Experiences with Product Line Development of Embedded Systems at Testo AG*

Ronny Kolb, Isabel John, Jens Knodel, Dirk Muthig
Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
{kolb, john, knodel, muthig}@iese.fhg.de

Uwe Haury, Gerald Meier
Testo AG
Testostraße 1
79853 Lenzkirch, Germany

Abstract

Product line practices are increasingly becoming popular in the domain of embedded software systems. This paper presents results of assessing success, consistency, and quality of Testo’s product line of climate and flue gas measurement devices after its construction and the delivery of three commercial products. The results of the assessment showed that the incremental introduction of architecture-centric product line development can be considered successful even though there is no quantifiable reduction of time-to-market as well as development and maintenance costs so far. The success is mainly shown by the ability of Testo to develop more complex products and the satisfaction of the involved developers. A major issue encountered is ensuring the quality of reusable components and the conformance of the products to the architecture during development and maintenance.

1. Introduction

Product line engineering stands for strategic reuse of significantly large portions of and among members in a line of similar systems. Independent of the exact strategy chosen to establish a product line [8], it is almost always required to invest into an infrastructure supporting all reuse-related activities, as well as the management of common product line assets. For such an investment, it is typically expected that ROI turns positive between the delivery of the 2nd and the 4th product in a line [6][20][21].

Most organizations expect to produce more product variants than in the past because their markets split into smaller segments requesting specialized variants, or the size of the covered market(s) is expected to grow.

In such scenarios, organizations address new markets with different products after migrating to product line engineering. Consequently, the cost curves shown in product line literature can rarely be observed and measured directly in practice. They are overlaid by changes to markets, business, and product portfolios. This is especially true for organizations that follow an incremental improvement and change strategy to migrate to product line engineering while continuously releasing new products to their customers. Product line benefits become even less visible if family and application engineering is not organizationally separated into different teams and the development team itself mainly profits from its home made improvements.

If such organizations are nevertheless satisfied with results achieved through the introduction of product line technology, there must be other aspects than pure cost savings that justify investments into product line engineering. Analyzing these aspects in organizations already migrating towards product line engineering may help other organizations to better understand the variety of benefits achievable through product line technology. Unfortunately, product line literature has not published much measurement data, detailed experience or lessons learned in this direction.

From our experience, the introduction of product line practices into an organization generally improves the overall quality of all kinds of development artifacts. It particularly unifies styles, patterns, technologies, and documentation templates applied consistently in diverse development teams, projects, or

* This work was performed in the project ArQuE – “Architecture-Centric Quality Engineering) which is partially funded by the German ministry of Education and Research (BMBF) under grant number 01IS F14A.
organizational units. Both, unification and consistency support reuse.

This paper presents results of assessing success, consistency, and quality of an industrial product line after its construction and the delivery of three commercial products. The assessment was performed right before the development of the next set of envisioned product variants has been started.

The investigated product line is owned by Testo AG, which is one of the world’s leading suppliers of portable measurement devices for industry and emission business. These devices (see Figure 1) enable to measure parameters like temperature, pressure, humidity, flow rate, and gas concentration. With 25 subsidiaries and about 60 trading organizations on all five continents, the company currently has 1,200 employees. New products are being developed continuously, with a cycle time of about half a year to one-and-a-half years, depending on the overall complexity of the products.

Figure 1. Typical Testo Products

Before adopting Fraunhofer PuLSE™ (Product Line Software and System Engineering) \(^2\) [19], Testo developed their products mostly independently in two different departments – mostly from scratch and partially with some opportunistic reuse. The incremental transfer of Fraunhofer PuLSE™ and its components started in 2001. Section 2 introduces the incremental and architecture-centric strategies for constructing and maintaining software product lines.

Section 3 introduces the approach followed to assess an existing product line, which is part of the evolution and management component of Fraunhofer PuLSE™, which is then applied in Section 4 to Testo’s product line. One central aspect is a consistency check between the product line architecture and the architectures of the derived products. Section 5 then summarizes lessons learned during the introduction and application of product line engineering at Testo. Section 6 discusses problems and challenges that have been encountered and need to be addressed in the future. Finally, Section 7 concludes the paper by providing a brief outlook on future activities.

2. Product Line Introduction

The transfer of product line technology into an organization that already develops a set of systems in a particular application domain affects many aspects of the organization and poses a number of severe risks. Therefore, the transfer must be systematically planned and realized. To minimize the risks, it is important to start from existing practices in an organization and to understand the individual entry points. Also, it is generally required to minimize the up-front investment and enable quick incorporation into the standard practices of an organization. To keep the initial investments low, an incremental transition strategy is followed. This does not only mean that product line concepts are introduced incrementally, but also that the development of product line artifacts is not done up-front but is realized as part of product development activities. Further, it means to start with the most promising areas and phases first.

Figure 2. Quality Improvement Paradigm

The approach applied at Testo to transition to product line development and to improve productivity and quality while efficiently producing many product variants is Fraunhofer PuLSE™ [3]. The approach is developed and applied since 1997 and enables software and system product lines to be conceived and deployed within a large variety of contexts. This has been achieved via a strong product-centric focus throughout all life cycle phases, the capability to incrementally introduce components, a maturity model, and the customizability of processes and techniques to a few main product development situations.

The introduction of product line engineering using Fraunhofer PuLSE™ follows the Quality Improvement Paradigm (QIP) [1]. The QIP is a model that defines an incremental view of technology transfer and process

\(^2\) PuLSE is a registered trademark of the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern, Germany.
improvement. The cyclic QIP process is shown in Figure 2. As an initial step the actual organization and its practices are characterized and understood. In particular, potential improvement activities are identified and an organization’s existing products are analyzed. In the second step, the identified improvement activities are evaluated and prioritized with respect to the organization’s business objectives, and the goals for the improvement activities are set. In the third step, suitable methods, techniques, and tools to achieve the goals set are selected. The selected technologies must be transferred into the organization. The transfer itself is also performed iteratively, adapting continuously the technology to optimize the measured effects. After the technology has been transferred, the experience and achieved results are analyzed and finally prepared for reusing them in future cycles of the QIP. In the context of a product line, generic artifacts in the product line infrastructure, market trends, system-specific characteristics, as well as data from configuration management and quality assurance activities are analyzed to find ways for further improvement. The post-transfer analysis of the organization directly leads to the first step of the subsequent transfer cycle, characterization of the organization.

The individual activities performed during the introduction of product line development using this approach at Testo were described in [19]. In the paper, the individual activities and the experiences that have been made until the delivery of the first member of the product line to market are summarized. This paper, in contrast, presents the results of assessing the product line after the delivery of three products to market and before starting the development of the next set of envisioned product variants. It comprises a summary of steps 5 and 6 of the QIP.

The characterization of the organization and its practices has been performed in 2001. The analysis addressed existing software and technology, future product development plans, the organizational context, and the business situation. Since a lot of experience existed in the product domain, the main technical domains have been identified through an architecture-centric scoping. This had the additional purpose of acquainting the stakeholders with architectural concepts. The results of the analysis and characterization showed a high economic potential and a sound chance for a successful product line transition. Based on the analysis results, an incremental transition plan was defined. According to the transition plan, first an overall architecture framework would be developed and then the individual sub-domains would be transitioned one by one in the order of their benefits and potential risks (i.e., high benefit, low risk first). The transition of the sub-domains started with memory management and continued with measurement, printing, and measurement programs. In parallel, techniques, methods, and tools suitable for the context of Testo and the specified goals have been selected. In particular, the individual components of PuLSE™ for the different life cycle phases have been selected and customized. The methods addressed domain analysis, architecting, reengineering, component engineering, as well as configuration and variant management. A major activity was the development of a common architecture for the product line based on the software architectures of the existing systems. In the following section, the applied approach is introduced.

2.1. Architecture Development

One of the key artifacts during product line development is the so-called product line or reference architecture. This architecture supports current as well as future products in a domain by defining common and variable components for the members of a product line. PuLSE™-DSSA is the part of the Fraunhofer PulSE™ approach that focuses on the definition and development of architectures for new product lines [4]. The main underlying concepts of the PuLSE-DSSA method are:

- Scenario-based development in iterations that explicitly addresses the stakeholders’ needs.
- Incremental development, which successively prioritizes requirements and realizes them.
- Direct integration of reverse engineering activities into the development process on demand.
- View-based documentation to support the communication of different roles.

Since Greenfield scenarios [9] are found only rarely in industrial contexts, PuLSE-DSSA is designed to smoothly integrate reverse engineering activities into the forward-engineering process of defining a product line architecture. The main process loop of PuLSE-DSSA consists of four major steps:

1. Planning. The planning step defines the contents of the current iteration and delineates the scope of the current iteration. This includes the selection of a limited set of scenarios that are addressed in the current iteration. Scenarios are used similar to SEI’s ATAM [7]. The step also includes the identification of the relevant stakeholders and roles, the selection and definition of the views, as well as defining whether or not an architecture assessment is included at the end of the iteration.
2. Realization. In the realization phase, solutions are selected and design decisions taken in order to fulfill the requirements given by the scenarios. When the architecture team realizes the scenarios and internally defines the architecture, questions whether the architecture may reuse existing components often arise. If this is the case, the question is formulated as precisely as possible as a request triggering reverse engineering activities. Reverse engineering is the process of analyzing a system to identify its components and their interrelationships and to create representations of it in other forms or at a higher level of abstraction. The main goals in the context of product line engineering are a) recovery of lost information in order to benefit from field-tested solutions and experiences, b) localization of single features in the source code in order to reuse this functionality, and c) enabling reuse in order to integrate components (or whole subsystems) into the product line. In case of Testo, the first reverse engineering task was the redocumentation of the architectures of the existing systems. Based on the redocumented architecture, then a product line architecture was defined with some minor reverse engineering requests. The resulting architecture, which is briefly discussed in Section 4.2, comprised seven layers, but no strict layering was enforced.

3. Documentation. In this step the architecture is documented using an organization-specific set of views. It thereby relies on standard views as, for example, defined by Kruchten [14] or Hofmeister et al. [10], and customizes or complements them by additional aspects requested by one of the key stakeholders [5]. For Testo, a view model consisting of conceptual view, module view, behavioral view, and code view was defined. The architectural views were documented using an extended version of the Unified Modeling Language (UML) [19].

4. Assessment. The goal of the assessment step is to analyze and evaluate the resulting architecture with respect to the achievement of business goals and for meeting functional and quality requirements. In an intermediate state of the architecture, this step might be skipped and the next iteration is started. However, it must be performed after the final iteration. The Testo product line architecture was evaluated using ATAM [7] with respect to quality requirements and its suitability as a basis for the development of the envisioned products of the software product line. In particular, the architecture was evaluated with respect to maintainability (e.g., changing existing functionality, testability), extensibility (e.g., adding new functions, adding support for new input and output devices), reusability, performance (i.e., time behavior, resource usage), and reliability.

3. Product Line Assessment

This section introduces the assessment of the Testo product line. The assessment focused on the qualitative and quantitative evaluation of the product line infrastructure and its quality especially with respect to the initially defined product line architecture. The assessment has been done after an initial product line infrastructure has been constructed and three products derived from it have been delivered to market.

First, Section 3.1 introduces the goals of the assessment and the overall approach. Section 3.2 introduces the approach for checking the conformance of the existing product line members to the product line architecture.

3.1. Objectives and Approach

In order to get an understanding of the status of product line development at Testo and to identify problems with respect to the development of products according to the product line architecture and by reusing components, a detailed assessment was performed. First, a structured interview with the Testo department heads responsible for development and product line introduction was performed in order to determine the achievements and improvements to the product line development. Then, the implementations of three members of the product line have been analyzed by Fraunhofer IESE. The objectives were to determine:

- whether and to which extent the initially developed product line architecture was realized during the implementation of the product line members,
- how good the quality of the currently available implementations is with respect to product line specific qualities like reusability, extensibility, and modifiability,
- what problems with respect to systematic reuse and product line development exist at the level of architecture and implementation.

In particular, the assessment focused on checking the conformance of the architectures as realized in the available implementations to the originally planned and documented product line architecture. Also, the consistency of the documentation (e.g., architectural views, component engineering models) against the implementation was checked by means of a semi-automatic evaluation as described in Section 3.2.
The quality of the product line infrastructure implementation as well as the product implementations was assessed by means of source code metrics, clone detection results, detailed variability analysis results, and results of a manual review of selected product line components and interfaces. An overview of the results is provided in Section 4.3.

3.2. Architecture Conformance Checking

Architecture conformance checking is based on static evaluations of two models of a software system. Typically, an architectural model (the planned or intended architecture) is compared with a source code model (the actual or implemented architecture). Each model consists of a set of (hierarchical) model elements and different types of relations or dependencies (e.g., calls, variable access, includes) between them. The model elements and the dependencies between them can either be postulated (e.g., an architectural model) or extracted (e.g., a source code model). The comparison requires a mapping between the two models to be compared, which is a human-based task, usually done by the architects of the system. The comparison assigns one of the following types to each model element and relation between a pair of model elements:

- **Convergence** – a specific model element or relation exists in both the architectural model and in the source code model
- **Divergence** – a specific model element or relation is only present in the source code model
- **Absence** – a specific model element or relation is only present in the architectural model

The outcome of architecture conformance checking is summarized and documented in a results report (graphical and textual), which can be processed further. The results show whether or not the architectural model converges to the source code model. The architects can interpret the results by the total numbers of convergences, divergences, and absences and use them for different purposes. In some cases, it is necessary to calibrate the evaluations (i.e., refinement of the architectural model, the source code model, or the mapping), which means that the evaluation is performed iteratively. We conducted the checking with the SAVE (Software Architecture Visualization and Evaluation, see [15] and [18]) tool, an Eclipse plug-in (see [12]) developed by Fraunhofer IESE. SAVE performs static architecture evaluations and is inspired by the ideas of reflexion models [16][11].

4. Assessment Results

This section summarizes the results of the detailed assessment of the current Testo product line. Section 4.1 first summarizes the major achievements and improvements attained by Testo due to the introduction of product line development. Section 4.2 then presents the results of checking the conformance of the existing product line members to the product line architecture. Section 4.3 gives a brief overview of the results of checking the quality of the product line implementation.

4.1. Achievements and Improvements

In general, the product line project is considered very positive by Testo even though there is actually no reduction of time-to-market as well as development and maintenance costs so far. Furthermore, in contrast to the common belief that with product lines less developers are needed, the development organization has grown since product line introduction and even more developers will be hired in the near future. This is primarily due to the fact that more complex products and products addressing new market segments will be developed. According to Testo, the development of these products would not have been possible without product line engineering. Despite the growing complexity of the products and the development of product line components as part of concrete product development projects, however, all projects have been in time and the quality level has been maintained. Also, the development effort has not been increased. As the time-to-market and cycle time of new products are primarily determined by hardware and mechanics design, no significant reduction is expected for the future. Contrary to the situation in the past in which software development was often behind schedule and not in sync with hardware development, however, a major improvement was already achieved.

Three products have been released so far and all three products share a common core, the framework. The framework comprises functionality that is shared among the products and was developed at the same time as the first product was implemented. The further development (still ongoing) extends the framework with further reusable components incrementally. With the latest delivered product, the common core of framework components has been increased to about 40%. The development team expects that the common core will grow as more components now are realized as framework components and that the framework components will mature when being used in more
products. It is also expected that in consequence the development and maintenance effort will be reduced.

With respect to the activities of the product line introduction project, especially the design and documentation of the product line architecture were seen as major achievements for development. The architecture documentation is actively used by the developers and serves as one of the main references during product development. Furthermore, with the structure provided by the architecture and the defined interfaces, it is much easier to integrate new developers and to realize more features in the same time.

4.2. Architecture Conformance

Three members of the Testo product line of climate and flue gas measurement devices sharing a common reference architecture were analyzed with respect to their conformance. Full conformance means that the product’s architectures do not violate the reference architecture at all.

As mentioned, all three products share a common core called framework. The usage of the framework in each product was one aspect of architecture conformance checking. Figure 3 shows the results for “Product A” graphically: the check mark indicates an allowed usage (a usage here means an include, a function call or a variable access dependency) whereas the exclamation mark indicates a violation.

![Figure 3. Framework usage of Product A](image)

Figure 4 presents a table with the call dependencies between product A and the framework. The dependencies are either internal ones (calls within the product, or in the framework) and external ones (from the product to the framework, or vice versa). An unexpected high number of call dependencies from the framework-related source code to the product code was detected. By zooming into the decomposition hierarchy (either by using the graphical user interface of SAVE or textual output), the architects were able to learn about the affected components, files, functions, and variables.

<table>
<thead>
<tr>
<th>CALLER</th>
<th>CALLEE</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product A</td>
<td>9,226</td>
<td>9,284</td>
</tr>
<tr>
<td>Framework</td>
<td>1,021</td>
<td>1,879</td>
</tr>
<tr>
<td>Total</td>
<td>10,247</td>
<td>11,163</td>
</tr>
</tbody>
</table>

Figure 4. Product A vs. Framework – Call Dependencies

Thus, the architects at Testo learned which framework-specific components had dependencies to product-specific code violating the architectural design goals. This imposes a threat for the derivation of future product line variants: dependencies between the framework and the products are a major risk to the structure of the framework.

![Figure 5. Layered Reference Architecture](image)

The reference architecture comprised seven layers, but no strict layering was enforced. Figure 5 depicts an overview on the architectural layers. Subsystems and components comprised in each layer have been filtered out.

![Figure 6. Evaluation Layered Architecture](image)

---

Note that the names of the products have been anonymized due to confidentiality reasons.
Figure 6 shows the results of one product evaluated against the layered reference architecture. It was significantly violated by calls, includes, and variable accesses as depicted by the exclamation mark overlay icons. Convergences are represented with check mark overlay, and there were no absences. Although the figures let assume that the distribution of convergences and divergences was even, this was not the case. The number of dependencies leading to an arrow for convergences was much higher than for divergences. The architecture documentation of the product did not reflect the implemented architecture as it was captured by the SAVE tool. Violations, even when introduced on purpose due to performance issues, were not documented.

The overall results of architectural conformance checking are represented in the table of Figure 7. The cells show the percentage to which degree the product is conformant to the layering set by reference architecture and the framework usage. Conformance is expressed in terms of percentage and calculated by the number of convergences divided by the total number of dependencies (i.e., divergences plus convergences). It shows that Product C had the lowest conformance of the three products, which can be explained with development history: Product A and the framework were developed more or less at the same time, while product C was the latest development being part of the analysis. While product A and B are rather similar, product C is a variant addressing another market segment.

<table>
<thead>
<tr>
<th>Product</th>
<th>Conformance</th>
<th>Evaluation against</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Layering</td>
<td>Framework</td>
</tr>
<tr>
<td>Product A</td>
<td>95.73%</td>
<td>85.75%</td>
</tr>
<tr>
<td>Product B</td>
<td>89.79%</td>
<td>85.38%</td>
</tr>
<tr>
<td>Product C</td>
<td>72.74%</td>
<td>63.27%</td>
</tr>
<tr>
<td>Average</td>
<td>86.01%</td>
<td>78.13%</td>
</tr>
</tbody>
</table>

The outcome was an action list of where to adapt the architecture, and how to better support the derivation of future products with the help of the framework infrastructure. For instance, the action items derived were concerned with the adaptation of either the reference architecture or the product implementation. It was decided to refactor obsolete dependencies (e.g., includes, but no included element is used), as well as to encapsulate global variables.

In summary, the architecture conformance checking proved that the reference architecture was in place (with exceptions) and the adequacy of the architecture was reflected in the satisfaction of the engineers at Testo as well. Several problems in the architecture and its documentation which have been identified based on the assessment results were addressed by Testo in the meantime.

4.3. Implementation Quality

Besides checking the conformance of the product implementations to the initially defined product line architecture, the quality of the implementation itself has been analyzed. To get an understanding of the current product implementations and to detect potential problems and improvement areas, the implementations, and hence the source code, have first been analyzed by means of selected source code metrics. The metrics have been selected using the Goal Question Metric (GQM) approach [2]. The metric values have been determined for each of the products as well as the framework by means of a commercial metric tool. For each of the metrics, then a comparison against threshold values and acceptable ranges as proposed by literature and best practices was done. In addition, we looked for the outliers (i.e. values that are outside the defined range or extremely deviate from the mean determined for the analyzed code). The analysis of the metrics showed that a large number of files, functions, and data structures exceeded the suggested threshold values and acceptable ranges. In particular, the metrics indicated problems regarding complexity and structuredness which can have a negative impact on maintainability and reusability. The analysis also showed that in the first products only a very small percentage of code from the framework was used and that the percentage of code reused from the framework is increasing in the products that have been developed later. Analogously, the percentage of product-specific code is decreasing as shown in Figure 8.

Figure 7. Overall Architecture Conformance

The analysis of the implementation with respect to the realization of variabilities showed that there is no common strategy and that the variability is
concentrated in a small number of components. The usage of conditional compilation for realizing variability often resulted in code that is difficult to read and understand even though it is in principle an easy and straightforward extension mechanism in C.

As various problems are associated with code duplication, including increased code size and increased maintenance costs, the implementation was also analyzed with the goal of detecting code clones in the source code. Code clones are defined to be fragments of the source code which are structurally or syntactically similar or identical to each other. The clone detection was performed using a tool based on simple text pattern matching that enables to detect internal clones (i.e. duplicated code lines that are found in a single file) as well as external clones which are found by comparing different files. The results of the clone detection showed that there existed some internal clones of medium size and a large number of external clones between product-specific code and framework code. As a detailed analysis showed, this was mainly due to the development of reusable framework components as part of a concrete product. Often, the original product-specific code has been moved to the framework, but not yet deleted in the application part.

5. Lessons Learned

During the introduction of architecture-centric product line development at Testo, a number of lessons have been learned. Most of the lessons are related to the definition of a product line architecture and the development of reusable product line components.

The early scoping activities (c.f. [19]) were reasonable because they could bind the further activities to realizable features and products. Nevertheless, experience showed that in the beginning too many variation points were modeled and realized in the code. Due to the high number of variation points the product line components turned out to be overly complex. A better strategy would have been to realize only the variation points that are essential for the next products and to leave the others for later. This not only helps to reduce complexity, but more importantly ensures testability. The high number of variation points also led to a high complexity of the domain requirements documentation. These documents are too complex and were therefore not often used so far. For further requirements activities, the documents should be made simpler and less abstract. Although the requirements activities started after the architecture definition, the documents do not reflect the architectural structure as to that point in time, experience (e.g., with the interfaces) was missing. In consequence, we recommend to focus domain requirements engineering on a few architecturally relevant domains and to use the requirements as clarification for the implementation.

The development of the product line architecture was primarily based on an analysis of the software architectures of the existing products rather than the results of an overall domain analysis. In case of Testo, this was a viable approach, as the previous recovery of the architecture led to a good understanding of the domain and the future systems would not include fundamentally different functionalities. The result of the activities, however, was an architecture that was not fully elaborated. In particular, the interfaces of the architectural components were not defined. Due to the development of product line components as part of concrete products, however, this later caused serious problems. Among others, there were problems with respect to the integration of components into products, unwanted dependencies, inconsistencies between interfaces, and the reusability and flexibility of components. A typical problem was that the resulting interfaces and components were too product-specific leading into time-consuming and difficult changes later on. From our experience, therefore, it is especially important in a reactive product line approach (where the product line infrastructure is built in parallel to a continuous product derivation [8]) to clearly define the interfaces for all components beforehand; if possible together with the different product teams. To have a fully elaborated architecture with defined component interfaces before staring the development of product line components and products would minimize problems with respect to integration of components as well as reduce the number of required adaptations.

To maintain the product line architecture and to ensure the architecture conformance and quality of framework components, a so called component team has been introduced to serve as an architecture board. The team currently consists of five senior developers from different development units. The experiences with such a team were very positive as the interests of the different units can be effectively aligned in the team. Additionally, the members of the component team serve as product line champions in their unit, communicating the ideas and structures of the product line and its architecture to the developers.

In terms of the documentation of the architecture, the usage of a tailored set of architectural views proved to be very helpful in both understanding and communicating the architecture. It showed that especially a behavioral view (showing using a number of usage scenarios how the architectural elements
defined in a structural view interact with each other and realize the required behavior) is very helpful. The scenarios also helped in identifying inconsistencies, defining the semantics of components, and discussing the effect of design choices on non-functional aspects like performance.

The reactive or incremental approach to product line introduction helps to significantly reduce the required up front investments and minimizes the risks for an organization due to introducing software product line engineering. Also, developing product line artifacts as part of concrete products is basically a viable approach as long as the product line architecture is actually followed and the interfaces for the reusable components have been agreed upon beforehand between the different product teams using them. It showed, however, that it is often very difficult to develop product line components that are not product-specific but reusable in a number of different products. To overcome this issue, the development has to be carefully planned and continuously coordinated with the product teams.

6. Open Issues and Challenges

The retrospective analysis of the product line activities at Testo also shows that some problems and challenges have been encountered, but are not – or not sufficiently – addressed or supported by the current approach yet. Thereby, the identified issues and challenges are primarily related to quality assurance as well as maintenance and evolution. Some of them are caused by the reactive nature of transitioning to product line development, while others would also have been observed in proactive situations.

In terms of quality assurance, the major problem was the missing support in efficiently and effectively planning and performing quality assurance of the reusable product line artifacts as well as the products derived from the product line infrastructure. In particular, ensuring the quality of the components intended for reuse in products was challenging. On the one hand, this was difficult due to the lack of adequate quality assurance and testing techniques and methods that take into account product line specifics like genericity [13]. On the other hand, the reactive nature of constructing reusable product line components as part of a concrete product and the tight deadlines for delivering a product made it very difficult to assure their general quality and reusability. As there often was not enough time and effort for adequately checking all parts of a generic component, quality assurance activities primarily focused on the product-specific aspects. As a result, the required level of quality for a component in the context of another product context was typically not achieved. Hence, each product team reusing a component from the product line architecture had to perform quality assurance for this component again in order to ensure high-quality of their end product. In the future, thus, an efficient quality engineering process that focuses on critical aspects, guarantees the required quality of the products, and does not have redundancies in the activities has to be established. This also includes establishing an automated testing of reusable software components as well as combining diverse quality assurance techniques such as testing, reviews, architecture analysis and evaluation, and code metrics.

Related to the quality and quality assurance problems are challenges with respect to maintenance and evolution of the product line infrastructure and the products derived from it. One of the major challenges encountered during evolution was the decision whether a new or modified feature required by a particular product is incorporated into the product line and its reusable components or restricted to the particular product. In case of Testo, evolution and maintenance of the reusable components have been performed as part of the product development. As a result, product line components have often been extended with very product-specific code and unwanted dependencies to other components have been created. This affected negatively the development of products in the product line. To reduce the risk that the different products evolve in different directions, the maintenance of the reusable components and the development and maintenance of the product line members must be carefully coordinated. Another issue that has to be addressed is not only to preserve and assure the quality levels of the reusable components during maintenance, but also to further optimize their quality in terms of product line specific qualities like reusability and flexibility.

7. Conclusions and Outlook

Product line development is an up-and-coming technology with a large potential to reduce cost and effort and to shorten time-to-market. In this paper, we presented an architecture-centric approach to product line development and its application to a family of embedded measurement systems at Testo AG. The paper analyzed in retrospective the results and experiences in introducing product line development. The goal was to determine whether the transition to product line development was actually successful and
what improvements have been achieved compared to the traditional single system development. Further, problems encountered during the introduction and application of product line development but not yet addressed have been identified to further improve the architecture-centric Fraunhofer PuLSE™ approach.

The results of the retrospective analysis showed that by following a systematic, carefully planned, and lightweight approach, product lines can successfully be introduced to an organization in short time. Testo is convinced that the effort for developing new products will be reduced significantly once the necessary reusable components have been developed.

From the study of a successful application of architecture-centric product line development, however, it also became evident that the development of reusable components as part of a concrete project poses the risk of minimizing the reusability of the components and violating the initially defined product line architecture. Moreover, the architectures as implemented in the concrete projects were not fully conformant to the proposed product line architecture and different components were sometimes realized in different ways in different products. A major issue encountered is ensuring the quality of the components and the conformance of the products to the product line architecture during development and maintenance.

In order to address the identified issues and shortcomings of the current state-of-the-art approach PuLSE™, an upcoming project will address the quality and cost optimized maintenance and evolution of embedded systems. In particular, it will investigate the combination of reverse engineering, code analysis, quality engineering, and architecture analysis and evaluation to support the decision processes during maintenance and evolution. The basic idea is to plan and optimize quality assurance and maintenance activities based on the quality profile of an organization and by using experiences with prior products.

References