# Evolutionary Introduction of Software Product Lines

Daniel Simon and Thomas Eisenbarth

Universität Stuttgart
Breitwiesenstrasse 20–22
70565 Stuttgart, Germany
simon@informatik.uni-stuttgart.de
eisenbarth@informatik.uni-stuttgart.de

**Abstract.** Software product lines have proved to be a successful and efficient means for managing the development of software in industry. The significant benefits over traditional software architectures have the potential to convince software companies to adopt the product line approach for their existing products. In that case, the question arises how to best convert the existing products into a software product line. For several reasons, an evolutionary approach is desirable. But so far, there is little guidance on the evolutionary introduction of software product lines.
In this paper, we propose a lightweight iterative process supporting the incremental introduction of product line concepts for existing software products. Starting with the analysis of the legacy code, we assess what parts of the software can be restructured for product line needs at reasonable costs. For the analysis of the products, we use feature analysis, a reengineering technique tailored to the specific needs of the initiation of software product lines.

## 1 Introduction

Suppose a company $C$ is developing and selling a software intensive product $P$. Product $P$ is very successful on the market, customers use $P$ and pay for maintenance and support. Because of $P$'s success, $C$'s marketing division wants to develop variants of $P$ that are tailored to customer-specific requirements. Being aware of anticipated future features of the software, the company's software engineers consider a software product line approach for $P$ and its relatives. The general question now arises: is $P$ product-line ready? If so, how can a software product line be best introduced?

According to Bosch [1], the initiation of a software product line has two relevant dimensions. On one hand, the organization may either take an evolutionary or a revolutionary approach to introduce product line concepts. On the other hand, the introduction of a new product line is different from the application of the approach to an existing product (resp. a set of products).

For the introduction of product line development, changes within the company's organization are due. These depend on several factors, such as the size and number of products, the staff involved, or the organizatorical culture. For a further discussion of organizatorical topics, we refer to [1] and to the framework provided in [2].

In parallel to organizatorical changes of the company, the software development is subject to changes. In cases where there are existing products, the mining of legacy assets for product lines helps protecting investments and the domain competence encoded in the software. There are several frameworks addressing the migration and reengineering for product lines. The Option Analysis Reengineering (OAR) [3] provides a framework for the mining of components for a software product line. Similarly, the PuLSE [4] methodology for the development of product lines incorporates migration steps. Both OAR and PuLSE provide means for the revolutionary introduction of product lines but do not fully support incremental and gradual transitioning.

In the case of company $C$, there is a number of reasons *not* to take the revolutionary approach. Instead, an evolutionary introduction can bring strategic advantages for the business. Some reasons for the application of an evolutionary approach are the following:

- The software developers have detailed knowledge about the structure of the software and its current architecture, but it is not obvious whether the product line approach is feasible.
- The software developers have no experience with software product lines and they are not used to software product line concepts. A disruptive switch-over to the new technology bears a high risk of failure.
- The product is available, used by customers, and needs continuous maintenance and support. In contrast to a revolutionary approach, the evolutionary introduction allows the simultaneous maintenance with little overhead.
- By using an evolutionary migration process, developers can "learn on the job". They get used to product line concepts, and have a better idea of how the product $P$ can be seen from a product line perspective.
- The changes to the product are accessible immediately. The first results of migration efforts will take effect in a short time.
- High domain competence is encoded in the software. Therefore, investment protection [5] is necessary.

There is no doubt that there are some risks. Prominently, the product has been developed over a long period of time without product line concepts at hand, and therefore might not fit the needs of a product line. The migration is difficult, because the legacy code has to be reengineered. For the evolutionary transition, at least a complete reengineering (which would be too expensive) can be avoided by rather conducting partial reengineering of the legacy software. Because the transition to a product line is experimental, the cost should be as low as possible.

*Overview* The rest of the paper is organized as follows. Section 2 introduces a migration process that is suited for incremental introduction of a product line architecture. In Sect. 3, we present a tailored reengineering method, so called *feature analysis*, that can significantly contribute to understanding of the legacy system from a product line perspective. In Sect. 4, we compare our work to related research. Finally, in Sect. 5, we summarize the results of this paper and give hints for future work.

## 2 Evolutionary Process

In this section, we propose a lightweight iterative process that assists the asset mining based evolutionary introduction of software product lines.

*Prerequisites* The prerequisites for the application of the migration process are modest. For one thing, we assume that the features of the legacy system are, at least to some extent, known to the programmers and/or users of the system. The software itself is largely understood, but not necessarily suitable for product line development. The specific parts of the software that could be used for product line development are not known for sure.

Because the transition proceeds in several iterations that have some time in between, it is further not necessary that the needs of product lines are well understood right from the start. Instead, all participants will gain knowledge and experience throughout the gradual process towards the product line.

*Participants* The process comprises the participation of several parties. We need an expert programmer and architect of the legacy software, a domain engineer, users and customers, a reengineer, and a product line expert. These people form different working groups during the process. However, for the development of the product line, the internal personal of the company should always work closely together. That way, the participants get accustomed to the new technology more easily.

*Process steps* The evolutionary process iterates the following steps. As an illustration, the proceeding is depicted in Fig. 1. The process as a whole can be iterated as well.

**Identification of available and accessible features.** The first step of the process uses expert knowledge of programmers, software architects, and users and customers to obtain a list of existing features.

**Priorisation of identified features.** The number of features a product provides is usually large. Because we do not want to perform a complete reengineering of the legacy code and because we want quick results, the domain engineer selects important features. With involvement of customers, the features are ordered with respect to where the important variations for the customer are spotted.

**Feature anticipation.** The domain expert performs a market analysis to anticipate features of future version and variants of the product.

**Feature analysis.** The reengineer and the product line expert analyze the legacy product according to the selected number of features. Feature Analysis is described in detail in Sec. 3. The individual steps are:
  1. Feature location. Where are the features implemented in the legacy code?
  2. Recognition of commonalities and variabilities. Where are the commonality and variability points in the legacy system?
  3. Recognition of dependencies among features. Which features do depend on which other features? Are there features not accessible for end-users?
  4. Metrics computation. The use of metrics facilitates the evaluation of the proposed changes. Is the intended architecture feasible for a product line approach?

**Effort estimation.** Based on the results of feature analysis, the product line expert and the reengineer decide if the restructuring is manageable and pays off. Further, the parts of the software where anticipated features can be hooked in are identified in the preliminary product line architecture. The product line expert and the architects of the old software perform an architecture analysis, e.g., SAAM [6] or ATAM [7]. If the effort seems not to be profitable, the evolutionary migration is cancelled.

**Reengineering.** The reengineer and the programmers restructure (parts) of the software according to its features and to the preliminary product line architecture.

*Collected data* During the process, all relevant artifacts are stored in a repository. After each iteration, the repository is updated. At least the following artifacts are needed:

– System's product line architecture. The architecture of the legacy system is used as a product line architecture initially.
– Product's feature list. The list of features along with their priorities and their potential for variations should be maintained as a separate document. The priorities of the features might change over some iterations.
– Feature map, feature commonalities, variabilities, and dependencies. All the information gained during feature analysis can be reused for subsequent iterations.
– Software sources. The software system is changed in the last step of each iteration.

*Summary* The proposed process allows a step-by-step migration of the software without disruption of the maintenance and support activities. After each iteration, the software system is more suitable for software product line development.

## 3 Partial Reengineering by Feature Analysis

Traditional reengineering methods focus on the legacy software's components and usually require an analysis of the complete system at hand. Unfortunately,

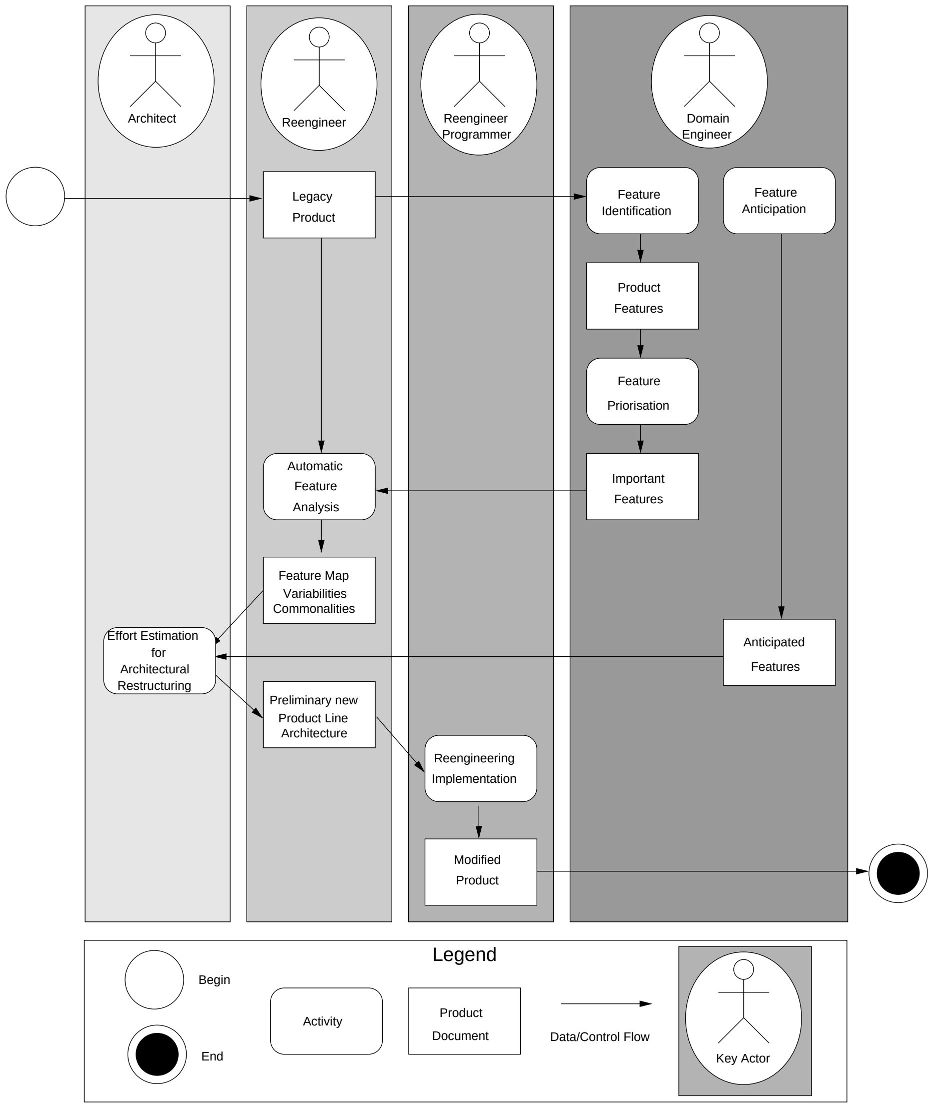

**Fig. 1.** One iteration step of the process for the evolutionary migration towards a product line architecture

this kind of analysis is an expensive task and therefore not feasible for experimental changes to the architecture. In our context, the cost for reengineering might well exceed the cost for reimplementation—because we know for sure that the architecture changes. In the following, we describe the reengineering method called *feature analysis* that compensates these shortcomings.

Firstly, feature analysis is tailored to needs of reengineering for product lines as it focuses on the features of a legacy software rather than its components. Second, feature analysis allows for partial reengineering. The input for feature analysis is just the set of software features of interest. Starting from that, the rest of the software is inspected on a demand driven basis. Third, feature analysis can be based on dynamic program behavior and can be implemented easily. It quickly yields results that can be combined and refined with conventional reengineering methods.

The remainder of this section explains feature analysis in detail. The first step (Sect. 3.1) of feature analysis is the location of a feature in the source code (or the components) of the legacy system. Second (Sect. 3.2), the dependencies of features are investigated. The feature's variabilities and commonalities are inspected. The last step of feature analysis (Sect. 3.3) consist of metric computation that indicate the feasibility and cost of the implementation of proposed results. Based on the result of feature analysis, restructuring proposals can be made along with an estimation of the cost of the proposed changes. Finally, a example for the application of feature analysis is given in Sect. 3.4.

## 3.1 Feature Location

To find the feature's implementation, there are either static or dynamic analyses, or the combination of both. A static approach is taken by Chen and Rajlich [8]. They propose a semi-automatic method for feature location that involves a human-guided search on the *Program Dependence Graph*. Their method takes into account documentation as well as domain knowledge and, as experiments in [9] show, can be quite time consuming.

The general idea of dynamic feature location is founded in the observation that a feature exposed to a user can be invoked in a usage scenario. If we record the routines that were executed during the usage scenario, we know where to look for the feature's implementation. With source code instrumentation, recording a usage scenario is simple. For further reduction of the search space, we compare different records for specific variants of the feature invocation.

Wilde and Scully [10] pioneered in taking a fully dynamic approach by analyzing *sequential traces* of program executions. After instrumenting the source code with the output of trace information, their *Software Reconnaissance* executes the program once invoking the feature and once again not invoking the feature. By subtracting the trace results of the second from the first invocation, the parts of the source code implementing the feature can be identified. The software reconnaissance is implemented by the tool RECON [11].

For large programs, sequential traces grow voluminous and are difficult to handle. In those cases, execution profiles seem more suitable than sequential traces. The most simple code instrumentation solution is using a profiler that is usually a part of the development environment. The profiler extracts the information about which and how often routine were executed during the program run (and various other information). We have presented a method for feature location based on execution profiles in previous papers [12,13,14].

Other than Wilde, we employ several program traces (resp. program profiles) at a time. We collect the profiling result of a number of feature invocations and merge them into a relation table that contains the information *routine r is invoked in scenario s*.

Subsequently, we apply the mathematically founded method *concept analysis* to investigate the data. The result of concept analysis, the *concept lattice*, is interpreted manually, but guided by generally applicable interpretation hints. The concept lattice is represented graphically and provides useful means for the investigation of the software system.

### 3.2 Feature Dependencies, Variablities, and Commonalities

For a product line architecture, it is important to understand the dependencies, variablities, and commonalities of product features. To analyze relations between features, it is not sufficient to look at one feature at a time. Rather, it is essential to investigate a set of features. Because Wilde as well as Rajlich only analyze individual features, their approaches would have to be applied repeatedly whereas our approach investigates sets of (more or less) related features.

The idea of our method is based on the following consideration: When we invoke a series of (related) features, and simultaneously record the execution profiles, we are interested in what routines are executed for all of the features (indicating commonalities) on one side. On the other side, the routines that are exclusively invoked for individual features indicate the variabilities. What we need is a useful grouping of routines. This grouping is achieved automatically by our method and can serve as a hint for the product line architecture of the involved features.

### 3.3 Restructuring Cost Assessment by Metrics

After we identified the features in code, we want to assess the complexity of a restructuring effort. Metrics are useful as a first indication in this regard.

One approach to quantify the relation between program components and features is studied by Wong et al. [15]. This work is based on Wilde's approach, and tries to complement Wilde's results by providing a more complete picture of how the features are spread across the software source. Wong proposes three metrics: the disparity between features, the concentration of the feature implementation in components, and the dedication of components to features.

In the context of product lines, the disparity serves as an indicator for commonalities (the greater the disparity between features, the less they have in common). The concentration and the dedication allow for assessment of the restructuring cost. However, the quantification does not take dependencies into account, as the feature location technique it is based on does not. Currently, we are investigating metrics that are based on our concept lattices.

### 3.4 Example for Feature Analysis

In [12,13,14], we applied feature analysis to various software systems with promising results. In the following, we present a case study we performed in [14] with the drawing tool XFIG [16] for the purpose of program understanding. The results of the analysis are re-interpreted for product line goals.

XFIG is a menu-driven tool that enables the user to draw and manipulate graphical objects interactively under the X Window System. It is written in the programming language C and consists of about 75KLOC in 175 files containing over 1.900 C routines. Without a further look at the sources of XFIG, we compiled the program with profiling information.

We looked at the facilities to draw various shapes, e.g., *circles*, *ellipses*, *boxes*, *arcs*, and others. We executed the corresponding usage scenarios for creating the objects and collected the execution profiles. Then we applied our feature analysis tool. The resulting lattice is shown in Fig. 2.

The boxes in Fig. 2 represent so called *concepts*. They contain the features' names and the routines that were invoked during the features' invocation. A concept on the topmost level of the lattice corresponds to a single feature and contains the routines specifically required for that feature (e.g., Concepts #1–#5). Below the first level, interdependencies of features are made explicit. The concept contain the routines that are jointly used for the feature atop. Concept #6, for example, lists the routines that are jointly used for the variants of ellipses and circles. The more we descend in the lattice, the more general are the routines in the concepts. At the bottom, Concept #10 lists the routines that are used in all scenarios, e.g., initialization code and the `main` start routine.

The lattice reveals the variabilities and commonalities for drawing circles and ellipses: the top level concepts represent the variants for drawing round objects, Concept #6 below holds the common code. In the special case of XFIG, the following activities could be guided directly by our results.

*Source Code Restructuring* XFIG's source files are not structured according to the concepts found by the analysis using shape features. If a file structure pursuant to the shape objects is desired, the concept lattice indicates how to restructure the sources.

*Adding and Removing Features* New shape variants can be added easily by using the structure of the other shapes as a template. E.g., defining a circle object by three points can reuse the functionality as grouped in Concept #6. Existing variants can be removed easily by removing the routines of the variability concept of the corresponding shape. In order to remove the ellipse-by-radius feature, we remove the routines of Concept #1 and the GUI elements identified by a static dependency analysis [14].

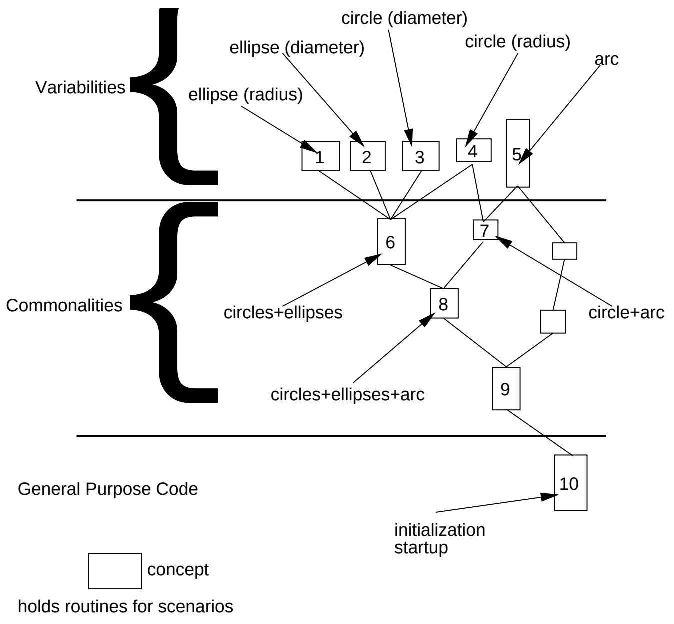

**Fig. 2.** Example lattice (excerpt) for an experiment with the drawing tool XFIG. The lattice reveals variablities and commonalities of selected features

*Product Configuration* Since we have identified the groups of routines that implement the variants, we can try to parameterize the build process for XFIG. With respect to the variants, the routines that vary are included on demand. For example, all circles and ellipses depend on Concept #6, so if a product is configured to include circles or ellipses as product features, the routines in Concept #6 have to be included.

For XFIG, our technique leads us directly to the routines realizing the variabilities currently included in the system. But the system is not necessarily structured "configuration friendly". We do not get the knowledge how to restructure the software to be more configurable for free, but it is fairly easy to spot the underlying structure of XFIG. E.g., the manipulation features such as moving or resizing objects are organized as routines handling all kinds of shapes by a case statement. That leads to at least three options to organize the configuration of an evolving product line:

1. Give the programmer a list of all locations where the code has to be changed to add or remove kinds of shapes according to product configurations (i.e., a manual configuration of products, the least preferable solution).
2. Generate the case statements automatically according to product configurations (automatized configuration, higher effort for tool support).
3. When migrating from C to C++ (this decision in itself has a long term strategic impact), the kinds of shapes correspond to classes and the case

statements would be replaced by virtual functions. Subsequently, the class hierarchy can be parametrized according to product configurations (use of modern implementation technology).

These options can be implemented such that the functionality of an accordingly configured product is exactly the same as that of the old product. That way, regression testing can be supported. Our method helps spotting potential variability points that can be restructured so that the product line becomes more configurable.

## 4 Related Research

The need for mechanisms to develop software product lines starting from existing, successful products is generally recognized. The main research areas today are product line processes, i.e., frameworks for product line development and changes to a company's organization. Further, the support for the low level modification of the legacy code itself is of vital interest as this is a topic where investments can be leveraged.

*Frameworks:* The Software Engineering Institute promotes the product line practice framework [2]. The framework collects information, including studies from various organizations and practitioners with the goal that the product line community can use the growing body of knowledge involving the development, acquisition, and evolution of a software product line. Bergey's framework [17] discusses software migration issues in general. The migration and mining efforts for software product lines they propose are not evolutionary and incremental; they do not provide points to make a strategic withdrawal from product line technology when the migration is underway. Bosch [1] proposes methods for designing software architectures, in particular product line architectures. Bosch explores the concepts of product lines and discusses the pros and cons of the evolutionary resp. revolutionary instantiations of software product lines in the face of existing products. PuLSE [4] implements a complete framework for the introduction, scoping, evolution, and maintenance of software product lines.

*Asset Mining for Product Lines:* Bergey et al. [18] propose a method for mining assets for product line reuse at a high level of abstraction. Before the process can start, a product-line architecture is required. Based on the Horseshoe Model and on OAR [3], they try to leverage components on a large scale. Reengineering techniques are used for architecture reconstruction to prepare the legacy software for modifications. The modification and restructuring of components according to product line needs is not considered. Within the PuLSE methodology for the development of product lines, RE-PLACE [19] is presented as an approach to support the transitioning of existing software assets towards a product line architecture. RE-PLACE uses architecture recovery for legacy systems. The main goal for reengineering is the identification of components that can be reused in the product line and the integration of components into the product line. Recovered components are treated as black boxes and are only reused as a whole. In the provided case study, the core of a legacy system is wrapped up for reuse. The transitioning process itself is not incremental and has no intermediate products. Instead, the a software product line architecture is reached in one step.

## 5 Conclusion and Future Work

In the previous sections, we have introduced a process supporting the evolutionary introduction of software product line concepts. The process starts out from a legacy software system that supports no product line concepts. After each iterative step in the process, the legacy product is restructured so that parts of it conform to needs of product lines. Throughout the process, there are dropout points, i.e., if restructuring of the legacy code seems not feasible, the migration process is cancelled, and instead a replacement of the software is indicated.

Our method avoids an expensive architectural reconstruction of the legacy system. Since the architecture will change anyway, we are interested in the features of the software rather than the architectural components. To restructure software according to the needs of software product lines, we use feature analysis. Feature analysis has proved to be a cheap and easy to implement yet powerful method to reinvestigate legacy code under a completely different paradigm.

Our future research goals include the further refinement of feature analysis and the evaluation of the iterative, incremental process in an industrial project that aims at the initiation of a software product line. The refinement of feature analysis is carried on in the context of the Bauhaus Project [20] at the Universität Stuttgart. We plan to improve that method by integrating further automatic source code analyses.

## References

1. Bosch, J.: Design & Use of Software Architectures. Addison-Wesley and ACM Press (2000)
2. Northrop, L.M.: A Framework for Software Product Line Practice. Available at `http://www.sei.cmu.edu/plp/framework.html` (2001)
3. Bergey, J., O'Brien, L., Smith, D.: Options Analysis for Reengineering (OAR): A Method for Mining Legacy Assets. Technical Report CMU/SEI-2001-TN-013, Software Engineering Institute (SEI), Carnegie Mellon University (2001)
4. Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.M.: PuLSE: A Methodology to Develop Software Product Lines. In: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), Los Angeles, CA, USA, ACM Press (1999) 122–131
5. Eisenbarth, T., Simon, D.: Guiding Feature Asset Mining for Software Product Line Development. In Schmid, K., Geppert, B., eds.: Proceedings of the International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing, Erfurt, Germany, Fraunhofer IESE (2001) 1–4

6. Kazman, R., Bass, L., Abowd, G., Webb, M.: SAAM: A Method for Analyzing the Properties of Software Architectures. In: Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, IEEE Computer Society Press (1994) 81–90
7. Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carrière, S.J.: The Architecture Tradeoff Analysis Method. In: Proceedings of the 4th International Conference on Engineering of Complex Computer Systems, Monterey, CA, USA, IEEE Computer Society Press (1998) 68–78
8. Chen, K., Rajlich, V.: Case Study of Feature Location Using Dependence Graph. In: Proceedings of the 8th International Workshop on Program Comprehension, Limerick, Ireland, IEEE Computer Society Press (2000) 241–249
9. Wilde, N., Buckellew, M., Page, H., Rajlich, V.: A Case Study of Feature Location in Unstructured Legacy Fortran Code. In Susa, P., Ebert, J., eds.: Proceedings of the 5th European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, IEEE Computer Society Press (2001) 68–75
10. Wilde, N., Scully, M.: Software Reconnaissance: Mapping Program Features to Code. Journal of Software Maintenance: Research and Practice **7** (1995) 49–62
11. Wilde, N.: RECON. Available at `http://www.cs.uwf.edu/~recon/` (2001)
12. Eisenbarth, T., Koschke, R., Simon, D.: Derivation of Feature-Component Maps by Means of Concept Analysis. In Susa, P., Ebert, J., eds.: Proceedings of the 5th European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, IEEE Computer Society Press (2001) 176–179
13. Eisenbarth, T., Koschke, R., Simon, D.: Feature-Driven Program Understanding Using Concept Analysis of Execution Traces. In: Proceedings of the 9th International Workshop on Program Comprehension, Toronto, Canada, IEEE Computer Society Press (2001) 300–309
14. Eisenbarth, T., Koschke, R., Simon, D.: Aiding Program Comprehension by Static and Dynamic Feature Analysis. In: Proceedings of the International Conference on Software Maintenance, Florence, Italy, IEEE Computer Society Press (2001) 602–611
15. Wong, W.E., Gokhale, S.S., Hogan, J.R.: Quantifying the Closeness between Program Components and Features. The Journal of Systems and Software **54** (2000) 87–98
16. The XFIG drawing tool, Version 3.2.3d. Available at `http://www.xfig.org/` (2001)
17. Bergey, J.K., Northrop, L.M., Smith, D.B.: Enterprise Framework for the Disciplined Evolution of Legacy Systems. Technical Report CMU/SEI-97-TR-007, Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, PA, USA (1997)
18. Bergey, J., O'Brien, L., Smith, D.: Mining Existing Software Assets for Software Product Lines. Technical Report CMU/SEI-2000-TN-008, Software Engineering Institute (SEI), Carnegie Mellon University (2000)
19. Bayer, J., Girard, J.F., Würthner, M., DeBaud, J.M., Apel, M.: Transitioning Legacy Assets to a Product Line Architecture. In: Proceedings of the Seventh European Software Engineering Conference (ESEC'99). Lecture Notes in Computer Science 1687, Toulouse, France, Springer (1999) 446–463
20. The New Bauhaus Stuttgart. Available at `http://www.bauhaus-stuttgart.de/` (2001)