Proceedings of the

First International Software Product Lines Young Researchers Workshop (SPLYR)

Boston, MA - August 30th, 2004
In conjunction with the 3rd Software Product Line Conference
(SPLC)
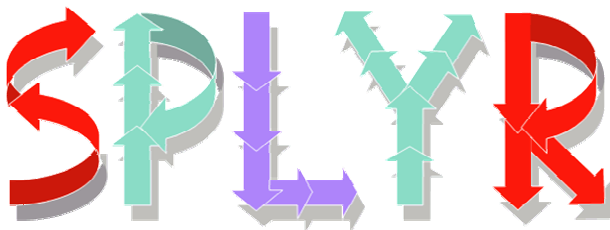
Editors
Birgit Geppert
Avaya Labs, USA
Isabel John
Fraunhofer IESE, Germany
Giuseppe Lami
ISTI - Italian National Council of Researches

Introduction: Family Planning beyond the Software

In the past years product-line engineering has emerged as the standard practice in industry for large-scale software reuse. Different to many other techniques, product-line engineering arose from practical experience in industry. But as with every successful technique product line-engineering can also be considered only a real standard approach when it is not only applied in practice but also widely researched and especially taught in academia. Practice and academia are the two sides of the same coin. While industry sets the requirements, academia prepares the practitioners of tomorrow. The peculiarity of the SPLYR workshop is that it is specifically addressed to young researchers, in particular Ph.D. students, having original ideas and initiatives in the product line field.

Though this is the first SPLYR workshop, we had submissions from students from all over the world, including Africa, Europe, and the Unites States. Each student was assigned one product line expert who reviewed the proposal and discussed pros and cons of the work with the student. We would like to thank Len Bass, Jan Bosch, André van der Hoek, and Dirk Muthig for reviewing the proposals and for the effort they spent in discussing the work with the students. We selected seven proposals for presentation at the workshop. The topics range from product line adoption over variability management and product line architectures to product line evolution.

For many of the students this is the first time to present their work to an international audience. The workshop aims at providing the right platform for this and for discussing the presented work among each other and with experts in the field. We hope that with this workshop each of the students will get valuable feedback for the further development of the work.

The SPLYR Organizers

Birgit Geppert
Isabel John
Guiseppe Lami

Keywords : Software Product Lines, Software Product Line Young Researchers Workshop, Proceedings, SPLYR.

Organization

SPLYR is co-located with the 3rd Software Product Line Conference, SPLC 2004 Boston, USA, August 30, 2004.

Workshop Chairs :

- Birgit Geppert
  Avaya Labs, Software Technology Research
  Basking Ridge, NJ, USA
  bgeppert@research.avayalabs.com

- Isabel John
  Fraunhofer IESE
  Sauerwiesen 6, D-67661 Kaiserslautern
  john@iese.fraunhofer.de

- Giuseppe Lami
  Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo"
  Area della Ricerca CNR di Pisa, Via G. Moruzzi 1
  Giuseppe.Lami@isti.cnr.it

Reviewers / Panelists:

- Len Bass
  Software Engineering Institute, USA

- Jan Bosch
  University of Groningen, The Netherlands

- André van der Hoek
  University of California, Irvine, USA

- Dirk Muthig
  Fraunhofer IESE, Germany

Workshop website and email:

http://www1.isti.cnr.it/SPLYR/

SPLYR@isti.cnr.it

Table Of Contents

1   *Yu Chen*
     A Process-Centric Approach for Software Product Line Evolution
     Management

A Process-Centric Approach for Software Product Line Evolution Management [⋆]

Yu Chen [⋆⋆]

Department of Computer Science and Engineering
Ira A. Fulton School of Engineering
Arizona State University, Tempe AZ 85287-8809, USA
yu_chen@asu.edu

Abstract. Evolving a software product line involves more people, organizations, and dependencies than evolving a single software product. Without advanced process management approaches, the impact of the increased process complexity can outweigh the benefits of using a software product line approach. Currently, there is a lack of support for software product line evolution management. I propose using a process-centric approach within a workflow-based environment to support software product line evolution management, which can effectively assist process design, enforcement, and measurement.

1 Introduction

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way [1]. A software product line approach promises large-scale productivity gains, shorter time-to-market, higher product quality, increased customer satisfaction, decreased development and maintenance costs [1]. However, it also imposes new challenges and risks. Evolving a software product line is more difficult than evolving a single product, because 1) the high dependencies between core assets and individual products require core assets and individual products to be evolved together, 2) new requirements originated from individual products may conflict with each other, 3) the involvement of more products, people, and organizational units makes the evolution process more complicated, and 4) it requires a higher initial investment, involves new risks, and needs different management strategies under different situations.

Process definition, enactment, and measurement are identified as practice areas that are critical to software product line success [1]. A process definition describes the responsibilities of people and organizational units that involved in

---

the software development, and specifies the procedures to be followed for product development. A good process definition facilitates the common understanding as to how the products are development and maintained, provides guidance from the best practices, and fits well into the organizational environment. As a complex and human-intensive process, software product line evolution without a well-defined process specification can become chaotic, unpredictable, and unrepeatable. Having a well-defined process description would not help if the process is not actually enforced. Currently, most organizations place their process definitions into documents and rely on their employees to read and follow the defined processes. A problem with this practice is that the *used* processes often differ from the *defined* processes due to human errors, reluctance to follow the processes, etc. Another problem with this practice is that efficiency reduces as more people and organizations become involved. In the context of a software product line, these problems become more prominent because of the increased number of people, organizational units, and dependencies. For software product lines, manually documenting and enforcing software processes is error-prone and inefficient. Hence, more effective process enactment is needed. That in turn raises the need for process measurement. Process measurement involves process monitoring, metrics data collection, and process analysis. It lets organizations know the enforced process status and helps them understand the effectiveness of the enforced process. Although the importance of applying the above process management activities has been widely recognized, not many organizations are applying them because under project delivery pressure those activities can be overwhelming without automatic tool support. As for software product lines, to date, there is no existing approach that effectively supports all those process management activities.

This research proposes using a process-centric approach within a workflow-based environment for software product line evolution management. The approach provides a systematic way to define, enact, and measure software product line processes. The environment effectively assists process definition, enactment, and measurement by utilizing underlying workflow management services and providing metrics data collection as well as process analysis tools. By following standards and supporting open architecture, the environment allows easy tool integration and data exchange. Using the process-centric approach within the workflow-based environment, software product line evolution can be managed in a more controllable and repeatable way.

The contributions of this work are threefold. First, an approach for software product line evolution management will be developed. The approach will be based on the analysis of existing methods for software product line evolution management and the evaluation of current workflow technologies for supporting software product line evolution. Second, the investigation of software metrics relevant to software product line evolution will be performed. The results can assist organizations in selecting appropriate software metrics to meet their process management goals. Finally, a prototype environment with the goal of facilitating software product line cost-benefit analysis will be developed. The prototype is

meant to demonstate the feasibility of building such an environment and usefulness of the proposed approach. As cost-benefit is often the concern of the organizations that adopt a software product line approach. The prototype will be found useful in providing decision support on both adoption and evolution issues.

The remainder of this proposal is organized as follows. Section 2 describes background and related work. Section 3 discusses the proposed work and research approach. Section 4 discusses the work status. Section 5 draws conclusions.

2   Background and Related Work

This section provides background and related work in the following fields: software product line evolution, software process modeling and simulation, workflow management, and software metrics.

2.1   Software Product Line Evolution

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way [1]. Software product line evolution is affected by many factors, such as market demands and available resources, and involves feedback loops between core asset development, product development, and management activities. Evolving a product line is more complicated than evolving a single software product. To cope with the difficulties, some guidelines and methodologies have been suggested. Schmid and Verlage discussed some software product line initiation situations (*independent*, *project-integration*, *reengineering-driven*, and *leveraged*), adoption approaches (*big bang* and *incremental*), and evolution strategies(*infrastructure-based*, *branch-and-unite*, and *bulk-integration*) [2]. Bosch presented some approaches to adopt and evolve software product lines, and related them to software product line maturity levels and organization models [3]. The proposed work will assist product line evolution under typical product line initial situations, adoption approaches, and evolution strategies.

2.2   Software Process Modeling and Simulation

A software process is a set of activities, methods, practices, and transformations that people use to develop and maintain software and associated products, such as project plans, design documentations, code, test cases, and user manuals [4]. Adopting new software processes is expensive and risky, so software process simulation modeling is often used to reduce the uncertainty and predict the impact. Discrete Event Specification (DEVS) [5] is a discrete event based modeling and simulation theory which supports characterizing and simulating concurrent dynamics. DEVSJAVA [5] is a Java implementation of DEVS. The external view of a DEVSJAVA model is a black box with input and output ports. A model

receives messages through its input ports and sends out messages through its output ports. Ports and messages are the means and the only means by which a model communicates with the external world. A DEVSJAVA model is either "atomic" or "coupled". An atomic model is undividable and generally used to build larger models while a coupled model is the composition of other models. A coupled model includes a finite number of (atomic or coupled) models and couplings. The couplings are essentially message channels and provide a simple way to construct hierarchical models. DEVSJAVA provides sequential, parallel, and distributed simulation engines, and its source code is available. Thus, it provides a powerful, flexible, and open framework for process modeling and simulation. The proposed work will use DEVSJAVA as the process modeling formalism.

2.3 Workflow Management

A workflow is "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules" [6]. A workflow management system (WFMS) is a system that "defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications" [6]. As a database management system (DBMS) assists data management, a WFMS facilitates process management. Although there are many WFMS implementations available and most of them provide similar functionalities, few of them can be used together becuase of the lack of interoperability. To address that issue, Workflow Management Coalition (WFMC) has proposed several standards. Among which the workflow reference model has been widely accepted. The reference model identifies six components and five interfaces in a WFMS. The components are *process definition*, *workflow enactment service*, *administration and monitor tools*, *workflow clients*, *invoked applications*, and *other enactment service*. The interface between the *process definition* and *workflow enactment service* includes a common meta-model for describing the process definition and an XML schema specifying XPDL [7]. The proposed workflow-based environment will follow the workflow reference model and use XPDL as the standard process description language.

2.4 Software Metrics

A metric is a quantitative indicator of an attribute of a thing. Software metrics provide a structured method for measuring, evaluating, and estimating software products and processes [8]. Many software metrics have been developed and investigated, only few of them are designed specifically for software product lines. A class of metrics for product line architectures has been developed by Hoek et al. [9]. The metrics are based on the concept of service utilization, which measures the percentage of the provided or required services of a component that are used in an architectural configuration. A metrics model specifies relationships among

metrics. COnstructive Product Line Investment MOdel (COPLIMO) [10] is a COCOMO II [11] based metrics model for software product line cost estimates. It has a basic life cycle model and an extended life cycle model. The basic life cycle model can be used for early stage trade-off considerations. The extended life cycle model allows modeling with more details and can be easily extended from the basic life cycle model.The prototype environment will use COPLIMO as the metrics model and provide an approach that collects relevant metrics data from configuration management and process history.

3   Approach

Software product line evolution spans the whole life cycle of a product line and requires continuous process monitoring and adjustment. To support that, I propose a process-centric approach (illustrated in Fig. 1) to manage software product line evolution.
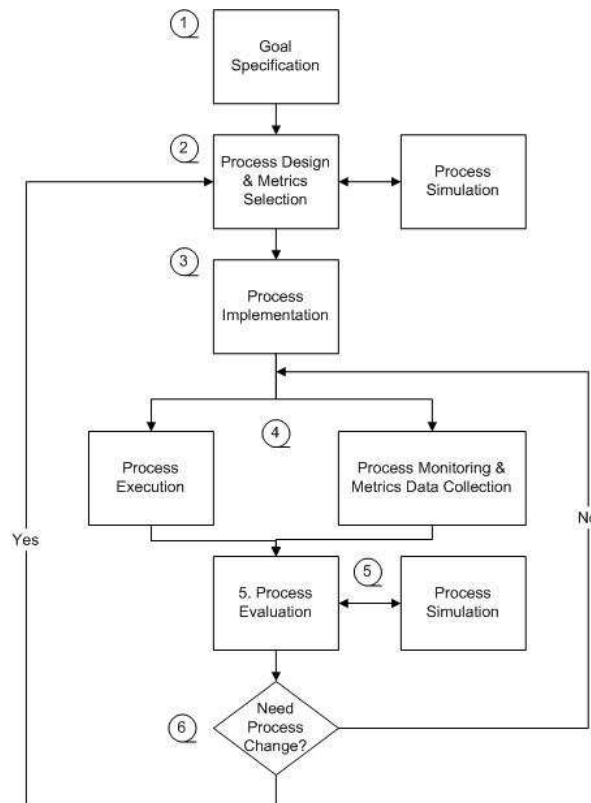
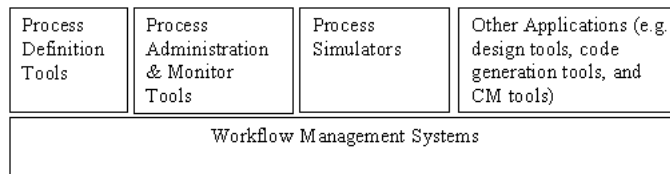**Fig. 1.** An Approach for Software Product Line Evolution Management

The approach is adapted from the workflow life cycle model [12], and consists of the following steps: 1) specify the evolution goal; 2) design, evaluate, and select candidate processes, choose appropriate metrics for process measurement; 3) implement the selected process; 4) execute and monitor the process, and collect metrics data; 5) evaluate the process by analyzing the collected metrics data and comparing the results against the specified goal; 6) go to Step 2 if the process evaluation suggests process improvement, otherwise go to Step 4. In step 2 and 5, process simulation can be used to assist process evaluation. The above steps constitute a loop that terminates when the product line is phased out. Also, any changes made to the goal will end the current iteration and start a new one.

I also propose a workflow-based environment to support the above approach. The environment architecture (illustrated in Fig. 2) conforms to the workflow reference model [13] and uses XPDL [7] as the standard process definition language. With this architecture, process design can be supported by workflow definition tools, process implementation and process execution can be handled by workflow management systems, metrics data can be collected by monitor tools, and process evaluation can make use of simulation tools. By following standards, this environment allows easy tool integration and data exchange. By providing an integrated environment for software product line development and process management, this environment facilitates effective process enactment and accurate process measuring.

Process Definition Tools	Process Administration & Monitor Tools	Process Simulators	Other Applications (e.g. design tools, code generation tools, and CM tools)
Workflow Management Systems			

Fig. 2. Workflow-based Environment for Product Line Evolution Management

My research will consist of the following topics:

1. The analysis of existing techniques for managing software evolution in general and software product evolution in particular. The analysis will be concentrated in the field of software metrics, process modeling and simulation, and workflow management. The goal of this analysis is to identify the strengths and weaknesses of the current approaches.
2. The investigation of software metrics relevant to software product line evolution. As the result, relevant metrics will be categorized in a way that can assist metrics selection.
3. The evaluation of current workflow technologies on supporting software evolution in general and software product line evolution in particular.
4. The prototype implementation of the proposed environment with the goal of assisting cost-benefit analysis. The implementation involves:

(a) The selection of an appropriate cost-benefit metrics model and a work-flow management system.

(b) The creation of a software product line process simulator that can assist cost-benefit analysis. The simulator will be based on DEVAJAVA [5] formalism and be implemented with two versions: one for early stage trade-off consideration and one for late stage cost-benefit analysis.

(c) The development of an approach that maps XPDL [7] process descrptions onto DEVSJAVA [5] models.

(d) The development of an approach that collects metrics data from config-uration management and process history.

5. The evaluation of the proposed approach and the environment via getting feedback from experts and conducting case studies.

4   Work Status

An initial analysis of existing methods for software product line evolution has been conducted, and part of the results are presented in Sect. 2.

The early stage version of the Software Product Line Process Simulator (SPLPS) has been implemented [14]. The goal of the SPLPS is to facilitate software product line decision making by providing time and cost estimates under various situations. In implementing the simulator, DEVSJAVA [5] and COPLIMO [10] are choosen as the modeling formalism and the cost model, re-spectively. SPLPS models the interaction between major software product line engineering activities, such as core asset development, product development, technical management, human resource management, and market demands. The inputs to SPLPS include *general* parameters and *product (core asset)* parame-ters. The general parameters are used to describe software product line pro-cess attributes and organization characteristics. These parameters include the maximum number of products that will be supported by the product line, the number of products that will be created during the creation stage, the product line adoption and evolution approaches, the number of employees in an organi-zation, and the market demand intervals. The product (core asset) parameters are primarily determined by the employed cost model (COPLIMO, in this case). These parameters include the size of the product (core assets), fraction of code segments(product unique code, reused code, and adapted code), percentage of modification required for design, code, and integration, software understanding, software unfamiliarity, and average change rate caused by new market demands. At the end of each simulation run, some statistic results will be generated. For products and core assets, their first release time, time-to-market, initial devel-opment effort, initial development time, accumulated development and mainte-nance effort, accumulated development and maintenance time, and the number of finished requirements will be provided. For the entire product line, the to-tal evolution effort, the time when all the requirements are finished, the average annual effort, the number of total requirements generated, and the average time-to-market will be given. SPLPS can also visually present how major product line engineering activities progress and interact over time.

5 Conclusion

Managing software product line evolution is a complex process, but currently there is a lack of support. This research proposes a process-centric approach within a workflow-based environment to support product line evolution. The proposed approach provides an effective way to manage product line evolution through systematic process definition, enactment, and measurement. The proposed environment can effectively support process management activities by providing tool support within an integrated software development and process management environment. The contributions of this work are: 1) a approach for software product line evolution management, 2) the investigation of software metrics relevant to software product line evolution, and 3) a prototype environment to assist software product line cost-benefit analysis.

References

1. Clements, P., Northrop, L.M.: Software Product Lines : Practices and Patterns. Addison-Wesley, Boston MA U.S.A. (2001)
2. Schmid, K., Verlage, M.: The economic impact of product line adoption and evolution. IEEE Software **19** (2002) 50 – 57
3. Bosch, J.: Maturity and evolution in software product lines: Approaches, artefacts and organization (2002)
4. Paulk, M., Weber, C., Garcia, S., Chrissis, M.B., Bush, M.: Key practices of the capability maturity model. Technical Report CMU/SEI-93-TR-25, Software Engineering Institute, Pittsburgh Pennsylvania U.S.A. (1993)
5. Zeigler, B.P., Sarjoughian, H.S.: Introduction to devs modeling & simulation with java (2003)
6. WFMC: Workflow management coalition terminology and glossary. Technical Report WFMC-TC00-1011, Workflow Management Coalition (1995)
7. WFMC: Workflow process definition interface - XML Process Definition Language. Technical Report WFMC-TC-1025, Workflow Management Coalition (2002)
8. Mens, T., Demeyer, S.: Future trends in software evolution metrics. In: International Workshop on Principles of Software Evolution. (2001) 83–86
9. van der Hoek, A., Dincel, E., Medvidovic, N.: Using service utilization metrics to assess the structure of product line architectures. In: Ninth International Software Metrics Symposium, IEEE Computer Society Press (2003) 298 – 308
10. Boehm, B., Brown, A.W., Madachy, R., Yang, Y.: A software product line life cycle cost estimation model (2003)
11. Boehm, B.W., Clark, B., Horowitz, E., Westland, J.C., Madachy, R.J., Selby, R.W.: Cost models for future software life cycle processes: COCOMO 2.0. Annals of Software Engineering **1** (1995) 57–94
12. Zur Muehlen, M.: Organizational management in workflow applications. Information Technology and Management **5** (2004) 271–291
13. Hollingsworth, D.: The workflow reference model. Technical Report WFMC-TC00-1003, Workflow Management Coalition (1995)
14. Chen, Y., Gannod, G.C., Collofello, J.S., Sarjoughian, H.S.: Using simulation to facilitate the study of software product line evolution. In: Seventh International Workshop on Principles of Software Evolution, Tokyo, Japan (2004)

2   *Marius Dragomiroiu, David L. Parnas and Markus Clermont*
     On Variabilities in Program Families

On Variabilities in Program Families

Marius Dragomiroiu, David L. Parnas and Markus Clermont

Software Quality Research Laboratory,
Computer Science and Information System,
University of Limerick, Limerick Ireland
{marius.dragomiroiu, david.parnas, markus.clermont}@ul.ie
www.sqrl.ul.ie

Abstract. The program family development process is concerned with reuse of artifacts related to requirements specification, software architecture, detailed design and code. There are many different approaches to producing families of programs: domain engineering, frameworks and program families. Sometimes they are viewed as competing but, in fact, they are complementary, compatible, and mutually supportive. To succeed, all these processes have to help the developers to manage the variability among the product members. Variabilities among the family members are determined during all phases of the development process: requirements analysis, module decomposition and implementation. This paper proposes a simple way to manage variabilities from identification phase to the implementation and beyond. The focus is on the design solutions to handle variabilities. The design solutions discussed promote reuse and provide flexibility in designing software families.

1   Introduction

Today's sophisticated customers expect reliable, flexible software systems that can be easily adapted to their special needs. On the other hand, the pressure on the cost and time side for software producers is always increasing. Although this seems like a hopeless situation, the use of software product lines and related concepts can help to satisfy these new requirements. Software families can provide significant gains in productivity by systematic reuse of core assets. The potential advantages of the program family approach to develop, maintain and reuse of software has been long recognized (e.g. [1], [5], [9], [12], [19], [24]).

In 1976, Parnas defined a program family as "a set of programs for which it is worthwhile to study their common properties before determining the special properties of the individual family member"[19]. The differences among the family members are called *variabilities*. Variabilities among the program family's members are determined during all phases of the development process i.e. requirements analysis (problem space) and respectively in system design and implementation (solution space). There are several crucial tasks involved in designing the software families for limiting the effect of variabilities. These involve (1) the identification of variabilities in both problem and solution spaces, (2) mapping of variabilities from the problem space to modules and (3) flexible design solutions to bind possible variant modules' implementations.

These issues have been advocated before: commonality analysis in [1], [6], [24] distinction between the variability in problem space and solution space, mapping of variability

from the problem space into the system design [10], [15] and design solutions to handle variant modules into the system [7], [12], [14]. However, solutions to these issues are still a challenging subject of research. Proper management of these tasks is a key issue for the program family approach to succeed.

This paper presents an effective way for handling program families' variabilities with the focus on design solutions. In the next section we present definitions of the concepts of product line, program family and frameworks; the third section describes the identification of variabilities in the requirements specifications of the family's members and summarizes the key design principles for dealing with variabilities in the system design phase. The fourth section presents solutions for mapping requirement variabilities into system modules. The fifth section presents detailed design solutions for binding system modules that encapsulate variabilities and finally we give some concluding remarks.

2   Terms and Related Concepts

In this section we discuss the terms and concepts of the program family approach, frameworks and domain engineering that are used throughout this paper.

Program Families. The program family approach looks at both problem and solution spaces in order to identify the variabilities. It aims, as much as possible, to hide every changeable aspect of the system in individual modules with aspects that will change separately being in separate modules [18]. There will also be modules hiding secrets that were not identified as variabilities among the capabilities or requirements of the family members.

This is the distinction between domain engineering, as some authors [1], [6], [24] define the concept of the program family and the more general meaning of program family, stated by Parnas in mid 1970's. The latter definition of program family promotes the reusability above the boundaries of the domain. The program family approach promotes the reuse of the system architecture across the domain but also promotes the reuse of components[1] or parts of the system design across several domains.

Product Line/Domain Engineering. Domain requirement analysis is a top-down approach to identify and structure the variabilities of a set of systems in that domain. Domain engineering analyses the requirements of a set of similar applications in a certain domain. Unfortunately, not everything that may change is identified in the domain requirement analysis as it has only a restricted scope. For instance it excludes variabilities in design decisions or variabilities introduced by the generalization of some components that can be used across different domains, as well as variabilities induced by some aspects that are likely to change in the future.

Frameworks. A *framework* is an application core that can be customized by the application developer [17]. It can be seen as defining a program family, with the restriction that decisions used for variability binding are postponed until configuration, runtime or 'extension' time. Generally, module variants that encapsulate variabilities are chosen or implemented, on the basis of an interface specification, by the application developer who uses the framework. With a well-designed framework, members of the family are produced by adding new components, rather than by modifying old ones.

All three approaches have the same ultimate goal to promote reuse by providing a common architecture for a set of systems and to advocate flexibility by postponing decisions of binding the variable parts.

---

1. A software component is a unit of composition with precisely specified interfaces and explicit support dependencies only. We consider both objects and modules to be components.

3 Dealing with Variabilities in Requirements

In this section, we want to discuss how to identify variabilities in the requirements and structure them to see relations between variabilities that seem to be independent at first sight. The variabilities will then be organized in variation points, in such a way that each variation point is encapsulated in exactly one module. A module can encapsulate several closely related variation points, but only if all of the variabilities are expected to change when any one of them changes. Otherwise they should be placed in separate modules.

3.1 Domain Variability Identification

The variabilities in the problem space are determined during domain requirements analysis (see [24]). The commonalities and variabilities across a set of applications in a specific domain are identified using the requirements analysis of all potential members of the family. These variabilities all relate to the observable behaviour of the system.

Members of a specific program family can be considered black boxes that have observable variations in one of the following properties:

1. **external interface**, i.e. the interface of the system exposed to the users: the system services provided to users along with the acceptable input from users and output produced by the system.
2. **required support interface**, i.e. what the system requires in order to behave properly, for instance system platforms, external devices, and other external software used or controlled by the system.
3. **action flows**: the sequence of actions that the system users will execute or the flow of actions the user has to perform on the system.

3.2 Domain Variability Specification

During the commonality analysis the variations in the observable behaviour among the considered family members are identified. Dependencies between variations in the requirements can take one of the following three forms [14]:

- **mutual exclusion (single variant)**: a range of alternative requirements from which only one can be chosen in any family member.

- **alternative list (multiple variants)**: a range of alternative requirements from which one or many are chosen in any family member.

- **optional**: requirements that are valid only for a subset of the family members. Sometimes, this can be considered a special case of mutual exclusion.

At this stage, there are two issues regarding identified variabilities that have to be taken into account when structuring them:

1. the variabilities may depend on one another, i.e. the presence/absence of an optional requirement might be influenced by the selection of another alternative requirement
2. mapping the variabilities from the problem space to the solution space. At this level, the requirements are defined in terms of services provided to the system's user. Therefore, one variability in the requirements may influence more than one module in the system's design. Changes of several modules that are caused by the same variability are not necessarily due to inter-module dependencies, but may be caused by the way that requirement variabilities are mapped to modules.

3.2.1   Interdependent Variabilities

The issue of dependencies between variabilities has been addressed by many research-ers, some of them modelling the relations among variabilities using trees or tables [6], [24] others using directed graphs or lattices (see [8], [15]). The structure of the variabil-ities will be used to validate the product line commonality analysis and it also will make up the *decision model* document [24] that will be used in the development of product line members. The relations among the variabilities influence the system structure. The way in which we specify these relations is beyond the scope of this paper.

3.2.2   Variability Decomposition

Concerning the situation where one variability affects several modules, all variabilities identified in the requirements should be refined to a set of *variation points* such that each variation point affects only one module and cannot be further decomposed. A clear decomposition of the variabilities into variation points may not be achieved before start-ing the module decomposition.

As it can be noticed from the description above, the *domain variability specification* and the *domain module decomposition* are two interdependent tasks. The domain variability analysis provides an input to the module decomposition. The system designer has to consider the variabilities identified in the domain commonality analysis and the varia-bility dependencies, in order to develop a system design that is flexible enough to han-dle these variabilities. In turn, the module decomposition may reshape the variabilities specification by refining variabilities into variation points that vary independently and by identifying new module variabilities, i.e. variations that haven't been identified in the problem space.

To achieve this, the decomposition of the system into modules has to be based on a set of principles that increase the degree of reusability, flexibility and localization of chang-es in the system. The key principles that lead to these properties are briefly introduced below:

- the information hiding principle [18] leads to a decomposition of the system into components, each of them hiding a secret that can vary independently of the rest of the system. It is the primary principle that is applied in the domain module de-composition process. It is guided by variabilities across the family members and by a separation of system aspects that are likely to change over time and are not necessarily stated in the commonality analysis.

- the generalization principle is guided by the generalization of modules that ex-press domain specific aspects by separating the domain specific aspects from the parts that are independent from the business domain. The application specific module will represent a specialization of the general module, achieved by param-eterisation, composition or inheritance. This leads to modules that can be used across domains.

- the virtual machine principle is concerned with defining common interfaces for hardware and software systems used by the program family, in such a way that the family members are not depending on a particular hardware or software used.

The last two principles might be seen as special cases of information hiding but are mentioned because they are worth special consideration in developing program fami-lies.

4   Mapping of Variabilities into Modules

After the variabilities of the program family's members have been identified and de-composed into variation points, each independent variation point is encapsulated into

one module. A variation point can affect a module and the family design in one of the following ways: by varying the module interface, by supplying a different module implementation, and 3) by the presence of a variant module.

4.1 Module Interface Variations

A variation in the module interface influences programs that use this module, too. Obviously, this should be avoided. Following the principles of **information hiding** and **virtual machines** [18], a good design can restrict the effect of variation points to a module's internal implementation whilst preserving the same module interface for all module variants considered.

Nevertheless, there are cases when designers have to deal with variations in the module interfaces i.e. using legacy software, COTS components, interacting with other software systems. It is well known that by abstracting from the module variants, one can define a common abstract interface that all the module variants will satisfy. The real issue is how to design a meaningful abstraction. Table 1 presents some solutions to build an abstract interface for the case when the variation might affect the module interface.

4.2 Module Implementation Variations

From the implementation perspective, a variation point that is encapsulated inside a component can take one of the following forms, described by Weiss in [24]:

- **values of parameters**: the module implementation contains some parameters which values can be set during any stage in the application engineering phase.

- **template modules:** the module implementation represents a template in which lines of code are usually edited during the implementation phase. This solution limits binding to the implementation phase. This form of variation is not appropriate especially in the case of black-box frameworks, where the application developer does not have access to the implementation. Moreover, if not done very carefully, and perhaps automated, the management of these template modules can become a nightmare for maintainers.

- **alternative module implementation** having the same interface (see also [4]): the framework provides variants of the module implementation with the same interface. The corresponding variant implementation can be chosen during application design, implementation, configuration or run-time.

Variants Differences	Solution
different program signatures	1. define the abstract interface as module interface adapter, which exposes a generic interface and provides adaptation of these methods into the ones requested by the variant. This module acts like a module interface, i.e. it implements the Adaptor Pattern or Wrapper [12]. E.g. module interface for system support programs or hardware devices (virtual machine, virtual device (see [4])) 2. solutions that rely on the polymorphism[a]
different number of programs	1. expose the "largest" interface, which contains the union of all programs and leave the methods un-implemented if they are not going to be used[b] (see [4]). 2. use an introspection mechanism, i.e. interrogate in the module that uses one of the variants to find the methods of that variant module.

Table 1: - Solutions to keep the variability inside the module internal definition

a. Instances of this principle are: the static parameterised types (like generics in Java [3]) or inheritance.
b. The program implementations that are required only by some of the module variants can even be encapsulated in separate submodules or subclasses according to the separation principle.

The two important goals of the program family approach are to promote **reuse**[1] of the code as much as possible and to provide **flexibility** in choosing the variant. Different

binding times have an influence on the flexibility, e.g. a configuration or run-time binding provides more flexibility than binding in the implementation phase. However, binding variabilities at a late phase of the software development process implies problems with respect to system performance. Thus, there has to be a trade-off between flexibility and other system properties.

Parameterisation satisfies the above-mentioned goals best, as the entire module implementation is reused and the parameters' values can be set at any stage in the application engineering process, including run-time, but it has a limited applicability. The template module approach limits the binding of variability to compilation time, because it allows variations in the code, while alternative module implementations sometimes reduce the reuse of the common code.

To improve reusability and flexibility in binding the variability, the following 'framework component reorganization' solutions (see [11]) can be used.

- **separation principle**: Exploit the commonality of the possible module variants by factoring the common part(s) in a separate module and express the variability between the module's variants in separate sub-modules. In this way variations are localized in small submodules. This solution provides great flexibility by allowing the variant submodules to be chosen during any stage in application development and promotes reuse of the common code. Design patterns like the Proxy or the Strategy [12] are realizations of this principle.

- **unification principle:** In object-orientation the mechanism of inheritance is introduced. In this paradigm, the concept of classes replaces the notion of modules. Inheritance can be used to increase flexibility and reusability by creating a base class that defines the common parts. Each variant class inherits the common parts from the base class and overrides the varying parts. In this way the common implementation is reused.

One can see these solutions as information hiding and generalization principle respectively, but at a lower level of granularity. Usually, the separation principle is preferred, because it can be used in a black-box approach to assemble systems (see [7]), whereas inheritance is often considered problematic (see [23]). Inheritance is easily misused to introduce links between things that should not be linked. If there is no shared aspect of the specification, there should be no shared aspect of the code. Instead, both units should use a common third component.

5   Binding Variability

The variabilities are bound during different stages of application engineering[1]: **application design, compilation, linking, configuration or run-time** [13], [16]. The decision when to allow the binding of variabilities is made during the product line architecture development and influences the design and the implementation of the family architecture. In the case of product lines that are maintained and extended by the same company, binding of variabilities is more frequent during design, implementation and compilation. In contrast, in the case of frameworks (especially black-box frameworks), variations are bound during installation, configuration or run-time by the application developer.

As it is stated in Section 4.2, a variation point can be a parameter value, some lines of code or a completely different module implementation. A variability that is comprised

---

1. The "reuse" may not be seen as an end in itself but a means to another end such as cost reduction, reduced maintenance effort, quicker response to module demands, etc.
1. In [24] it is stated that application engineering has the purpose *to explore quickly the space of requirements for an application and to generate the application.*

by several variation points of different kinds will require synchronization of the binding, although it can take place at different stages, in order to adhere to all imposed constraints.

As an example, let us consider an optional requirement that imposes the presence of a component *A* and a variant *b1* of a program in component *B*. The implementation of the program *b1* depends on the presence of component *A*. Thus, if at high-level design is decided that the component *A* will be used then at detail design, or implementation phase the possible variant of the program *b1* is already determined.
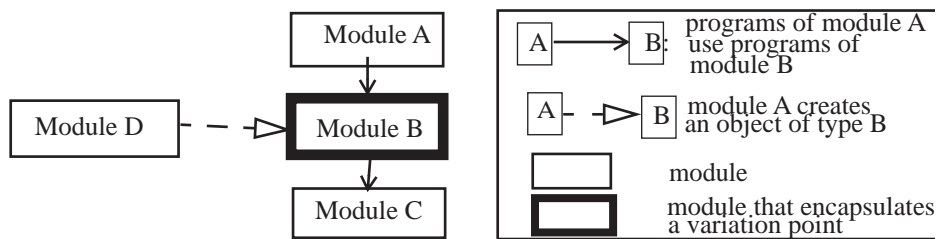

Figure 1   Module Relations

5.1   Binding the Module Variants

In this section we want to identify how a module, which encapsulates a variability, and thus a module that varies, can influence the design or implementation of other modules in the system. Based on the module-uses relation[1] we distinguish three main cases for modules that encapsulate a variation point. These cases are depicted in Figure 1, where the "Module B" is considered to be the variant module (encapsulates a variation point). Subsequently, each of the possible cases will be briefly discussed. The possible relations between modules might also be combined, e.g. a module creates the variant module first and then uses it. However, we argue that the relations can still be treated independently.

- **the variant module's programs are used by other modules**. Given that all the alternative implementations of the module that vary have the same interface, there is no influence on the modules that use programs of the variant module. In this case we assume that a reference to an instance of the variant module is obtained from outside. E.g. In Figure 1, programs of variant Module B are used by Module A. As long as all variants of B have the same interface there is no need for any concern in the modules that use B about the encapsulated variability. In the case of variations in the module interface we suggest a redesign according to the solutions specified in Section 4.1.

- **programs of the variant module use programs of other modules.** In Figure 1, the variant Module B uses programs of Module C. We consider this trivial because the module that is used by the variant module either

  - does not depend on the variation encapsulated in the variant module, or deals with the variation encapsulated in the variant module but it is designed to be general enough not to depend on the module variant chosen, or

  - depends on the module variant chosen and, thus, contains another variation point that is part of the same variability. In this case, the dependency is not a problem, because it was planned during module decomposition.

---

1. The module-uses relation refers to modules that use other modules. The module A uses module B if a program in A invokes a program in B, or A contains a reference to module B

- **the variant module is created by another module**. This case concerns modules that have to create (to obtain a reference) to a particular implementation of a variant module. E.g., in Figure 1, Module D instantiates a variant of Module B. Depending on the form of variability and the binding time, the selection or instantiation of a particular variant module might itself depend on a parameter (see Table 2). In the case of alternative implementations we need a parameter to make the distinction which particular implementation has to be instantiated. We refer to that parameter as *binding parameter*.

For example, let us consider two variant modules, i.e. two alternative implementations of a module that have the same interface. Both modules are supported, but only one will be used. The selection can take place at run-time by evaluating the binding parameter that identifies which one of the alternative module implementations will actually be instantiated.

In the case of object oriented approach instantiating a variant module is a tighter relation then the use-relation as the module that instantiates the variant module has to point to a specific implementation of the variant module. This lead to a strong dependency between the modules (beyond the interface level). According to the type of variation of the module that has to be instantiated, solutions to avoid this strong coupling are described farther.

Type of Variant (implementation)	Binding Time	Type of Variation	Instantiation Method	Solutions
parameter	any	all	-	not necessary
lines of code	impl.	all	-	template module
alternative module implementation	design impl.	all	binding param.	set the binding parameter during the implementation.
	compil. config. run-time	single	binding param.	single variant pattern solution
		multiple	binding param.	multiple variant pattern solution
		optional	binding param.	optional variant pattern solution

Table 2: - Instantiating Module Variants

Single Variant Pattern Solution. Using the inheritance principle, all the module variants, in this case classes, have a common base class. The classes that use programs of a variant module, i.e. a variant class, use instead the program of the base class. The base class can be responsible for instantiating a variant module. All the alternative module implementations are present in the system during the binding phase. Based on the value of the binding parameter set during the binding phase, the base class determines the corresponding variant to be instantiated (e.g. Factory Pattern [12]).

Multiple Variant Pattern Solution. The multiple variant pattern solution assumes a repository of module variants during run-time, with all variants having the same interface. The value of the binding parameter determines the corresponding module variant that has to be instantiated. Concrete solutions in this case are: (1) the (Abstract) Factory Pattern [12] that will instantiate the variant, (2) a repository of module variant instances that allows the selection of the corresponding module variant based on the value of the binding parameter or (3) a dynamic loading mechanism.

The value of the binding parameter can be set during compilation, configuration time or can be determined during run-time. Obviously, this might allow several instances of a module to be created at run-time. However, we recommend keeping the number of modules that are in charge of instantiating the variant module(s) as low as possible, e.g. by keeping a repository or a factory object, as well as the number of instances of the same module in order to improve performance and eliminate unnecessary sources of errors.

Optional Variant Pattern. There are two ways to model optionality:

1. to model it **as a single variant** where one variant is the module variant containing the methods implementation and the second variant is an empty module that contains only the interface declaration but does not provide any implementation.
2. to ignore the presence of an optional module in the system. Employ a factory object or a repository that is in charge of the creation of the module if the module variant exists in the system or will not do anything otherwise. This approach is less flexible when several other modules use programs of the optional module. All these modules have to be designed taking the absence of the variant module into consideration.

6   Summary and Future Work

The program family development process is concerned with the reuse of artifacts related to requirements specification, software design and implementation. Managing variabilities is the cornerstone of the program family development process.

This paper raises issues and provides solutions for dealing with variabilities in the software families that involving the identification of variabilities in both problem and solution spaces, mapping of variabilities to modules and flexible software design and implementation solutions to handle these variabilities. An important contribution of this paper is the discussion of the relation between product lines, program families and frameworks.

Previous work [1], [24] uses this terms as if they were synonyms. We argue that the program family concept is more general and the others are special cases; this means that the specialized approaches are compatible and mutually supportive. It also means that other approaches can be added to this arsenal of methods.

The paper also emphasises the distinction between variabilities in the requirements and variation points, the actual unit of variation, which are meant to be encapsulated in exactly one module.

We focus on the translation of the variation points into the system's module design and implementation, and on the design solutions to manage the resulting module variants. Therefore, flexible solutions to express the variation in modules are presented. The central point in this process of expressing the variability in the module are the issues regarding binding the modules that encapsulate the variation points by making the clear distinction between the means to express the variation in a module on one hand and of the types of module-uses relations on the other hand. The presented design solutions for instantiating the module variants are based on the form of variation in the module and the time chosen for binding.

The ideas presented in this paper are referring to one side of the process of managing variability in program families only. Other important issues are the specification of the variabilities in the requirements and the specification of the module variants. The current approaches based on the natural language specification, UML [1], [7], or on development of specific domain languages together with the compilers or translators for these languages [24] do not enable us to precisely define software or are difficult to use. We believe that a set of documents must characterize any program family and are investigating the use of functional documentation [20], [21] with tabular notation as a way to improve our ability to develop high quality program families. The research is focused on the adaptation of the tabular specification method to handle program families' variabilities.

References

1. Atkinson C., Bayer J., Muthing D.: Component Base Product Line Development: The KobrA Approach, 2002
2. Bosch J.: Design et Use of Software Architecture (Adopting and evolving a product-line approach), Addison Wesley NY, 2000
3. Bracha G:, Cohen N., Kemper C., Marx S., Odersky M., Panitz E., Stoutamire D., Thorup K., Wadler Ph.: Adding Generics to the Java Programming Language - Spec. Draft, 2001.
4. Britton H. K., Parker A. R., Parnas D.: A Procedure for Designing Abstract Interfaces for Device Interface Module, Software Fundamentals, Addison-Wesley, 2001, 295- 314
5. Clements P., Northrop L.: Software Product Lines: Practices and Patterns, Addison Wesley, 2000
6. Coplien, J., Hoffman, D., Weiss, D.: Commonality and Variability in Software Engineering, IEEE 1999, 37-45
7. Dragomiroiu, M.: Enterprise Frameworks for Web-Applications, Master Thesis, University of Limerick, 2003
8. Faulk S.: Product-Line Requirements Specification (PRS): an Approach and Case Study", IEEE Software 2001, pp. 48-55
9. Fayad M.E., Schmidt D.C., JohnsonR.E.: Building Application Frameworks: Object-Oriented Foundation of Framework Design, Wiley, 1999
10. Ferber S., Haag J., Savolainen J.: Feature Intercation and Dependencies: Modeling Feature for Reengineering a Legacy Product Line, Proceedings SPLC2, USA, 2002, 235-256
11. Fontuara M., Pree W., Rumpe B.: The UML Profile for Framework Architectures, Addison-Wesley, 2002, 68-112
12. Gamma E., R. Helm, R. Johnson, J. Vlissides: Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing (1995).
13. van Gurp J., Bosch J., Svahnberg M.: On the Notion of Variability in Software Product Lines, Proceedings. Working IEEE/IFIP Conference on Software Arhitecture, 2001, 45 - 54
14. Keepence B., Mannion M.: Using patterns to Model Variability in Product Families, IEEE Software, 1999, 102-108
15. Mannion M.: Using First-Order Logic for Product Line Model Validation, Proceedings Second International Conference, SPLC2, USA, 2002, 176-187
16. Jaring M., Bosch J.: Representing Variability in Software Product Lines: A Case Study, Vol. 2379. Springer-Verlag, 2002, 15 -36
17. Johnson R. E.: Frameworks = (Components + Patterns), Communications of the ACM, Vol.40/No.10, 1997, 38-42.
18. Parnas D. : On the Criteria Used in Decomposing Systems into Modules, Communications of the ACM, vol. 15, 1972, 1053-1058
19. Parnas D.: The Design of Program Families, IEEE Trans. Software Eng. vol 2/1, 1976, 1-9
20. Parnas D.: Tabular Representation of Relations, CRL Report 260, McMaster University, Communication Research Laboratory, 1992
21. Parnas D., Madey J., Iglewski M.: Precise Documentation of Well-Structured Programs, IEEE Trans., 1994, 948-376
22. Theil S., Hein A.: Systematic Integration of Variability into Product Line Architecture Design, Proceedings SPLC2, USA, 2002, 130-153
23. Wegner, P.: Panel on Inheritance, Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Florida, United States, 1988, ACM Press.
24. Weiss D., Lai R. T.: Software Product Line Engineering- FAST, Addison Wesley, 1999

3   *John M. Hunt*
     The Library Considered as a Product Line

The Library Considered as a Product Line

John M. Hunt

Department of Computer Science
Clemson University
Clemson, SC 29631 USA
hunt2@clemson.edu

Abstract. Software libraries provide the oldest and most widespread form of software reuse. Due to a variety of factors traditional software libraries have reached their limit of usability. This paper contends that the limiting factor on library use is the current one-size-fits all approach to providing libraries. Library users have overlapping but not identical requirements. A better way of satisfying these requirements is to use a product line approach to provide customized libraries. These customized libraries should allow the library to be more adaptable to different related uses, be easier to use and provide more of the product's functionality. Using a product line approach to customize libraries can provide a fundamental change in library use: the library can be extended into related areas more easily, library users obtain better usability through the elimination of irrelevant options, and products should have higher reuse levels because of the better fit of the library to a particular project.

Note: I am a second year Ph.D. student. I hope to present a dissertation proposal in Fall 2004 based on the ideas in this paper.

Importance of Software Libraries

Reuse has been identified as one of the key issues in software engineering since this first conference on software engineering was held under NATO auspices in 1968, where McIlroy identified the need for a catalog of "software IC's" [McIlroy 68], to enable reuse of software. To provide these components today, libraries are almost universally used. Most introductory programming courses include the use of collections (lists, maps, etc.) from a class library. Many such courses teach collections before arrays and, increasingly, omit arrays altogether.

Most current software projects will use a variety of libraries. Typically a general purpose library supplied with the language of choice will be used for collections and utility routines, such as dates. There will be a library to interface with the operating system for IO, processes and threads, and simple communication such as pipes and sockets. More advanced communication methods come from specialized libraries for services like CORBA infrastructure or message queues. There will be a library to provide a language specific interface to a database supplied by the database vendor. GUI libraries support a variety of user interface styles, Microsoft Windows, X11, Web, etc. In addition to these general purpose libraries, are a variety of domain specific libraries.

Problems of Software Libraries

While libraries have been an undisputed success in providing reuse, there are signs that they have reached their limit. Working against component libraries are:
- Sheer Size
- Generality
- Complexity
- Difficulty in integration
- Difficulty in evolution

While libraries are not the only large scale software product, they differ from other products in that they are designed to be used by another set of developers. For the developer using the library, selecting a library element that provides a lot of functionality, like a socket, provides more assistance than using a library element that provides a small amount of functionality, like a string copy. However, in providing a lot of functionality, these elements quickly become increasingly specialized. As they become specialized they are usable in fewer places. The more functionality an element provides the fewer opportunities there are to use it [Krueger 92]. This means that conventional software engineering techniques, such as inheritance, are helpful in mitigating code count for library development organizations but they do not address the size problems that are unique to libraries.

Based on this observation, there are two approaches to growing a library. One is to have lots of simple lowest common denominator functions. However, unless these functions are very commonly used, the overhead of finding and understanding them will exceed the small amount of functionality they provide. The number of sufficiently common functions will soon be exhausted.

Alternately, the library can supply functions that handle increasingly large pieces of work; however, the functionality provided will become increasingly specialized and thus will be applicable to fewer situations. In addition, as the functionality increases it becomes hard to avoid an architectural impact. It is desirable for a GUI library to help manage keyboard and mouse events rather then just drawing the screen. To do this they need to read these devices. This quickly leads to a need to provide an event loop. To get the event loop to work the GUI library typically provides the programs main procedure. This in turn dictates how the application can be structured.

The developer using the library is concerned with library size, not in terms of line count, but interface complexity, which can be measured by the services provided and parameters required. For reuse to occur, the developer must complete the following: find the element, select the element, understand the element, and (optionally) adapt the element [Dusink 95]. Here size creates a point of tradeoff, that ultimately limits the functionality that can be supplied. More elements make it more likely that the needed element is available, but less likely that it can be found. Having more elements means that out of those found it is more likely that the selected one will be a good fit, but at the cost of a larger selection process. More elements mean more to understand. It may be hoped that more elements means that there is one available that does not need to be adapted, but it may instead force more adaptors to be written. The

library must be big enough to have all the functionality that the client wants, but ideally, no bigger.

Libraries for a given project are likely to come from different sources. This is almost inevitable as one of our motivations for using a library is that it embodies expertise in a particular area, and it is highly unlikely that a single source will have expertise in all areas. The libraries necessarily make assumptions about use. These may be as simple as a choice between using doubles instead of floats, as subtle as who is responsible for freeing memory, and as pervasive as how to report errors (exceptions vs. return codes vs. special values). Different libraries will take different approaches. The library will then need to be adapted so that it will work with other code, the application code and that from other libraries. The problem is common enough to spawn its own term – glue code.

Finally, libraries bring with them the problem of evolution. Libraries are intended to be long lived in order to amortize their higher cost, which comes with their reusability, over many projects. Compounding this is the need for multiple libraries, each of which will tend to evolve at an independent rate. Since the consequences of removing (or modifying the interface) an element still being used is grave and since the library developer has no way of knowing what is still in use, libraries evolve solely by growing. The versioning system for DCOM is a prime example, although all libraries share it to some degree. Items may be marked as depreciated, but support for actually removing them is minimal. A problem here is that different projects have very different needs. A new project may have a great desire to avoid depreciated elements; while one at the end of its life cycle may have an economic imperative to avoid changes.

Customized Libraries through Product Lines

I would suggest that all these issues point to an underlying problem in the current approach to providing libraries – which is the incorrect assumption that *one size fits all*. It is unlikely that any project will actually use all or even a significant portion of a library. The reason multiple products make use of a given library is an overlapping, although not identical, set of requirements in the area provided by the library. Very different products that provide a GUI, still have a need for many of the same elements - windows, menus, etc. The question is whether the current one size fits all approach is the best way to satisfy these sorts of overlapping requirements.

One of the most promising approaches in software engineering to satisfying related and overlapping requirements is the Product Line approach. I propose that libraries be architected using a product line perspective. This in turn will allow the user of the libraries to work with a library that is customized for their particular project. Normally, such a customized library would be too expensive for most projects. However, a product line approach is able to drastically cut the number of users needed to make a custom version economically attractive.

My focus here is on product lines for *libraries* NOT on product lines for *components*. There is still much valuable work to be done on components; however, what I am interested in researching is what happens to additional *dimensions* of a

library of *many* related elements. The most obvious difference are concerns that effect many items in the library. For example, memory management, or a particular security requirement (perhaps signed objects). Current libraries address these issues but they do so by providing a single one-size-fits-all solution, which they try to make adaptable through techniques, such as parameters, which often result in a complex interface. By using a product line approach, one library customer, with high security needs, can have signed objects, while another customer can have it without.

The fundamental change that I am suggesting is to allow the user to have a library customized to his needs. It is ironic that the purchaser of a car has a more customized product then the purchaser of a software library. The application of a product line approach can provide an economically viable way to allow the library consumer to select those areas that are important to a particular project.

Overall, using a product line approach creates two fundamental changes for libraries: First, the size barrier of the library is shattered because size of the provided features is de-coupled from the size of the library used in a given product. Second, there is a fundamental change in the relationship between the library and user because the library becomes customized to fit the user's needs. This means that the library can be extended and provide a larger part of the final product.

Domain Analysis of Libraries

The first step in undertaking a product line approach is domain analysis. What I am attempting to provide a domain analysis of libraries, not domain analysis of a library, or even a domain analysis procedure to be used for a library. What I want to identify are those elements of interest that are common across all or at least many libraries. This is not a common approach. To take one recent example Czarnecki's DEMRAL [Czarnecki 00] work provides a process for providing a library for a particular domain. While the approach may be applied to different domains, it does not attempt to analyze issues that span domains or the general effect of libraries on an application. In a sense, this is the compliment of what I am trying to achieve. DEMRAL looks at providing specific instances of libraries and my proposed work looks at what is not specific to any particular library.

To begin our domain analysis of libraries, a definition of a library is needed. Finding a meaningful definition[1] for library has proven surprising elusive. As a result, I provide my own definition.

---

[1] All too typical are definitions like this from the Oxford's Dictionary of Computing: "program library (software library) A collection of programs and packages that are made available for common use within some environment; individual items need not be related. A typical library might contain compilers, utility programs, packages for mathematical operations, etc. Usually it is only necessary to reference the library program to cause it to be automatically incorporated in a user's program. See also DLL." [Oxford 96]

Definition of Library

A library is a collection of code intended for repeated use, across multiple applications, bottom up in design, with a passive control flow being invoked by a client, providing small to medium grain members, primarily functions and objects. A library provides implementation reuse; although, of course, it embodies some design and domain expertise. A library attempts to be architecturally neutral. Libraries may use other libraries but they do not form and can not be composed into complete applications, as they have no main, and preferably no externally visible control structure. Libraries are also passive in the sense that they do not receive their own allotment of CPU cycles, either as a process or a thread; although an element of a library may require its client application to provide it with a thread.

The primary connection between the library and the application consists of object instantiation and method calls by the client application. Applications combine or aggregate libraries.

A library has a release and deployment cycle independent of its applications; although, its release cycle may be tied to system software such as operating systems and language releases. Libraries are visible to system software, relying on languages to support separate module compilation, linkers to statically assemble or operating system support to dynamically assemble libraries and applications.

Library Compared with Other Reuse Approaches

Libraries may be contrasted with other reuse approaches, particularly frameworks. Frameworks are intended to create an individual application, providing a top down design. The framework provides the application control flow including the program's main, this control flow is characterized by *inverted control* where client code is executed primarily via callback initiated by the framework. Frameworks, generally, aim to provide the complete structure for an entire application; they provide an implementation of a particular architecture and are not intended to work with other frameworks. The connection between the framework and application specific code is provided by callbacks and sub-classing from classes defined by the framework. It is not possible to deploy a framework independently of a particular application. Frameworks directly provide implementation reuse. They indirectly provide design reuse by embodying a particular architecture. Frameworks are not visible to and do not require system software support. Applications specialize frameworks.

Another reuse approach is the platform. A platform defines the runtime environment for a program. At a minimum this is a hardware / operating system combination; although, it may be quite a bit more. An EJB container is a platform that provides considerably more in the way of services then most operating systems; for example, object invocation. Since libraries enhance the runtime environment, some collection of libraries would typically be included in a platform. The main role of a platform is to provide a standard set of services to an application.

Table 1. Comparison of implemenation reuse approaches

|	Library	Toolkit	Framework	Middleware	Platform	Component
Design approach	Bottom up	Middle out	Top down	Bottom up	System	Middle
Granularity	Small	Medium	Large	Medium	Large	Medium
Function Access	Method call	Call-back	Call-back	Call-back	Interprocess	Interprocess
Control flow	Passive	Events	Main loop	Events	Process	Interprocess
Deployment	Independent	W/App	W/App	Independent	System	Independent

Table 1 compares a number of reuse approaches, to place the library approach in perspective. All of these are primarily methods for implementation reuse, as opposed to design reuse provided by patterns and architectures.

From this several continuums of traits emerge. As shown in figure 1 moving from libraries to frameworks the reuse elements become more tightly coupled with each other and they have more control over the architecture of the application.
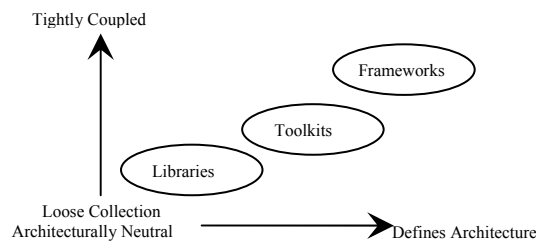
Fig. 1. Comparison of 3 different reuse approaches in terms of relationship to the architecture and coupling between elements

Another continuum is how intermingled the reusable and application specific code is. Here a continuum is formed consisting of platform, library, and framework. Platform being the most separated from the application and framework being the most intermingled with the application. This can be expressed by saying that an application runs *on* a platform, *using* libraries, *within* a framework.
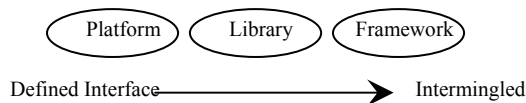
Fig. 2. Comparison of 3 different reuse approaches in terms of separation of reuse elements from the application

Other Facets of Libraries

Having defined what a library is and how it differs from other approaches of reuse, we can also discuss desirable qualities of libraries. A short list might include:

Completeness, Consistency, Ease of learning, Ease of use, Efficiency, Adaptability, Extendibility, Ease of integration, Intuitiveness, and Robustness [Korson 92]. Many of these are rather general, such as being efficient; although, in many cases there are additional challenges present in satisfying these for a library.

There are also a number of issues in developing good libraries that are not present in other approaches:
- Size – since libraries provide building blocks for other applications there is not the same sort of natural limit to what is provided that exists for even large applications.
- Integration issues – libraries are not useful unless they are used by application, which in turn is designed in isolation from the library. Noteworthy here is the possibly impractical assumption that the library is architecturally neutral.

Research Ideas

It is my hope to have been persuasive that libraries constitute a specific and interesting domain for study and that they have problems, which at first glance, seem amenable to a product line approach. I see the following issues to be pursued:
- Further definition of library commonalities. Particularly, functional commonalities among libraries. For example, it would seem that all libraries must provide for error handling.
- Definition of variability points for libraries. There are some obvious candidates such as the ability to ship the library for multiple platforms.
- The problem of adapting to multiple applications. For example, a library must provide for an error handling policy and mechanism. It must also integrate with multiple applications each of which may use a different error handling. There is some work in the "active library" area that may be related to this.
- The problem of customization vs. general deployment. I have argued that customizing a library for an application provides many advantages. However, in the definition section I pointed out that one of the characteristic advantages of the library as a reuse approach was the ability to deploy the library independently of a particular application and have it accessed by multiple applications on the same system. How to reconcile these two?
- Does the resulting library code produce the promised improvements? Is there good code reuse? Is programmer productivity improved? Are library variations reliable?
- Does using a customized library affect the development process? Possible effects include: an extra stage in the architecture definition to specify library variants, changes in deployment to include the correct version of the library, and feedback from coding phase that a different version of the library is required.
- How should this be implemented? Particularly, which techniques would allow a feature to be applied across a library without developer intervention? How to

apply a feature to subsets of the library? For example, a search may be applicable to all containers but not to GUI components.

Conclusion

Software reuse is a key strategy in our ability to develop affordable, reliable software. While the importance of reuse has long been known, having effective techniques for reuse has often been surprising elusive. Libraries are the oldest most widespread techniques for reuse. However, the current one size fits all approach to providing libraries seems to be reaching a practical limit. While the limit on library size has been extended through standard software engineering techniques, simply re-engineering the internals of the library will not have a fundamental effect. The product line methodology represents a much newer approach to software reuse, which focuses on providing related variations in software. Product lines make it economically feasible to provide mass customization of a product to fit the needs of a user. It is my contention that product line approach is a complementary technique to libraries. Applying product line techniques to libraries make it possible to create highly customized libraries that fit the needs of its client product and thus extend the libraries usefulness.

References

[Czarnecki 00] Czarnecki K. and Eisenecker U.: Generative Programming: Methods, Tools, and Applications. Addison Wesley, Boston (2000)

[Dusink 95] Dusink L. and van Katwijk J.: Reuse Dimensions. In Proceedings of the 1995 Symposium on Software reusability. Seattle, Washington ACM Press (1995) 137-149

[Korson 92] Korson T. and McGregor J.: Technical Criteria for the Specification and Evaluation of Object-Oriented Libraries. In Software Engineering Journal. IEEE 7(2) (1992) 85-94

 [Krueger 92] Krueger C.: Software Reuse. ACM Computer Surveys 24 (2) (1992) 131-183

[McIlroy 68] McIlroy M. D.: Mass produced software components. In Software Engineering; Report on a conference by the NATO Science Committee (Garmisch, Germany, Oct. Naur, P., and Randell, B., Eds. NATO Scientific Affairs Division,Brussels, Belgium, (1968) 138-150

[Oxford 96] Program Library. In A Dictionary of Computing. Oxford University Press, 1996. Oxford Reference Online. Oxford University Press. <http://www.oxfordreference.com/views/ENTRY.html?subview=Main&entry=t11.e4144>

4   *Waraporn Jirapanthong*
     Towards a Traceability Approach for Product Family Systems

Towards a Traceability Approach for Product Family Systems

Waraporn Jirapanthong

Department of Computing
City University
Northampton Square, EC1V 0HB, UK
w.jirapanthong@soi.city.ac.uk

Abstract. Traceability has been proposed as an important activity in software engineering to guarantee quality in the life-cycle of software system development. This work aims to enable traceability for product family software systems, in particular, for identifying both common and different aspects between the various members of a system. Our approach supports automatic generation of traceability relations among artefacts for product family software systems. We have identified nine types of traceability relations. The approach is rule-based: the rules are represented in XQuery and the requirements artefacts are represented in XML. In this paper, we describe the approach for different types of product family artefacts, namely feature model, use case specifications, and class diagram.

1. Introduction

Requirements Traceability (RT) has been proposed as a supportive technique for software system development [14][21][23]. Traceability can facilitate the development process and ensure quality of the system. In particular, traceability relations can support evolution and reuse of software systems by comparing artefacts of new and existing systems, completion of the final-system by validating the satisfaction of requirements and the system, understanding the rationale of design and implementation, and analysis of the implications of changes in the system.

Unfortunately, despite its importance requirements traceability is rarely established due to the fact that traceability is subject to manual tasks. Some approaches [5][16][26] assume that traceability relations should be established manually, which is error-prone, difficult, time consuming, expensive, complex, and limited on expressiveness. Other approaches have been contributed to support semi- or fully-automatic generation of traceability relations [1][10][12][19][20]. However, the traceability relations generated by the majority of these approaches do not have strong semantic meaning necessary to support the benefits that can be provided by traceability. Moreover, traceability of complex systems like product families is much more ambiguous and difficult to establish.

An exception is presented in [25][29], in which a rule-based approach has been proposed to allow automatic generation of traceability relations between different types of requirements documents such as customer requirements specifications, use-case specification, and analysis object model. There are three different types of traceability relations, namely *overlaps*, *requires* and *realises* relations. In the approach, traceability rules identify the relationships between requirements documents by matching syntactically related terms of customer requirements and use-case specification with semantically terms in the object model. The traceability relations are established by the matching of rules. In this paper we extend the contributions of this previous work [25][29].

Our approach has been proposed to allow automatic generation of traceability relations between documents under the domain of product family software systems, in particular, to identify common and variable aspects of the product members. The documents of our concern are based on feature-based methodologies due to the fact that customers and system developers communicate with each other in terms of product features when developing product family systems, and also based on object-oriented methodology due to its support for software development. FORM [18] is applied as the basis of our approach due to its simplicity, maturity, practicality, and extensibility in software development process for product family systems. The goal of our work is to allow generation of traceability relations between documents proposed in FORM [18] i.e. feature, process, and subsystem models, and some object-oriented diagrams such as component, class, and state chart diagrams. The work presented in this paper concentrates on functional requirements specification, feature models, and class diagram.

In the approach these documents are represented in XML and the rules are expressed in XQuery [28]. The documents are translated into XML-format due to several reasons: (a) XML has become the

de facto language to support data interchange among heterogeneous systems, (b) the existence of applications that use XML to represent information internally or as a standard export format, and (c) to allow the use of XQuery as a standard way of expressing the traceability rules.

The remaining of this paper is structured as follows. In Section 2, we describe the main documents we propose to use for product family system development. In Section 3, we present an overview of our approach: the different types of relationships existing in product family software systems; and the traceability rules including representing in XQuery. In Section 4, we describe existing related work. Finally, in Section 5 we summarise our approach and discuss directions for future work.

2. Types of documents

In this section, we give an overview of documents used in our work. Due to our XML-based approach, the documents are translated into XML. We have created DTDs for the functional requirements specifications and feature models, and use XMI for the class diagrams. We present examples for some of these documents for a product family related to mobile phone systems.

Functional Requirements: Functional requirements specifications are use-cases defined according to a template proposed in [8]. A use case contains *title*, *status*, *region*, *description*, *level, preconditions*, *postconditions*, *primary_actors*, *secondatry_actors*, *flow_of_event*, *exceptional_events*, *superordinate_use_case*, and *subordinate_use_case*. In order to deal with the natural language sentences present in the use case description, we propose to mark up the words of the sentences by using XML elements that indicate their grammatical role in the sentence. The grammatical role is identified by using the part-of-speech tagger called CLAWS [9]. Figure 1 (a) presents an example of a use case description for sending data from a mobile phone. This use case is related to a member (MP1) of the product family. In this example the word "Send" appears as an element <VVB> denoting that it is a base form of a lexical verb, while the word "data" appears as an element <NN0> denoting that it is a neutral noun. For a complete description of the tags given by CLAWS, please refer to [9].

Feature Model: In our work we use an XML representation of the feature model proposed in [17]. The model is structured like a graphical tree. Each feature of the system is represented in a feature model and includes the following information: *feature_name*; *description; issue_and_decision* describing issues and decisions that arise during the feature analysis process; *type* that classifies the feature in terms of *capability*, *operating environments*, *domain technology*, or *implementation techniques*; *commonality* denoting if a feature is *mandatory*, *alternative* or *optional*; and *relations* representing relationships with other features. An example of a feature model for a "Bluetooth" feature presented in mobile phone systems is shown in Figure 1 (b). This feature is "optional" (i.e. not every member in the family needs to have this feature) and of type "capability" (i.e. Bluetooth is concerned with an ability of the system).

Class Diagram: The class diagram is actually UML class diagram, including classes, attributes, operations, associations, and generalisation relations between classes and is represented in the eXtensible Metadata Interchange (XMI) format.

Component Diagram: The components are defined in UML component diagrams. The diagram includes components, classes, and relationships between components and is also represented in the eXtensible Metadata Interchange (XMI) format.

Subsystem Model: According to [18], a subsystem model represents the high-level characteristics of systems, which can be distributed to and executed by different machines. Particularly, the model is packing of physical boundaries i.e. services and operations that are related to logical boundaries (capability and operating environment features). The model shows *subsystem_name*; *external_systems*; and *messages* that are classified as *closely_coupled_message_queue, loosely_coupled_message_queue*, and *message_without_reply*.

Process Model: The process model shows the activities executed by each system in the family. In other words, the subsystem model is elaborated in the process model that shows a set of processes and interactions between the processes. The process model basically includes *process_name*; *information* describing data dependency, functional cohesion, execution frequency, etc; *messages* that are similarly categorized as subsystems (i.e. *closely_coupled_message_queue, loosely_coupled_message_queue,* and

message_without_reply); and *shared_data* denoting which data is being shared by the different processes.

Module Model: The process model is elaborated in the module model, in which a set of components and classes are identified and logically grouped as modules. The module model is tightly associated to all levels of feature model. The module consists of *module_name*; and *links* between modules like *uses* or *inherits*.

State Chart Diagram: The state chart diagram represents behavior aspects of the system. We use UML state chart diagram with *states*; *transitions* indicating possible paths between states; and *events,* represented in the eXtensible Metadata Interchange (XMI) format.

<**FunctionalReqSpec** System="**MobilePhone**" Product_Member="MP1"> <**Use_case** UseCaseID="1"> <**Title**> <VVB>*Send*</VVB> <NN0>*data*</NN0> </**Title**> <**Status**>Common</**Status**> <**Region** Name = "All"/> <**Description**>…<AJ0>mobile</AJ0> <NN1>phone</NN1> <VM0>can</VM0> <VVI>send</VVI> <NN0>data</NN0> <VVN>kept</VVN> <PRP>in</PRP> <AT0> the</AT0> <NN1>phone</NN1> <PRP>to</PRP> <DT0>another</DT0> <NN1>phone</NN1> <CJC>or</CJC> <NN1>device</NN1> <PRP>via</PRP> <NN1>communication</NN1> <NN2>channels</NN2> <AV0>i.e.</AV0> <NN1>*Bluetooth*</NN1>… </**Description**> <**Level**>Primary Task</**Level**> <**Preconditions**>…</**Preconditions**> <**Postconditions**>…</**Postconditions**> <**Primary_actor**>…</**Primary_actor**> <**Secondary_actors**>…</**Secondary_actors**> <**Flow_of_events**>…</**Flow_of_events**> <**Exceptional_events**>…</**Exceptional_events**> <**Related_Information**> <**Superordinate_use_case**>…</**Superordinate_use_case**> <**Subordinate_use_case**>…</**Subordinate_use_case**> </**Related_Information**></**Use_case**>…</**FunctionalReqSpec**>	<**Feature**> <**Feature_name**> <NN1>*Bluetooth*</NN1> </**Feature_name**> <**Description**>…<NN1>Bluetooth</NN1> <NN1>connection</NN1> <VM0>can</VM0> <VBI>be</VBI> <VVN>used</VVN> <TO0>to</TO0> <VVI>*send*</VVI> <NN0>*data*</NN0> <AV0>i.e.</AV0> <NN2>texts</NN2> <NN1>business</NN1> <NN2>cards</NN2> <NN1>calendar</NN1> <NN2>notes</NN2> <CJC>or</CJC> <TO0>to</TO0> <VVI>connect</VVI> <AV0>Wirelessly</AV0> <PRP>to</PRP> <NN2>computers</NN2>… </**Description**> <**Issue_and_decision**/> <**Type**>Capability</**Type**> <**Commonality**>Optional</**Commonality**> <**Relation**/> </**Feature**>
(a)	**(b)**

Fig. 1. Examples of documents: (a) functional requirement specification for use case "Send data"; (b) feature model for "Bluetooth" in XML

3. Overview of the approach

Figure 2 presents an overview of our approach. Initially, the documents are translated into XML by using an *XML translator*. In the case of the class, component, and state chart diagrams, the XMI formats are generated by using commercial XMI exporter (e.g. Unisys XMI exporter for Rational). The *XML translator* is also responsible to add the XML POS-tags in the sentence after identifying these tags using CLAWS. These XML documents are used as input to the *traceability generator* that creates traceability relations based on the rules represented in XQuery [28]. The traceability relations are also represented in XML format. As described below, some of the traceability relations will be used to support identification of new traceability relations (*derived relations*). In the figure this is represented by using the XML-formatted relationships documents as input to the traceability generator.
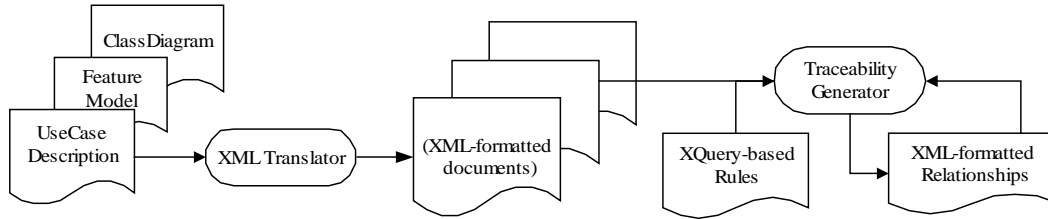
Fig. 2. Overview of the approach

Types of Traceability Relations

Based on the different types of documents described in Section 2, we identified nine types of traceability relations for product family systems. We classify the traceability relations in two groups: *direct* and *derived* relations. The *direct* relations group is concerned with relations that are identified by matching terms in the documents. The *derived* relations group is concerned with relations that are identified based on the existence of other traceability relations identified by our traceability generator. Both *direct* and *derived* relations are sub-classified into other types of relations. Figure 3 present the different types of relations. In particular, *alternative* and *additional* relations allow identification of variable aspects of the product members in a family, while *similar* relation supports identification of common aspects among these products.
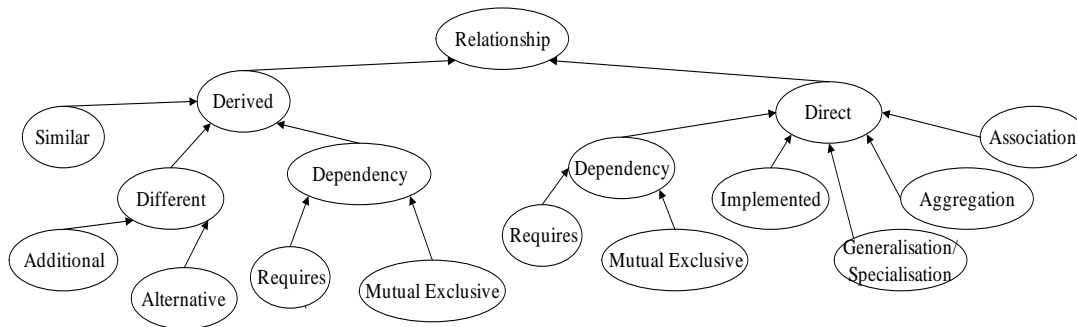

Fig. 3. Types of traceability relations

In the following, we explain the meaning of each type of relation being proposed and illustrate them with examples from mobile phone systems. The term *element* is loosely used to refer to a feature, an object, an item, a functionality, or a characteristic of the system.

1. Association – This relation expresses the association between two elements. For example, a component "network management" is associated with a process "network establishment". The relation is captured between the component in the component diagram and the process in the process model.

2. Aggregation – This relation expresses that an element is composed of other elements. For example, a process "displaying" in the process model is composed of features in the feature model i.e. a screen, a keypad, a speaker microphone.

3. Generalisation/Specialisation – This relation expresses an abstraction between an element and a set of its specialized elements. For example, a "screen" feature is specialized by algorithms, in which are realized into some methods in classes such as "text", "mono-colour", and "multi-colour displaying". The relations are captured between the screen feature in the feature model and methods in the class diagram.

4. Implemented – This relation expresses that an element is implemented by another element. The approach can capture the relation within the same type of documents e.g. within the feature model and between different types of documents. For example, there is *implemented* relation between a "read message" feature in capability layer and "textual display" feature in domain technology layer in the feature model. The relation is implied that the "read message" feature is implemented by the "textual display" feature. Between functional requirements specifications and class diagram, for

example, a "text messaging" feature in the feature model is related to a "sending data" method of a "text" class in the class diagram as an *implemented* relation.

5. Mutual Exclusive – This relation is bi-directional and expresses that two elements require the existence of each other. Note that *mutual exclusive* relations can be both *direct* and *derived* relations. A sample of *mutual exclusive* relation is that a "multi-media messaging" and "integrated digital camera" features requires the existence of each other in the feature model.

6. Requires- This relation expresses that an element requires an existence of another element. Like, *mutual exclusive* relations, the *requires* relations can be both *direct* and *derived* relations. For example, a "send message" feature requires a "signal notification feature". It is implied that a system requires displaying a notice to a user after sending a message.

7. Alternative – This relation expresses different ways of achieving the same functionality in different product members of the family. In particular, the relation is based on *implemented* and other relations. For example, the family of mobile phone systems has "text", "mono-colour graphic", or "multi-colour graphic screen" features. A mobile phone system is implemented by the "mono-colour graphic screen" feature while another mobile phone system is implemented by the "multi-colour mode screen" feature. Thus, two product members are related to each other in term of *alternative* of the "screen" feature. The relation is created between two functional requirements specifications.

8. Additional – This relation expresses extra functionalities that may exist between product members of the family. Like the *alternative* relationship, the *additional* relations are based on *implemented* and other relations. For example, a mobile phone system is implemented by a feature "integrated digital camera" while another system is not. Two product members are related to each other in terms of *additional* "integrated digital camera" feature. The relation is generated between two functional requirements specifications.

9. Similar – This relation expresses common elements in the product members of the family. The *similar* relations are also based on *implemented* and other relations. For example, two mobile phone systems are implemented by a "WAP service" feature. Two product members are related to each other in terms of *similarity* of "WAP service" feature. The relation is generated between two functional requirements specifications.
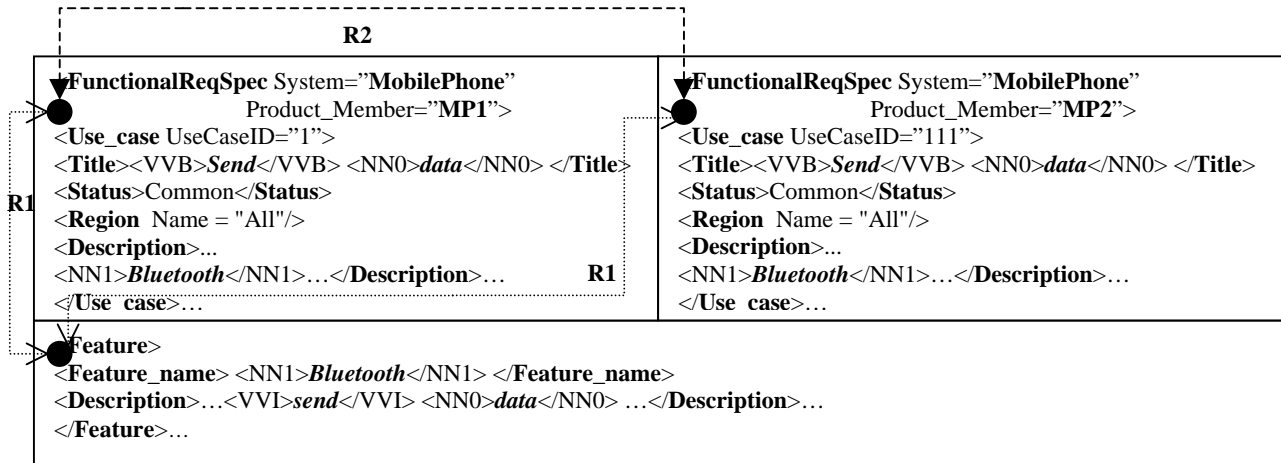
Traceability Rules

In our approach the traceability relations given above are identified by using traceability rules. Our *traceability generator* analysis the rules and establishes the relations if the conditions for the rules are satisfied. Each rule has a unique identifier and a type associated with the traceability relation identified by the rule. The rules can be concerned with the same or different types of documents.

In this paper, we have proposed a number of rules for creating *implemented*, *derived requires*, *derived mutual exclusive, alternative, additional* and *similar* relations. The rules for creating *implemented* relations identify matching terms in the textual contents of documents by comparing the XML POS-tags specifying grammar roles generated by the *XML translator*. The rules for creating *derived requires*, *derived mutual exclusive, alternative, additional* and *similar* relations are based on existing relations.

Figure 4 presents an example of two rules described below.

Rule1 identifies an *implemented* relation between use case and feature model if there is a singular noun (<NN1>) composing the name of the feature that appears in the description of the use case, and the title of the use case is composed of a neutral noun (<NN0>) and a verb (<VVB>) that appear in the description of the feature model. Figure 4 shows that two use cases are related to a "Bluetooth" feature as *implemented* relationship.

Rule2 identifies a *similar* relation between use cases. The relation signifies that two different product members of a family have some similarity if they implement the same feature. This relation depends on the existence of previous *implemented* relations between use cases and feature model (i.e. different use cases and the same feature names). Figure 4 shows a similar relation between two use cases from different product members "MP1" and "MP2" in terms of implementation of the "Bluetooth" feature.

Fig. 4. Examples of direct *implemented* traceability relations, which are created by Rule1, between a use case "Send data" of product member "MP1" and feature "Bluetooth"; and between a use case "Send data" of product member "MP2" and feature "Bluetooth" in the feature model; and *similar* traceability relation in term of "Bluetooth" feature, which is created by Rule2, between two use cases based on *implemented* relations,.

Rules in XQuery

We propose to represent the rules in XQuery [28] due to its power and support for retrieving data from XML documents, and due to its maturity, simplicity, and extensibility. With XQuery it is possible to locate specific elements and attributes in an XML document by using XPath expressions. Apart from the embedded functions offered by XQuery, it is possible to add new functions and commands. In this paper, we are using XQuery implementation of [24] in order to validate the XQuery rules. Even though the implementation of [24] is still ongoing, it covers most of the XQuery functions necessary in our approach. However, it is also necessary to extend XQuery with new functions in order to cover some of the traceability relations being proposed. Example of these functions are *synonym*, to identify words that have the same meaning, *distance_control*, to measure the distance between two words in a text.

Figure 5 shows a sample XQuery rule of Rule1, shown in figure 4. The XQuery part is composed of two parts. The first part (*for*) identifies the elements of the documents being compared (i.e. <Feature> and <Use_case>). The second part (*where*) contains the conditions that should be satisfied in order to create the traceability relations. The conditions of this rule verify if there is a singular noun (<NN1>) composing the name of the feature that appears in the description of the use case, and if the title of the use case is composed of a neutral noun (<NN0>) and a verb (<VVI>) that appear in the description of the feature model.

```
<TraceRule RuleID="R1" RuleType="Implemented">
<Query>
  <![CDATA[ for $fm in doc("file:///c:/FeatureModel.xml")//Feature_model/Feature,
    $uc in doc("file:///c:/UseCases.xml")//Use_case
    where some $t in $fm/Feature_name/NN1 satisfies (contains(string($uc//Description), $t))
      and (some $p1 in $fm/Description/NN0 satisfies contains(string($uc//Title), $p1))
      and (some $p2 in $fm/Description/VVI satisfies contains(string($uc//Title), $p2))]]>
</Query>
<Action> <Relation type="Implemented"/>
          <Elements>$fm/feature_name</Elements>
          <Elements>$uc/use-case-ID</Elements>
</Action>
</TraceRule>
```

Fig. 5. A sample rule in XQuery

The parts of the documents to be related when the conditions of the rule are satisfied, is specified by element <Action>. In order to illustrate, consider the use case and feature model shown in Figure 4. In this case, a relation of type *implemented* is created between the use cases and the feature model, since

the feature name "Bluetooth" appears in the description of the use cases, and "Send data" appears in the title of each use case and the description of the feature model.

4. Related work

Our work is related to two main areas of research: product family software systems; and requirements traceability.

Product Family Software Systems

The field of product family software systems is large and growing and there are many projects from both academia and industrial environment. Some projects [2][4][7][11][13][22] define frameworks for system development. Other methods [3][27] describe the conceptual framework of product family processes and focus on the reuse of product family systems. However, the majority of the methods are complex, do not tackle the problem of how to manage requirements, as well as how to support traceability generation between the systems.

Methods like [6][15][17][18] use the *feature model* for representing common and variable aspects of product family systems. However, some of these methods do not cover the life-cycle of product family system development and some of these methods do not focus on requirements engineering.

Requirements Traceability for Product Family Software Systems

Research into requirements traceability for product family software systems has been proposed to investigate the roles and importance, and the methods for establishing traceability relations. Approaches like [5][16][26] support manual generation of traceability relations, while other approaches [1][10][12][19]20] have been proposed to support automatic traceability generation. However, these approaches do not fully represent the semantics of traceability relations and have limitations of how to generate traceability relations in complex systems.

5. Conclusion & Future work

This paper is part of my PhD work, which focuses on two main areas: requirements traceability and product family software systems. The paper presents a rule-based approach to allow automatic generation of traceability relations in documents for product family systems. In particular, the traceability relations allow the identification of common and variable aspects and also describe the semantics of relationships between artefacts for product members. In this paper, we have identified nine different types of traceability relations for product family systems. Our approach generates relations between three types of documents i.e. functional requirements specifications, feature models, and class diagrams.

We propose to use XQuery to represent the traceability rules. Although XQuery is apparently powerful, it is still subject to changes. However, based on our study we believe that XQuery is a very good answer for the problem being investigated.

Currently, we are implementing the *traceability generator* in order to evaluate our work. Before large scale experimentation and use, we are extending our work to cover traceability generation between other types of documents for product family software development i.e. process models, subsystems models, module models, component diagrams, and state chart diagrams. As mentioned earlier, we plan to add new functions in XQuery in order to support identification of all types of traceability relations being proposed. We also plan to evaluate our approach in terms of recall and precision.

References

1   Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering Traceability Links between code and Documentation. IEEE Transactions on Software Engineering, Vol. 28 No. 10 (2002) 970-983

2   ARES: http://www.cordis.lu/esprit/src/20477htm, (1999)
3   Atkinson, C., Bayer, J., Muthig, D.: Component-based product line development: The KobrA approach. Proceedings of the 1st Software Product Line Conference, (2000)
4   Bayer, J., DeBaud, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T.: PuLSE: A methodology to develop software product lines. Proceedings of the 5th ACM SIGSOFT Symposium on Software Reusability (SSR'99), (1999) 122-131
5   Bayer, J., Widen, T.: Introducing Traceability to Product Lines. Proceedings of Software Product-Family Engineering: 4th International Workshop (PFE), Spain (2001), Lecture Notes in Computer Science, ISSN: 0302-9743.
6   Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach. Addison-Wesley, (2000)
7   CAFÉ: http://www.esi.es/en/projects/cafe/cafe.html, (2003)
8   Cockburn, A.: Structuring Use-Cases With Goals. JOOP, Sep-Oct (1997)
9   CLAWS: https://www.comp.lancs.ac.uk/ucrel/claws.
10 Cleland-Huang, J., Chang, C.K., Sethi, G., Javvaji, K., Hu, H., Xia, J.: Automating Speculative Queries through Event-based Requirements Traceability. Proceedings of the IEEE Joint International Requirements Engineering Conference, Essen (2002)
11 Clements, P., Northrop, L.: A Framework for Software Product Lines Practice – Version 4.1 [Online]. Carnegie Mellon, Software Engineering Institute URL: http://www.sei.cmu.edu/plp/framework.html, Pittsburgh (2003)
12 Egyed, A.: A Scenario-Driven Approach to Trace Dependency Analysis. IEEE Transactions on Software Engineering, Vol. 9 No. 2 (2003)
13 ESAPS: http://www.eso.es/en/projects/esaps/esaps.html, (2001)
14 Gotel, O., Finkelstein, A.: An Analysis of the Requirements Traceability Problem. The 1st International Conference on Requirements, England (1994) 94-101
15 Griss, M., Favaro, J., Alessandro, M.: Integrating feature modeling with the RSEB. Proceedings of the 5th International Conference on Software Reuse, Vancouver BC (1998) 76-85
16 Integrated Chipware; RTM: www.chipware.com
17 Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990, also available at http://www.sei.cmu.edu/domain-engineering/FODA.html.
18 Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: a feature-oriented reuse method with domain-specific architectures. In Annals of Software Engineering, Vol. 5, 354-355.
19 Marcus, A.; Maletic J.I.: Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. ICSE, (2003)
20 Pinheiro, F., Goguen, J.: An Object-Oriented Tool for Tracing Requirements. IEEE Software, (1996) 52-64
21 Pohl, K.: Process-Centered Requirements Engineering, John Wiley & Sons, Inc., (1996)
22 PRAISE: http://www.esi.es/en/projects/praise/praiseproject.html
23 Ramesh, B., Jarke, M.: Towards Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering, Vol. 37 No. 1 (2001)
24 Sourceforge; Saxon: http://saxon.sourceforge.net/
25 Spanoudakis, G., Zisman, A., Perez-Minana, E., Krause, P.: Rule-based Generation of Requirements Traceability Relations. Journal of Systems and Software, Vol. 72 No. 2 (2004) 105-127 (to appear).
26 Teleologic; Teleologic DOORS: www.teleologic.com/products/doors
27 Weiss, D.: Software Synthesis: The FAST Process. Proceedings of the International Conference on Conference on Computing tin High Energy Physics (CHEP), (1995)
28 XQuery: http://www.w3.org/TR/xquery/
29 Zisman, A., Spanoudakis, G., Perez-Minana, E., Krause, P.: Towards a Traceability Approach for Product Families Requirements. The 3rd ICSE Workshop on Software Product Lines: Economics, Architectures, and Implications, Orlando (2002)

5 *Periklis Sochos*
  Mapping Feature Models to the Architecture

Mapping Feature Models to the Architecture

Periklis Sochos

Technical University Ilmenau, Process Informatics, Postfach 10 00 565
98684 Ilmenau, Germany
Periklis.Sochos@tu-ilmenau.de
http://www.theoinf.tu-ilmenau.de/~pld

Abstract. Many software product line (PL) methodologies use features to model variability and express requirements. Furthermore, they set features as the main driver for the development of the PLs' architecture. Nevertheless, the existing PL methods do not provide the developer with a concrete and efficient process to map features to the architecture. This paper directly addresses this issue by providing a method that will allow for a strong mapping between PL features and architecture. The method makes use of feature transformation, inter-process communication protocols and plug-in architectures.

Classification: PhD, 2nd year

1   Introduction

A number of software product line development methodologies make extensive use of features, e.g. [5], [6], [4] and place feature models in the center of the development efforts. Nonetheless, the resulting architectural components lack a strong mapping to features. Extensions to these methods come from the combination of generative programming techniques (e.g. [7], [8]) with the aforementioned methodologies (e.g. [2]). Although these extensions provide an improvement on the issue of mapping between features and architectures, they introduce new problems, such as lack of sufficient tool support, decreased maintainability and evolution.

This paper introduces the Feature-Architecture Mapping (FArM) method, which directly addresses the issue of weak mapping between PL features and the PL architecture. Section 2 will provide an overview of FArM, section 3 illustrates the core of the FArM method, sections 4 to 8 provide more details on the method's processes, while section 9 states the conclusions and further work.

2   Feature-Architecture Mapping (FArM)

The Feature-Architecture Mapping (FArM) method is divided, as most PL development methods, in a Product Line and Product Engineering phases. The
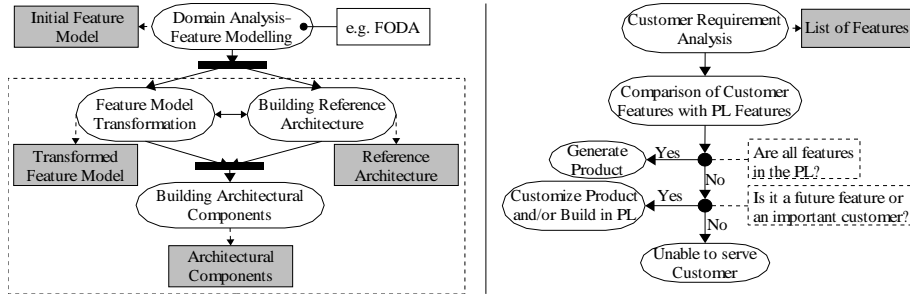
Fig. 1. The FArM Product Line and Product Engineering workflows.

former develops the PL core-architecture and components, while the latter supports the instantiation of PL end-products. Figure 1 illustrates these two phases.

The Product Line Engineering in FArM performs a domain analysis with an independent domain analysis method. The resulting initial feature model is then used in the two main processes of FArM, namely the Feature Model Transformation and Building Reference Architecture processes. These processes take place in parallel and require the exchange of information between them. The produced artifacts are a transformed feature model and the PL reference architecture. These in turn serve as the basis for the development of the PL architectural components, where each component implements exactly one feature in the transformed feature model. A noticeable point in the FArM Product Engineering is the fact that FArM explicitly supports the generation of custom PL products rather than placing a demand for extra effort into writing "glue-code" to instantiate end-products.

3   FArM Core Processes

In the heart of FArM lays the Feature Model Transformation (FMT) and Building Reference Architecture (BRA) inter-related processes. During this stage of FArM the initially developed feature model is subject to transformations. This is due to the fact that a domain analysis method yields mainly a customer-oriented feature model, which may lack the needed features for a direct mapping to the system's architecture. FArM strives to preserve the customer-view features and embed in the feature model the architectural view of the PL.

The FMT process is performed from a feature modelling team. It transforms the initial feature model making use of the high level feature model information. An inter-process-communication protocol is built containing the candidate features, the suggested transformations and the rational for these transformations. Possible transformations are: **adding** features, **integrating** features within other features, **dividing** features and **reordering** the hierarchy of features on the feature model.

The protocol is then examined from the PL architecture team for feasibility and compliance with the architectural view of the system in the BRA pro-

cess. Changes to the protocolled suggestion(s) or an acceptance decision may occur, whereby specifications for the respective architectural component(s) are recorded. At the end of this process traceability links are created where needed between the initial and transformed feature models denoting the transformations and recording their rational, thus maintaining backwards traceability.

The transformation steps performed throughout the FMT and BRA processes are based on feature model and architectural structures considered in the following order: **quality features**, **architectural requirements/implementation details**, **grouping features** and **interacts relationships** (figure 2).
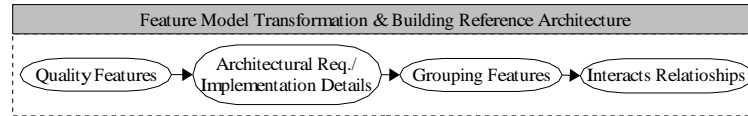
Fig. 2. The FArM transformation flow.

Armed with the knowledge gained throughout these transformation phases the PL developers implement the PL components for the target platform. FArM follows a hybrid iterative development motif: If a PL requirement change occurs or a problem in the design logic is discovered throughout the transformation phases or component implementation, the developers can always return to one of the previous phases and carry out the desired transformation steps to resolve the problem. They should then proceed down the transformation chain from that point on to preserve the consistency of both the feature model and system architecture.

4 Transformations Based on Quality Features

In order to achieve a strong mapping between feature models and architecture the initial feature model is transformed to resolve all quality features. Through this transformation step FArM explicitly supports the developers into resolving quality features that do not allow a direct implementation in architectural components. Quality features transformation takes place in two levels, i.e. Feature Model and Architectural level.

4.1 Preconditions

All not architecture-related quality features are "removed" from the feature model since they can have e.g. a managerial resolution. For example a feature such as *Competitive Market Price* can be resolved by using already registered stable software platforms and tools, acquiring field experts in the developer team or performing thorough periodical risk analysis. Thus only architecture-related

quality features are addressed by FArM, i.e. quality features that can be resolved through an architectural solution. The resolution of non-architecture-related quality features is recorded in the transformation documentation.

Furthermore, each of the quality features should have a quantitative, explicitly defined specification. If this is not the case, the developers can follow the method of defining Profiles [3] for these quality features or may consider "breaking down" the feature to finer features relating to particular aspects of the system and then provide a specification.

4.2 Feature Model Level Quality Feature Transformation

First the developers approach the quality feature transformation on the feature model level. Based on the features' specification the developers search for relations between the quality features and the functional features already present in the PL. This process can take place upon explicitly performing a syntactical analysis of the features' specification text and identifying lexical structures (e.g. nouns, adjectives, verbs) referring to an existing functional feature or being part of the specification of an existing functional feature.

After identifying the related functional features, an augmentation of their specification (e.g. through a restriction) is protocolled for further consideration with the PL architects. During this process it may be the case that identified lexical structures imply the addition of new features. This information is also protocolled for later consideration with the system architects as a candidate new feature/component. Finally, there may exist quality features that neither relate to existing functional features nor imply the addition of new functional features. These features are noted as such for further consideration by the architectural team.

4.3 Architectural Level Quality Feature Transformation

In this phase the developers in collaboration with the system architects consider the protocols developed during the previous phase of the transformation. For the quality features for which related functional features were identified, the architects consider the augmented specification proposals of the PL feature model analysts from an architectural point of view, e.g. whether the extra specifications can be architecturally realized.

For the quality features that imply the addition of new features, the PL architects consider the possible implementation issues involved with the new features (components) and provide a rational for their role in the architecture. If the new feature-components successfully "pass" both phases they are accepted in the PL. Such a quality feature can be a *Usability* feature in a cell phone PL. Its specification could be: "the placing of a phone call shall require at most 2 steps: type phone-number, press dial button, which will be acknowledged within 150 ms", etc. This feature is well defined and can be integrated into the PL's *PhoneCall* and *Keyboard* functional features. The *PhoneCall* feature will implement the phone call functionality, while the *Keyboard* feature implementing the phone's

keyboard driver and listening to the dial button press, will provide feedback to the *PhoneCall* feature for acknowledgement within 150 ms (e.g. by calling its `PhoneCall->Acknowledge()` method).

Features having neither related functional features nor suggesting their implementation in new functional features are examined by the architects by assigning each feature to one of the following categories based on the ISO 91260: Development (e.g. maintainability, reusability, flexibility and demonstrability) and Operational (e.g. performance, reliability, robustness and fault-tolerance). For each category FArM has collected all architectural and design patterns found in related literature that can be applied to each category to satisfy the quality features from an architectural perspective.

5 Transformations Based on Architectural Requirements/Implementation Details

This transformation phase is initiated with the architectural team considering the requirement and implementation details imposed on the PL architecture. These can lead to one or more of the aforementioned transformations i.e. adding, dividing, integrating or reordering features. The resulting components are protocolled and after approval from the feature analysts they are recorded as features into the transformed feature model, which should now ideally include all PL features.

Using the cell phone PL example, one could consider the requirements placed upon such a PL by the wireless network infrastructure in which the cell phones should operate. Such requirement may impose the addition of a *Wireless Network* feature that would provide a uniform information access and exchange to ensure a consistent mechanism for global reach, regardless of the underlying wireless technology. Such a feature is not directly visible to a custom user and thus may not have been included into the initial feature model but is nonetheless a dominant architectural requirement.

6 Transformations Based on Grouping Features

In order to achieve a strong mapping between feature model and architecture, FArM binds the feature model hierarchy to the architecture design process by reflecting it onto the relationships between the architectural components. That is, the feature model hierarchy places restrictions on the communication flow between architectural components and their interfaces.

This transformation takes place both on the feature model and architectural level. In the feature model level all feature - sub-feature relationships are validated. The feature model analysts compare the grouping feature specifications with those of their sub-features to make sure that a substantial degree of accordance regarding their operational aspects in the PL is present. More precisely, a sub-feature should have one or more of the following relationships with the

grouping feature it belongs to: extend its functionality, present a functional alternative or functionally complement the grouping feature.

The results are propagated to the architectural team in a protocolled form and are further examined for compliance to the system's architecture. Possible adaptation of the suggestions optimize the inter-component communication. The feature hierarchy should now define which components interact with each other. Finally, the architects using the gathered information on the architectural components, namely, operational restrictions from system quality requirements, allowed feature communication, feature specifications, architectural requirements and implementation details, define the interfaces of the components.

A valid feature - sub-feature relationship in the cell phone PL is shown in figure 3 where the *Languages - English,German,French* hierarchy presents a feature with its functional alternatives. This in turn would imply that communication is allowed between the *CellPhone* and *Languages* features, as well as the *Languages* feature and its sub-features.

Component interfaces referring to component interactions beyond the ones imposed from hierarchical structure are created on the next transformation phase based on interacts relationships.

7 Transformations Based on Interacts Relationships

Interacts relationship transformations support the developers in the explicit modelling of the extra-hierarchical relationships between features and transferring this knowledge to the system's architecture. They also support the design for maintenability as well as PL end-product instantiation.

7.1 Creating Interacts Relationships

Interacts relationships are established between features in different hierarchy branches needing to communicate to complete a task. Their main purpose is to model the kind of interactions between the features and thus propagate this information to the architectural level. Information contained in interacts relationships may be requested services, kind of data, data format, etc. and may be captured in a formal (OCL) or semi-formal form (structured language). Interacts relationships are addressed initially from the feature analysts on a feature specification level and in turn on an architectural/implementation level by the system architects.
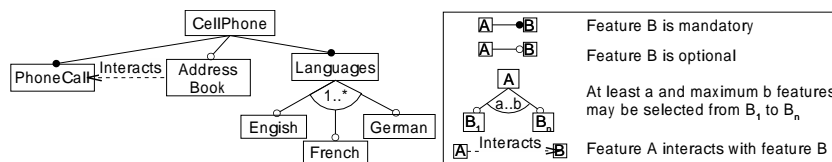

Fig. 3. A partial cell phone product line feature model.

Figure 3 shows an interacts relationship between the *PhoneCall* and *Address-Book* features of the cell phone PL. This relationship models the present feature interaction: The *AddressBook* feature calls upon the *PhoneCall* feature providing the necessary information (e.g. phone number of person) to perform a phone call initiated from the user e.g. from the Address Book menu.

7.2 Resolving Interacts Relationships

The resolution of interacts relationships contributes to the enhancement of system maintenability and evolution. Related works on feature interactions may be found in [9]. Some of the FArM specific criteria on interaction resolutions for a feature are: number of interacts relationships, desired variability related to the feature, importance of the feature as to system evolution and/or maintenance.

It must be noted at this point that an assessment of the PL architecture should takes place after the transformations to assure the satisfaction of the system's quality requirements. This can be achieved through architecture assessment methods found in the literature (see [3]).

8 Building Architectural Components

In this final phase of FArM the architectural components of the PL are implemented. FArM supports plug-in architectures for the realization of the PL. Each feature in the transformed feature model is implemented in exactly one architectural plug-in component. The interfaces and communication between the components as well as their specification have been determined throughout the transformation process.

FArM does not require a specific plug-in architectural structure rather it adapts to the needs of the PL domain. The decision to support plug-in architectures in FArM is based on the following arguments: it simplifies PL end-product instantiation (features plugged into the plug-in platform instantly compose PL products), it supports a high level of encapsulation (features can be completely implemented in one plug-in component), decoupling of the features is achieved through the hierarchical plug-in structure, the design of the plug-in component interfaces is directly supported from feature interactions. Developers wishing to follow a non-plug-in architectural paradigm and at the same time use FArM should make sure to develop components adhering to the points mentioned above.

Finally, FArM's implementation demands no extra development tools as a number of the hybrid PL development methods do (see [2], [8], [7]). FArM can be applied with a documentation tool, a feature modelling tool and an industrial development environment for the target platform.

9 Conclusions & Further Work

This paper presented the Feature-Architecture Mapping (FArM) method for enhancing the mapping of PL features to the PL architecture. This mapping is

poorly supported by the present PL development methodologies having a large impact on PL maintenability and evolution, as well as PL end-product instantiation. The FArM method is based on a number of transformations (adding, dividing, integrating and reordering features) applied on the PL's initially developed feature model. The transformations are based on quality features, architectural requirements/implementation details, grouping features and interacts relationships. FArM supports the development of a modular plug-in architecture where each architectural component implements exactly one feature of the transformed feature model. Inter-component communication is explicitly modelled and supported in the transformation phases.

The FArM method has been already implemented on an IDE product line with positive results. More precisely, a feature model has been developed for the IDE PL based on features present in industrial tools (e.g. model-code synchronization, support for different compilers, etc) and was used as the initial feature model of FArM. The various FArM transformations were applied resulting to the implementation of a number of PL features. The case study results proved the method's feasibility and enhanced system maintenability. In order to test FArM in an existing industrial platform at the time of writing FArM is applied on the Blackberry [1] cell phone development platform. Further work includes the concretization of the FArM phases and testing on a variety of other PL domains for the identification of the method's limitations.

References

1. Blackberry Handheld, http://www.blackberry.com/
2. Boellert, K.: Object-Oriented Development of Software Product Lines for the Serial Production of Software Systems (Objektorientierte Entwicklung von Software-Produktlinien zur Serienfertigung von Software-Systemen). PhD Thesis, TU-Ilmenau, Ilmenau Germany (2002)
3. Bosch, J.: Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach. Addison-Wesley (2000)
4. Griss, D.; Allen, R. and d'Allesandro, M.: Integrating Feature Modelling with the RSEB. In: Proceedings of the 5th International Conference of Software Reuse (ICSR-5) (1998)
5. Kang, K.; Cohen, s.; Hess, J.; Novak, W.; Peterson, A.: Feature-Oirented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (1990)
6. Kang, KC; Kim, S.; Lee, J.; Kim, K.; Shin, E.; Huh, M: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering, 5 (1998) 143–168
7. Kiczales, G.: Aspect-Oriented Programming. Springer-Verlag, In Proceedings of the 1997 European Conference on Object-Oriented Programming (ECOOP '97), (1997) 220–242
8. Ossher, H.; Tarr, P.: Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In: Software Architectures and Component Technology. Kluwer Academic Publishers (2001)
9. Zave, P.: FAQ Sheet on Feature Interaction. AT&T (1999)

6   *Jan Suchotzki*
     A Strategic View on Software Product Lines – Adapting an Or-
     ganization's Structure for a new Approach in Software Develop-
     ment

A Strategic View on Software Product Lines – Adapting an Organization's Structure for a new Approach in Software Development

Jan Suchotzki [1]

[1] ABB Corporate Research Germany, Wallstadter Str. 59, 68526 Ladenburg, Germany
jan.suchotzki@de.abb.com

Abstract. In terms of technical issues the Software Product Line (SPL) approach quickly emerges to one of the most promising development methods. This includes that business manager can be convinced with the high potential in terms of a source for competitive advantage. However, for creating sustainable and thus long-lasting competitive advantage this approach has to be embedded into the business strategy. Organizational structure, as one of the key implementation instruments for strategy, is also one of the core elements for the success of the SPL approach in a company. This position paper reports on an MBA-thesis. This thesis will point out challenges arising from the SPL approach in the context of organizational structure, present a way to deal with these challenges, and from this level abstracts to more general implications for business strategy.

Background

The thesis, also called management report is done within the flexible MBA program of the University of East London. The hand-in date is the 16th of September in 2004. Currently the author is in the second and final year of his studies.

There are two main motivations for this report. First, there is the relevance to the MBA course. Organizations, which think about utilizing the SPL approach face several challenges. They have to adapt their complete strategy to the approach and have to implement a more or less radical change program. That means nearly all parts of an organization are influenced and thus it is related to nearly all disciplines of business and management studies.

Second, there is the personal motivation. The company the author is working with has recently started first projects for launching and institutionalizing software product lines. Even if the current focus is to evaluate the SPL approach from a technical point of view, it is very important to understand the implications for management. Due to the authors work in an R&D department, there is also a high interest to further develop existing theoretical frameworks and apply them. This corresponds with the fact that there is not much research done in the area of organizational management in relation with the SPL approach, so far. Three of the few publications in this context are [1], [4], and [6].

Objectives

Simplifying the decision on whether the software product line approach should be utilized or not, the following two questions have to be answered:

- How does the software product line approach improve the company's competitive advantage? That is, what benefits are gained?
- What needs to be changed for implementing a software product line strategy successfully? That is, what needs to be spent?

While the first question is heavily stressed by the management, the second is much more critical as it seems on the first view. For example, the following questions have to be considered:

- Which organizational structure (departments, business units, …) is needed for the approach? Who is responsible for the product line (core assets) and who is responsible for the products?
- What skills and core competences are needed in general and in the different organizational entities in particular?
- What strategy should be followed to build new product lines while maintaining the existing product base?

Helping managers to understand what needs to be changed is the overall objective of the MBA-thesis. Based on this, strategies towards an SPL-centric company will be worked out. It has to be mentioned here that especially companies, which are not only in the software business, might tend towards being Product Line centric. This in turn probably includes also a SPL. Addressing the author's personal resource and time constraints the report will focus on organizational structure and SPL. Thus the scope is narrowed and a discussion of Product Lines in general is not part of the thesis. The research question and the more detailed research objectives can be found in Table 1.

Table 1. Research Question and Objectives

Research question:	Why does an organization have to expect major changes in their structure when introducing software product lines?
Research objectives:	1. To extract major factors, important for a SPL oriented organization in general and their organizational structure in particular, from existing theory and research. 2. To determine general structural configurations and their influencing factors, as currently applied by organizations. 3. To analyze the current structure of a case study organization by using the general configurations and influencing factors. 4. To develop different organizational structures for the case study organization based on theory of organizational structure and SPL. 5. To develop a theory that explains the challenges a company faces when introducing SPLs by using the organizational structures as an example.

Realization

This research project is divided into three major parts. As mentioned in the objectives an important part is the conception of a theoretical framework based on literature review. This will include an analysis of three major areas. First of all there is the area of generic and mature organizational structure, which deals with structural configurations (e.g. Mintzberg's simple structure, divisionalized form, …) and contextual variables and success factors like size and strategy (Objective 2). The second area is based on new theories in organizational structure arising from research in organizational innovation and structural implications from new product development (Objective 2). For example, some research stresses that the mature structural configurations do not address the special needs of innovative and technology orientated organizations. The third area in this part will be characteristics and requirements resulting from the SPL approach (Objective 1). According to Clements and Northrop [1] the SPL approach heavily influences organizational structure and probably there is the need for modified structural configurations and new contextual factors have to be addressed by the structure of an SPL-centric organization.

The second major part is a case study analysis. In this part the structure of a case study organization will be analyzed with the previously created framework (Objective 3). That means it will be pointed out, which contextual variables are important in the environment of the organization and which structural configuration is applied by the company. It should also be analyzed which major problems might arise from this organizational structure in the context of SPL. From that analysis different organizational structures will be developed, which support the SPL approach best in the context of the case study organization. Finally one structure will be recommended, which should fit best from the experience gathered so far (Objective 4). Thus the case study will be used for verification, but is not the primary focus of the report.

In the third major and concluding part a theory, which explains the challenges an organization might face when utilizing the SPL approach, will be developed. Once again, also the focus for this part will be on organizational structure. This chapter will analyze whether, and if why, the currently available theory on organizational structure is not well suited for designing an SPL-centric organization. A special focus here is what challenges a manager has to expect. From that theory the author will provide some personal thoughts, supported by literature, on why the SPL approach influences the complete organization and their strategy and not only organizational structure. (Objective 5)

First Findings

This chapter provides a short overview of the results found so far. Here the focus is on the theoretical framework, because this is the basis for the next steps. Moreover it is at the heart of the complete thesis.

The scope of the framework is to build up a model that describes contextual variables, design parameters, design processes, dependencies between the former, and

finally structural configurations. That means this framework should be applicable for all organizations ranging from small and medium sized enterprises to large and global organizations. This generic and theoretical framework can then be applied for different types of organizations (e.g. global player) that act in different industries (e.g. mobile phone industry). That means it is similar to the generic models and descriptions of SPLs, like for example provided in Clements and Northrop [1]. Also these have to be adapted and modified for the concrete context they are applied in. For example, Muthig *et al.* [5] shows how to utilize the SPL approach in mobile phone industry.

Frameworks for Analyzing Organizational Structure

According to Nelson and Quick [7] organizational structure and especially its design process aims on constructing and linking departments and jobs to achieve the organizational goals. This type of management research and practices is called organizational design or organizational behavior. It provides several mature and generic frameworks. A general overview of the key organizational design elements is provided in Fig. 1.
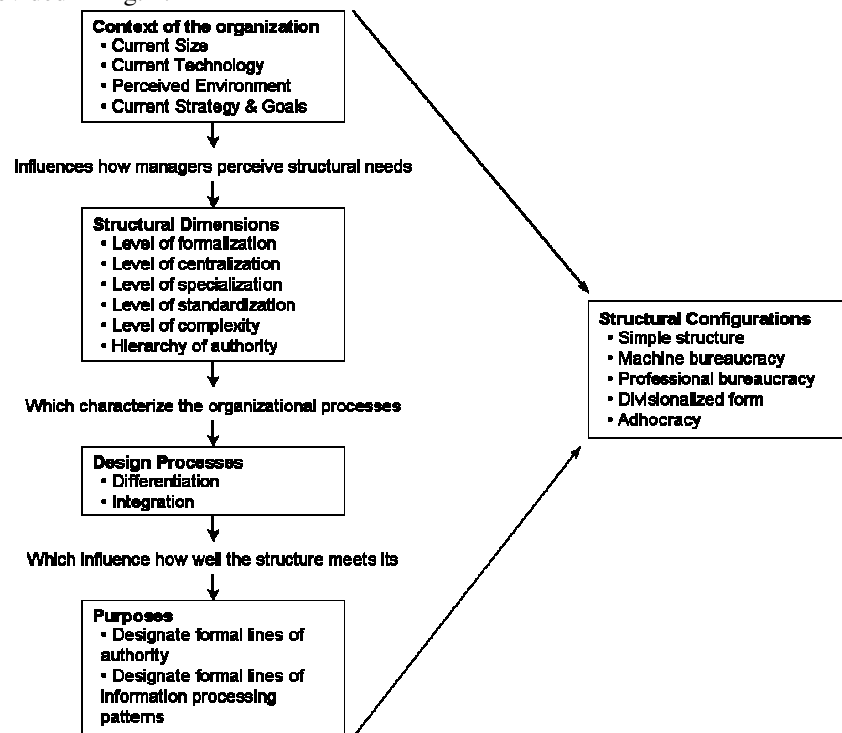
Fig. 1. Relationship among key organizational design elements (based on: Nelson, D.L. and Quick, J.C. (2001) Understanding Organizational Behavior, South-Western, Ohio)

In his basic research of organizational structure Mintzberg [5] identified five basic parts of an organization. (1) The operating core gathers all employees who perform the basic work related directly to the production of products and services. That means activities like securing inputs for production, transforming inputs to outputs, distribution of outputs, and direct support to these activities are placed in this part of the organization. (2) The strategic apex consists of managers that perform activities to ensure that the organization serves its mission in an effective way and that stakeholders and customers are satisfied. (3) The middle line is a group of managers with formal authority that connects the strategic apex and the operating core. (4) The technostructure includes analysts and supporting clerical staff that design, plan, and standardize the work done by the operating core. (5) The last part is the support staff. Here it is possible to find anything not directly related to the core business. Depending on the size this might range from legal counsels and public relations department to mailroom and cafeteria.

In combination with the key organizational design elements this framework develops its whole power. The structural configurations, which configure and combine the other design elements, relate directly to the five basic parts. These configurations can also be seen as design patterns. That means each structural configuration provides its key part of the organization, main design parameters, and important contextual variables. For example, the simple structure sees the strategic apex as the driving force and has nearly no technostructure and no supporting staff. That means there is nearly no standardization of work, but a high centralization (decisions are mainly made at the strategic apex). This is also called the entrepreneurial configuration, because the workforce is young, the amount of employees is relatively small and the environment is very dynamic.

Activities and Life Cycle of SPL

In this paper the general ideas of software product lines will not be discussed. The aim here is to identify major concepts that provide requirements and influencing factors for organizational structure.

Therefore a couple of different studies were analyzed. First roles and influencing factors were identified based on the product line engineering life-cycle as proposed by Muthig *et al.* [8], the three software product line practice areas (technical and managerial activities performed during the life-cycle of a product line) as proposed by Clements and Northrop [1] and first research of software product lines in the context of organizational structure as proposed by Bass *et al.* [2], [3] and Bosch [4].

The findings are presented in Fig. 2 and further discussed in the next section.

Towards a Framework for Analyzing SPL-centric Organizational Structures

A first step in analyzing and designing an organizational structure is to research the internal and external context of an organization; compare Fig. 1.

As stated in the previous section the software product line approach introduces several influencing factors. The most important for organizational structure are

presented in Fig. 2. At the heart of the proposed framework there are the five basic parts of an organization. In the context of software product lines the following roles, as proposed by Bass *et al.* [3], were assigned to the parts: core asset group, application group, marketers, support staff (training and consultancy), and managers. If this assignment is correct, it offers the possibility to use all existing findings and ideas (e.g. structural configuration) from theories and practices of organizational design to solve the problems in structuring an SPL-centric organization. A weak point in this assignment is the core asset group. It definitely standardizes work for the application group (e.g. provides an architecture and processes), but it may also provide completely designed and implemented components. Following Mintzberg [5] such activities should only be done in the operating core. However, this skeleton should be handled with flexibility. Even if there are configurations that might not have a core asset group, some of their activities have to be performed anyway. Then those activities have to be shifted into the operating core. This has to be kept in mind during the next steps in designing organizational structures.
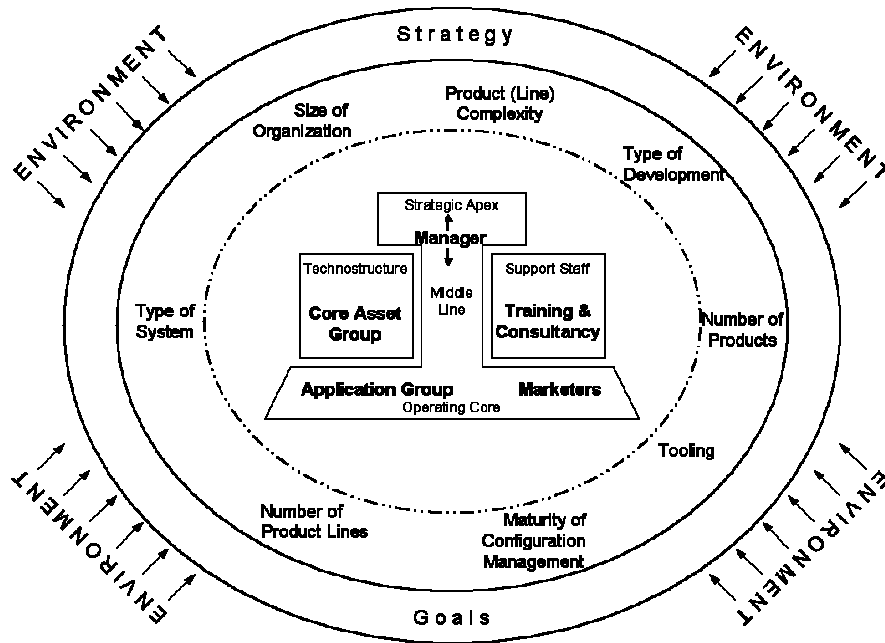

Fig. 2. Framework for analyzing SPL in the context of organizational structure

The framework (see Fig. 2) places internal and external influences around the five basic parts of an organization. Going from inside to outside the internal influences are first. For example, the degree of tooling (ranging from simple case tools to a completely integrated development environment with specialized code and documentation generators) and the type of development (sequential versus parallel product development) heavily influence the size and performed activities of the core asset group. But how to determine what degree of tooling is needed? First there is an inter-relationship between the internal factors. For example, if it is a specialized and

small product line and only a few products share the core assets a specialized code generator for the domain might be not the right choice. Second there are the external influences like for example hard competition. In the case an organization's strategy clearly formulates cost-leadership as the way to address this competition, the question regarding tooling is: Is it more cost efficient to have a specialized tooling or is it cheaper to write the code by hand? Obviously this is oversimplified, but shows the importance of strategy and environment.

The dependencies between influences (internal and external) and the organizational structure show another important point. During the design of an organizational structure there are two options according to the existing resources. (1) The existing resources have to be expanded (training employees, buying new development tools, …). (2) The organizational structure must be designed in such a way as to be realized with the existing resources. As outlined in the research objectives this work hypothesizes that only option one is applicable when introducing a SPL.

Due to the complexity of the environment the management report will not focus on the environment. Instead the organization's strategy and goals are seen as the major information and decision source. From a management point of view this sounds reasonable, because the strategy should always address all major issues from the environment.

Current Status and Summary

This paper described work in progress. Therefore it is as likely as not that some of the outlined ideas will not be part of the final report and others might be integrated. As shown in the last sections, a framework for analyzing organizational structures in the context of SPL is in place. Although not completely discussed here. It was pointed out that the framework is a tool to design a software product line centric organization. Because it is generic enough to be applied in many different domains (like the SPL approach too), it cannot be used without a strong and reliable data basis from the concrete environment an organization acts in.

In addition to this first part of the framework, structural configurations that are typical for software product line centric organizations have to be created. Therefore the generic existing configurations (e.g. simple structure) provide more than a starting point.

The topic is very recent and important as outlined in this paper. Due to the fact that this topic addresses managers on hierarchy levels, who might not have that much experience with software development, it is important to hide all the technical details. From a rough literature research it seems that it will be one of the first detailed research projects in the outlined area. Therefore it is very important to get feedback from a large group of people with different backgrounds. Preparing and writing the management report will be a quite challenging task, but with the defined scope and constraints it should be manageable.

References

1. Clements, P. and Northrop, L.: Software Product Lines – Practices and Patterns. Addison-Wesley, New York (2002)
2. Bass, L., Clements, P., Northrop, L. and Withey, J.: Product Line Practice Workshop Report. SAIC/ASSET, Morgantown (1997)
3. Bass, L., Chastek, G., Clements, P., Northrop, L., Smith, D. and Withey, J.: Second Product Line Practice Workshop Report. SAIC/ASSET, Morgantown (1998)
4. Bosch, J.: Software Product Lines: Organizational Alternatives. Proceedings of the 23rd international conference on Software engineering, 12 – 19, May, Toronto (2001)
5. Mintzberg, H.: The Structuring of Organizations. Prentice-Hall, Englewood Cliffs, NJ (1979)
6. Schmid, K.: People Management in Institutionalising Product Lines. IESE-Report 101.03/E, July 21, 2003
7. Nelson, D.L., and Quick, J.C.: Understanding Organizational Behavior: A Multimedia Approach, South-Western, Ohio (2001)
8. Muthig, D., John, I., Anastasopoulos, M., Forster, T., Dörr, J. and Schmid, K.: GoPhone – A Software Product Line in the Mobile Phone Domain, IESE-Report 025.04/E, March 5, 2004

Document Information

Title: Proceedings of the First International Software Product Lines Young Researchers Workshop (SPLYR)

Date: August, 2004
Report: IESE-Report No. 086.04/E
Status: Final
Classification: Public