0,75	0,5	0,6	1	1	0,57	0	1	0,75	1	0,64	0,41	1	0,76	0,21
d	0,7	0,62	1	0,5	1	0,44	0,33	0,86	1	1	0,75	1	0	0,64	1	1	0,21
e	0,06	0,05	0,5	0,5	0,6	0	0,27	1	0,53	0,61	1	0,75	0	1	1	0,82	1
f	0,14	0,08	0,25	0,5	0,4	0,5	0,13	0	0,37	0,23	0,5	1	0,55	0,25	0,67	0,04	0,03
g	0,1	0,1	0,25	1	0	0,38	0,13	0	0,21	0,21	0,5	0,25	0	0,26	0	0,02	0,08
h	0,1	0,08	0	0	0	0,25	0	0	0	0,06	0	0,25	0	0,01	0	0,02	0,02
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	0,21	0,15	0,38	1	0,6	0,38	0,13	0,14	0,21	0,41	0,75	0,25	0	0,44	1	0,36	0,03

avg_{goal}					avg_{all}
a	4,45	3,25	0,67	2,23	2,65
b	0,35	0,64	1	0,96	0,74
c	0,92	0,67	0,8	0,59	0,74
d	0,77	0,73	0,58	0,71	0,7
e	0,2	0,5	0,58	0,96	0,56
f	0,16	0,3	0,68	0,25	0,35
g	0,15	0,28	0,25	0,09	0,19
h	0,06	0,04	0,08	0,01	0,05
i	0	0	0	0	0
y	0,25	0,41	0,33	0,46	0,36

Table 54: Measurements after evolution step 4

	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}	NV_e	NV_i	NV_a	RR	NOD	$\nabla_{LOC,s}$	K_{var}
a	708	610	20	1	20	164	20	20	20	708	0	0	0	0	0	302	0,45
b	137	69	1	1	1	39	1	1	1	137	0	0	11	0	0	0	0
c	232	147	11	3	7	41	20	11	11	148	7	0	7	0,36	0	65	0,77
d	191	112	13	3	9	32	10	13	30	148	7	0	0	0,23	0	76	0,76
e	109	53	1	1	1	25	1	10	1	109	0	7	0	0	0	0	0
f	120	63	7	3	6	33	7	7	18	69	6	0	6	0,43	1	34	0,68
g	117	60	7	4	4	31	7	7	15	69	6	3	0	0,41	3	36	0,69
h	117	55	5	2	4	29	5	7	11	51	4	5	0	0,56	3	36	0,66
i	102	49	5	2	4	25	5	7	11	43	4	4	0	0,58	3	30	0,65
y	126	65	8	4	7	31	7	8	15	88	7	3	0	0,3	0	48	0,63
z_s																	
a	606	561	15	1	16	139	15	13	9	665	4	4	0	0,58	3	272	0,2
b	35	20	4	1	3	14	4	6	10	94	4	4	11	0,58	3	30	0,65
c	130	98	6	1	3	16	15	4	0	105	3	4	7	0,22	3	35	0,12
d	89	63	8	1	5	7	5	6	19	105	3	4	0	0,35	3	46	0,11
e	7	4	4	1	3	0	4	3	10	66	4	3	0	0,58	3	30	0,65
f	18	14	2	1	2	8	2	0	7	26	2	4	6	0,15	2	4	0,03
g	15	11	2	2	0	6	2	0	4	26	2	1	0	0,17	0	6	0,04
h	15	6	0	0	0	4	0	0	0	8	0	1	0	0,01	0	6	0,01
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	24	16	3	2	3	6	2	1	4	45	3	1	0	0,28	3	18	0,02
max	130	98	8	2	5	16	15	6	19	105	4	4	11	0,58	3	46	0,65
z_s/max																	
a	4,66	5,72	1,88	0,5	3,2	8,69	1	2,17	0,47	6,33	1	1	0	1	1	5,91	0,31
b	0,27	0,2	0,5	0,5	0,6	0,88	0,27	1	0,53	0,9	1	1	1	1	1	0,65	1
c	1	1	0,75	0,5	0,6	1	1	0,67	0	1	0,75	1	0,64	0,37	1	0,76	0,18
d	0,68	0,64	1	0,5	1	0,44	0,33	1	1	1	0,75	1	0	0,61	1	1	0,17
e	0,05	0,04	0,5	0,5	0,6	0	0,27	0,5	0,53	0,63	1	0,75	0	1	1	0,65	1
f	0,14	0,14	0,25	0,5	0,4	0,5	0,13	0	0,37	0,25	0,5	1	0,55	0,27	0,67	0,09	0,05
g	0,12	0,11	0,25	1	0	0,38	0,13	0	0,21	0,25	0,5	0,25	0	0,29	0	0,13	0,06
h	0,12	0,06	0	0	0	0,25	0	0	0	0,08	0	0,25	0	0,02	0	0,13	0,02
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	0,18	0,16	0,38	1	0,6	0,38	0,13	0,17	0,21	0,43	0,75	0,25	0	0,48	1	0,39	0,03

avg_{goal}					avg_{all}
a	4,09	3,19	0,67	2,06	2,5
b	0,32	0,67	1	0,91	0,73
c	0,92	0,68	0,8	0,58	0,74
d	0,78	0,75	0,58	0,7	0,7
e	0,2	0,43	0,58	0,91	0,53
f	0,18	0,31	0,68	0,27	0,36
g	0,16	0,28	0,25	0,12	0,2
h	0,06	0,05	0,08	0,04	0,06
i	0	0	0	0	0
y	0,24	0,42	0,33	0,48	0,37

Table 55: Measurements after evolution step 5

	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}	NV_e	NV_i	NV_a	RR	NOD	$\nabla_{LOC,s}$	K_{var}
a	1606	1508	40	1	40	388	40	40	40	1606	0	0	0	0	0	604	0,46
b	149	81	1	1	1	43	1	1	1	149	0	0	12	0	0	0	0
c	268	183	13	3	8	47	23	13	13	166	8	0	8	0,38	0	65	0,78
d	219	140	16	3	11	36	12	16	35	167	8	0	0	0,24	0	87	0,77
e	120	64	1	1	1	28	1	11	1	120	0	8	0	0	0	0	0
f	131	74	8	3	7	37	8	8	20	77	7	0	6	0,41	1	34	0,68
g	128	71	8	5	4	35	8	8	17	77	7	3	0	0,4	3	36	0,69
h	128	66	5	2	4	31	5	8	11	51	4	6	0	0,6	3	36	0,67
i	113	60	5	2	4	28	5	8	11	43	4	5	0	0,62	3	30	0,66
y	137	76	9	5	8	35	8	9	17	96	8	3	0	0,3	0	48	0,63
z_s																	
a	1493	1448	35	1	36	360	35	32	29	1563	4	5	0	0,62	3	574	0,2
b	36	21	4	1	3	15	4	7	10	106	4	5	12	0,62	3	30	0,66
c	155	123	8	1	4	19	18	5	2	123	4	5	8	0,24	3	35	0,12
d	106	80	11	1	7	8	7	8	24	124	4	5	0	0,38	3	57	0,11
e	7	4	4	1	3	0	4	3	10	77	4	3	0	0,62	3	30	0,66
f	18	14	3	1	3	9	3	0	9	34	3	5	6	0,21	2	4	0,02
g	15	11	3	3	0	7	3	0	6	34	3	2	0	0,22	0	6	0,03
h	15	6	0	0	0	3	0	0	0	8	0	1	0	0,02	0	6	0,01
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	24	16	4	3	4	7	3	1	6	53	4	2	0	0,32	3	18	0,03
max	155	123	11	3	7	19	18	8	24	124	4	5	12	0,62	3	57	0,66
z_s/max																	
a	9,63	11,8	3,18	0,33	5,14	18,9	1,94	4	1,21	12,6	1	1	0	1	1	10,1	0,3
b	0,23	0,17	0,36	0,33	0,43	0,79	0,22	0,88	0,42	0,85	1	1	1	1	1	0,53	1
c	1	1	0,73	0,33	0,57	1	1	0,63	0,08	0,99	1	1	0,67	0,39	1	0,61	0,18
d	0,68	0,65	1	0,33	1	0,42	0,39	1	1	1	1	1	0	0,62	1	1	0,17
e	0,05	0,03	0,36	0,33	0,43	0	0,22	0,38	0,42	0,62	1	0,6	0	1	1	0,53	1
f	0,12	0,11	0,27	0,33	0,43	0,47	0,17	0	0,38	0,27	0,8	1	0,5	0,33	0,67	0,07	0,03
g	0,1	0,09	0,27	1	0	0,37	0,17	0	0,25	0,27	0,8	0,4	0	0,36	0	0,11	0,05
h	0,1	0,05	0	0	0	0,16	0	0	0	0,06	0	0,2	0	0,03	0	0,11	0,02
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
y	0,15	0,13	0,36	1	0,57	0,37	0,17	0,13	0,25	0,43	1	0,4	0	0,52	1	0,32	0,05

avg_{goal}				avg_{all}
a	8,2	6,31	0,67	3,09	4,57
b	0,26	0,56	1	0,88	0,67
c	0,91	0,66	0,89	0,55	0,75
d	0,78	0,73	0,67	0,7	0,72
e	0,15	0,34	0,53	0,88	0,48
f	0,17	0,29	0,75	0,28	0,37
g	0,15	0,29	0,38	0,13	0,24
h	0,05	0,03	0,07	0,04	0,05
i	0	0	0	0	0
y	0,22	0,42	0,47	0,47	0,39

Table 56: Measurements after evolution step 6

G3	0	1	2	3	4	5	6
a	0,25	0,67	1,74	2	4,45	4,09	8,2
b	0,42	0,45	0,43	0,49	0,35	0,32	0,26
c	0,67	0,89	0,93	0,93	0,92	0,92	0,91
d	0,59	0,72	0,76	0,77	0,77	0,78	0,78
e	0,39	0,4	0,25	0,31	0,2	0,2	0,15
f	0,07	0,1	0,11	0,15	0,16	0,18	0,17
g	0,09	0,12	0,13	0,15	0,15	0,16	0,15
h	0,09	0,12	0,07	0,08	0,06	0,06	0,05
i	0	0	0	0	0	0	0
y	0,27	0,24	0,24	0,29	0,25	0,24	0,22

G4	0	1	2	3	4	5	6
a	0,49	0,65	1,44	1,6	3,25	3,19	6,31
b	0,87	0,86	0,73	0,81	0,64	0,67	0,56
c	0,85	0,61	0,76	0,75	0,67	0,68	0,66
d	0,7	0,49	0,8	0,75	0,73	0,75	0,73
e	0,66	0,7	0,58	0,67	0,5	0,43	0,34
f	0,1	0,1	0,35	0,35	0,3	0,31	0,29
g	0,1	0,09	0,26	0,25	0,28	0,28	0,29
h	0,1	0,09	0,06	0,05	0,04	0,05	0,03
i	0	0	0	0	0	0	0
y	0,44	0,26	0,42	0,4	0,41	0,42	0,42

G5	0	1	2	3	4	5	6
a	0,5	0,44	0,56	0,67	0,67	0,67	0,67
b	0,83	0,78	0,89	1	1	1	1
c	0,44	0,31	0,65	0,7	0,8	0,8	0,89
d	0,33	0,22	0,44	0,5	0,58	0,58	0,67
e	0,67	0,67	0,67	0,67	0,58	0,58	0,53
f	0,28	0,19	0,54	0,62	0,68	0,68	0,75
g	0	0	0,11	0,08	0,25	0,25	0,38
h	0	0	0,11	0,11	0,08	0,08	0,07
i	0	0	0	0	0	0	0
y	0,17	0,11	0,22	0,17	0,33	0,33	0,47

G7	0	1	2	3	4	5	6
a	0,83	0,81	1,31	1,42	2,23	2,23	3,09
b	0,97	1	0,93	1	0,96	0,96	0,88
c	0,64	0,62	0,64	0,62	0,59	0,59	0,55
d	0,66	0,65	0,72	0,69	0,71	0,71	0,7
e	0,97	1	0,93	1	0,96	0,96	0,88
f	0,08	0,05	0,19	0,22	0,25	0,25	0,28
g	0,04	0,04	0,06	0,06	0,09	0,09	0,13
h	0,04	0,04	0,03	0,01	0,01	0,01	0,04
i	0	0	0	0	0	0	0
y	0,6	0,53	0,48	0,37	0,46	0,46	0,47

all	0	1	2	3	4	5	6
a	0,52	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
y	0,37	0,28	0,34	0,3	0,36	0,37	0,39

Table 57: Aggregated complexity per goal

Appendix E Aggregated Results

The following tables list the results discussed in Sec.6.5.

E.1 Results for Hypotheses H1.1 and H1.2

G3								
a	0,246377	0,670776	1,740387	2,004046	4,445459	4,087009	8,195478	
b	0,42029	0,448382	0,427472	0,494857	0,352632	0,324437	0,255542	
c	0,666667	0,888889	0,933333	0,933333	0,916667	0,916667	0,909091	
d	0,586957	0,718884	0,755464	0,771089	0,773818	0,775824	0,778092	
e	0,391304	0,39548	0,246168	0,31289	0,203172	0,198221	0,147106	
f	0,065217	0,0981	0,11272	0,151262	0,156718	0,177106	0,167559	
g	0,086957	0,115049	0,131952	0,153264	0,151514	0,15921	0,152977	
	-0,52281	0,094413	1,946764	2,190941	6,916093	6,380107	14,68639	
G4								
a	0,486264	0,648526	1,435505	1,596252	3,246081	3,194455	6,311596	
b	0,872253	0,863492	0,726478	0,811814	0,643783	0,666174	0,560015	
c	0,851648	0,614512	0,762585	0,748299	0,667347	0,680952	0,657862	
d	0,696429	0,488889	0,79529	0,753295	0,732568	0,752976	0,734754	
e	0,664835	0,695692	0,581711	0,665926	0,50045	0,431651	0,342394	
f	0,103022	0,097506	0,349189	0,35437	0,304367	0,307053	0,293064	
g	0,096154	0,090703	0,255913	0,252746	0,275818	0,280926	0,294183	
	-0,36955	-0,02572	1,003451	1,143099	4,103603	4,047024	10,00049	
G5								
a	0,5	0,444444	0,555556	0,666667	0,666667	0,666667	0,666667	
b	0,833333	0,777778	0,888889	1	1	1	1	
c	0,444444	0,305556	0,652778	0,7	0,795455	0,795455	0,888889	
d	0,333333	0,222222	0,444444	0,5	0,583333	0,583333	0,666667	
e	0,666667	0,666667	0,666667	0,666667	0,583333	0,583333	0,533333	
f	0,277778	0,194444	0,541667	0,616667	0,681818	0,681818	0,75	
g	0	0	0,111111	0,083333	0,25	0,25	0,383333	
	-0,12195	-0,09859	-0,1623	-0,06977	-0,09974	-0,09974	-0,13669	
G7								
a	0,829545	0,807407	1,30853	1,423932	2,234921	2,055184	3,093301	
b	0,970588	1	0,929688	1	0,955556	0,913043	0,881579	
c	0,642191	0,623782	0,639491	0,622074	0,592039	0,579884	0,545365	
d	0,655935	0,651448	0,720062	0,68975	0,710836	0,695005	0,695841	
e	0,970588	1	0,929688	1	0,955556	0,913043	0,881579	
f	0,078975	0,054918	0,194115	0,217151	0,248214	0,266258	0,275429	
g	0,042219	0,041662	0,06218	0,056675	0,09101	0,120679	0,126881	
	0,024351	-0,01392	0,626044	0,719816	1,781494	1,651016	3,11841	
all	0	1	2	3	4	5	6	
a	0,515547	0,642788	1,259994	1,422724	2,648282	2,500829	4,566761	
b	0,774116	0,772413	0,743132	0,826668	0,737993	0,725914	0,674284	
c	0,651238	0,608185	0,747047	0,750927	0,742877	0,743239	0,750302	
d	0,568163	0,520361	0,678815	0,678534	0,700139	0,701785	0,718839	
e	0,673349	0,68946	0,606058	0,66137	0,560628	0,531562	0,476103	
f	0,131248	0,111242	0,299423	0,334862	0,347779	0,358059	0,371513	
g	0,056332	0,061853	0,140289	0,136505	0,192085	0,202704	0,239344	avg
	-0,22674	-0,00744	0,816174	0,950608	2,863797	2,701504	5,973411	-0,11709

Table 58: Results for H1.1 and H1.2 (Cloning complexity)

E.2 Results for Hypothesis H2.1

G3								
b	0,42029	0,448382	0,427472	0,494857	0,352632	0,324437	0,255542	
c	0,666667	0,888889	0,933333	0,933333	0,916667	0,916667	0,909091	
d	0,586957	0,718884	0,755464	0,771089	0,773818	0,775824	0,778092	
e	0,391304	0,39548	0,246168	0,31289	0,203172	0,198221	0,147106	
f	0,065217	0,0981	0,11272	0,151262	0,156718	0,177106	0,167559	
g	0,086957	0,115049	0,131952	0,153264	0,151514	0,15921	0,152977	
	0,923077	1,014702	1,183745	1,057164	0,975219	0,894293	0,869792	
G4								
b	0,872253	0,863492	0,726478	0,811814	0,643783	0,666174	0,560015	
c	0,851648	0,614512	0,762585	0,748299	0,667347	0,680952	0,657862	
d	0,696429	0,488889	0,79529	0,753295	0,732568	0,752976	0,734754	
e	0,664835	0,695692	0,581711	0,665926	0,50045	0,431651	0,342394	
f	0,103022	0,097506	0,349189	0,35437	0,304367	0,307053	0,293064	
g	0,096154	0,090703	0,255913	0,252746	0,275818	0,280926	0,294183	
	1,209507	1,153287	0,502508	0,539836	0,446203	0,51994	0,463446	
G5								
b	0,833333	0,777778	0,888889	1	1	1	1	
c	0,444444	0,305556	0,652778	0,7	0,795455	0,795455	0,888889	
d	0,333333	0,222222	0,444444	0,5	0,583333	0,583333	0,666667	
e	0,666667	0,666667	0,666667	0,666667	0,583333	0,583333	0,533333	
f	0,277778	0,194444	0,541667	0,616667	0,681818	0,681818	0,75	
g	0	0	0,111111	0,083333	0,25	0,25	0,383333	
	1	1	0,748031	0,821429	0,711191	0,711191	0,619048	
G7								
b	0,970588	1	0,929688	1	0,955556	0,913043	0,881579	
c	0,642191	0,623782	0,639491	0,622074	0,592039	0,579884	0,545365	
d	0,655935	0,651448	0,720062	0,68975	0,710836	0,695005	0,695841	
e	0,970588	1	0,929688	1	0,955556	0,913043	0,881579	
f	0,078975	0,054918	0,194115	0,217151	0,248214	0,266258	0,275429	
g	0,042219	0,041662	0,06218	0,056675	0,09101	0,120679	0,126881	
	0,845584	0,857845	0,646528	0,652163	0,543261	0,49668	0,441554	
all	0	1	2	3	4	5	6	
b	0,774116	0,772413	0,743132	0,826668	0,737993	0,725914	0,674284	
c	0,651238	0,608185	0,747047	0,750927	0,742877	0,743239	0,750302	
d	0,568163	0,520361	0,678815	0,678534	0,700139	0,701785	0,718839	
e	0,673349	0,68946	0,606058	0,66137	0,560628	0,531562	0,476103	
f	0,131248	0,111242	0,299423	0,334862	0,347779	0,358059	0,371513	
g	0,056332	0,061853	0,140289	0,136505	0,192085	0,202704	0,239344	avg
	0,994768	0,996648	0,728158	0,741975	0,644833	0,637752	0,57779	0,7602749

Table 59: Results for H2.1 (Binding time complexity)

E.3 Results for Hypothesis H2.2

G3								
b	0,42029	0,448382	0,427472	0,494857	0,352632	0,324437	0,255542	
c	0,666667	0,888889	0,933333	0,933333	0,916667	0,916667	0,909091	
d	0,586957	0,718884	0,755464	0,771089	0,773818	0,775824	0,778092	
e	0,391304	0,39548	0,246168	0,31289	0,203172	0,198221	0,147106	
f	0,065217	0,0981	0,11272	0,151262	0,156718	0,177106	0,167559	
g	0,086957	0,115049	0,131952	0,153264	0,151514	0,15921	0,152977	
	0,818182	1,109826	1,947463	1,521206	2,101099	2,069175	2,516163	
G4								
b	0,872253	0,863492	0,726478	0,811814	0,643783	0,666174	0,560015	
c	0,851648	0,614512	0,762585	0,748299	0,667347	0,680952	0,657862	
d	0,696429	0,488889	0,79529	0,753295	0,732568	0,752976	0,734754	
e	0,664835	0,695692	0,581711	0,665926	0,50045	0,431651	0,342394	
f	0,103022	0,097506	0,349189	0,35437	0,304367	0,307053	0,293064	
g	0,096154	0,090703	0,255913	0,252746	0,275818	0,280926	0,294183	
	0,657942	0,312572	0,572031	0,451976	0,512406	0,689052	0,763881	
G5								
b	0,833333	0,777778	0,888889	1	1	1	1	
c	0,444444	0,305556	0,652778	0,7	0,795455	0,795455	0,888889	
d	0,333333	0,222222	0,444444	0,5	0,583333	0,583333	0,666667	
e	0,666667	0,666667	0,666667	0,666667	0,583333	0,583333	0,533333	
f	0,277778	0,194444	0,541667	0,616667	0,681818	0,681818	0,75	
g	0	0	0,111111	0,083333	0,25	0,25	0,383333	
	0,416667	0,125	0,625	0,877778	0,836364	0,836364	0,80303	
G7								
b	0,970588	1	0,929688	1	0,955556	0,913043	0,881579	
c	0,642191	0,623782	0,639491	0,622074	0,592039	0,579884	0,545365	
d	0,655935	0,651448	0,720062	0,68975	0,710836	0,695005	0,695841	
e	0,970588	1	0,929688	1	0,955556	0,913043	0,881579	
f	0,078975	0,054918	0,194115	0,217151	0,248214	0,266258	0,275429	
g	0,042219	0,041662	0,06218	0,056675	0,09101	0,120679	0,126881	
	0,159001	0,118476	0,251859	0,196666	0,197558	0,187064	0,189047	
all	0	1	2	3	4	5	6	
b	0,774116	0,772413	0,743132	0,826668	0,737993	0,725914	0,674284	
c	0,651238	0,608185	0,747047	0,750927	0,742877	0,743239	0,750302	
d	0,568163	0,520361	0,678815	0,678534	0,700139	0,701785	0,718839	
e	0,673349	0,68946	0,606058	0,66137	0,560628	0,531562	0,476103	
f	0,131248	0,111242	0,299423	0,334862	0,347779	0,358059	0,371513	
g	0,056332	0,061853	0,140289	0,136505	0,192085	0,202704	0,239344	avg
	0,455955	0,339122	0,653665	0,623681	0,679782	0,722127	0,757599	0,604561

Table 60: Results for H2.2 (Programming language-dependence complexity)

E.4 Results for Hypothesis H3.1

G3								
g	0,086957	0,115049	0,131952	0,153264	0,151514	0,15921	0,152977	
y	0,268116	0,237459	0,244673	0,286622	0,246595	0,24096	0,216185	
	-0,67568	-0,5155	-0,4607	-0,46527	-0,38557	-0,33927	-0,29238	
G4								
g	0,096154	0,090703	0,255913	0,252746	0,275818	0,280926	0,294183	
y	0,442308	0,259637	0,415829	0,400924	0,409785	0,4163	0,415562	
	-0,78261	-0,65066	-0,38457	-0,36959	-0,32692	-0,32518	-0,29208	
G5								
g	0	0	0,111111	0,083333	0,25	0,25	0,383333	
y	0,166667	0,111111	0,222222	0,166667	0,333333	0,333333	0,466667	
	-1	-1	-0,5	-0,5	-0,25	-0,25	-0,17857	
G7								
g	0,042219	0,041662	0,06218	0,056675	0,09101	0,120679	0,126881	
y	0,601533	0,530695	0,483708	0,365015	0,456692	0,475171	0,469534	
	-0,92981	-0,9215	-0,87145	-0,84473	-0,80072	-0,74603	-0,72977	
all	0	1	2	3	4	5	6	
g	0,056332	0,061853	0,140289	0,136505	0,192085	0,202704	0,239344	
y	0,369656	0,284726	0,341608	0,304807	0,361601	0,366441	0,391987	avg
	-0,84761	-0,78276	-0,58933	-0,55216	-0,46879	-0,44683	-0,38941	-0,58241

Table 61: Results for H3.1 (Lack of Default complexity)

E.5 Results for Hypothesis H3.2

G3								
e	0,391304	0,39548	0,246168	0,31289	0,203172	0,198221	0,147106	
g	0,086957	0,115049	0,131952	0,153264	0,151514	0,15921	0,152977	
h	0,086957	0,115049	0,065286	0,077465	0,061308	0,05887	0,048518	
	-0,63636	-0,5493	-0,65468	-0,66764	-0,6543	-0,6706	-0,67664	
G4								
e	0,664835	0,695692	0,581711	0,665926	0,50045	0,431651	0,342394	
g	0,096154	0,090703	0,255913	0,252746	0,275818	0,280926	0,294183	
h	0,096154	0,090703	0,057038	0,053857	0,044189	0,046599	0,031773	
	-0,74729	-0,76932	-0,86381	-0,88275	-0,88615	-0,86921	-0,90018	
G5								
e	0,666667	0,666667	0,666667	0,666667	0,583333	0,583333	0,533333	
g	0	0	0,111111	0,083333	0,25	0,25	0,383333	
h	0	0	0,111111	0,111111	0,083333	0,083333	0,066667	
	-1	-1	-0,71429	-0,7037	-0,8	-0,8	-0,85455	
G7								
e	0,970588	1	0,929688	1	0,955556	0,913043	0,881579	
g	0,042219	0,041662	0,06218	0,056675	0,09101	0,120679	0,126881	
h	0,042219	0,041662	0,025843	0,011888	0,012515	0,042648	0,03733	
	-0,91663	-0,92001	-0,94789	-0,9775	-0,97608	-0,91749	-0,92597	
all	0	1	2	3	4	5	6	
e	0,673349	0,68946	0,606058	0,66137	0,560628	0,531562	0,476103	
g	0,056332	0,061853	0,140289	0,136505	0,192085	0,202704	0,239344	
h	0,056332	0,061853	0,06482	0,06358	0,050336	0,057862	0,046072	avg
	-0,8456	-0,83535	-0,8263	-0,84063	-0,86625	-0,84239	-0,87121	-0,84682

Table 62: Results for H3.2 (Open/closed variant complexity)

CV

Name	Thomas Patzke	
Address	Buchenheckenstraße 17a 67661 Kaiserslautern	
Date of Birth	January 9, 1968	
Studies	1987-1994	Ruhr-Universität Bochum Degree: Dipl.-Ing. Electrical Engineering
Professional Life	1994-2001	Software Developer at Kassenärztliche Vereinigung Westfalen-Lippe, Dortmund
	2001-today	Scientist at Fraunhofer Institute Experimental Software Engineering (IESE), Kaiserslautern

Kaiserslautern, March 30, 2011

PhD Theses in Experimental Software Engineering

Volume 1 **Oliver Laitenberger** (2000), *Cost-Effective Detection of Software Defects Through Perspective-based Inspections*

Volume 2 **Christian Bunse** (2000), *Pattern-Based Refinement and Translation of Object-Oriented Models to Code*

Volume 3 **Andreas Birk** (2000), *A Knowledge Management Infrastructure for Systematic Improvement in Software Engineering*

Volume 4 **Carsten Tautz** (2000), *Customizing Software Engineering Experience Management Systems to Organizational Needs*

Volume 5 **Erik Kamsties** (2001), *Surfacing Ambiguity in Natural Language Requirements*

Volume 6 **Christiane Differding** (2001), *Adaptive Measurement Plans for Software Development*

Volume 7 **Isabella Wieczorek** (2001), *Improved Software Cost Estimation A Robust and Interpretable Modeling Method and a Comprehensive Empirical Investigation*

Volume 8 **Dietmar Pfahl** (2001), *An Integrated Approach to Simulation-Based Learning in Support of Strategic and Project Management in Software Organisations*

Volume 9 **Antje von Knethen** (2001), *Change-Oriented Requirements Traceability Support for Evolution of Embedded Systems*

Volume 10 **Jürgen Münch** (2001), *Muster-basierte Erstellung von Software-Projektplänen*

Volume 11 **Dirk Muthig** (2002), *A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*

Volume 12 **Klaus Schmid** (2003), *Planning Software Reuse – A Disciplined Scoping Approach for Software Product Lines*

Volume 13 **Jörg Zettel** (2003), *Anpassbare Methodenassistenz in CASE-Werkzeugen*

Volume 14 **Ulrike Becker-Kornstaedt** (2004), *Prospect: a Method for Systematic Elicitation of Software Processes*

Volume 15 **Joachim Bayer** (2004), *View-Based Software Documentation*

Volume 16 **Markus Nick** (2005), *Experience Maintenance through Closed-Loop Feedback*

Volume 17 **Jean-François Girard** (2005), *ADORE-AR: Software Architecture Reconstruction with Partitioning and Clustering*

Volume 18 **Ramin Tavakoli Kolagari** (2006), *Requirements Engineering für Software-Produktlinien eingebetteter, technischer Systeme*

Volume 19 **Dirk Hamann** (2006), *Towards an Integrated Approach for Software Process Improvement: Combining Software Process Assessment and Software Process Modeling*

Volume 20 **Bernd Freimut** (2006), *MAGIC: A Hybrid Modeling Approach for Optimizing Inspection Cost-Effectiveness*

Volume 21 **Mark Müller** (2006), *Analyzing Software Quality Assurance Strategies through Simulation. Development and Empirical Validation of a Simulation Model in an Industrial Software Product Line Organization*

Volume 22 **Holger Diekmann** (2008), *Software Resource Consumption Engineering for Mass Produced Embedded System Families*

Volume 23 **Adam Trendowicz** (2008), *Software Effort Estimation with Well-Founded Causal Models*

Volume 24 **Jens Heidrich** (2008), *Goal-oriented Quantitative Software Project Control*

Volume 25 **Alexis Ocampo** (2008), *The REMIS Approach to Rationale-based Support for Process Model Evolution*

Volume 26 **Marcus Trapp** (2008), *Generating User Interfaces for Ambient Intelligence Systems; Introducing Client Types as Adaptation Factor*

Volume 27 **Christian Denger** (2009), *SafeSpection – A Framework for Systematization and Customization of Software Hazard Identification by Applying Inspection Concepts*

Volume 28 **Andreas Jedlitschka** (2009), *An Empirical Model of Software Managers' Information Needs for Software Engineering Technology Selection A Framework to Support Experimentally-based Software Engineering Technology Selection*

Volume 29 **Eric Ras** (2009), *Learning Spaces: Automatic Context-Aware Enrichment of Software Engineering Experience*

Volume 30 **Isabel John** (2009), *Pattern-based Documentation Analysis for Software Product Lines*

Volume 31 **Martín Soto** (2009), *The DeltaProcess Approach to Systematic Software Process Change Management*

Volume 32 **Ove Armbrust** (2010), *The SCOPE Approach for Scoping Software Processes*

Volume 33 **Thorsten Keuler** (2010), *An Aspect-Oriented Approach for Improving Architecture Design Efficiency*

Volume 34 **Jörg Dörr** (2010), *Elicitation of a Complete Set of Non-Functional Requirements*

Volume 35 **Jens Knodel** (2010), *Sustainable Structures in Software Implementations by Live Compliance Checking*

Volume 36 **Thomas Patzke** (2011), *Sustainable Evolution of Product Line Infrastructure Code*

Software Engineering has become one of the major foci of Computer Science research in Kaiserslautern, Germany. Both the University of Kaiserslautern's Computer Science Department and the Fraunhofer Institute for Experimental Software Engineering (IESE) conduct research that subscribes to the development of complex software applications based on engineering principles. This requires system and process models for managing complexity, methods and techniques for ensuring product and process quality, and scalable formal methods for modeling and simulating system behavior. To understand the potential and limitations of these technologies, experiments need to be conducted for quantitative and qualitative evaluation and improvement. This line of software engineering research, which is based on the experimental scientific paradigm, is referred to as 'Experimental Software Engineering'.
In this series, we publish PhD theses from the Fraunhofer Institute for Experimental Software Engineering (IESE) and from the Software Engineering Research Groups of the Computer Science Department at the University of Kaiserslautern. PhD theses that originate elsewhere can be included, if accepted by the Editorial Board.

ISBN 978-3-8396-0315-4

Fraunhofer IESE

Technische Universität Kaiserslautern
AG Software Engineering

PhD Theses in Experimental Software Engineering