

---

## Practice

# Refactoring a legacy component for reuse in a software product line: a case study



Ronny Kolb<sup>1,\*</sup>, Dirk Muthig<sup>1</sup>, Thomas Patzke<sup>1</sup> and Kazuyuki Yamauchi<sup>2</sup>

<sup>1</sup>*Fraunhofer Institute for Experimental Software Engineering (IESE), Fraunhofer-Platz 1, D-67663 Kaiserslautern, Germany*

<sup>2</sup>*Ricoh Company, Ltd, 1-15-5 Minami-Aoyama, 107-8544 Minatoku, Tokyo, Japan*

---

## SUMMARY

Product lines are a promising approach to improve conceptually the productivity of the software development process and thus to reduce both the cost and time of developing and maintaining increasingly complex systems. An important issue in the adoption of the product-line approach is the migration of legacy software components, which have not been designed for reuse, systematically into reusable product-line components. This article describes activities performed to improve systematically the design and implementation of an existing software component in order to reuse it in a software product line. The activities are embedded in the application of Fraunhofer PuLSE<sup>TM</sup>-DSSA—an approach for defining domain-specific software architectures (DSSA) and product-line architectures. The component under investigation is the so-called Image Memory Handler (IMH), which is used in Ricoh's current products of office appliances such as copier machines, printers, and multi-functional peripherals. It is responsible for controlling memory usage and compressing and decompressing image data. Improvement of both the component's design and implementation are based on a systematic analysis and focused on increasing maintainability and reusability and hence suitability for use in a product line. As a result of the analysis and refactoring activities, the documentation and implementation of the component has been considerably improved as shown by quantitative data collected at the end of the activities. Despite a number of changes to the code, the external behavior of the component has been preserved without significantly affecting the performance. Copyright © 2006 John Wiley & Sons, Ltd.

*Received 14 September 2005; Revised 2 December 2005; Accepted 15 December 2005*

KEY WORDS: refactoring; reuse; legacy component; software product line; code analysis; variability

---

\*Correspondence to: Ronny Kolb, Fraunhofer Institute for Experimental Software Engineering (IESE), Fraunhofer-Platz 1, D-67663 Kaiserslautern, Germany.

†E-mail: kolb@iese.fraunhofer.de



## INTRODUCTION

Organizations developing software face a number of challenges today. In particular, there is a critical need to reduce the cost, effort, and time-to-market of software, but, at the same time, the complexity and size of systems are rapidly increasing and customers are requesting increasingly quality systems tailored to their individual needs [1].

A promising approach to improve conceptually the productivity of the software development process and thus to reduce both cost and time of developing and maintaining increasingly complex systems is the notion of software product lines [2]. Product-line software engineering refers to techniques for creating a collection of similar software systems from a shared set of software assets using a common means of production. A software product line is defined as a family of products designed to take advantage of their common aspects and predicted variability [3]. Product-line engineering is based on strategic reuse as opposed to opportunistic reuse, which usually comes into play in single-system development. Strategic reuse means that attributes of software such as generality, modularity, environment independence, and self-descriptiveness are planned and implemented with respect to the predictable benefits concerning customer needs and market trends. In product-line engineering, the overall development life cycle is split into two concurrent phases: development for reuse, which is done in the domain-engineering phase, and development with reuse, which is done in the application-engineering phase. During domain engineering, the development of the product-line infrastructure consisting of reusable assets takes place. These assets are work products (e.g., requirement specifications, architectural diagrams, code, etc.), which however are flexible and therefore applicable to the whole family of systems under development. To that end these assets contain variation points, which must be resolved on delivering a concrete product of the family. Product delivery takes place in *application engineering*. Therein, assets from the product-line infrastructure are being reused and customer-specific assets are being developed.

The benefits associated with product-line engineering increasingly convince organizations to study underlying concepts and ways for implementing it in practice. In particular, when planning and engineering activities focusing on a next generation of an existing product family are started, organizations are often forced to install product-line technologies into their practices in order to avoid all of the known and existing problems experienced in the past while creating family members; that is, constructing product variants in a single-system manner.

An important point for organizations is the question of how to migrate from single-system development towards product-line engineering. A promising migration strategy in order to achieve this organizational change is to transform the existing system into reusable components where appropriate. The reusable (often generic) components can then form the core of the product-line infrastructure, basis of the future product line. This leads to a reverse-engineering-driven approach, which may also help saving knowledge encoded in the existing software (including knowledge on necessary technical variations).

This article presents an extended discussion of an industrial case study [4] in applying such an approach, namely Fraunhofer PuLSE<sup>TM</sup> (Product Line Software Engineering) [5]. The article focuses on the architecture component of PuLSE<sup>TM</sup> and reports on the aspect of migrating legacy software components, which have not been designed for reuse, systematically into reusable product-line components. Reusable product-line components are software components that capture commonalities of all members of the product line and provide variation points to support the variabilities



among the different products. The article describes in detail the analysis of existing components' implementations with respect to different quality aspects and derived improvement steps of its design and implementation. In particular, the article focuses on the quality aspects flexibility (i.e., the capability to meet changing situations and diversified operations with minimum disruption or delay) and reusability (i.e., the capability to reuse an existing asset for different products). Both of these attributes contribute to maintainability, that is, the capability of an existing artifact to be corrected, improved, or adapted to new or changing requirements or to a changed environment. In addition to the initial description of the case study [4], the article describes a second improvement cycle that has been performed on the component and presents quantitative data regarding the achieved quality improvements.

The overall case study described in this article has been done over the course of four months and in total five people have been involved. Two of these people were researchers from the Fraunhofer IESE, which coordinated the case study, and have a background in component refactorings, architecture analysis and design, and documentation of components. They have been supported by two students, who mainly performed the previously agreed upon refactoring steps. Detailed knowledge about the component and the products making use of it was continuously provided by one expert from Ricoh. If necessary, additional input was requested from the original component developers.

The remainder of this article is structured as follows. First, the context of the case study and the investigated component are introduced. Then, an overview on the approach followed is given. That is, on the process of PuLSE<sup>TM</sup>-DSSA<sup>‡</sup>, and especially the integration of reverse-engineering activities in the forward-engineering process of defining an architecture for a new product line. After that, the main activities done in order to prepare the component as a potential product-line asset, namely the redocumentation of the component and analysis of the quality of the component's implementation are covered. Next, the improvement activities performed to reflect the different (reusable) aspects covered by the component and to increase the overall component quality are outlined. Then, the achieved results are summarized and the actions taken are analyzed in retrospect. Finally, the article presents related work on refactoring and clone detection, and provides some concluding remarks.

## COMPONENT OVERVIEW

The case study presented in this article is based on a software component called Image Memory Handler (IMH), which is used in Ricoh's current products of office appliances. Ricoh is one of the world's leading developer and producer of office equipment, including copiers and printers and other electronic equipment. In the following, a brief overview of the component is provided. One of the main issues in Ricoh's products is how to process image data with small memory space and best performance. To solve this issue, a number of techniques have been developed and continuously refined. The IMH component realizes such techniques and has been used in most of Ricoh products in the past four years. The responsibilities of the component are, among others, to control memory usage among several client programs, manage image data stored in persistent data storage such as a hard disk drive, compress and decompress image data, and to synchronize image-processing timing

---

<sup>‡</sup>DSSA stands for domain-specific software architecture.



among specific hardware. Consequently, the component is required to adapt to new hardware and new algorithms for image handling whenever they change or are added and removed. In addition, since the component is very close to hardware and its control software, it is influenced by non-functional requirements such as performance and constraints such as the amount of memory that usually vary among the different products. Finally, the component has been extended over time to accommodate the requirements of new products, which provide new or differing features.

At the beginning of the development of the IMH component, main functionality, all external interfaces, data structure, and internal dynamic behavior were explicitly documented to some extent. However, as development was ongoing, the necessary information was not written explicitly as a design document but embedded into source code implicitly. Whenever a developer changed, coding and documenting style also changed. Finally, the slight variations have been introduced regularly in the common source code. This is due to the fact that the original design did not, or only to a limited degree, take aspects such as maintainability, extensibility, and reusability into account. Owing to the numerous variations, the component is hard to maintain with keeping high quality and it requires a lot of time to change some part of the component, fix the defects, and add new functionality.

The component under investigation is part of a large number of systems and exhibits both commonalities and variabilities. It is implemented in standard C and the required variability is realized by using conditional compilation, which means that differing parts were enclosed by `#ifdef` macros, and by defining or undefining the macros, appropriate parts of the code are included or excluded. Currently, the component is used in more than 30 different products and consists of about 200 000 lines of source code.

The IMH component plays a center role of image handling in Ricoh's office appliances. From the viewpoint of software productivity, the effort for customizing it product by product will be significantly reduced and the product will be able to ship earlier if all commonality and variability in this component are captured and controlled appropriately. From the viewpoint of software quality, improving this component's quality will lead to a reduction not only in the cost of fixing defects but also in the effort to maintain it. As a result, it is expected that Ricoh's products get more competitive.

## APPROACH

PuLSE<sup>TM</sup>-DSSA is the part of Fraunhofer PuLSE<sup>TM</sup> that deals with activities at the architectural level. PuLSE<sup>TM</sup>-DSSA focuses on the definition and development of architectures for new product lines. Since 'greenfield' scenarios (i.e., situations in which an organization has not built products of a certain type before) [6] are found only rarely in industrial contexts, it 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<sup>TM</sup>-DSSA consists of four phases, as shown in Figure 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. Here, scenario is defined similar to SEI's Architecture Tradeoff Analysis Method (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.

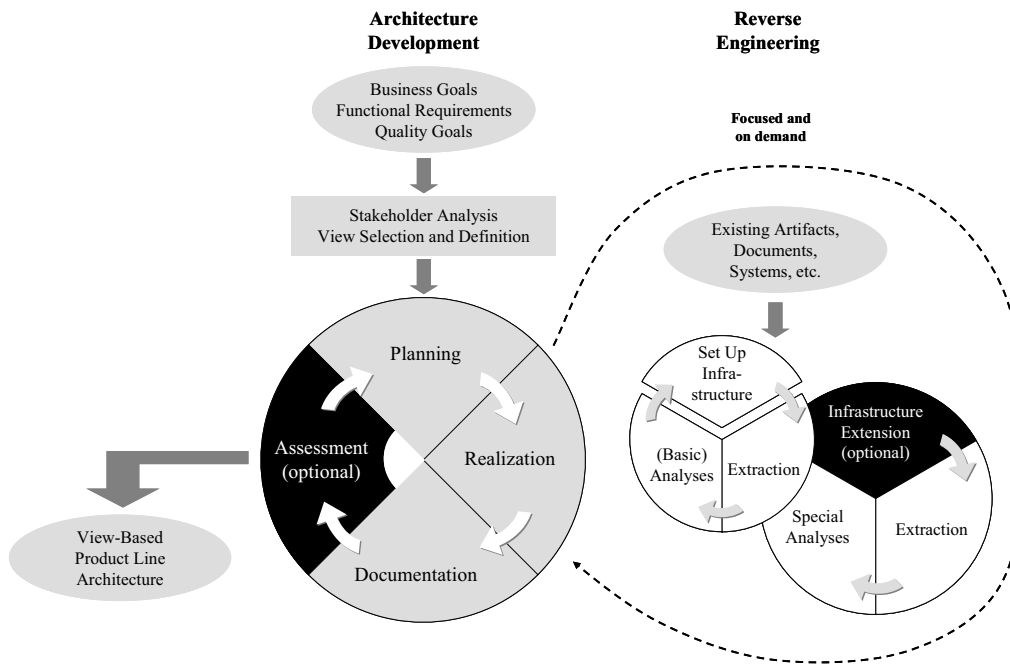


Figure 1. PuLSE™-DSSA process.

- **Realization.** In the realization phase, solutions are selected and design decisions taken in order to fulfill the requirements given by the scenarios. When selecting and applying the selected solutions, an implicit assessment regarding the suitability of the solutions for the given requirements and their compatibility with design decisions of earlier iterations is made.
- **Documentation.** As part of this step, the architecture team creates consistent documentation of the new architecture status by using an organization-specific set of views. If the architecture is an intermediate status, the team may directly return to the first step selecting the next set of scenarios. It may also involve their stakeholders and assess the overall architecture.
- **Assessment.** The goal of the assessment step is to analyze and evaluate the resulting architecture with respect to functional and quality requirements and the achievement of business goals. In an intermediate state of the architecture, this step might be skipped and the next iteration is started. However, the assessment must be performed after the final iteration.

In the realization phase, the architecture team realizes the scenarios and internally defines the architecture. Thereby, often questions whether the architecture may reuse existing components 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 [8]. 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 in the product line; and (c) enabling reuse in order to integrate components (or whole subsystems) into the product line. In PuLSE<sup>TM</sup>-DSSA, reverse engineering is performed asynchronously to the architecture definition. That is, the iteration of the architecture definition may proceed if the answer to the reverse-engineering request is delayed. Then, the request response is handled in one of the following iterations.

The advantage of such a request-driven approach is that investment in reverse engineering is kept as small as possible. In addition, any reverse-engineering activity is directly related to a request from the architecture team and thus has an explicit and clear meaning to activities focusing on the future of an organization. According to our experience from numerous large-scale industry projects this is a clear advantage over the too often implemented approach of first reverse engineering everything from the past before forward engineering is started. Such an approach consequently requires a larger initial investment and it is less clear whether and how it will provide benefits to an organization.

As mentioned earlier, the process documents architectures by using an organization-specific set of views. It thereby relies on standard views as defined, for example, by Kruchten [9] or Hofmeister *et al.* [10], and customizes or complements them by additional aspects requested by one of the key stakeholders [11].

The architectural views are systematically mapped to models used by Fraunhofer's method for engineering component-based product lines [1,12]. The structural view, for instance, maps to the component containment tree, dynamic views map to interaction models of system or subsystem components. According to Fraunhofer's method, components are modeled by using the Unified Modeling Language (UML) and consist of a specification and a realization. Each specification consists of a structural, a behavioral, and a functional model. Each realization consists of a refined structural model, an activity model, and an interaction model. Figure 2 gives an overview on models used.

Figure 3 shows the general approach followed to refactor the existing IMH software component into a reusable product-line component. As the figure shows, there are two separate activities. The 'Document Component' activity deals with the documentation of the design and variabilities of the component. There is a direct feedback between both activities and an expert in the component and the functionality provided by it is involved in the activity. In the activity 'Improve Component', first an analysis of the component is done. Based on the analysis results, an improvement of the design as well as of the implementation are done. Note that there might be additional analyses and subsequent improvements of the component. In case of the IMH software component, two improvement cycles have been done. In the first cycle, first a basic analysis of the component and its quality are performed. Then, based on the analysis results initial refactoring activities addressing major design and implementation problems are carried out. In the second cycle, first the achievements with respect to maintainability and reusability of the performed refactorings are analyzed and further areas for improvement determined.

The activities for the documentation and improvement of the component follow the Quality Improvement Paradigm (QIP) [13,14]. The QIP is a model that defines an incremental view of technology transfer and process improvement that allows a good integration of support activities, such as goal-oriented measurement [15] and knowledge-management activities [16].

The cyclic QIP process is shown in Figure 4. As an initial step the actual organization and its practices are characterized and understood. In the second step, the identified improvement activities are

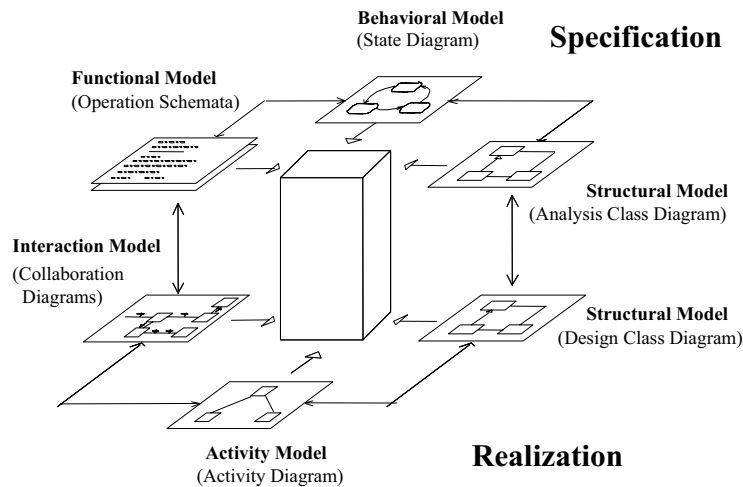


Figure 2. Specification and realization models.

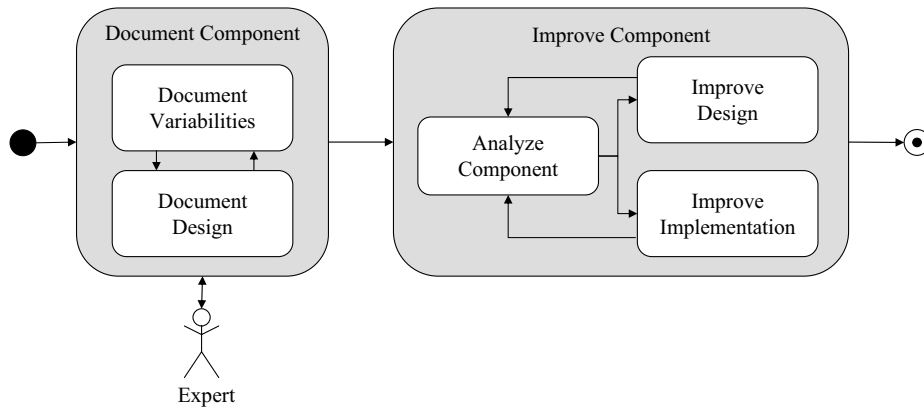


Figure 3. General refactoring approach.

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 set goals 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. The post-transfer analysis of the

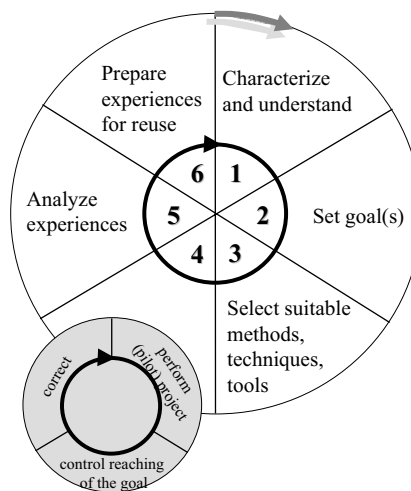


Figure 4. QIP.

organization directly leads to the first step of the subsequent transfer cycle, characterization of the organization.

## DOCUMENTATION

This section presents the approach that has been followed to improve the documentation of the component and briefly presents the results of the activities. The goal of the activities was to improve the existing documentation of the IMH component. In particular, the design specification and documentation of the variabilities realized by the component should be made more complete and up-to-date. Complete and up-to-date documentation is the key to successful reuse and maintenance and evolution of a component. Also, if a system is well documented it can be analyzed more easily.

As a first step, we first checked the available documentation of the component and analyzed which information is available and which is missing. The result of this investigation was that there is a general document available for the IMH component that gives a brief overview of the provided functionality, the role of the component in the overall architecture, and the design of the component. The documentation of the design of the component includes information about the realization by means of processes and threads and the communication between the different threads<sup>§</sup>. In addition, the document provides a summary of the provided interfaces and briefly specifies the externally provided

<sup>§</sup>Processes consist of a number of threads that run concurrently and perform different jobs, such as waiting for events or performing a time-consuming job that the program does not need to complete before going on.



functions and handled notifications. With respect to the variabilities realized by the component, there was no explicit information available. Neither was there information about which functions vary and how they vary, nor was there a complete and up-to-date overview of all the decisions and variabilities. Based on the characterization of the existing documentation and taking into account Ricoh's objectives, concrete goals for improvement of the documentation have then been defined. The major goals for the documentation have been to specify the functional decomposition of the component into subcomponents and to show how the various components interact with each other and realize the required behavior. Furthermore, a variability model consisting of a decision model that lists the decisions realized by the component and a product map that shows the value of the relevant decision variables should be provided for the various products.

### Component design

The documentation of the component design provides detailed information about the structure and behavior of the component and its subcomponents. Specifically, there is a description of the functional decomposition of the component into subcomponents, the interfaces provided by the individual subcomponents, and the relationships between the components. For each provided function, there is a specification that describes the signature, the externally visible effects, as well as constraints and preconditions for use.

The design of the component has been extracted from the existing component implementation and documented using Fraunhofer's method for engineering component-based product lines [1]. According to the approach, a component is documented using a specification and a realization. The specification of a component describes all the properties of a component that are externally visible and hence can be viewed as the interface of a component. The information provided by the specification includes the set of services that the component instances make available to others (commonly called the supplied or server interface), and the set of server instances that the component instances need to acquire (commonly called imported or used interfaces). The specification also defines the behavior of the component instances over time, including the different states that they may exhibit. A component has exactly one component specification, which consists of a structural model, a behavioral model, a functional model, and in the case of a product-line component, a decision model. The purpose of the structural model is to describe any structure of the component that is visible at the interface and the nature of the classes and relationships by which the component interacts with its environment, and also to describe. In the behavioral model, the behavior of the component as a whole in response to external stimuli is shown by means of externally visible states. The model may take one of two forms: one or more UML state chart diagrams and/or state chart tables. The functional model describes the externally visible effects of the operations supplied by the component. The model consists of a set of operation specifications, one for each operation, constructed using the template illustrated in Table I.

The realization of a component describes how the private design of the components realizes the properties defined in the component specification. This includes a description of any server creations performed by the component, and the identification of any subcomponents. The realization also describes how the various server instances (created and acquired) are linked together within the component's instances.

The necessary information have primarily been extracted using 'Understand for C++', a commercial reverse-engineering tool that offers code navigation for C and C++ source code using a detailed cross



Table I. Operation specification template.

Name	Name of the operation.
Description	Identification of the purpose of the operation, followed by an informal description of the normal and exceptional effects.
Constraints	Properties that constrain the realization and implementation of the component.
Receives	Information input to the operation by the invoker.
Returns	Information returned to the invoker by the operation.
Sends	Signals that the operation sends to imported components. These can be events or operation invocations.
Reads	Externally visible information accessed by the operation.
Changes	Externally visible information changed by the operation.
Rules	Rules governing the computation of the result.
Assumes	Precondition on the externally visible state of the component and on the inputs (in receives clause) that must be true for the component to guarantee the post-condition (result clause).
Result	Strongest post-condition on the externally visible properties of the component and the returned entities (returns clause) that become true after execution of the operation with true assumes clause.

reference, and Doxygen, a free tool for extracting the code structure from source files and automatically generating formatted, browsable, and printable documentation. Based on the extracted information, the design documentation has been created by one expert from Fraunhofer IESE over the course of two months. The work focused on creating the structural and functional models of the component and its subcomponents.

### Variability model

For the component, a variability model has been created based on information extracted from the existing source code. The variability model consists of a decision model, which lists the decisions found in the different code files of the component, and a product map. In case of the IMH component, the decisions are the macros used in conditional compilation statements (i.e., `#ifdef` or `#if`). The decisions have been grouped according to product-related decisions, product series related decisions, architecture related decisions, operating system related decisions, IMH specific decisions, and debugging related decisions. The decision model not only provides a good overview of which decisions exist, but also which files are influenced by the decision and which decisions are used in a certain file. In this way, spots of high variability and commonality can be identified in the component. The information can be the basis for improving the structure of the component and the way how variability is realized using variability mechanisms.

The approach used to analyze the component with respect to variability and to create the decision model and product map are shown in Figure 5. The identification of the variability mechanism revealed the use of conditional compilation. Therefore, the detection of variabilities was done by automatically extracting `#ifdef` or `#if` macro uses in the source-code files with the help of scripts or tools such as `ifnames` and `grep`. Once a list of all macros uses and hence potential variabilities exists, relevant

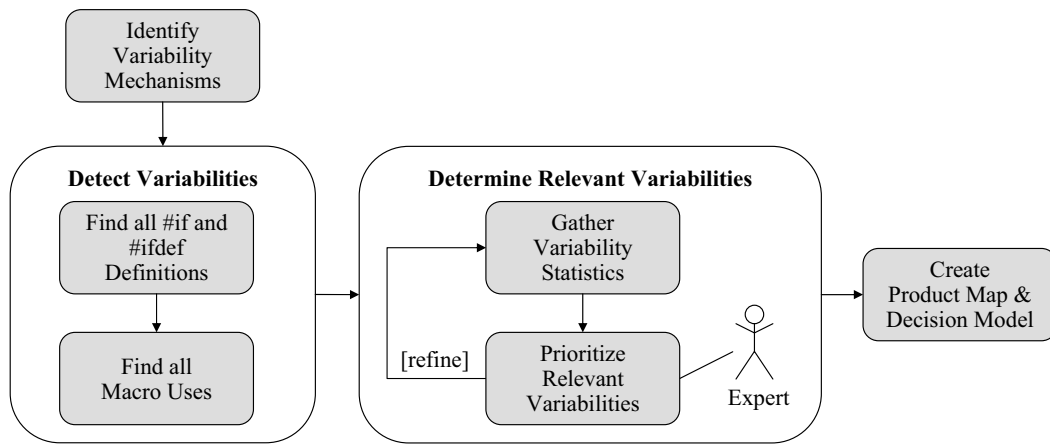


Figure 5. Approach for variability model creation.

Table II. Excerpt of the product map.

	P1	P2	P3	P4	P5	P6	P7	P8	P8	P9	P10
VOUT_QUEUE_ENABLE	—	—	—	×	—	×	×	×	×	×	—
LS_SMALL	—	—	—	—	—	—	—	×	—	—	—
STAMP_ALL_PATTERN_EXIST	—	×	—	—	×	—	×	—	—	—	—
VOUT_OVERLAY_FUNC	—	—	—	—	—	—	×	—	—	—	—
TRAN_400_600	—	—	—	—	—	—	—	—	—	—	—
MULTI_BAND	—	×	—	—	×	—	×	—	—	—	—
SKIP	×	—	×	—	—	—	—	—	—	—	—

variabilities have been determined. This was done with the help of usage statistics, such as the number of occurrences of a macro or the files that are affected most by macros, and by identifying groups of macros and prioritizing variabilities together with an expert. The last step was then the documentation of the decisions and the mapping of decisions to products, and so the construction of a product map. The product map shows the value of the relevant decision variables for the various products. In case of the IMH component, the decision variables correspond to macro names either used in the conditional compilation statements or for the definition of constant values. Table II provides an excerpt of the product map. Note that the names of products have been anonymized. The resulting product map showed some interesting aspects. For example, there are a number of macros that are defined for every product and hence should be considered to be removed from the code. Also, there are macros that are only defined for one particular product. Finally, there are macros that are never defined for a product.

In Table III, a part of the decision model that has been extracted from the source code is shown. The table shows which decisions exist as well as which files are influenced by the decision and which decisions are used in a certain file.



Table III. Example of a decision model.

	imh.h	imh_sys.h	mode_macro.h	imh_main.c	imh_proc.c	imh_proc_ctrl.c	imh_util.c	imh_util_ctrl.c	imh_hsm.c
GW_COLOR_COPY	—	—	×	—	—	×	—	—	—
GW_COLOR_MACHINE	—	×	—	—	—	—	—	—	—
GW_TWIN_COLOR_COPY	—	—	×	—	—	×	—	—	—
GW_TWIN_COLOR_MACHINE	—	×	—	—	—	—	—	—	—
IMH_COLOR	×	—	×	—	—	×	—	×	—
IMH_COLOR_1DRUM	—	—	×	—	×	—	—	—	—
IMH_COLOR_1DRUM_MFP	—	—	—	—	—	—	—	—	—
IMH_COLOR_4DRUM	—	—	×	—	×	×	—	—	—
IMH_COLOR_4DRUM_LP	—	—	—	—	—	—	—	—	—
IMH_COLOR_4DRUM_MFP	—	—	×	—	—	—	—	—	—
IMH_COLOR_LP	—	—	—	—	—	—	×	—	—

## COMPONENT ANALYSIS

In this section, the analysis of the component with respect to selected quality attributes is discussed. To get an understanding of the current implementation and to detect potential problems and improvement areas, the implementation, and hence the source code, of the component has been analyzed using a number of different techniques. The goal of the analyses was to define objective goals for improvement. This section presents the approach and gives a brief overview of the applied analysis methods and techniques.

### Metrics

Software metrics are an unbiased means to objectively qualify the source code of a component or application. Typical software metrics are the size of the code (measured in lines of code, number of statements, and so on) and the code complexity (measured through complexity figures such as the Cyclomatic complexity). The measurement of source code provides useful information for the assessment of its quality, predicting to some extent the external system quality characteristics, such as maintainability, reliability, extensibility, and portability.

In order to get an understanding of the quality of the component implementation, a number of source-code metrics have been used. To select the metrics, the Goal Question Metric (GQM) approach was used [15]. GQM is an approach that determines the metrics that allow a quantitative analysis as to what extent a goal has been reached. A goal is formulated in a structured and explicit way, and then



Table IV. Applied software metrics.

Metric	Category	Description
Cyclomatic complexity	Function	A measure of the number of decisions in the control flow of a function. It is calculated as the number of decisions plus 1.
Number of lines of code	Module, Function	The total number of lines in a module or function. Blank lines and comment lines are included.
Number of maintainable lines of code	Module, Function	The total number of lines in a module or function without blank lines and comment lines.
Percentage of comments	Module, Function	The ratio of comment lines in the code to all lines and hence a quality indicator for how much of the code is commented.
Fan-in	Function	The number of distinct functions calling a certain function.
Fan-out	Function	The number of distinct functions called within a function. Functions with a large number of function calls are more difficult to understand because their functionality is spread across several components.
Cumulated data structure members	Structures	The total number of members in a structure cumulated over its substructures. If there is a data structure with five members, one of it being of the type of another data structure with four members then there are nine data structure members in total.
Associations with other structures	Structures	The number of associations a structure has directly or indirectly with other structures.
Maximum nesting	Function	The maximum nesting level of control constructs.

questions are formulated that being answered give an indication of goal fulfillment. Finally, metrics are collected for each question that allow the question to be answered in a quantitative way. As a result of the GQM analysis, the metrics summarized in Table IV have been selected. The category describes to which implementation element the metric is applied. Metrics are applied at the module, function, and structure level. A *module* or *file* corresponds to a physical implementation file whereas a *component* is a logical group of one or more related implementation files realizing a coherent set of functionality. Note that the counts of Fan-in and Fan-out are regarded as a measure of the structural quality of a program, with high values of either (and particularly high values of both within the same module) indicating increased risk of changes required in one module requiring changes across other modules.

For each of the metrics, a comparison against threshold values and acceptable ranges as proposed by literature (e.g., [17–20]) and best practices was made. The comparison of measured values for metrics to the acceptable ranges can lead to a representative view of the quality of the analyzed source code. 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 outliers indicate areas where further analyses such as manual code reviews should be performed.

The analysis of the metrics showed that many files, functions, and data structures exceed the suggested threshold values and acceptable ranges. In particular, the number of maintainable lines of code per function, maximum nesting, and Cyclomatic complexity are very high for a large number of functions. For most of the files, the number of maintainable lines of code was above 2000 and there



have also been a large number of files with more than 5000 lines of code. This indicates problems with respect to understandability of the code and in consequence maintainability and extensibility. For Fan-in and Fan-out, top values of 391 and 68, respectively, have been measured.

In addition to the traditional metrics, the source code was analyzed with respect to compile-time and link-time dependencies using a set of tools from Lakos [21]. Among others, the tools enable to find cyclic dependencies among implementation files and to calculate a metric called Cumulative Component Dependency (CCD). The CCD of a system or subsystem is defined to be the sum of the Component Dependencies (CDs) for each component in that system. Note that here a component is a physical implementation file. The CD is the number of components needed in order to use a given component. This metric characterizes the internal coupling of components within a subsystem. The value of CCD will range from  $N$  to  $N^2$  where  $N$  is the number of local components. A CCD value of  $N$  implies a horizontal subsystem of independent components. A value of  $N^2$  implies a completely interdependent subsystem. A value such as  $N \times \log(N)$  would suggest a tree-like dependency graph. In general, a lower CCD indicates a less tightly coupled, more flexible, more understandable subsystem in which the components can be tested and reused more independently. For the IMH component the number of components has been determined to be 78. Hence, the ideal value for CCD would be  $78 \times \log(78)$  or 148, but we have calculated a value of 1143. Based on the CCD, another metric called Average Component Dependency (ACD) [21] can be calculated. For the IMH component, this metric has a value of 15; that is each component depends on average on 15 other components.

### Variability analysis

The IMH component uses conditional compilation to realize variability. In conditional compilation, variable code parts are differentiated from the common parts by enclosing them in `#ifdefs`. In this way, variability is managed at a medium level of scale by controlling the inclusion of lines of code. As part of the analysis, all uses of the variability mechanism have been detected by statically analyzing the source code using the ifnames Unix tool. This tool enables one to detect all macros that are used in `#if`, `#ifdef`, `#ifndef`, `#elif`, and `#if` defined preprocessor statements. In total, 309 different macros were detected in all `.c` and `.h` files of the IMH component. According to their names, however, only a portion of them are used for variability purposes, whereas many others serve development aspects such as debugging or are used as include guards to prevent multiple header inclusion. Furthermore, many `#if 0` or `#if 1` preprocessor statements have been found in the source code.

In general, there is a high total use of macros in the source code. The problem of a heavy use of conditional compilation is that it can overwhelm a human trying to understand the code. Owing to the limited understandability of the code, maintainability and reusability is low. The main problems regarding the use of macros for the realization of variabilities in the IMH component are the lack of consistency in definition and use. According to the macro usage statistics, which are shown in Table V, many macros are used in a large number of different implementation files and numerous macros are used in each file. The macros in the IMH component are mainly used to realize optional and alternative variabilities. In addition to the use of macros, different implementation files are used. These source-code files implement the same interface and are used during the build process depending on the selected product configuration.



Table V. Variability statistics.

Macro	Total number of uses	Percentage of files used	Number of files used
SPEC_INJI	396	42	39
DEV_HDD_EXIST	266	19	18
UNI_DRV	234	34	32

### Clone detection

The goal of this analysis was to detect code clones in the source code and to analyze those clones. Code clones are defined to be fragments of the source code that are structurally or syntactically similar or identical to each other. One of the segments is usually a copy of the other, perhaps with minor changes. Often, code cloning happens due to code reutilization by just copying existing solutions. For example, developers may copy and paste code. Code cloning, the gratuitous duplication of source code within a software system, is a common practice in software development. However, various problems are associated with code duplication, including increased code size and increased maintenance costs. Source codes with code clones are more difficult to maintain than codes without [22,23]. In particular, errors and bugs may be copied as well and errors can be difficult to fix as they are not located in just one place.

Several methods exist for detecting code clones in software, such as simple string matching, using statistical fingerprints of code segments, function metrics matching, parameterized string matching, and program graph comparison [24,25]. In the case study, the clone detection was performed using a tool developed by Fraunhofer IESE. This tool is based on text pattern matching and can therefore be used independently of the applied programming language. The tool enables one to specify the number of lines of code that have to be identical. For the IMH component, we detected only clones with 25 or more identical lines of code. By specifying a smaller number of lines, too many clone pairs would have been detected. Both internal and external clones have been detected. Internal clones are duplicated lines of code that are found in a single file, whereas external clones are found by comparing different files. The tool also enables one to detect multiple instances of the same code fragment in one file and across multiple files.

The result of the clone detection was an overview of source files and the clones found in the files, as illustrated in Tables VI and VII. For each original source-code file, all of the files that contain cloned code and the number of clone pairs is provided. In addition, the size of the clone pair and information about the location of code clones inside a file are provided.

### COMPONENT IMPROVEMENT

This section describes the approach that has been followed to improve the existing design and implementation of the component. The improvement activities have been determined based on the results of the component analysis and the goals set for the component.



Table VI. Example of external clone statistics.

Original file	File with cloned code	Number of clone pairs
job_proc_func.c	job_proc_func.c	35
mrm_page_color_1drum.c	mrm_page_color_4drum.c	14

Table VII. Example of internal clone statistics.

Clone size	File name	Original start	Clone start
146	mrm_page_color_4drum.c	1512	3356
125	mrm_page_color_1drum.c	1660	2718

As the results of the component analysis showed, the existing implementation of the IMH component has some problems regarding maintainability, extensibility, understandability, and reusability. In particular, the implementation is very hard to understand, which makes it difficult to make changes, add new features, or to reuse the component. This is mainly because the component is very complex, is not structured optimally, lacks up-to-date and complete documentation, and has been changed and extended to accommodate the features required by various products. Due to many enhancements that the code has undergone, it showed clear signs of degradation.

The primary goal was to improve the maintainability and reusability of the component without changing or breaking existing functionality and without deteriorating performance. In other words, the aim was to alter the internal code structure without changing the code's external behavior. In detail, the understandability, changeability, and usability of the component should be improved. Furthermore, the realization of variability in the code should be improved.

As mentioned earlier, the improvement of the IMH component was done iteratively in two cycles. The goal of the first cycle was to resolve major design and implementation problems which had been detected by the different analyses. Then, in the second cycle, the achievements with respect to maintainability and reusability of these refactorings are analyzed and further areas for improvement are determined. The refactorings in the second cycle have been focused on a selected subcomponent and targeted specific problem areas whereas in the first cycle primarily basic refactorings of the whole component have been performed. In the following, the activities performed in the first improvement cycle (Basic refactorings) and the second cycle (Advanced refactorings) are discussed.

### Basic refactorings

The overall goal of the implementation improvement activities was to improve the existing implementation of the IMH component with respect to maintainability and reusability by changing the existing code structure. All changes to the code structure should not break existing functionality



and hence should be semantics-preserving. Also, the changes should have no (or no significant) performance impact. To achieve this, we followed the following three-step approach.

- (1) Perform automatic improvements of the whole code. As part of this first step, the conditional preprocessor statements have mainly been changed.
- (2) Improve selected aspects of the complete code manually. Examples of refactorings performed include the renaming of files and functions, the splitting of long files, the moving of functions from one module to another, the conversion of macros to inline functions, and the changing of data types.
- (3) Improve the design and implementation of a number of selected subcomponents. In this final step, sequences of small code transformations have been applied to components that have been identified during the analysis to be very complex and/or risky.

In the following, some of the improvement activities and the performed code transformations are presented. The existing IMH component contains a lot of variability that has been realized by means of conditional compilation and hence the use of macros. There were some inconsistencies between the variability in header files and source files. As part of the improvement activities, we looked at the variability in the source files and checked whether it matches with the header files. If not, either the header or source file has been updated by introducing corresponding preprocessor statements. For all macros that are used by IMH exclusively, we introduced a common naming scheme. We also wanted to introduce a common scheme for realizing variability. In particular, we wanted to use `#if` instead of `#ifdef` or `#if defined`. As many macros are also used by other components, however, we could not change the definition and preprocessor statements without breaking functionality and so it was not possible to come up with one common `#if` scheme and a common macro naming.

In the original implementation of the IMH component, macros were commonly used to mimic functions without the overhead of a function call. Macros that are used to implement functions are a persistent source of bugs in C programs, as they may not behave like the intended function when they are invoked with certain parameters or used in certain syntactic contexts. Therefore, function macros have been refactored to inline functions in order to avoid the problems associated with macros. The potential performance loss of inline functions was negligible and, due to the many other benefits, acceptable.

The existing header files have been very large and there was no clear separation of concerns. As a consequence, almost all source code files had to include all header files. In order to improve this situation, we introduced a header file for common data types (`imh_types.h`) and one header for common definitions (`imh_defs.h`). Also, separate header files for the various subcomponents have been introduced.

As part of the improvement of the implementation structure, the product configuration has also been improved. In particular, a separate file `imh_conf.h` and individual product configuration files have been introduced, as illustrated in Figure 6. The product configuration files, which are placed in a folder 'conf', are named according to the respective product and contain all relevant definitions for the product. In the file `imh_conf.h`, the respective configuration file is loaded.

In addition to a restructuring of the source code, common language problems have also been identified and resolved as part of the implementation improvement activities. The activities performed included, among others, the resolving of cyclic header inclusions, the removal of redundant and obsolete header inclusions, removal of unreachable code, the removal of unused or obsolete functions

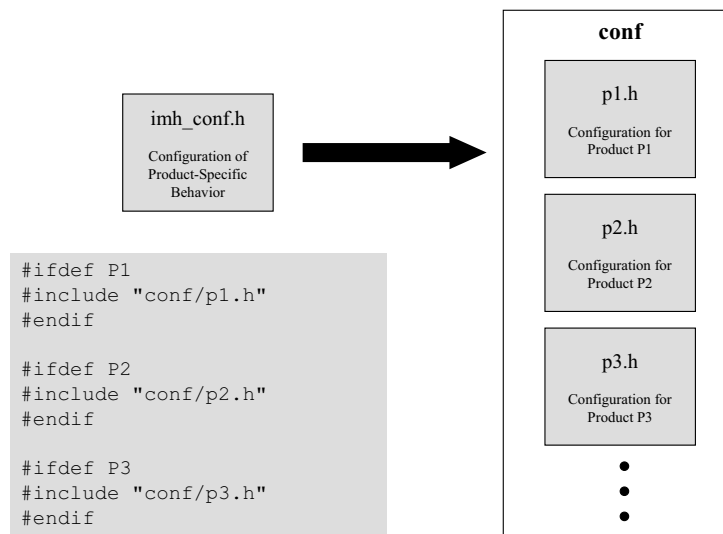


Figure 6. Product configuration.

and variables, the removal of dependencies on external variables, the reduction of ‘extern’ statements in implementation files, and the transformation of functions that are only used internally into static functions.

In addition to the implementation improvement activities, some design improvements have also been made. The goal of the design improvement activities was to improve the modularity and usability of the IMH component and its subcomponents. As part of the improvement activities, the following major design changes have been performed: splitting of components into distinct subcomponents, introduction of new components, and factoring out functionality of components. This was mainly done because in the existing design related functions are not clearly separated into individual subcomponents and there was no clear separation of concerns. Note that due to the confidential nature of the design and the changes performed to it no detailed information can be provided in this article.

### Advanced refactorings

The results of the basic refactoring of the component were very promising. Due to the limited time and effort, however, not all possible improvements of the design and code could be performed. Nevertheless, the quantitative data collected at the end of the refactoring cycle showed a significant improvement of the component with respect to maintainability and reusability. In particular, the values for the selected code metrics showed substantial positive changes. Despite a number of changes to the code, the external behavior of the component had not been changed and the semantics of existing functionality had been preserved as shown by tests performed with the component on two different copier machines. Therefore, it was decided to continue with the refactoring activities. Based on an



analysis of the results of the basic refactorings and the goals of Ricoh with respect to the component, additional refactoring activities have been determined. As the analysis results showed a large number of code clones and problems in terms of code metrics in a subcomponent called memory resource manager and this component is very critical for the realization of different products in the product line of Ricoh, it was decided to focus the refactorings exclusively on this subcomponent.

In the original version of the IMH component, the memory resource manager component existed in four different implementations for the different types of copiers and printers. In particular, the processing of pages in the case of color machines and monochrome machines was distinguished. All implementations realize the same interface and similar functionality. In contrast to the majority of the other subcomponents and implementation files of the IMH component, no conditional compilation has been used to realize the required variability. Depending on the product currently built, the makefile selects the appropriate files and compiles and links it. As was shown by the clone detection, there are many code clones in these four implementations. Therefore, the following refactorings have been done.

- *Removal of internal and external code clones.* The goal of this activity was to remove the code clones detected in the various implementations of the subcomponent. The code clones have been removed manually by introducing internal or external functions for commonly used code fragments. Depending on the size of the functions, they have been realized as function macros, inline functions, or normal functions. Note that the removal of code clones and the introduction of functions for code clones has been done in conjunction with a merging of the different implementations. In particular, the functions have been made variant using conditional compilation if necessary.
- *Merging of the different implementations and realization using conditional compilation.* As a basis, the existing implementations in which the code clones had been removed were used. The first step was the definition of a new header file defining the interface of the subcomponent. Then, the realization of this interface was done based on the existing implementations. The required variability was determined by looking at the makefile and the product or feature dependencies defined there for the files. For variabilities, conditional compilation statements have been added in the code. As the major functions used a large number of helper functions, additional files for these utility functions have been created.
- *Reduction of the scale and complexity of functions.* Functions identified during the component analysis to be too complex (i.e., in terms of Cyclomatic complexity) or too large (i.e., in terms of lines of code per function) should have been split or refactored in order to reduce complexity and so improve maintainability and understandability.

The aforementioned refactorings have been limited to a small part of the IMH component. Therefore, they could be done manually within reasonable time (i.e., six weeks) and effort. In fact, the refactorings have been done by one student working full-time under the supervision of an experienced researcher. The refactorings have been selected as they could be done without detailed knowledge of the domain and the functionality.

## ANALYSIS AND DISCUSSION

This section discusses the quality improvements of the component that have been achieved compared with the original implementation. Also, it presents some lessons learned and experiences.



Table VIII. Comparison of selected metrics.

Metric	Original code	After first improvement cycle	After second improvement cycle	Change to original
Total files	93	194	196	110.75%
Source files	71	88	89	25.35%
Header files	22	106	107	386.36%
Ratio of C files with corresponding header	0.15	0.78	0.81	440.00%
Total lines of code	199 572	186 800	183 305	−8.15%
Average file size (lines of code)	2146	963	935	−56.43%
Maximum function size (lines of code)	3201	3104	668	−79.13%
Maximum Cyclomatic complexity	122	83	66	−45.90%

The use of tools for extracting architecture and design information from source code provided a good basis for understanding the component and creating a design document. The problem with the tools, however, is that the results are very detailed and much effort is required to abstract and consolidate the information. Furthermore, expert knowledge is important to check the results of the tools and to improve the design document with information that cannot be extracted from the source code. The tools that have been used to extract variability information worked quite well and provided reliable data. Despite the large size of the component source code, no scalability issues have been encountered with the tools used.

Table VIII provides a comparison of some selected metrics. It compares the original code of the IMH component with the version after the first improvement cycle and the version after the second improvement cycle. Compared with the original code, the size of the component was reduced by 8.15% after the improvement cycles. Furthermore, the maximum function size in terms of lines of code was reduced by 79% and the average file size by 56%. Finally, the maximum Cyclomatic complexity was reduced by 46%.

As mentioned earlier, the refactorings of the second improvement cycle focused on a small part of the IMH component, namely the memory resource manager. Table IX gives an overview of this subcomponent as it was after the first improvement cycle and as it is now after the second improvement cycle. With respect to the total size of code, there is a reduction of 20%. This is mainly due to the removal of code clones. On the other hand, the number of preprocessor line increased by 69% because of the introduction of conditional compilation. In addition to the total number of lines of code, the number of lines of code per file and the average number of lines per function have also been decreased. Finally, the complexity as measured by Cyclomatic complexity has been reduced in average by 43% and in total by 18%. This was done manually by modifying the most critical functions.

Using the results of the clone detection and the results of the clone removal in the memory resource manager component, it is estimated that by just removing all clones the size of the IMH component can be further reduced by around 15%.

As a result of the analysis and refactoring activities, the documentation and implementation of the component has been considerably improved. Due to the limited time and effort available for the



Table IX. Comparison of resource manager component metrics.

Metric	After first improvement cycle	After second improvement cycle	Change to original
Total files	4	7	
Average Cyclomatic complexity	5.25	3	−43%
Total Cyclomatic complexity	237	194	−18%
Average file size (lines of code)	486	129	−73%
Total lines of code	14 730	11 729	−20%
Total lines of preprocessor code	538	910	+69%
Maximum function size (lines of code)	3104	668	−78%

refactoring, however, the component documentation is not complete and not all possible improvements of the design and code could be performed. In particular, not all detected code clones have been removed and changes to reduce the Cyclomatic complexity of functions or to improve the data structure metrics have been limited to a small part of the component. Nevertheless, the quantitative data collected at the end of the two improvement cycles shows a significant improvement of the component with respect to maintainability and reusability. For the changes of the design, it was difficult to present quantitative data that show the benefits of the improvement.

Despite a number of changes to the code, the external behavior of the component has not been changed and the semantics of existing functionality has been preserved. This has been checked by testing the improved component on two different products (i.e., one monochrome machine and one color machine). The improvements regarding reusability of the component could not be checked so far, as currently no new product that makes use of the component is under development. The case study indicated that software refactoring is sufficiently fail-safe and allows to existing code to be cleaned up with only little risk of introducing bugs, if applied correctly. The key is to apply a sequence of small code-transformation steps that together achieve the desired change. It is critical to change code incrementally and to order refactoring steps properly. The problem, however, is to find arguments that the code transformation is semantics-preserving and does not degrade performance. In the case of the IMH component, considerable testing effort has been invested in the past, which in turn provided an increasing level of confidence in the code. Since there is no base of automated regression tests for the target component, however, any change to the code will diminish confidence. Thus, it is very important to change code incrementally, to order refactoring steps properly, and validate each step (e.g., follow the mechanics of a refactoring). In particular, it is important for future projects to have short coding-test cycles and to apply automated tests that check the results of a refactoring step.

## RELATED WORK

Early work on refactoring dealt with refactoring object-oriented software. In his thesis [26], Opdyke characterized a refactoring as a behavior-preserving transformation, categorized major refactorings of several levels of scale, together with the necessary pre- and post-conditions that make each refactoring behavior-preserving. Building on this work, the first tool support was realized in the Smalltalk refactoring browser [27,28]. Current tool support is often integrated into IDEs, such as the Java



refactoring support in IntelliJ IDEA [29], Java or C++ support in Eclipse [30] or Emacs [31], or C# support in Visual Studio .NET [32,33].

Fowler's catalog of object-oriented refactorings [34] raised interest in refactorings for a wider audience; it was adopted as a major practice in agile approaches [35].

The problems of refactoring programs written in C, a language that is still among the major languages used in practice for implementing embedded systems, have only been addressed recently [36]. Among others, the presence of a preprocessor, used for conditional compilation, complicates refactoring issues considerably, as observed by several authors [36,37].

An overview of refactoring support for a number of industry relevant programming languages, such as C, C++, Java, Smalltalk, Fortran, Cobol and Lisp, is given in [38].

Other major work on refactoring dealt with the evolution of software designs with refactorings [39], or combining refactoring and patterns [40]. More related work can be found at [41].

Work on plagiarism detection [42] preceded software clone detection. Most recent work deals with categorizing software clones [25], and identifying crosscutting concerns [24], an important issue for product-line implementation. There is also a tendency towards language-independent clone detection, as summarized in [43].

The DMS program transformation tool has been used for both clone detection [44] and managing conditional compilation. Other recent tool support for clone detection has been presented in [45].

## CONCLUSIONS

This article presented a case study in systematically refactoring an existing software component for reuse in a product line. The article has described the activities performed to migrate the IMH of Ricoh's current products of office appliances into a reusable product-line component. In particular, the article provided an overview of the activities done while redocumenting and redesigning the component applying Fraunhofer PuLSE<sup>TM</sup>. Also, it outlined the systematic analysis of the component that provided the basis for the systematic improvement of the component with respect to maintainability and reusability and hence suitability for use in a product line. Finally, it has provided a discussion of the activities performed to improve the existing implementation and has presented the results of a comparison of the improved component with the original.

## REFERENCES

1. Atkinson C, Bayer J, Bunse C, Kamsties E, Laitenberger O, Laqua R, Muthig D, Paech B, Wust J, Zettel J. *Component-based Product Line Engineering with UML*. Addison-Wesley: Reading MA, 2001; 464 pp.
2. Clements P, Northrop LM. *Software Product Lines: Practices and Patterns*. Addison-Wesley: Reading MA, 2001; 563 pp.
3. Weiss DM, Lai CTR. *Software Product-line Engineering: A Family-based Software Development Process*. Addison-Wesley: Reading MA, 1999; 448 pp.
4. Kolb R, Muthig D, Patzke T, Yamauchi K. A case study in refactoring a legacy component for reuse in a product line. *Proceedings IEEE International Workshop on Software Maintenance (ICSM 2005)*, September 2005. IEEE Computer Society Press: Los Alamitos CA, 2005; 369–378.
5. Bayer J, Flege O, Knauber P, Laqua R, Muthig D, Schmid K, Widen T, De Baud J-M. PuLSE: A methodology to develop software product lines. *Proceedings of the 5th ACM SIGSOFT Symposium on Software Reusability (SSR'99)*, May 1999. ACM Press: New York NY, 1999; 122–131.
6. Böckle G, Clements P, McGregor JD, Muthig D, Schmid K. Calculating ROI for software product lines. *IEEE Software* 2004; **21**(3):23–31.
7. Kazman R, Klein M, Barbacci M, Longstaff R, Lipson H, Carriere SJ. The architecture tradeoff analysis method. *Technical Report CMU/SEI-98-TR-008*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, 1998; 11 pp.



8. Chikofsky E, Cross JH. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 1990; **7**(1):13–17.
9. Kruchten P. The 4 + 1 view model of architecture. *IEEE Software* 1995; **12**(6):42–50.
10. Hofmeister C, Nord R, Soni D. *Applied Software Architecture*. Addison-Wesley: Reading MA, 1999; 397 pp.
11. Bayer J. View-based software documentation. *PhD Thesis*, Fraunhofer IRB Verlag, Stuttgart, Germany, 2004.
12. Muthig D, Atkinson C. Model-driven product line architectures. *Proceedings of the 2nd Software Product Line Conference (SPLC2)*. Springer: Berlin, 2002; 110–129.
13. Basili V, Green S. Software process evolution at the SEL. *IEEE Software* 1994; **11**(4):58–66.
14. McGarry F, Pajerski R, Page G, Waligora S, Basili VR, Zelkowitz MV. Software process improvement in the NASA Software Engineering Laboratory. *Technical Report CMU/SEI-94-TR-22*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, 1994; 80 pp.
15. Basili V, Calidera G, Rombach D. The goal/question/metric paradigm. *Encyclopedia of Software Engineering*, vol. 1, Marciniak J (ed.). Wiley: New York NY, 1994; 528–532.
16. Basili V, Calidera G, Rombach D. The experience factory. *Encyclopedia of Software Engineering*, vol. 1, Marciniak J (ed.). Wiley: New York NY, 1994; 469–476.
17. Brandl DL. Quality measures in design. *ACM Sigsoft Software Engineering Notes* 1990; **15**(1):68–72.
18. McCabe T. A complexity measure. *IEEE Transactions on Software Engineering* 1976; **2**(4):308–320.
19. McCabe TJ, Butler CW. Design complexity measurement and testing. *Communications of the ACM* 1989; **32**(12):1415–1425.
20. Software Assurance Technology Center. Recommended thresholds for non-OO languages. [http://satc.gsfc.nasa.gov/metrics/codometrics/non\\_oo/c/index.html](http://satc.gsfc.nasa.gov/metrics/codometrics/non_oo/c/index.html) [28 February 2006].
21. Lakos J. *Large-Scale C++ Software Design*. Addison-Wesley: Reading MA, 1996; 896 pp.
22. Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. *Proceedings IEEE International Conference on Software Maintenance (ICSM 1999)*. IEEE Computer Society Press: Los Alamitos CA, 1999; 109–118.
23. Kapser C, Godfrey MW. Toward a taxonomy for source code cloning. *Proceedings of the 1st International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*, 2003; 67–78. Available at: <ftp://ftp.umh.ac.be/pub/ftp-infofs/2003/ELISA/proceedings.pdf> [8 March 2006].
24. Bruntink M, Deursen A, Tourwe T. An evaluation of clone detection techniques for identifying crosscutting concerns. *Proceedings International Conference on Software Maintenance (ICSM 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 200–209.
25. Kapser C, Godfrey MW. Aiding comprehension of cloning through categorization. *Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 85–94.
26. Opdyke W. Refactoring object-oriented frameworks. *PhD Thesis*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992; 206 pp.
27. Roberts DB. Practical analysis for refactoring. *PhD Thesis*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.
28. Roberts D, Brant J, Johnson R. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems* 1997; **3**(4):253–263.
29. JetBrains. IntelliJ—the most intelligent IDE. <http://www.jetbrains.com/idea> [28 February 2006].
30. Eclipse Foundation. Eclipse project overview. <http://www.eclipse.org> [28 February 2006].
31. XRef-Tech—a C/C++ refactoring browser for Emacs and XEmacs. XRefactory. <http://xref-tech.com> [28 February 2006].
32. JetBrains. ReSharper—the most intelligent add-in to VisualStudio.NET. <http://www.jetbrains.com/resharper> [28 February 2006].
33. Xtreme Simplicity. C# refactory tool. <http://www.xtreme-simplicity.net> [28 February 2006].
34. Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Reading MA, 1999; 464 pp.
35. Beck K, Andres D. *Extreme Programming Explained*. Addison-Wesley: Reading MA, 1999; 224 pp.
36. Garrido A, Johnson R. Refactoring C with conditional compilation. *Proceedings 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, 6–10 October 2003. IEEE Computer Society Press: Los Alamitos CA, 2003; 323–326.
37. Spinellis D. Global analysis and transformation in preprocessed languages. *IEEE Transactions on Software Engineering* 2003; **29**(11):1019–1030.
38. Mens T, Touwe T. A Survey of software refactoring. *IEEE Transactions on Software Engineering* 2004; **30**(2):126–139.
39. Tokuda L. Evolving object-oriented design with refactorings. *PhD Thesis*, University of Texas, Austin TX, 1999.
40. Kerievsky J. *Refactoring to Patterns*. Addison-Wesley: Reading MA, 2004; 400 pp.
41. Fowler M. Refactoring homepage. <http://www.refactoring.com> [28 February 2006].
42. Aiken A. A system for detecting software plagiarism. <http://www.cs.berkeley.edu/~aiken/moss.html> [28 February 2006].
43. Rieger M. Effective clone detection without language barriers. *PhD Thesis*, University of Bern, Switzerland, 2005; 191 pp.



44. Baxter ID, Yahin A, Moura L, Sant' Anna M, Bier L. Clone detection using abstract syntax trees. *Proceedings International Conference on Software Maintenance (ICSM 1998)*. IEEE Computer Society Press: Los Alamitos CA, 1998; 368–377.
45. Burd E, Bailey J. Evaluating clone detection tools for use during preventative maintenance. *Proceedings 2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*. IEEE Computer Society Press: Los Alamitos CA, 2002; 36–43.

#### AUTHORS' BIOGRAPHIES



**Ronny Kolb** works as an applied researcher in the Product Line Architectures (PLA) department at the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern. He works in several research and industrial projects in the context of product-line engineering and has experience in transferring product-line engineering into practice. His main research interests are quality assurance for software product lines and the design and analysis of product-line architectures.



**Dirk Muthig** heads the PLA department at the Fraunhofer IESE. He has been involved in the definition, development, and transfer of Fraunhofer PuLSE (Product Line Software Engineering) methodology since 1997. He received a diploma in computer science, as well as a PhD, from the Technical University of Kaiserslautern.



**Thomas Patzke** is an applied researcher at IESE's PLA department. His research interests are product-line implementation technologies and software evolution approaches.



**Kazuyuki Yamauchi** is a software architect for office appliances at Ricoh Company, Ltd. Since the collaborative project with Fraunhofer IESE started he has been in charge of transferring product-line technology to several software development departments at Ricoh.