Fraunhofer Institut
Experimentelles
Software Engineering

Proceedings of

Software Product Lines: Economics, Architectures, and Implications

Workshop #15 at 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, June 10th 2000

Editors:
Peter Knauber
Giancarlo Succi

IESE-Report No. 070.00/E
Version 1.0
June 2000

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competetive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach
Sauerwiesen 6
D-67661 Kaiserslautern

Program Committee Members

Luigi Benedicenti, University of Regina, Canada
Jorge Diaz-Herrera, Southern Polytechnic State University, USA
Loris Gaio, Università di Trento, Italy
Peter Knauber, Fraunhofer IESE, Germany
Masao J. Matsumoto, University of Tsukuba, Japan
Frank Maurer, University of Calgary, Canada
Maurizio Morisio, University of Maryland, College Park, USA
Giancarlo Succi, University of Alberta, Canada
Tullio Vernazza, Università di Genova, Italy
Enrico Zaninotto, Università di Trento, Italy

Table of Contents

[1] Paper is not provided because the author failed to submit a signed copyright agreement.

New product line approaches

Perspectives on Software Product Lines: Report on First International Workshop on Software Product Lines: Economics, Architectures, and Implications

Perspectives on Software Product Lines

Report on First International Workshop on Software Product Lines: Economics, Architectures, and Implications
Workshop #15 at 22nd International Conference on Software Engineering (ICSE)

Peter Knauber
Fraunhofer Institute for
Experimental Software Engineering (IESE)
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
(+49) 6301 – 707 242
Peter.Knauber@iese.fhg.de

Giancarlo Succi
Department of Electrical and Computer Engineering
The University of Alberta
238 Civil / Electrical Building
Edmonton, Alberta, Canada, T6G 2G7
(780) 492-7228
Giancarlo.Succi@ee.ualberta.ca

1 INTRODUCTION

Product line engineering is a concept that has emerged in the 80's in the business schools and is now among the hottest topics in software engineering.

Software product lines aim at achieving scope economies through synergetic development of software products. Diverse benefits like cost reduction, decreased time-to-market, and quality improvement can be expected from reuse of domain-specific software assets. But also non-technical benefits can be expected as result of network externalities, product branding, and sharing organizational costs.

Product lines introduce additional complexity. In a sense they go against the common adage of "divide and conquer." Planning and/or developing of more than one product at a time have to be managed technically and organizationally.

However, the rate of innovation of the technology and the intrinsic nature of software products do not let alternatives to developers: users like to jump into the bandwagon of new products, and old products often drive preferences to new products.

Research has been conducted in software product lines for the past few years. Some of it has focused on demonstrating that existing systems and approaches were indeed instrumental for product line development, such as generative techniques, domain analysis and engineering and software components. Another portion of the research effort has tried to determine how it is possible to create a comprehensive methodology and an associated tool for product lines, starting from the business idea of line of products down to the development of a product and trying to exploit all the possible synergies existing at each phase, from network externalities to component reuse.

2 WORKSHOP STRUCTURE

The workshop was structured into the following parts:

- Two invited talks were starting point and introduction into the morning and the afternoon sessions. The first one was given by Stefano De Panfilis who reported about his experience using a product line approach in the domain of fund management. In the second talk the largest European project addressing software families, ESAPS, was presented by its project leader, Frank van der Linden.
  More details on these talks are given in section 3.
- Three technical sessions gave room to present theoretical and practical issues concerning product lines and their use in practice. The first session addressed economic and organizational aspects of product line engineering. In the second session, experience from using product line methods in case studies, experiments, or industrial projects was reported. Several new product line approaches and their specifics were presented in the third technical session. At the end of each session, some time was reserved for discussion.
  The most important topics of each session are briefly summarized on section 4.
- A final panel discussion concluded the workshop. Six panelists answered questions from the audience and discussed with each other. Unfortunately, there was too little time to resolve many open issues.

P. Knauber, G. Succi (2001), "Review of the Workshop on Software Product Lines: Economics, Architectures, and Implications", ACM Software Engineering Notes, 26(2)

More information about the panel and the topics discussed there is given in section 5.

3 THE INVITED TALKS

Experience in the domain of fund management
Stefano De Panfilis reported about the application of product line in banking systems, and their specific fund management systems. There is a large common core of functionality among the products that are then customized to the customer's needs. De Panfilis pointed out that this fact is one out of two inevitable prerequisites for successful product line application. According to him, the second prerequisite is, that the customization of products offers a competitive advantage.

The ESAPS project
ESAPS (Engineering Software Architectures, Processes and Platforms for System Families) is Europe's largest research project, coordinating the work of 21 companies and research institutions across Europe. The project manager, Frank van der Linden, reported on the goals as well as the history of ESAPS: some of the project partners have already worked together in two previous projects, ARES and PRAISE. Overall goal of ESAPS is the achievement of significant higher levels of reuse and improved system quality through better engineering of architectures, processes, and platforms for system families. The first phase of ESAPS will concentrate on the development of the approach and laboratory scale validation of the individual technologies and technology integration framework. The second phase will focus on the integration of the individually validated technologies, automation of the approach and industrial scale validation in various domains.

4 THE SESSIONS

Session 1: Economic and organizational aspects of product line development
In this session, the economic aspects of software product lines were addressed: when and where it is worth to introduce a product line approach in an organization, how the benefits and risks involved can be quantified, how the product line can be focused best to match customers' needs, and how the ageing of a product line can be determined in order to decide about its retirement.

The four papers presented here tried to answer these questions and the discussion showed strong interest of the audience.

- The approach presented by *Goiuria Sagarduy* tries to qualtify the potential benefits and risks involved in the decision to go towards a product line development in order to give company owners and managers a good idea about the economic impacts of such a decision.
- In his talk, *Stuart Faulk* presented a new software development process including Customer Value Analysis to link relevant software design decisions to tactical and strategic business objectives. Customer Value Analysis here is used to denote a product's perceived oberall benefit.
- *Klaus Schmid* structured his product line scoping approach in three steps: the first one (product line mapping) produces a high-level dscription of the product line in terms of domains; the second one (domain-based scoping) does basically an assessment of reuse potential and viability of these domains; this information is then used to select the most promising ones. The third step (feature-based scoping) produces the quantitative benefit of implementing certain features in a generic way, that is, reusable. These three talks were about product line introducing and running them most effectively.
- In the last talk, *Susanne Johnsson* discussed the evolution of product lines, how the costs of maintenance increase over time and the relative division of effects resulting from maintenance tasks. These considerations are the basis of a model for identifying architecture erosion which can be used in the decision processes surrounding the reorganization and retirement of software product lines.

Session 2: Case studies, experiments, reports from industrial projects
In this session, industrial projects, experiments, and case studies together with the respective approaches used, their results, as well as lessons learned were presented. The intention was to help other organizations that intend to invest in product line development to get a feeling for the main risk factors and the critical issues to consider. The very lively discussion after this session was continued in the final panel discussion.

- *Ali Mili* reported about a classroom experiment, where four domain engineering methodologies (FODA, JODA, Synthesis, Reuse Business Methodology) were analyzed and compared. Criteria for the comparison were mainly the support for the domain engineering lifecycle, the rationale for domain definition, the support for legacy assets, the guidelines for domain architecture development, the domain engineering deliverables, the reusable assets, the technology and language dependency, and the effort for domain and application engineering.
- *Oliver Lewis* proposed a set of experiments that would help to quantify the space and time overhead due to variability in a product line implementation versus a single system implementation. The resulting data can inform embedded systems engineers about the

behaviour of overhead they might expect in a single system solution built from a product line model.

- In his talk, *Dan Paulish* described his experience with applying the technique of global analysis to plan software projects better by designing product line architectures that anticipate change. The purpose of global analysis is to analyze the factors that influence the architecture and to develop strategies for accommodating these factors in the architecture design. The technique was applied to the design of a meter data processing and control central station platform. The resulting high-level design proved to be very flexible and expandable.
- *William El Kaim* reported on product line experience from an experiment concerning systems used in the simulation for ground vehicle pilot training. Strong emphasis was on the modeling of the architecture from different perspectives (or views), each perspective corresponding to the concern of a stakeholder. The experiment, which was performed in the PRAISE Esprit project, showed also the importance of traceability among assets and the need to convince the people involved to explicitly describe and reuse assets. Tools, even though considered very important, are very expensive while not yet providing the support needed for product line development.
- In his talk, *Paolo Predonzani* described a case study regarding the introduction of domain analysis and object-oriented frameworks in a small software firm with the purpose to set up a development environment based on product lines. Goal of this project was to evaluate the impact and the benefits of the introduction of a domain engineering approach in a specific domain, laying the groundwork for the definition of a corporate reuse program toward the introduction of a product line. The experiment showed satisfactory results in general, the new concepts had positive effects on the software process.

Session 3: New product line approaches

In this session, new concepts for the development of product lines and the management of their evolution over time are presented. These include addressing single aspects or the complete product line life cycle. Specific attention has also been posed in tools to support product lines, government initiatives and small and medium size companies, which constitute a large and significant part of the software market.

- *Paul Clements* detailed a framework for the establishment of software product line practices. He identified the following key areas to target before the introduction of software product lines: domain understanding, asset mining, architecture exploration and definition, architecture evaluation, COTS utilization, software systems integration, data collection, metrics, and practices, product line scooping, configuration management, technical risk management. In addition he emphasized the importance of a suitable launch of the product line initiative in a company.
- *William El Kaim* presented a work of *Michel Coriat* and *Frédéric Waeber*. He introduced Wheels, a process framework to introduce and institutionalize product line practices. Three are the tenets of Wheels: **(a)** the use of UML for its metamodeling; this is intended to increase the understandability of the metamodel; **(b)** the adoption of a matrix approach in defining those practices that are required in a company to implement a product line approach; **(c)** the definition of a written handbook to detail the process patterns of the company. Wheels is being experimented in real projects by Alcatel and Thomson-CSF.
- *Giancarlo Succi* discussed the essential requirements for the establishment of a tool to support product lines. The key problem is the need to track and integrate the multiple activities that are required for the sound design of a product line. There is also the need to support domain specific advice. In brief, these requirements are: **(a)** link consistency management, to ensure that link traces make sense; **(b)** queries on dependency links: filtering, sorting, and other relationships; **(c)** change consistency management between different activities, to ensure that all changes are propagated correctly; **(d)** management of multiple users working concurrently; **(e)** data integration with COTS tools: to allow COTS tools to communicate with the framework; **(f)** semantic domain specific assistance. He also detailed the relevance of a design critiquing system.
- *Jorge Díaz-Herrera* presented a methodology for establishing product lines in the domain of embedded systems, a part of the Yamacraw Embedded Systems (YES) program, funded by the Georgia government. A detailed description of the project can be found in its web site: http://www.yamacraw.org. The idea is to exploit a synergistic design of multiple embedded systems together. Reuse is expected to play a major role, and a major promoter of reuse is the definition of standardized interfaces for the different pieces of hardware devices. The overall product line effort is divided in two groups of activities: **(1)** Modeling activities, including *(1.1)* requirement engineering for embedded systems, and *(1.2)* smart compilers for embedded systems; and **(2)** Engineering activities, including *(2.1)* personal embedded computing environments, *(2.2)* networked and enterprise embedded applications, and *(2.3)* home computing applications.

- *Joachim Bayer* raised the problem of how to help small and medium size companies to implement software product lines. He suggested that a product line methodology for such kind of companies should be able to **(a)** cope with immaturity of the development environment; **(b)** introduce techniques and concepts step by step, to evidence clearly the progress; **(c)** continue the work-in-progress, to avoid any disruption of the development, which would not be bearable in a small environment; **(d)** focus on the evolution of the products and not the variants, since SMEs are more likely to have new releases of products, rather than different versions of the same product; **(e)** rely on existing techniques and tools, for which there might be already expertise in the company: SMEs are not likely to take risks associated with new ideas. He introduced KobrA, a product line methodology specifically targeted to SMEs.
- *David Rine* evidenced that in a product line there are three major stakeholders: a management, system developers, and a reuse team. They all need to have coherent views of the product line, however, such views may be significantly different. Typical views for software product lines are the "product line overview," the "product line architecture," the "products," the "product release architecture," the "components." Each view is characterized by its own attributes. These views can be provided for the major steps involved in developing a product line: **(a)** deciding on the adoption of a product line strategy; **(b)** planning the product line; **(c)** utilization and management of a product line; **(d)** expansion of a product line.

5 THE PANEL

A panel concluded the workshop. The panelists were J. Bayer, P. Clements, J. Díaz-Herrera, D. Rine, and W. El Kaim.

The panelists and the workshop participants discussed several topics. Here below there is a review of the discussion. We have organized it by few major topics: the costs to establish a product line, the CMM, problems related to small companies, the role of management in a product line initiative, the importance of domain analysis and variability analysis in a product line effort, staging models for introducing a product line, how product lines support competitive biddings for large government contracts, and the relationships between product line efforts and lightweight methodologies. Clearly, there are significant overlaps among these topics.

Costs. Clearly, the overall cost to develop a product line depends on the size, the kind, the peculiarities etc of the target domain. Everything depends on how the costs are defined and measured, and this is not a trivial task.

Whenever a domain is scoped for a product line initiative there is an economic rationale of optimality. This then results in all the subsequent decisions, including the tradeoffs between generality and specific implementation. Studying such optimality is key in determining the overall cost/benefit analysis of the product line effort. This is not purely an economic analysis but also involves other issues related to management. For instances, there are problems related to legal aspects. Small businesses often have to conform to additional legal requirements to receive targeted business supports.

Simple numeric answers based on immediate cash flow have been proposed in the past for the CelsiusTech and the Boeing experience; the old adage has been repeated: "the cost of doing a product line is 2 to 2.5 the cost of doing one product the old way."

A non-monetary cost is represented by the time to market. For the introduction of a product of a brand new line, a product line approach may result is longer time-to-market, with risk of failure of the marketing effort. However, once the product line is institutionalized, it is faster to come up with new products in the line. An approach consists of starting small, incrementally, possibly growing from existing products.

A further approach to reduce the time to market is to create a statewide infrastructure, like what the state of Georgia is doing with the Yamacraw project.

There are also circumstances when time to market is not the key consideration. There are companies who take a break from production to grow in size. In these cases, a product line strategy is a way to provide a reasoned growth, with a clear definition of the core of the company and all the additions.

CMM. There is no one-syllabus answer on the relationships between the maturity in the CMM scale and the feasibility of a product line development. It seems that maturity helps but it is not a prerequisite. A speaker suggested that the CMM level 3 should be required for the business unit.

Anyway, it is evident that product lines practices and process improvement –in whatever scale it is measured, go hand-in-hand. This is especially true dealing with frameworks-based product lines.

The situation is particularly critical for **small companies.** Small companies represent a large part of the software development market; for instance, 80% of the companies in the Washington DC area are small companies.

There are indeed differences on how large and small companies can approach a product line initiative. A survey by Rine and Nada, to be published shortly, will detail these differences.

One of the problems is getting to the point where small companies leave a service oriented approach and start to see a substantial profit from a line of products, perhaps in synergy with other small companies. Also, the focus of product lines is not just reusing components, but to share knowledge across products, which may be even a more critical issue, since turnover in personnel is harder to manage in small companies.

Small companies alone often cannot afford to undertake certifications, such as the CMM certification. However, often in small companies the amount of variability for new requirements is usually narrow and they relay on a single product. Once the product is shipped, then it is possible to look at further requirements and expand the market.

As previously mentioned, Fraunhofer IESE has developed KobrA, a methodology to introduce product lines in small and medium size companies.

Management. The role of management in a product line effort is essential, especially when there are common assets, shared across departments.

In small companies this is not usually an issue. The management group is often part of the development as well, so there is not an "independent, non-technical management" to convince.

In large companies, the situation is different. For an effective product line strategy, everyone in the command line needs to be involved, and everyone in the command line, from the top down to the bottom, makes key decision.

This is a difficult issue, because there are several units to involve. A possible approach is to focus on the technology viability of the product line initiative first and then, if proven suitable, a limited pilot can be launched. If the pilot is proven successful, then the scope of the product line initiative is broadened. This is iterated to involve always larger parts of the company, till everything suitable is inside a product line.

The SEI framework requires a heavy management support: 2/3 of it is management oriented. Management has to fund, take the risks, put incentives. In addition, management has to participate to issues related to aging of software systems and controlling the evolution of the product line, since such issues are critical for the business successes and costs of the line.

It is also important to notice that customers may be exposed to tradeoffs of product lines and decide accordingly. The future of the company owning the product lines can be prosperous, since they have a baseline to compete on the market. Product lines can be considered a way to build the future.

Domain analysis and variability analysis. It is not yet clear whether an upfront effort in domain analysis is the requirement for a successful introduction of a line of software products. This is especially important for small companies, which often to don't have the resources for such initial commitment.

Variability analysis is an important part of the establishment of a product line. Variability analysis is scattered across multiple phases of the process of developing a product line. First, variability is studied while understanding the relevant domain for the line of products and while scoping the domain. Variability also lives in architectures.

Two critical issues deal with the situation of embedded systems, when variability is to be solved also with hardware components, and with instilling the knowledge of variability in the corporate knowledge base.

Anyway, a very comprehensive marketing analysis is a crucial prerequisite for a successful analysis of variability.

Standard staging models for introducing a product lines. At the Software Engineering Institute there is ongoing research on staging the introduction of product lines. The idea is to start where most of the benefits occur. A risk assessment is performed at the beginning and then a set of steps is identified, with ample room for improvements and modification.

Competitive bidding for large government contracts. Competitive biddings occur very early in the software lifecycle, when very limited information on the target system is available. Product lines are a big advantage in these situations. Business people can make decisions with more information: **(a)** they can reuse previous instances of the line –analogy can be performed to a much larger extent, and **(b)** they have already byproducts that can be used in the final system.

Lightweight methodologies. Lightweight methodologies are not antithetic to a product line. There are parts of the product line strategy that live well with, say, extreme programming. An example is domain scoping. The development methodology depends on the situation, the business environment, the company skills, etc.

6 LESSON LEARNT AND LINES FOR FUTURE RESEARCH

The workshop has consolidates some aspects of the state of the art on software product lines.

- *Methodologies* have been classified, reviewed, and experimented.
- The pivotal role of *staging approaches* and of *champions* for the introduction of product lines has been reaffirmed.

Several new ideas have been presented, that set the lines for future research in the field.

- There is a need of understanding better how to shape software product lines for *small software companies*. Research has already been performed. Hoiwever, there is not a well established understanding of the issues. More models and more experiments are required, as the ongoing effort at Fraunhofer IESE on the KobrA methodology.
- The *importance of a product line approach beyond the simple reuse* of software components should be more clearly stated and defined. Suitable supporting tools should be developed.
- The *role of government agencies and initiatives* could be critical in establishing state- or country-wide framework that could support product line initiative of local companies or even product lines that span multiple local companies, as is the case for the state of Georgia.
- Clear taxonomies and experimentations of *economic models* for product lines should be developed, to provide companies more precise figures of what they can expect from product lines and what should be their upfront investments.
- *Relationships between product lines*, corporate environments, and other methodologies, process improvement frameworks, and tools should be clarified, to better understand when and how it is appropriate to start a product line. This is in particular important for the ISO and the CMM certifications and for extreme programming and other lightweight methodologies, given their relevance in the software industry.

7 CONCLUSION

Altogether, the workshop has been a very large success, due to the quality of the submitted papers, the level of participation of the audience, and the profile of the panelist. Several positive feedbacks have been received; for this reason, we have decided to publish the papers as a collection in [1].

At ICSE 2001 in Toronto the "Second International Workshop on Software Product Lines: Economics, Architectures, and Implications" will be held. We look forward a lot of papers and participants, to discuss the advancement in the discipline brought by the Y2K and to exchange our experience.

REFERENCES

[1] Software Product Lines: Economics, Architectures, and Implications, Peter Knauber, Giancarlo Succi (Editors), Proceedings of Workshop #15 at 22nd International Conference on Software Engineering (ICSE), Limerick, 2000, Fraunhofer IESE Report No. 070.00/E, 2000

Economic and organizational aspects of product line development

Multi-Staged Scoping for Software Product Lines

Klaus Schmid

Fraunhofer Institute for
Experimental Software Engineering (IESE)
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
+49 (0) 6301 707 158
Klaus.Schmid@iese.fhg.de

ABSTRACT[1]

Scoping is a core planning activity in product line development. It is central to determining and optimizing the economical benefits of product line development. In this position paper we discuss the requirements on a sound and practically useful product line development approach and will propose a specific approach which fulfills these requirements.

Keywords

Product line scoping, economic evaluation, feasibility analysis, scoping requirements, PuLSE-approach

1 INTRODUCTION

In this paper, we discuss the scoping problem in software product line development and propose an effective approach to solving this problem.

Software product line development centers around systematic and planned software reuse. This is to be contrasted with traditional reuse approaches, which are often referred to as *opportunistic reuse*. From an operational point of view the key difference of product line engineering (and domain engineering) to opportunistic reuse is that the former explicitly relies on *development for reuse* as opposed to assuming that any kind of software may be developed and later reused in an ad hoc fashion. This explicit development of assets for reuse is usually referred to as *domain engineering*. The characterization of product line development as *systematic* and *planned* relates to making the distinction between generic and application-specific software development explicit and repeatable. This distinction is at the core of *scoping* product line development.

However, as product line engineering is a form of domain-specific software reuse, the notion of a domain needs to be included in the evaluation. In particular, many aspects that impact the viability of software reuse need to be evaluated on a per-domain basis, as they are inherently linked to the concept of a domain, e.g., the maturity of the domain or the interference with organizational constraints.

In practical situations several technical domains are usually relevant to the systems in the product line. At the same time not all of the aspects of a domain will be relevant to the systems. This is depicted in a simplified manner in Figure 1. In the approach we propose, the linkage between systems and domains is made explicit via a step called product line mapping (cf. Section 3.2).

As a consequence of the above discussion we believe that scoping needs to be performed on two levels:

- Determining the (sub-)domains that are particularly relevant to software reuse.
- Determining the specific assets that should be developed for reuse.

Performing reuse in a planned manner is a key differentiator between product line development and other forms of reuse. Thus, it is extremely surprising that hardly any disciplined product line scoping approach has been published so far [5]. While without such planning product line development may still succeed (as any software development effort can in principle succeed without adequate planning), this is then more or less by chance. Scoping is the step in the development process where the economic foundation of the product line effort is analyzed and where one seeks to maximize this benefit.

The importance of adequate scoping is actually well known in literature [6]. The problem is that, both having a too large scope, as well as having a too small scope has very problematic implications:

- If the scope is too large, the project may fail, as too few resources are available for a successful completion of the project. Additionally, this adds complexity and costs to the project, which results in added risks and a delution of the overall benefits.
- If the scope is too small, the project may fail as the sup-

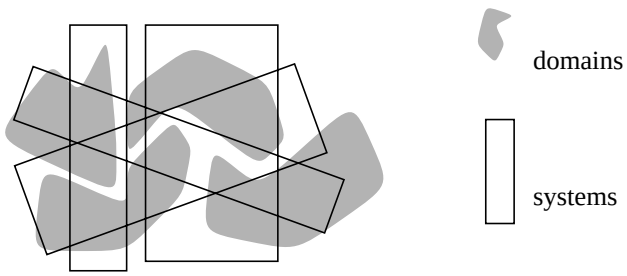


Figure 1. Relationship between domains and systems

1. This work has been funded by the ESAPS project (Eureka S! 2023 Programme, ITEA project 99005).

port for the overall product family may be too limited and repeatedly problems will arise during product instance development.

There is actually a third type of failure mode: you may choose a domain which looks promising, but which has problems embedded that are hard to detect up-front and will severely limit the return on the invested resources. This actually happened to us in an industrial transfer study based on [1] and triggered the introduction of the domain-based evaluation into our original approach (cf. Section 3.2).

2 SCOPING REQUIREMENTS

There are many aspects that we found insufficient with existing scoping approaches and also with earlier versions of our approach (cf. [1]). We want to detail these aspects here, by discussing the requirements we found relevant for practical scoping approaches.

The motivation behind software reuse in general and product line development in particular is primarily economical as the systems that are developed will basically fulfill the same requirements independent of whether they are based on product line development or not. Thus, it is a key requirement on scoping that it provides a detailed economic argument for the proposed scope [2]. In particular, an evaluation of the overall, to be expected benefits of product line development should be given and the various kinds of benefits (reduced time-to-market, improved quality, reduced risks of software development, etc.) should be possible to assess independently.

Further, it should be expected that scoping — as it is primarily a planning activity — addresses the risks of product line development and supplies a well-founded analysis of the risks of performing product line development with a certain scope. As we discussed above, we regard the risks as inherently linked to the characteristics of a domain and thus our approach addresses them on this level (cf. Section 3.2).

As a result of scoping it should be clearly possible that the scope is empty, i.e., that the scoping effort results in the proposition not to perform product line based software development at all, while we don't assume this to happen very often, we regard this as an important requirement on a sound scoping approach.

Further, as scoping is mainly a planning and controlling activity to product line development, this activity may only consume a very low amount of resources. This should be particularly true in cases where due to an inappropriateness of the domain or due to other reasons (e.g., resource or organizational constraints) the result is that product line development should not be pursued.

These requirements are not fulfilled by most existing scoping approaches, as they usually provide no way to give a sound economic basis to the scopes they propose (cf. [5]). Further, the few approaches that are around that actually address the issue of economic return (e.g., [8, 1]) do not provide a sound way for analyzing the overall feasibility of the product line and they have no way for staging the resources they consume, so that an assessment of the principal feasibility and benefits of product line development can be performed with relatively little effort. On the other hand there are approaches around which provide this at least partially (cf. [3, 4]). They in turn lack the ability to be linked to an approach that provides a clear economic argument for a scope.

Further, there are some additional requirements we find relevant to a scoping approach in order to make it practically useful. These are:

- A scoping approach should be integrated with an overall product line development approach, so that it is ensured that the results are directly useful to the development approach and so that no work is duplicated. As so far — except for some domain scoping approaches (i.e., [3, 4]) — all scoping approaches are not integrated with any domain engineering or product line development approach, this is a rather strong requirement.
- The approach should provide detailed guidance so that it is also applicable by somebody else but the developer of the method. This is actually a major requirement in order to make the approach repeatable and thus to assess the usefulness of the approach independent of the expertise of the method performer.
- The approach needs to be tailorable, as different organizations embark on product line for different reasons. These reasons or business goals need to be reflected in the scope selection process, as otherwise one will arrive at a scope which optimizes the wrong goals and thus at an inappropriate scope. This is particularly problematic with most existing scoping approaches, as they either do not give explicit criteria at all or embed a fixed set of criteria.

3 SOLUTION APPROACH

3.1 An Overview

In the preceding sections we discussed the requirements for a disciplined scoping approach. Here, we will outline the solution we propose and describe how it relates to the requirements we identified.

Note, that we identified two main levels of scoping: scoping a domain and analyzing whether we can expect to get the benefits we expect to gain from developing software for reuse in this area of functionality. On the other hand we want to be able to develop a concrete economic argument for this product line. In particular, we want to answer the question which functionality should be developed in a generic manner and which ones should be developed on a per-system-basis based on an economic argument.

Both of these "types" of scoping are complementary and seem to have little to do with each other on first sight. For this purpose we decided to have two main process components, each of which addresses one of the issues. These correspond to the boxes labeled *Domain-based scoping* and *Feature-based scoping* in Figure 2. Another way to distinguish these two steps is to label them as *qualitative* and *quantitive scoping* due to their different focus on evaluation.

However, in both cases, in order to arrive at a repeatable and understandable form of evaluation, there needs to be a common understanding of what the different domains and features actually mean. This actually was a problem with a previous approach we developed and reported on in [1], making the basis for the evaluation rather problematic.

In order to avoid this kind of problems with our revised approach, we added a main process component which aims at analyzing and deriving a high-level descripition of the product line and the domains relevant to it. We termed this component *product line mapping* (cf. Figure 2). This component addresses the need to establish a reference model which provides a sound basis to the evaluation, so that the different stakeholders share a common understanding of what is the exact extent of the identified features.

3.2 The Solution Components

The product line scoping approach we propose consists of three components:

- Product line mapping
- Domain-based scoping
- Feature-based scoping

An overview of the interaction of these steps is given in Figure 2. Below we will discuss each of these components in detail.

Product Line Mapping

The purpose of the *product line mapping* step is to develop a reference framework for the further steps of the scoping approach. In this step a description of the product line is developed. This happens on two levels: first a description of the product line is developed by describing the various systems that are part of the product line in terms of the major functionalities they provide, the market segments they address, etc.

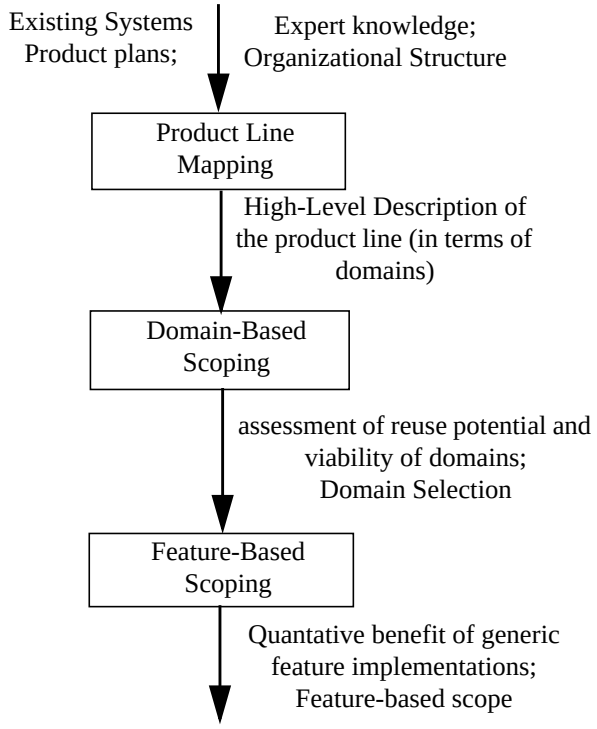


Figure 2. Solution Components

Second, based on this product plan, along with additional input (e.g., expert input, existing software architectures, organizational structure) a structuring of the product line functionality in terms of domains is developed. Each domain is in turn described in terms of the functionalities it provides, the data it handles, etc.

With this approach a high-level description of the relevant functionalities and domains is developed. Besides the usual benefits associated with such a description (cmp. domain analysis), we are in particular interested in the aspect of this method providing a well-founded basis for the communication with the stakeholders, which is later on relevant to develop sound evaluations of the domains and features. Without such a basis the same term is sometimes associated with different interpretations at different points in time by the stakeholder, leading to situations where for example first simply the user interface is associated with this functionality, while later the whole implementation from the user interface level down to the database level is associated with the term, leading to inconsistent evaluations.

This approach can also be applied to simply develop high level domain descriptions or develop more concrete product line plans.

In addition this step elicits the main objectives for introducing product line development. This is used in the next two steps for tailoring of the approaches.

Domain-based scoping

In this step the domains that have been identified and described during product line mapping are evaluated with respect to their potential for successfully fielding a reuse approach. This takes basically the form of an assessment, i.e., the experts are interviewed with respect to the various dimensions of evaluation and the resulting information is aggregated to arrive at the final evaluation. Two main dimensions of evaluation are used:

> *viability dimensions* – can a reuse approach be successfully fielded in this context?
>
> *benefit dimensions* – what benefits can be expected from introducing reuse in this domain?

Each of these main dimensions are further sub-divided into subdimensions which describe specific types of criteria (e.g., effort saving, domain maturity, etc.). Obviously, a sufficient score along the viability dimensions is a precondition for recommending a domain for product line development. If the viability evaluation is sufficient, then the scores along the benefit dimension can be used for selecting domains.

The approach provides a framework of evaluation dimensions, which can be tailored based on the specific objectives that are relevant to an organization. This can happen both by selecting only a subset of the evaluation dimensions for the assessment as well as by weighting the evaluation results. In particular, the former approach reduces the amount of effort

needed for the evaluation.

As the viability dimensions can be evaluated independently of the other dimensions, in cases where the domain is inappropriate for product line development the evaluation effort is kept to a bare minimum. In this case any effort can be abandoned after the focused interviews on the viability dimensions.

Feature-based scoping

Our approach to feature-based scoping builds on the results of the previous two steps and deepens them. The basic concept of this approach has already been described in [1]. During this step the individual features are evaluated and it is determined which benefits can be expected from developing generic assets for implementing them. While in the previous steps standardized (but adaptable) questionnaires were used for evaluation, in this step the evaluation criteria are further refined and more organization-specific aspects are elicited in a GQM-based fashion. Based on the information thus elicited, models are developed that describe for each objective identified by the company the benefit that can be expected from reuse [2].

Based on this characterization for each feature the benefit of having reusable assets that implement it can be determined. This provides a sound economical forecast that can be used to arrive at a scope that optimizes the objectives of the company.

Integration of the approach

As we described above, the integration of a scoping approach with a complete product line development approach is very important to avoid rework and to ensure that the information is appropriately used in later stages. The approach we described is tightly integrated with the PuLSE™ product line approach [7]. In particular the following relationships exist:

- The high level domain descriptions that are produced provide a basis for the later detailed domain analysis. Additionally this effort is bounded by the derived scope.
- Similarly, the scope helps to bound the architecting effort. Additionally, the specific approach used in PuLSE for architecting builds on the benefit evaluations for deriving an optimally adapted reference architecture.
- The scope and product development plans serve as an important input to the PuLSE-EM product line management component.

4 CONCLUSION AND FUTURE WORK

In this paper, we discussed what we regard as the core requirements to a scoping approach. As there is currently no scoping approach available which addresses all these requirements, we described our concept for such an approach and discussed how it addresses the various requirements. This approach is currently under development at the Fraunhofer IESE. The basic components have been developed and partially applied with a validation partner. Further work is needed to refine the method and the quality model which is used for evaluation.

References

[1] J.-M. DeBaud and K. Schmid. *A Systematic Approach to Derive the Scope of Software Product Lines.* International Conference on Software Engineering (ICSE'21), Los Angeles, CA, USA, pp. 34–43, 1999.

[2] K. Schmid. *An Economic Perspective on Product Line Software Development.* First Workshop on Economics-Driven Software Engineering Research, Los Angeles, 1999.

[3] Software Productivity Consortium Services Corporation. *Reuse-Driven Software Processes Guidebook, Version 02.00.03,* Technical Report SPC-92019-CMC. November 1993.

[4] Software Technology for Adaptable, Reliable Systems (STARS). *Organization Domain Modeling (ODM) Guidebook, Version 2.0.* Technical Report STARS-VC-A025/001/00. June 1996.

[5] Klaus Schmid. *Scoping Software Product Lines —An Analysis of an Emerging Technology.* First Software Product Line Conference (SPLC1), 2000. To appear.

[6] H. Mili, F. Mili, and A. Mili. Reusing Software: Issues and Research Directions. Transactions on Software Engineering. Vol. 21, No. 6, 1995.

[7] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. PuLSE: A Methodology to Develop Software Product Lines. Symposium on Software Reusability, Los Angeles, CA, USA (SSR'99), pp. 122–131, 1999..

[8] J. Withey. Investment analysis of software assets for product lines. Technical report, Software Engineering Institute, Carnegie Mellon University, 1996.

Product-line Analysis: Do we go ahead?

Goiuria Sagarduy
European Software Institute
goiuria.sagarduy@esi.es

Sergio Bandinelli
European Software Institute
Parque Tecnológico de Zamudio, 204
Zamudio, Bizkaia
Spain
+34 944209519
sergio.bandinelli@esi.es

Ramón Lerchundi
European Software Institute
ramon.lerchundi@esi.es

ABSTRACT
We have often heard that the product-line approach is a very good idea. However what we find when we try to introduce this concept into an organisation is a kind of reluctance to take this step forward. In general, company owners and executive managers ask for a report in economical terms of the risks or impact of implementing such an approach. This paper presents a practical way to quantify the benefits, and to relate them to the risks involved of embracing product-line. This approach has been taken during the execution of the PRAISE [1] project.

Keywords
Assessment, Reuse economics, Reuse benefits, Risk analysis

1 MOTIVATION

Product-lines represent a natural step in the evolution of software development to an industrial practice. A product-line approach intrinsically leads to systematic reuse and reuse is supposed to have a positive impact in business terms: saving development and maintenance costs, time to market reduction, quality improvement, more predictable project execution, etc.

In an industrial context, the decision of adopting a product-line approach must take into account a wide range of factors. Technology is, of course, one of these factors, but it is not necessarily the most important one and, certainly, it is not the "driving factor". The drivers for introducing a product-line approach are generally related to the general company strategy, taking into account market considerations. The product-line technology should be evaluated and used in this business context.

However, this previous analysis is not always performed and there is a tendency to jump directly into the technical implementation of a product-line: architecture, components, middleware technology etc. Firstly one must reason, with discipline, on what domain (or sub-domain) is the most appropriate and on whether the selected domain has the potential to justify the effort.

Not all domains are equally appropriate to be approached as a product line. A successful adoption of product-line approach requires some conditions such as potential demand for similar products, in-house knowledge and experience, existing regulations and standards, etc. A domain potential analysis evaluates the degree to which these conditions exist and serves as a reference for:

- Defining a product-line adoption strategy, and setting realistic goals for it.
- Deciding on the most appropriate domains or sub-domains for a product-line approach.
- Reaching consensus on a shared vision for the domain.
- Evaluating progress in product-line adoption.

Some models are documented to assist in performing this kind of analysis. Most of them are economic models and base the analysis on economic figures (cost vs. savings) to determine the benefits at different levels of reuse granularity: single component, project or whole domain. Other models include some analysis of the level of preparation of the organisation. (See [2] and [3] for a survey of all these models).

The domain potential analysis [4] presented here uses these models as a basis for the analysis of reuse benefits and combines this with a risk analysis. The two combined dimensions, benefits and risks, give an overall picture of the potential of a domain. The combined picture provides a clear indication on whether it is convenient to approach a domain as a product-line in absolute terms and by comparing different domains. In addition, the analysis may be adapted to be used with the available data in the organisation.

2 A SIMPLE ANALYSIS OF THE PRODUCT-LINE POTENTIAL IN A DOMAIN

The product-line potential in a domain provides an indication of the opportunities that derive from adopting a product-line approach to develop applications in a domain and the ability of the organisation to exploit these opportunities to obtain benefits from them.

The concept of domain that we use is a very broad one. It includes

- the technical description of the domain in terms of the existing and potential applications that share some common features (technology, functionality, etc.),
- the market of the domain (customers, competitors, regulations, etc.)
- the organisational structures that participate in the business.

When identifying a domain all these elements must be taken into account, since all of them are part of the analysis.

The potential analysis is similar to taking an investment decision. This is why both benefits and risks must be taken into account:

1. The benefits are the ones that the organisation expects from the product-line approach.
2. The risks are the ones associated with the introduction of product-line practices in the organisation.

The combined analysis benefits vs. risk gives the complete picture to take an investment decision.

3 ANALYSING THE BENEFITS

The right context for an analyses of benefits is the set of goals established by the organisation for embarking in a product-line. The organisation, depending on these goals, can give more or less importance to one potential benefit over the others.

The list of these benefits can be very long, for example:

- Higher productivity
- Better quality
- Higher Reliability
- Faster time to market
- Better bid estimation
- Better life-cycle estimates
- More on-time delivery
- Cost improvements and savings
- Improved maintenance

This long list can be generally shortened to the classical better, faster, cheaper:

- Quality Improvement
- Time to Market reduction
- Cost reduction

Ultimately, all the benefits should result (in the short term or in the long term) in economical benefits for the organisation. Some of the benefits, such as cost reduction, can be directly translated into economic results. Other benefits, such as quality improvement and time to market reduction, have an indirect impact on economic results. To simplify, we consider that quality improvement and reduced time to market have an impact as a reduction of maintenance costs and as an increment on the number of requests that may be satisfied in the same period (more production capacity translated into more units sold).

With these hypothese, there are three main elements which are included in an economic benefits analysis:

- **INVESTMENTS (I)**: which result from the activities of establishment and development of the product-line infrastructure;
- **EXPENSES (E)**: which result from the activities of maintainence of the product-line infrastructure during its life cycle;
- **SAVINGS (S)**: savings achieved as a result of the development of applications using the domain assets, comparing the development costs with and without reuse.

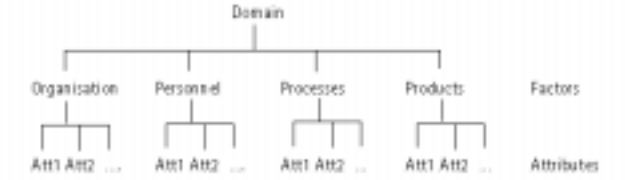
The depth of the analysis in terms of investments, expenses and savings can vary according to the data available in the organisation. In general, the process model can be a guide in the breakdown of the activities so that the effort associated with each of the activities can be evaluated separately and the results aggregated.

The economical analysis may be as simple as considering RoI (Return On Investment) in terms of savings against investments and expenses. It could also involve an additional analysis taking into account the time at which investments and expenses are made and savings are obtained. In this case, we are calculating time sensitive indicators such as the NPV (Net Present Value) or (PI) Profitability index.

4 ANALYSING THE RISKS

The objective of the risk analysis is to understand and quantify the major sources of risk when introducing product-line practices in a domain. The analysis is supported by a risk model. This model identifies a set of risk attributes organised into four risk factors (see Figure below):

- **ORGANISATION**, with attention given to the adequacy of the organisational structures for adopting reuse
- **PERSONNEL**, with attention given to staff experience and preparation
- **PROCESS**, looking at the presence of supporting processes for reuse
- **PRODUCTS**, looking at the existence of beneficial reuse characteristics in domain products



The risk analysis is performed by rating each of the risk attributes and by giving an impact weight for each attribute. The model provides a set of guidelines to interpret each of the attributes consistently [5]. The results of the risk analysis are a risk profile and an aggregated risk level that represent the overall risk for the domain.

5 TYING IT ALL TOGETHER

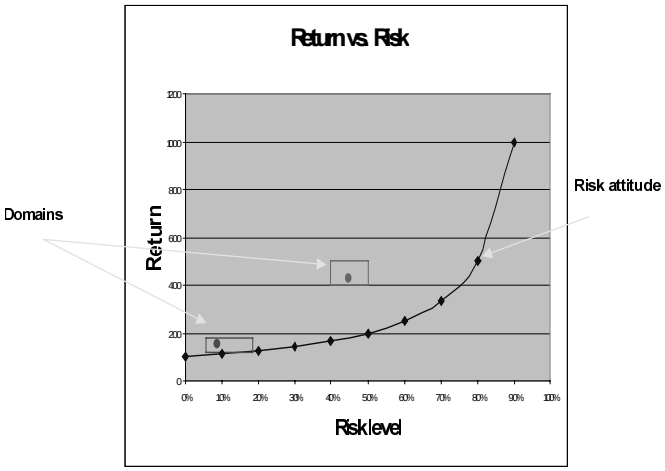
The potential analysis is completed with the evaluation of the organisation's attitude toward risk. This is determined through a questionnaire in which the respondent must choose between a set of possibilities regarding investments in a given situation. Three main attitudes are identified [6]:

- *Risk Taker:* This kind of organisations gives preference to obtaining more benefits at the expense of taking much higher risks.
- *Risk indifferent:* A risk indifferent organisation is ready to take some more risk decisions only if there is a proportional increase in the benefits that may be obtained.
- *Risk adverse:* For a risk adverse organisation, the main objective is to reduce risks. The organisation does not look for increments in benefits if this implies taking more risk. A more positive result is adequate.

Most organisations fall in the risk adverse category.

The benefits analysis, the risk analysis and the attitude towards risk are combined in a single graph that summarises the situation for one or more domains. The graph looks like the one in the figure below, in which the x-axis represents the risk level and the y-axis represents the economic return.

The risk attitude is represented by a curve that divides the



area of the graph into two parts. The area above the curve represents the zone for which the organisation considers that the domain potential is sufficiently high for an investment on a product-line approach. Below that curve, the risk is considered too high for the expected benefits and therefore the domain should be rejected as a candidate for product-line investment.

6 CONCLUSION

The paper describes a simple, concise and effective way of determining the potential of a domain for introducing a product-line approach. The main contribution of this analysis model is that it summaries a lot of information in one single picture:

1. Support for an investment decision based on benefits vs risk dimensions.
2. Comparative analysis of several candidate domains and sub-domains to help the organisation to concentrate on the initial efforts in those domains which have a higher potential (i.e., achieve more benefits with fewer risks).
3. Indication in absolute terms of the convenience of investing in a domain taking into account the organisation's attitude toward risk.

The analysis has been successfully applied in several European organisations, including small and big software development teams from diverse domains, including utilities, banking, control systems, etc.

The analysis is usually performed in a one-day workshop with the participation of representatives from all the departments involved in the domain. It is especially important to involve not only the software development department, but also systems and marketing/sales departments to bring a customer perspective to the analysis.

These workshop-type meetings have proved to be extremely effective for reaching consensus among the different departments of stakeholders of a domain in a given organisation. When consensus is not reached, the differences among the participants are also recorded and shown in the graph. The area associated with the domain is a representation of these differences and provides an indication of the uncertainty of the information collected.

The domain potential analysis is presented as a tool to facilitate the inclusion of a economic perspective in setting a product-line strategy. It has been conceived with the idea of being simple enough to be usable, without requiring much effort or data that is not available in the organisations and, at the same time, complete enough to ensure a disciplined and repeatable analysis.

REFERENCES.

1. PRAISE (Product-line Realisation and Assessment in Industrial Settings) is pursued by Thomson-CSF (France), Robert Bosch GMRH (Germany), and the European Software Institute (Spain) under ESPRIT

Project 28651.
(http://www.esi.es/Projects/Reuse/Praise)

2. Wayne C. Lim, *Managing Software Reuse*, Prentice Hall, 1998.
3. Jeffrey S. Poulin, *Measuring Software Reuse*, Addison Wesley Longman, Inc., 1997.
4. PRAISE Consortium, Benefits Assessment model, P28651-D1.2, 1999
5. Barry W. Boehm, *Software Risk Management*, IEEE Computer Society Press Press, 1989.
6. Richard Pike and Bill Neale, *Corporate Finance and investment,* Prentice Hall, 1996.

Quantifying Software Product Line Ageing

Susanne Johnsson & Jan Bosch
Department of Software Engineering and Computer Science
University of Karlskrona/Ronneby
Soft Center
S-372 25 Ronneby, Sweden
+46 457 385 800
[susanne.johnsson|jan.bosch]@ipd.hk-r.se

ABSTRACT
It is a generally accepted fact that software ages over time. This requires software products to be replaced once their ageing inhibits the goals of the organization owning the product. However, few techniques are available for quantifying the negative effects of ageing software. In this extended abstract, we present an approach to quantifying the ageing of a software architecture. The approach can, among others, be used to decide upon the retirement of a software product line. Validation of the approach is in progress, but not reported upon in this extended abstract.

Keywords
Software product lines, architecture replacement, software erosion.

1 INTRODUCTION

Software evolves over time in response to new requirements. Due to this, the structure of the software typically degrades over time, i.e. the software ages. Consequences of the structural degradation process include, among others, decreased understandability and increased maintenance cost. At some point, it is no longer economically viable to continue to maintain the software product and the product should be retired and replaced with a newly developed software product.

In the domain of software product lines, the problem of identifying ageing and deciding when a software product line should be replaced is even more relevant. This is because several products and a larger part of the organization is depending on the software product line. In addition, the reusable assets of the software product line, i.e. the product-line architecture and the reusable components, are more exposed to evolution because these assets must satisfy the requirements of multiple products.

One can identify two extreme approaches to replacing a software product line. First, the software product line is used until it simply is not possible to use it any more, because it has eroded completely. On the other extreme, whenever it is possible to define a more optimal architecture, the software product line is replaced with a new. Neither approach is economically feasible, thus the optimal solution is somewhere in between. The decision of replacing a software product line is, in the end, an economical one. Consequently, it should be based on economical calculations.

Erosion of software product line assets can be caused by several types of changes. For instance: the incorporation of new features in the set of existing products; the incorporation of new technology; and the incorporation of new products.

Two types of maintenance tasks may cause architectural erosion. First, the change to one or more software products causes the product line to expand beyond its initial intent and scope. This frequently demands a decomposition of the product line architecture that is different from the one originally chosen. However, since the product line assets have been developed based on the initial decomposition, the required change will most likely be integrated in a less than optimal way in the architecture and the components. Second, due to time and resource constraints, a change may be implemented by a so-called quick-fix. That is, the functionality is added to the product line such that least effort is required now, but without any consideration of the future. Although erosion happens in both types of maintenance tasks, erosion will be relatively rapid in the latter case. In this extended abstract, we focus on maintenance-related erosion.

In this extended abstract, we present an approach, identifying and quantifying software ageing, that is based on the effort associated with maintenance tasks. This approach can be used to support the decision process concerning the replacement of a software product line, but also for deciding on major reorganizations of the software assets in a product line. Since validation of the approach has started, but we lack concrete results at this point, the

intention of this extended abstract is to present the approach to the research community and to obtain feedback from other researchers before continuing with validation.

The remainder of this extended abstract is organized as follows. In the next section, we categorize the effects of a maintenance task on a software system in three categories. In section 3, we discuss two models for identifying software ageing. In section 4, we discuss the application of the models to decision processes surrounding reorganization and retirement of a software product line. Finally, in section 5 we briefly discuss some related work and conclude the extended abstract.

2 MAINTENANCE TASKS

As discussed in the introduction, maintenance tasks affect the architecture and the components of the software product line. Maintenance activities may affect these assets in three different ways, i.e. new components may be added to the software product line, existing components can be extended with new units of functionality and, finally, functionality present in existing components may need to be changed. Below these types of maintenance effects are discussed in more detail.

First, new components may be added to the architecture. When adding a new component in a product line, this may be a product-specific component that is used as the implementation of an architectural component. A second type of new component may affect the product line architecture by its presence due to the rearrangement of component relations where the component is added. The productivity of this type of maintenance is likely to be similar to that of new development since the dependencies on the existing system are minimal and only present through interfaces.

The second type of effect of a software maintenance task is the extension of an existing component with new functionality. For instance, a product line component is extended at one of its configuration points with a software module containing extended functionality. The range of possible extensions is generally very wide, depending on the type of component. For instance, a white-box object-oriented framework will generally require considerable amounts of extension code both during instantiation and when incorporating new requirements. On the other hand, a communication protocol component will generally allow for very limited extension and will have relatively few configuration points. The productivity associated with this type of maintenance is lower than the type first discussed since the relation between the extension code and the existing component is more intimate. However, since it avoids changing the component internals, productivity will be considerably higher than traditional maintenance.

The final type of effect on the software assets of a product line is traditional maintenance, i.e. changes to the code in existing components. Changes to existing components may just affect the component internals, or also affect the interface of the component. In the latter case, this type of maintenance may cause ripple effects in the product line because components referring to the changed component may be affected as well. This type of maintenance has typically very low productivity. In some publications [1], productivity measures as low as 1.7 lines of code per day and per software engineer are reported.

In the remainder of this extended abstract, the effects of maintenance tasks will be referred to as N (New), NE (New in Existing), and CE (Change within Existing) respectively.

The effect of software ageing (or erosion), we believe to be twofold. First, over time the cost of implementing changes in the software product line will increase, due to, among others, the less than optimal structure and component implementations.

Second, the division of effects of maintenance tasks changes over time. Early in the life of a software product line, most requests for maintenance will lead to effects in the N category, i.e. adding new components. Some effects will be in the NE category, whereas relatively few effects will be of the CE category. One reason is that the initial design of the product line architecture is performed based on an implicit or explicit assumption of likely future changes to the architecture. Assuming that the actual maintenance tasks match reasonably well with the assumptions, these tasks will be relatively easy to perform.

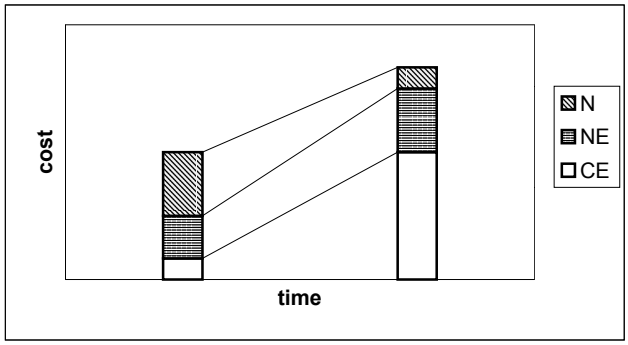


Figure 1 – Maintenance of different categories

Once the software product line has aged and has been maintained for some time, a gradual shift in the division of effects will occur towards an increased frequency of CE effects. The current scope of products and features may no longer be the same as the initial scope, which leads to a need to change the existing components in order for them to fit with the new requirements. A number of maintenance tasks may have been performed through quick fixes, which tend to seriously erode the product line in most cases because little thought is spent concerning the long term effects on the software product line. Thus, the relatively occurrence of N effects will decrease, whereas CE effects

will become much more frequent.

As discussed earlier in this section, traditional maintenance of the CE category has an associated productivity that is around an order of magnitude lower than N type maintenance. Thus, sufficient arguments are available for minimizing maintenance activities that erode the software product line, such as quick fixes; and for justifying structure-improving reorganizations of the product line when unacceptable levels of erosion have been detected.

3 QUANTIFYING SOFTWARE AGEING

Software ageing, as discussed earlier in this extended abstract, expresses itself in two ways. First, the average cost per maintenance task increases over time. Second, the relative division of effort for each maintenance task moves from the development of new components (N) to changing code in existing components (CE). In the sections below, we discuss approaches to measuring these indicators of software maintenance.

Average Maintenance Task Cost

The first measure we use to identify erosion of a software product line is the average cost per maintenance task. More and more changes to the product will have architectural impact; due to quick fixes and due to the 'drift' of the scope of the product line that causes the product line architecture to not fit the products now included in the product line. Thus, the maintenance cost will most likely increase as the architecture grows older.

Based on the average cost per change request and the change in the change request frequency over time, the cost of not reorganizing or replacing the current product line can be calculated. The cost of developing a new architecture together with the, historically derived, lower cost of future changes following the new replacing architecture, the cost of replacing the product line architecture can be calculated. The cost of developing a new architecture is based on the cost of developing the previous architecture.

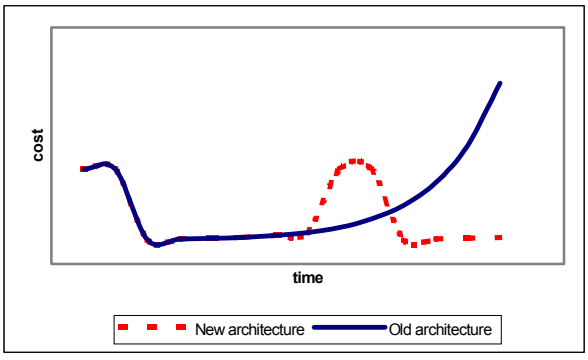


Figure 2 – Development and maintenance effort

Figure 2 is an example where the cost of using the architecture is shown over time. The graph is a simplification and the proportions may not match an actual industrial case. The cost for a set period of time is the area under the graph, within that time period, and the big bumps are the development cost of the entire architecture. From this example, we can conclude that it is important to start the development of a new product line before the cost of maintaining the existing product line becomes totally unacceptable. It takes considerable time to develop a new software product line, and in the meantime the cost per change in the existing product line, which needs to be maintained until it is retired, will continue to increase.

So far, we have not taken into account that the type of maintenance needed changes over the life time of the software product line. In the next section, we discuss the relative division of effort for maintenance tasks.

Relative Division of Effort

Since the cost of each change is related to the type of change needed, i.e. N, NE, or CE, the use of the overall average cost when calculating future cost of maintenance does not provide a completely accurate perspective. Using the overall average cost of changes leads to estimations where the cost of maintenance are overestimated in the beginning of the life of the product line and, consequently, underestimated later in the lifecycle.

When, on the other hand, taking into account in the calculations the cost variation of maintenance, the results will be more accurate. We can collect the average cost, or effort, per maintenance task for each category. Analyzing this allows us is to determine the trend of the relative division between the different categories. As long as the major part of the maintenance is of category N, the architecture should not have suffered much from erosion. Eventually, an increasing part of the maintenance will shift towards the CE type. Maintenance then gets significantly costlier. The analysis of the figures is done in nearly the same way as described earlier, except that the different costs of the different type of maintenance is also considered. Here too it is the area under the graph that is interesting, i.e. the cost for a given period of time, the graph is slightly more complex though since there are additional factors to take into consideration, i.e. the sum of average costs of N, NE, and CE. The graph will be flatter in the beginning and steeper later on than if the simple approach was used.

Discussion

Calculating average maintenance task cost does, in most cases, not require planning ahead of its use, but can be performed afterwards. Typically, all the needed figures can be retrieved from the administration surrounding software maintenance projects and activities. If this information is not available for each maintenance activity, it may be possible to collect the total maintenance costs for the entire lifetime of the architecture. Using the total cost and the period, an overall average cost for maintenance can be

calculated. This figure can be used to predict future maintenance costs, but the increase of maintenance cost over time has to be considered for achieving fair predictions. The decision concerning reorganization and replacement of the software product line should aim at minimizing the total lifecycle cost.

The relative division of effort for maintenance tasks does require either upfront planning or detailed documentation of the actual changes and extensions made during each maintenance tasks. If this information can be retrieved, this can be used to calculate the trend of the average cost per maintenance task over time.

4 SOFTWARE PRODUCT LINE EVOLUTION

Although the approach discussed in the previous section support the decision process concerning reorganization and retirements of software product lines, other aspects should influence the decision process as well. The two main factors that should be considered as well, are the occurrence of major changes with architectural impact and the evolution of the scope of the software product line.

Major changes with architectural impact can be caused by e.g. availability of new technology; a major change of the context in which the product line members are used. In the case of a major shift in technology it can be decided whether a replacement is needed depending on the overall impact on the software product line. In cases where it is not obvious, an assessment should be performed to analyze the possibility of incorporating the new requirements into the current software product line. This estimation can be compared with the effort needed to replace it.

The second influencing factor is the evolution of the scope of the software product line. The scope of a software product line typically expands over time, but one can identify three types of scope evolution. The different types are shown in the figure below.

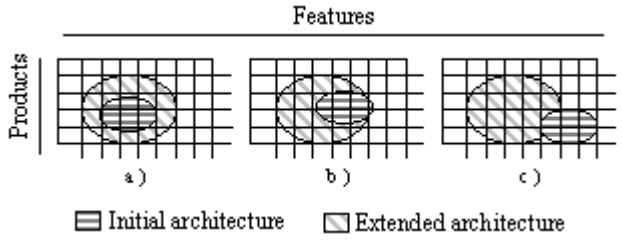


Figure 3

The examples in figure 3 illustrate the different types. The domain of alarm systems is used in the demonstration; the original scope (the small circle) is fire alarm systems for individual buildings in all three examples.

a) The scope is extended to fire alarm systems for multiple buildings; the core of the scope remains as the core.

b) The scope is extended to also include intruder alarm systems and systems for passage control; the core slightly shifts.

c) The initial use of the fire alarm has shifted from warning when there is a fire burning, to indicate when someone smokes cigarettes in a no smoking area. Thus, the purpose of the system has shifted.

The old architecture can be appropriate to keep and extend if the case is as in ‘a’, in the case of ‘b’ and ‘c’, it is most likely that it should be replaced.

In the case of a major change with architectural impact or evolution of the scope of the product line, the approach presented in section 3 can be used to support the decision process. The presented approach can be used to quantify the erosion of the software product line. With this as an evaluation base, analysis can be done concluding whether to replace the existing software product line or to perform a reorganization of it. In this analysis consideration of the eroding impact of the extensions to the product line must be taken into account.

5 CONCLUSION

In this extended abstract we have discussed the evolution of software product lines, how the cost of maintenance tasks increases over time and the relative division of effects resulting from maintenance tasks. The types of effects that we identified are (1) adding components to the product line, (2) adding extensions to existing components and (3) changing code in existing components. The main contribution of this extended abstract is the identification of the relation between different types of effects and the ageing or erosion of a software product line. We claim, and plan to validate this in the near future, that over time the relative division between these effects moves from the first to the last category.

The discussed model for identifying architecture erosion can be used in the decision processes surrounding the reorganization and retirement of software product lines.

Finally, since this is an extended abstract, rather than a full paper, not all appropriate references and text has been included. Our main objective is to discuss the presented ideas here with the workshop participants and to obtain feedback while preparing a full paper.

REFERENCES

1. Henry, J. E., Cain, J. P., “A Quantitative Comparison of Perfective and Corrective Software Maintenance”, *Journal of Software Maintenance: Research and Practice*, John Wiley & Sons, Vol 9, pp. 281-297, 1997.

2. Mikael Svahnberg, Jan Bosch, ’Evolution in Software Product Lines: Two Cases’, *Journal of Software Maintenance*, Vol. 11, No. 6, pp. 391-422, 1999.

Case studies, experiments, reports from industrial projects

A Comparative Analysis of Domain Engineering Methods: A Controlled Case Study

Ali Mili and Sherif M. Yacoub
CSEE Department,
West Virginia University
Morgantown, WV26506
{amili, yacoub}@csee.wvu.edu
Tel: +1 304 293 0405 x {2548,2537}

Abstract

The deployment of product line engineering requires a profound understanding of domain engineering as an essential phase in a product line lifecycle. A domain engineering method defines domain activities which include: domain definition, domain analysis, derivation of generic domain architectures, identification of commonalties and variabilities in product families, and identification and specification of domain-wide software assets. To gain insight on how to do domain engineering in a product line context, we need to analyze and compare existing domain engineering methods using controlled experiments and case studies.

This paper reports on a classroom experiment in which we analyze and compare a sampling of domain engineering methods. We discuss the details of the experiment and the lessons which we draw from it, then we discuss some preliminary conclusions about the experiment.

Keywords: Domain engineering, domain analysis, software reuse, FODA, JODA, Synthesis, Reuse Business.

1. Domain Engineering Methodologies

There is an increasing recognition nowadays that specialized forms of software reuse, such as product line engineering, component based software engineering, and COTS-based software engineering, offer the greatest promise in software reuse practice. The much promised but seldom observed gains in productivity, quality, and time to market, have not generally materialized outside the confines of systematic software reuse initiatives, such as those advocated by these specialized paradigms of software reuse. Sound domain engineering is one of the common features that these paradigms rely on: there is an increasing recognition that reusable assets are best developed as part of a carefully planned, soundly executed, thoroughly validated, domain engineering effort [2].

In this paper we discuss a controlled classroom experiment in which we have student teams use different domain engineering methodologies to tackle a domain engineering/ application engineering term project. The methods that student teams chose are: Feature-Oriented Domain Analysis (FODA) by the Software Engineering Institute [3,4], the Joint Object-Oriented Domain Analysis (JODA) by the Joint Integrated Avionics Working Group (JIAWG) reuse subcommittee [5], Synthesis by the Software Productivity Consortium [6], and Reuse Business by Jacobson *et.al*. [7]. The purpose of the experiment is to analyze and compare the candidate domain engineering methods with respect to the following criteria.

- *Rationale for Domain Definition.* In [1], we have observed that a domain definition must be driven by an economy of scale rationale, and that there are three alternative rationales:
  - *Common Expertise*. This criterion identifies a field of domain expertise and attempts to cater to it through specialized development of software assets that span the range of applications of the expertise at hand. This criterion is producer-focused, and could be a natural criterion for a domain expert.
  - *Common Design*. This criterion identifies a problem solving pattern that is embodied in some generic software assets and attempts to cater to it through the development of generic assets that can be specialized for specific needs. This criterion is product-focused, and could be a natural criterion for a programming expert.
  - *Common Market*. This criterion identifies a segment of the software market and attempts to cater to it through specialized development of software assets that cover the range of needs of the market; its rationale is to be a one-stop shop for the selected market segment. This criterion is consumer-focused, and could be a natural criterion for a (marketing) manager.

  Of course these criteria are not mutually exclusive: an ideal domain is one that satisfies all three criteria, a poor domain definition is one that satisfies neither. The question we have is whether a domain engineering methodology promotes one criterion or another in attempting to define a domain, be it perhaps implicitly.
- *Processes for Domain Definition.* A domain may be defined in one of two ways:

– *By Increasing Commonality*. We start from a comprehensive definition, which includes a wide range of (possibily heterogeneous) applications, and we reduce this set in a stepwise manner, until we are satisfied with the level of commonality of the applications that remain. This process can also be referred to as *stepwise exclusion*.

– *By Increasing Variability*. We start from a small set of applications, which have presumably a great deal of commonality, and we expand the set until we are satisfied that the domain is fairly large, while still maintaining adequate commonality. This process can also be referred to as *stepwise inclusion*.

We are interested to know, for each method, whether the method advocates a specific process. Also, for methods that advocate domain definition by increasing variability, we are interested to know whether the method requires that we know ahead of time the list of applications in the domain.

- *Domain Engineering Lifecycle*. We are interested in analyzing the lifecycle advocated by each method, and assessing to what extent the lifecycle gives concrete guidelines in producing the deliverables of domain engineering.
- *Support for Legacy Assets*. Some domain engineering methods make provisions for integrating legacy assets in the domain analysis activity. We are interested to analyze which methods make such provisions, and how legacy assets affect the domain analysis process.
- *Guidelines for Domain Architecture*. We are interested to know, for each method, whether the method advocates a specific architectural style (e.g. layered, client/server, or pipes-and-filters architectures), and whether it provides constructive guidelines for how the architecture is derived. In particular, does the method provide guidelines for mapping commonalities and variabilities into architectural features. Are the architecture guidelines *product-based*, i.e. describe what the architecture is and how it is represented or *process-based*, i.e. describe how to create the architecture.
- *Domain Engineering Deliverables*. Each domain analysis method produces a set of deliverables, which include domain definition, assumptions, analysis models, etc. We are interested in analyzing the deliverables of each method, and assessing to what extent these deliverables capture information about the domain of the applications family.
- *Reusable Assets*. We are interested in cataloging the set of reusable assets that are produced for this application domain. An analysis of this set for each method and a comparison across methods should give us some insights into design features of each method.
- *Technology Dependency*. Some domain engineering methods advocate a specific technology, and can only be applied within that technology.
- *Language Dependency*. Some domain engineering methods advocate a specific set of languages (for specification, architecture description, design, implementation, etc), and can only be applied using these languages.
- *Domain Engineering Effort*. How much effort was needed to conduct the domain engineering phase? how was this effort distributed on the various activities/ phases? The cost of domain engineering cannot be used in isolation to assess a domain engineering method: a method may cost more than another, but may also produce better reusable assets, thereby reducing application engineering costs. Hence we consider that it is the sum of domain engineering costs and application engineering costs that reflects on the cost effectiveness of a domain engineering method.

2. A Controlled Case Study

Course CS 477, *Software Reuse*, is offered in the *CSEE* Department at West Virginia University. As part of their evaluation, students are required to work on term projects which include a domain engineering task and an application engineering task. Section 2.1 describes the domain engineering task, and section 2.2 presents a selection of sample applications in the selected domain. Students are organized in teams; each team turns in a collective domain engineering deliverable, and each student subsequently turns in a selected application developed using the team's domain engineering deliverables. We notify the students that the applications given in section 2.2 are only a sample, and that the proposed domain contains all queue simulation applications, as defined by the range of variability discussed in section 2.1. Also, the assignment of specification applications to students is not carried done until *after* domain engineering is complete, and is done by the instructor. So that when students perform domain engineering, they do not know which application they will end up developing; this ensures that they make their assets as generic as possible. Student teams select a domain engineering methodology among a set of six candidates; generally, they have no prior knowledge of these methodologies. For the purposes of this study, student teams are asked to fill out questionnaires. The conclusions provided in this paper stem from their replies, as well as our own analysis.

2.1 Domain Engineering

In this section, we present the application domain of interest by presenting in turn our perception of the commonalities and variabilities of the proposed domain. Individual groups, guided by their respective methodologies, may alter these definitions of commonalities and variabilities by refining them or otherwise reformulating them.

2.1.1 Commonalities

We want to develop a set of reusable components for the purpose of producing applications that simulate the behavior of waiting queues. Examples of application include: simulating the traffic of processes through a CPU dispatcher; simulating the traffic of passengers through check-in counters; simulating the traffic of customers at a post office, etc. Among the commonalities between all the applications of our domain, we cite the following:

- **System Topology**. All the applications include service stations, service users, and queues of service users.
- **Simulation Events**. Users appear in the system at random intervals of time, and have time requirements that are drawn at random upon their arrival.
- **Statistical Measurements.** All the applications collect statistical information about the quality of service that users have received (waiting time) or the quantity of service that the system has delivered (throughput).

2.1.2 Variabilities

Individual applications vary in a number of ways, including:

- **Topology of service stations.** The number of service stations may be fixed or variable (in time); we may have one service station or more than one; if more than one service station, the service stations may be interchangeable (they deliver the same service) or not (e.g. check out counters for 10 items or less; check out counters for more than 10 items).
- **Service time**. The service time may be constant or variable; if it is variable, it may be determined by the customer or by the service station or by both (e.g. customer determines amount of service needed, and service station prorates that with its own productivity factor); also, if it is variable, it may be subject to a maximum service value (as is the case in CPU dispatching algorithms); when the maximum is reached, the customer may be thrown out, queued at the end of the queue where it was, or considered as a new arrival.
- **Topology of queues**. A single queue; multiple interchangable queues; multiple queues with different service categories (each customer may line up at queues of a given category).
- **Types of queues.** First In First Out queues; Last In First Out queues; priority queues; limited size queues.
- **Arrival distribution.** Markovian distribution; Poisson distribution; clustered distribution (if the service stations are immigration posts at an airport, then passenger arrivals are clustered around/after flight arrivals).
- **Dispatching policy.** Customers are assigned to queues at random, and may not change queues after the first assignment; customers are assigned to the shortest queue upon arrival and may not change queues subsequently; customers assigned to the shortest queue upon arrival and switch queues to take the shortest subsequently, until they are served.
- **Measurements.** Average waiting time; standard deviation of waiting time (as a measure of fairness); maximum waiting time; throughput.

The object of this domain engineering exercise is to develop a set of generic software components that can be easily combined to produce any queue simulation we may want; samples of such simulations are discussed in the next section.

2.2 Application Engineering

In order to exercise the domain engineering activity that is carried out according to the requirements discussed above, we propose a testbed of software applications, which are all instances of queue simulations.

1. **CPU Dispatching**. We want to simulate the behaviour of a CPU dispatching mechanism. We are interested in measuring fairness and throughput. There is a single priority queue, with maximum service time (quantum service, *Q*); once a process has exhausted its service time, it is queued back, with an increased priority.
2. **Self-Serve Carwash**. We have a set of self-serve interchangeable carwash stations. Arriving cars line up at the shortest queue (queues of equal length are interchangeable) and do not change queues subsequently; queues are FIFO, of course. Service is limited to a maximum value (but may take less time), and cars are expected to clear the station once the maximum time has expired. Arrival distribution is Markovian. We are interested in monitoring maximum waiting time (we do not want anybody to leave before being served) and throughput (we want to serve as many people as possible).
3. **Check-Out Counters**. We have a number of check out counters at a supermarket, some of which are reserved for shoppers with 10 items or less. Shoppers with 10

items or less line up in the shortest express check-out queue; others line in the shortest queue reserved for them. Once they are lined up in some queue, shoppers do not leave the queue until it is their turn. Service time is determined by the shopper (size of his/her cart) and by the productivity of the cash register attendant (a factor p between, say n and h, where $0<n \leq 1 \leq h <2$). Arrival rate is Markovian distribution. The number of stations increases whenever the longest queue exceeds a threshold value *L* and decreases by one whenever the number of stations of each category is greater than one and the length of the queue is zero. Whenever a new cash register is open, shoppers at the end of the queue rush to line up at the station (talk of fairness!) until the length of the queue equals the shortest current queue of the same type (express checkout, regular checkout). We are interested in average waiting time and fairness.

4. **Immigration Posts**. We have a number of immigration stations at an airport, some of which are reserved for nationals, the others are for foreign citizens. There are two queues: one for nationals, the other for foreigners; each queue feeds into the corresponding set of stations, and there is no transfer between queues. The number of stations that handle nationals increases by one whenever the length of the nationals' queue exceeds some value, say *L*; and the number of stations that handle foreigners increases by one whenever the length of the foreigners' queue exceeds some other (larger?) value, say *M*. The arrival rate is a clustered distribution, as passengers come by planeloads. Service time for nationals is constant, and service time for foreigners is determined by the passenger and by the productivity of the immigration agent attendant (a factor p between, say n and h, where $0<n \leq 1 \leq h <2$). We are interested in monitoring throughput.

5. **Check In Counters**. We have two FIFO queues for passengers at an airline checkin station: a queue for first class and a queue for coach. We have two categories of service stations: first class and coach; the number of stations does not change for the length of the experiment. The duration of the service is the same for all passengers and all stations of the same class, but differs from first class to coach. The arrival rate is Markovian distribution; passengers lines up at their designated queue and do not leave it until they are served. Whenever one queue is empty, the corresponding service stations may serve passengers of the other queue (typically: first class stations serve coach passengers when no first class passengers are waiting). We are interested in monitoring the average waiting time and the maximum waiting time for each class of passengers.

6. **Round Robin Dispatching**. Same as example 1, but with a FIFO queue; processes that exceed their time quantum are inserted at the back of the queue.

7. **Self-Serve Carwash, arbitrary service time**. Same as example 2, but without limit on the service time.

8. **Fair Check-Out Counters**. Same as example 3, but whenever a new counter is open, it is filled by shoppers at the front of the longest queues (although in practice they are least motivated to go through the trouble, their queue swapping will probably minimize average waiting time and maximize fairness).

9. **Multiqueue Immigration Posts**. Same as example 4, but with one queue for each post; use the policy of example 3 for queue swapping when a new post is created. Assume also that passengers go from queue to queue whenever their position in the current queue is farther (from the head) than the length of another queue.

10. **Fair Multiqueue Immigration Posts**. Same as example 9 with the policy of example 8 for queue swapping when a new post is created.

2.3 Candidate Methodologies

The class includes four teams; each team has chosen a domain engineering methodology among a set of six candidates. The four that were selected are: *FODA*, *JODA*, *Synthesis*, and *Reuse Business* methods. We review them briefly below. Henceforth, we refer to the team by the name of the methodology they have chosen.

2.3.1 Feature-Oriented Domain Analysis, FODA

Feature-Oriented Domain Analysis (FODA) method is developed at the Software Engineering Institute (SEI), Carnegie Mellon University [3,4]. FODA focuses on identifying features that characterize a domain and hence gives the approach its name. Products in a domain provide several capabilities. These capabilities are modeled in FODA as features. To model these features, FODA defines a process for domain analysis that is based on three activities (discussed later in Section 3.1): context analysis, domain modeling, and architecture modeling.

2.3.2 Joint Object-Oriented Domain Analysis, JODA

The JODA domain analysis method advocates the idea that software objects are more understandable and customizable than traditional functions and subroutines. JODA is developed by the Joint Integrated Avionics Working Group (JIAWG) reuse subcommittee [5]. JODA defines domain models using Coad/Yourdon Object-Oriented Analysis technique. These domain models are then used to define the domain architecture. Domain models are developed using the Coad/Yourdon whole-part and inheritance diagrams.

JODA is the domain analysis part of the reuse-based software development approach defined by JIAWG. This approach also includes business and methodological planning. Business planning identifies the high level domain that the analysis approach will be applied to. This definition of the domain usually includes the domain scope, technology dependencies, and whether domain expertise is available. The methodological planning defines the domain engineering activities, application-engineering activities, and how they integrate. The JODA domain analysis method defines the domain structure and requirements, and captures them in domain models. These domain models are implemented and stored as a repository of reusable software objects.

2.3.3 The Synthesis Domain Analysis Methodology

Domain analysis is part of the Reuse-Driven Software Process (the Synthesis approach) that is developed by the Software Productivity Consortium [6]. As a part of both the opportunistic and the leveraged approach, domain analysis activity is defined as the scoping and specification of a domain based on the analysis of the needs of a targeted project in an organization. The activities of domain analysis include: domain definition, specification, and verification.

2.3.4 Reuse Business Methodology

The Reuse-driven Software Engineering Business is a framework developed by Ivar Jacobson *et.al.*[7] to define a set of guidelines and models that help ensure success with large-scale object-oriented reuse. The framework is referred to as Reuse Business. It deals systematically with business, process, architecture, and organization issues in a product line. The Reuse Business processes are categorized under three main categories: Component System Engineering, Application Family Engineering, and Application System Engineering. Component System Engineering is responsible for creating component systems. Each component system is a set of customizable configurable software components where a component is a type, class, or any work-product that has been specially engineered to be reusable. Application Family Engineering creates the overall system architecture and identifies the component systems that will be used with the architecture to develop applications that belong to the same domain. Application System Engineering is the process of creating a specific application that belongs to the domain by selecting, specializing, and assembling components from component systems. Reuse Business does not have an explicit domain engineering process, but distributes the main domain engineering processes between Application Family Engineering and Component System Engineering.

3. Analytical Observations

In this section, we review the questions that we had raised in section 1 and attempt to answer them for each of the four selected methods.

3.1 Domain Engineering Lifecycle

We give a summary of the activities that each team has experienced when applying the domain analysis method that they selected. This synopsis of the domain engineering lifecycle gives insight into the activities defined by each method and helps in rationalizing the comparison made in subsequent sections. The deliverables of these activities as applied to our case study are discussed in Section 3.8

3.1.1 FODA

FODA advocates a three-phase domain analysis lifecycle:

- *Context Analysis.* The purpose of context analysis is to scope the domain. It consists of two steps: the derivation of structure diagrams and context diagrams.
- *Domain Modeling.* Domain modeling identifies commonalities and differences that characterize applications in the domain. It has three sub-activities: feature analysis, information analysis, and operation analysis. Feature analysis captures the capabilities of applications in the domain. Information analysis captures domain knowledge in terms of entities and their relationships. Operation analysis captures the functions that applications in the domain perform.
- *Architecture modeling.* Architecture modeling captures the high-level structure of applications in the domain.

3.1.2 Synthesis

The Synthesis domain analysis method has three main activities:

- *Domain definition*. The Synthesis team developed a set of domain definitions in terms of glossary, domain synopsis, assumptions, and domain status as discussed later.
- *Domain specification*. The domain specification process is concerned with:
  - Defining decision models that define what is inside and what is outside the domain.
  - Identifying product requirements that define the domain concept, context, content, and constraints.
  - Identifying the process that defines how to create applications that belong to the family.

- Defining the product design process that develops the architecture, and designs reusable components.

- *Domain verification*. The domain verification process verifies the domain definitions, domain specifications, and implementations.

Synthesis is an iterative process; continuous feedback is provided from domain implementation and application engineering to the analysis processes.

3.1.3 JODA

JODA reuse lifecycle has four main activities: business planning, methodology planning, domain engineering, and application engineering. The domain engineering phase has an analysis process, an implementation process, and a process for constructing a component repository. The domain analysis activity defines three main processes:

- *Domain preparation.* This process consists of:
  - Interviewing domain experts, and analyzing existing legacy systems.
  - Identifying future trends. The JODA team considered future extensions to the domain by considering applications that simulate queuing networks. These applications have links connecting servers, sinks, and sources. Different sources generate customers that can possibly request services from multiple cascaded service stations. The team perceived that queuing networks is a future extension that is worth investigation in their domain analysis phase. The focus on future trends is viewed as a way to depart from the specific set of applications that are envisioned at domain engineering time.
  - Identifying the stability and maturity of technology in the domain. OO models are becoming more stable due to the unification of OO models in one modeling language, the Unified Modeling Language [8].
- *Domain definition and scoping*. The process consists of:
  - Defining domain glossary and services.
  - Developing a subject diagram. The subject diagram shows the subject of interest, a process, and an output in the form of a block diagram.
  - Developing a whole-part diagram that shows relationships between a subject at its constituents.
  - Developing inheritance diagrams that show how common concepts are captured as abstractions and concrete components provide different implementations.
- *Domain modeling.* Domain modeling packages reusable objects and defines reusable scenarios, object life history, and state-event-response diagrams.

3.1.4 Reuse Business

The domain engineering phase of the Reuse Business lifecycle has two main activities: the Application Family Engineering that is concerned with developing the domain architecture, and the Component System Engineering that is concerned with the development of reusable components. The main activities that the Reuse Business team experienced in Application Family Engineering are:

- *Capturing requirements*. Reuse Business captures requirements using OO use case models. Each way an actor (a system user) uses that system is a distinct use case. Each use case is further analyzed using a set of interactions within the system, i.e. scenarios.
- *Performing robustness analysis*. Robustness analysis is concerned with identifying analysis types and then clustering them into subsystems that represent candidate components. For example, use cases and business models that are highly related to each other are grouped into analysis models.
- *Designing the layered architecture*. The Reuse Business team defined the architecture for our case study into two layers: the application system layer and the component subsystem layer. The component subsystem layer contained a number of components that are general for the domain, e.g. Source, Service Facility, Queue Facility, Sink, and Measurements.

Reuse Business is an iterative process.

3.2 Rationale for Domain Definition

FODA defines the domain by instantiating a structure diagram and a context diagram. The common characteristic among all applications in a domain is the aggregate of a structure diagram and a context diagram. The structure diagram maps events and concepts, and the context diagram shows the data flow between service stations, customer queues, and the simulation program. Clearly, FODA's rationale for domain definition is *Common Design*.

The Synthesis domain analysis team found that Synthesis domain analysis emphasizes the *Common Market* characterization of a domain definition, because Synthesis emphasizes a business reuse level solution. As part of the domain definition, Synthesis documents the domain status, which includes an endorsement that the domain synopsis and assumptions define a viable domain. It also defines the confidence and risks associated with the endorsement. To meet the common market rationale defined by the Synthesis process, the team considered

themselves a company targeting the market with this product-line and analyzed their background and experiences in the field. The team considered the other teams as market competitors and analyzed the market from that perspective.

The JODA domain analysis method is a part of a larger reuse lifecycle that is composed of: business planning, methodology planning, domain engineering, and application engineering. JODA identifies the domain as part of its business planning phase. The criteria for identifying a domain are as follows: is the domain well understood, is the technology predictable, and is the domain expertise available to support domain engineering? These criteria are mostly related to our *Common Expertise* classification. Though the JODA team did not have the specific domain expertise in our case study, the team managed to relate the project to their previous experience in simulation of queuing networks.

Reuse Business defines an application family as a set of applications with common features. These set of applications work together to help some users accomplish their work, such as Microsoft Office (Word, Access, Powerpoint, etc.). Reuse Business also considers the same application systems that need to be reconfigured, packaged, and installed differently for different users as a family. In addition, a family is a set of fairly independent application systems that are built from the same lower level reusable components and the same architecture. Therefore, we consider this method a *Common Design*.

Observations. Table 1 summarizes the observations made by the teams. Domain analysis methods differ in the criteria by which they characterize and define a domain. The teams were able to use the classification of *Common Design*, *Common Expertise*, and *Common Market* to characterize the domain of our case study differently according to the analysis method they each team used.

FODA	Synthesis	JODA	Reuse Business
Common Design	Common Market	Common Expertise	Common Design

Table 1 Rationale for domain definition

3.3 Processes for Domain Definition

FODA advocates to define the domain by collecting relevant information, based on the study of existing systems and their development histories. Hence its approach to domain definition appears to be driven by *increasing variability*.

The Synthesis team pursued a bottom-up approach based on sample applications and *increasing variability*. However, the Synthesis method itself does not imply an increasing commonality or increasing variability process. The team reported that because the domain in our case study is limited, they pursued the increasing variability approach. They also tried to increase variabilities by adding other applications to the domain such as train unit dispatchers and telephone waiting queues.

The JODA method heavily relies on domain expertise and interviewing and reengineering existing systems. We tend to classify this as an *increasing commonality* approach because the method collects a wide amount of information and uses this information to narrow down the domain.

The Reuse Business team found that the process is better described as *increasing variability*. They point out that the main feature of the method is modeling with use cases. They started with the set of common applications, given in section 2, from which they identified use cases.

Observations. We note that the teams found it hard to characterize the domain analysis process as *increasing variability or increasing commonality.* They were inclined to use the *increasing variability* approach. This may reflect more on the way the domain was described than on meaningful differences between methods.

FODA	Synthesis	JODA	Reuse Business
increasing variability	Both	increasing commonality	increasing variability

Table 2 Process for defining a domain

3.4 Support for Legacy Assets

FODA supports reengineering because the process studies existing systems, their development histories, and knowledge captured from domain experts[3]. Synthesis supports reengineering legacy systems as part of its domain analysis activities. JODA is heavily based on reengineering existing systems and interviewing domain experts. Reengineering is the first process in the domain preparation phase of JODA. Reengineering of legacy systems is also encouraged in Reuse Business. Each legacy component or subsystem can be wrapped using one or more OO wrappers.

Observations. The teams found that reengineering legacy systems is an important activity in all domain analysis methods. Source code (developed in an object-oriented language) from an earlier project [9] was available, however, we restricted the teams from using this code. We found that since reengineering is supported by all methods, the experiment would not benefit from providing legacy code and models, and on a worst case could cause unbalance in the comparison of domain analysis methods especially in favor of methods that are based on OO technology.

3.5 Guidelines for Domain Architecture

3.5.1 FODA

The guidelines provided by the FODA team about the domain architecture are *product-based*. The architecture is defined in terms of component interfaces, model execution (scenarios), and nature of interconnections (specifications of architecture connectors). The FODA team developed a *layered* architecture style for the domain such that reuse can occur at the layer appropriate for a given application. In general, FODA advocates a layered architecture that has four layers:

- the domain architecture layer, which is a set of processes and their interactions,
- the module structure chart layer, which shows the packaging of functions, features, and data into modules that are common to applications in the domain,
- the common utilities layer, which defines utilities that are general across several domains, and
- the subsystem layer, which defines operating system, languages, etc.

The FODA team also identified some guidelines for creating the architecture which include defining concurrent processes, defining common modules, and then mapping features, functions, and data to these processes and modules.

3.5.2 Synthesis

The Synthesis team developed the architectures as a consequence of activities for defining commonalities and variabilities. The Synthesis team described a *process-based* guideline for creating architectures. An interesting result that the Synthesis team found is that they can map commonalities into components and variabilities into parameters to these components. The Synthesis method is flexible on representation and definition of the architecture style. The team used UML.

3.5.3 JODA

The JODA team found that the guidelines for creating architectures are mostly *process-based*. The team used scenarios for developing subject diagrams, whole-part diagrams and further refined these diagrams by analyzing detailed scenarios of each part. There are no restrictions on how JODA represents the domain. The team emphasized the role of scenarios in deriving the domain architecture and identifying components.

3.5.4 Reuse Business

The Reuse Business team found that the architecture guidelines are *product-based*. In an OO context, the team described the architecture as a set of subsystems, their interfaces, and nodes on which these subsystems are executing. The Reuse Business domain analysis method produces a layered architecture where a layer is defined as a set of subsystems with the same degree of generality. Upper layers are application specific and lower layers are generic. Thus the representation of the architecture is more emphasized than the process of developing it.

Observations. Table 3 illustrates the observations made by the teams on the architecture guidelines that they inferred from their selected domain analysis methods. The table shows whether the guidelines are process-based or product-based. It also shows the main characteristic of the architecture style or the main guideline for the creation process.

	FODA	**Synthesis**	**JODA**	**Reuse Business**
Product-based	Layered			Layered
Process-based		Mapping commonalities and variabilities to components	Scenarios	

Table 3 Guidelines for domain architecture

3.6 Domain Engineering Deliverables

In this section we analyze the deliverables of each domain analysis method as applied to our case study.

3.6.1 FODA

The FODA team developed the following domain models and artifacts:

- *Context diagrams*. FODA uses Structured Analysis and Design (SA/SD) context diagrams to show data flows and relationships between the domain under consideration and the environment, i.e other entities and abstractions that are outside the scope of the domain.
- *Structure diagrams*. Structure diagrams are block diagrams that describe the domain under consideration and all other domains in a layered form. The target domain is placed in the structure diagram relative to higher, lower, and peer level domains. All other domains that interface with our domain are placed in the structure diagrams.
- *Feature models*. Feature models are arrow-and-box diagrams that capture the capabilities of applications in the domain. They are the attributes of applications that directly affect the end users. These features include services provided by applications in the domain, performance, hardware platform required for the

domain, etc. For instance, in our case study, the FODA team documented some alternative features such as a queue can be a FIFO or LIFO, and other mandatory features such as an event should have a type and time. They also reported on some operational features such as service operations (selecting a server, dispatching, and scheduling events).

- *Information models*. Information models are represented as entity relationship diagrams. These models are a representation of domain data and knowledge. In our domain, a client, a queue, and a server are examples of domain entities.
- *Operational or functional models*. These models capture functions and behaviors. Functions are captured as control and data flow diagrams, which describe inputs, outputs, activities, internal data, and data flow relationships. Behaviors are captured using state machines that capture events, states, state transitions, and outputs. The FODA team produced a functional model for the system; however, behavior models were found of less significance and were not recorded.
- *Domain architecture*. Architecture models are defined in terms of process interaction models and module structure charts. The team decided to use a single processing node and hence they were more concerned with developing structure charts.

3.6.2 Synthesis

The Synthesis team produced outputs for each of the phases defined in Synthesis:

The outputs of the domain definitions process are:

- *Domain synopsis*. An informal statement describing the domain. The Synthesis team named the project Waiting Queue behavior Simulation (WQBS). They used the description provided in Section 2 and added possible applications such as rail/train dispatches, telephone waiting queues, and certain layout of manufacturing production flow.
- *Domain glossary and references*. These are definitions of standard terms and references to external sources. For the case study, some of the terms that the Synthesis team defined include: Arrival, Channel, Counter, Productivity, Queue, Queue Discipline, Task, Service Station, LIFO, FIFO, etc. As for external references, the team referenced the class note material, the reuse-driven development environment from the ASSET web site [10], and a book on theoretical concepts of waiting lines and queuing theory.
- *Domain assumptions.* The Synthesis team documented assumptions and their rationale. The assumptions include commonalities, variabilities and exclusionary assumptions. The Synthesis team did not only consider the commonality and variability assumptions given in Section 2, but they also developed more assumptions as they went into the definition of the domain. The team added assumptions regarding minimum set of components involved in a scenario, the minimum set of components in a product, the assumption about customers staying in the queue until served, etc. The team elaborated on the variabilities defined in Section 2 and added new assumptions as related to the domain. Among variability assumptions that the team added we mention: assumption about assigning customers to the queue at random or to the shortest queue, customers having attributes that restrict them to specific queues, assumptions about the user interface and the display, etc. Exclusionary assumptions deal with things that are excluded from the domain. For instance, the team assumed that the domain does not handle the situation in which customers leave the system because of long service times.
- *Domain Status*. Domain status is an assessment of the maturity and viability of the domain. The team has done some feasibility study and analysis of project risk as if they are a company with market competitors (other teams).
- *Legacy products*. A list of any legacy products that may provide information or material for developing the domain. For our case study, none was created.

The outputs of the domain specification process are:

- *Decision models*. Decision models capture the set of variability assumptions and the alternatives that application engineers will later choose from. These decisions were grouped into conceptual components. Constraints about these conceptual components are also added. The team identified the following conceptual components: Customer, Simulator, Scheduler, Queue, Server, Controller, and Metric Collector.
- *Product requirements.* The product requirements define: the concept, the purpose and objectives of the domain; the context, the relationship to the environment; and the information content and constraints, i.e. the architectural components. The team restated the concept from the domain definition. They also identified some external environmental elements such as keyboard for entering simulation parameters and platform on which the product runs (the team intended to develop Java components to make the product platform independent). The team then combined decision models with the context and concept definitions to define architectural components. The architectural components were one-to-one map with conceptual components with the addition of input and output components.

- *Process Requirements and Product Design*. Synthesis domain analysis specifies the process that will be later used by application engineers to develop applications. The Synthesis team is currently working on this phase.

3.6.3 JODA

The JODA team produced outputs for each of the domain method processes:

The outputs of the domain preparation process are:

- *Domain expert information*.
- *Future trends in technology and domain stability*. In analyzing the technology on which JODA is based, the team found that OO models used in JODA have been updated by new models as defined in UML. However, the study of OO modeling trends show that UML is becoming a stable modeling language that many designers and analysts in OO world confidently use.

The outputs of the domain definition process are:

- *Subject diagrams*. The team reported a subject diagram for our case study that includes: Client as the main subject, the Waiting Queue System as the process block, and the Report as the output block.
- *Top-level whole part diagram*. This diagram defines the domain as part of a big system and parts of the domain. The team used scenarios to identify parts of the system. For instance the scenario of a customer (first part) coming for service identified the Source (second part) where the customer is generated, the server (third part) where it will be served, and finally the sink (fourth part) where the client exits the system. Another scenario of two clients competing for a server identified the queue (the sixth part) where one of the client should wait.
- *Top-level generalization-specialization diagram*. This diagram models the variations in each part of the whole part diagrams. The team developed a generalization-specialization diagram showing the general application and its specialized instance as applications that belong to the domain of our case study.

The outputs of the domain modeling process are:

- *Lower level subject diagrams*. The high level subject and whole-part diagrams are used with aid of scenarios to further develop a lower level subject diagrams. The team developed a second level subject diagram that is composed of Source, Sink, Service Facility, Queue Facility, Queue, Event, Queue Event, Client, and Report.
- *Lower level generalization-specialization diagrams.* These diagrams model the variations for each individual subject of the subject diagrams. Variations are modeled as special types of the general object with names, attributes, and operations that are specific for the application.
- *Object diagrams*. These diagrams capture the abstraction among different types of objects into class diagrams. For example, the team developed a Queue object diagram showing different types of queues as a specialization of the common Queue interface. Similarly, the team abstracted every element in the simulation as a Node (Source, Sink, and Service Facility are all inherited from Node).

3.6.4 Reuse Business

The domain analysis products defined by the Reuse Business team consisted of:

- *Architecture diagrams* showing how the architecture was composed of subsystems and how subsystems were layered. The team used UML package diagrams for this purpose.
- *Use case diagrams* that were used in analyzing the domain and developing the architecture model.

3.7 Reusable Assets

This section summarizes the reusable assets that each team produced. Each team will reuse these components together with the reference architecture to develop one or more applications that we randomly assign. Though these assets are fully understood in the reference architecture context, we mention them here for comparison purposes.

3.7.1 FODA

The FODA team reported the following set of components: a *Simulation Controller* that runs the simulation procedure; a *Source* component that generates inter arrival of customers; a *Queue* component; an *Eventlist* component that holds all simulation events; and a *Server* component.

3.7.2 Synthesis

The Synthesis team reported the following set of reusable components: a *Simulator* component that creates a new Customer and sends it to the Scheduler; a *Scheduler* component that schedules placing customers in the appropriate queue; a *Controller* component that monitors, creates, and deletes Queues and Servers; a *Queue* component; a *Server* component that dequeues and provides services to a customer; a *Customer* component; a *Metric Collector* component that monitors customers events; and an *Input* and an *Output* components to read simulation parameters from the user and print simulation results.

3.7.3 JODA

The JODA team reported the following set of reusable components: a *Node* component which is an abstraction of all model elements that are used in the simulation; a *Source* component that generates customers; a *ServiceFacility* component; a *Sink* component; a *Link* component that connects nodes; a *Queue* component; a *Client* component; and an *Event* component.

3.7.4 Reuse Business

The Reuse Business team reported the following components: *Source*; *Service Facility*; *Queue Facility*; *Sink*; *Measurements*; and *Event List*.

Observations. Whereas different teams developed different components, we can conceptually recognize components that have similar functionalities. The following table compares reusable components produced by each team. The table shows that some conceptual components are similar and some are not.

FODA	Synthesis	JODA	Reuse Business
Source	Simulator	Source	Source
Queue	Queue, Scheduler	Queue	Queue Facility
Server	Server	Service Facility, Sink	Service Facility, Sink
		Node, Link	
EventList		EventQueue	EventList
Simulation Controller	Controller		
	Customer	Client	
	Metric Collector		Measurements
	Input		
	Output		

Table 4 Reusable assets produced from different domain analysis methods

We attribute the differences in the set of reusable assets to either: the approach that each team used, the analysis domain method itself, or the skills of the team members. We attribute some of the above differences to the approach that a team pursued. For example the Synthesis team did not use an event list to handle events instead they continuously check events by incrementing simulation time. Other teams used an event list. We attribute some differences to the analysis method itself; for example the JODA method heavily emphasizes domain experiences and hence the team produced the *Node* and *Link* components based on their experience in simulation of queuing networks. Other differences are attributed to the skills of the analysts. For example the Synthesis team developed an *Input* and an *Output* component to handle inputs and outputs of the simulator respectively. Also, the FODA and Reuse Business teams did not develop a *Customer* component though obviously needed by all applications.

3.8 Technology and Language Dependency

The FODA team reported that the method is heavily based on structure analysis and design (SA/SD). The domain models that the team produced are based on context diagrams, data and control flow graphs, structured charts for module designs, and function and operation models. Therefore we can consider FODA technologically dependent on structured design and programming. FODA does not advocate a specific implementation language. The Synthesis team reported that the method does not advocate a particular technology nor a specific implementation language. JODA is based on the assumption that objects are more understandable and customizable than traditional functions and subroutines. Therefore, the method is heavily based on the object-oriented technology. The notations and models used are the Coad/Yourdon models. These models are now integrated into the UML models. Reuse Business has an object-oriented domain analysis method therefore it is heavily based on OO analysis and design models such as use cases and scenarios. The implementation language is only restricted to be object-oriented.

Observations. The following table summarizes the dependency of the domain analysis methods on technology and programming languages.

	FODA	Synthesis	JODA	Reuse Business
Technology Dependency	Yes	No	Yes	Yes
Language Dependency	No	No	No	No

Table 5 Dependency of domain analysis methods on technology and programming languages

None of the domain analysis methods involved in our case study is programming language dependent though most are technology dependent (object-oriented or structured). Synthesis is the only method that is neither technology nor language dependent; the team selected object-oriented models and languages as well.

3.9 Domain and Application Engineering Effort[1]

The following table summarizes the cost that each team reported in domain analysis, domain implementation

[1] The data collection is currently undergoing and will be completed by end of March 2000.

and testing, and application engineering in terms of person hour. The table shows how the cost of domain analysis and domain implementation pays off in application engineering phase. Some teams have also reported a reuse ration of 70% of the components in the library.

	FODA	Synthesis	JODA	Reuse Business
Domain Analysis and Design	98	131	43	21
Domain Implementation and Testing	96	97		
Application Engineering	~3hrs/ appl.		~4hrs/ appl.	~ 2.5/ appl.

Table 6 Cost of domain analysis and domain implementation in person-hour

4. Summary and Prospects

In this paper we have discussed some preliminary results pertaining to the deployment of four different domain engineering methodologies. We compared the four methodologies using a product line case study for simulating the behavior of waiting queues.

Even though our study is still preliminary, we can discern some distinct trends in the way the different methods approach a given problem. Some methods focus on processes while others focus on products and deliverables; some methods focus on business considerations whereas others focus on technical considerations; some methods propose guidelines whereas others impose standards; also some methods require more detailed deliverables, and it remains to be seen whether more detail is synonymous with better quality or merely with bigger overhead. Interestingly, these differences have a profound influence on the deployment of each method and these differences permeate the whole lifecycle and the deliverables. The comparisons we make between the methodologies must be qualified with three premises:

- First, many of the features we observe on the deployment of the four distinct methodologies on a common application domain stem, not from the methodologies, but from the domain.
- Second, some of the variances we observe between the methodologies stem, not from the different methodologies, but from the different skill levels and interest levels of the student teams.
- Third, some of the features that we observe are dictated solely by the methodology, and do not depend on the example on which the methodology is applied.

5. References

[1] A. Mili, S. Yacoub, E. Addy, and H. Mili, "Towards an Engineering Discipline of Software Reuse," *IEEE Software*, 16(5):22-31, September/October 1999.

[2] W. Tracz, "Confessions of a Used Program Salesperson," Addison Wesley, Reading, MA, 1995.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University www.sei.cmu.edu/publications/documents /90. reports/ 90.tr.021.html

[4] Feature Oriented Domain Analysis (FODA) Bibliography, Software Engineering Institute, Carnegie Mellon University, http://www.sei.cmu.edu/domain-engineering/FODA_bib_ref. html.

[5] R. Holibaugh, "Joint Integrated Avionics Working Group (JIAWG) Object-Oriented Domain Analysis Method (JODA)," Software Engineering Institute, Carnegie Mellon University, CMU/SEI-92-SR-003.

[6] Software Productivity Consortium, "Reuse-driven Software Processes Guidebook, Version: 2.0.3", December 1993.

[7] I. Jacobson, M. Griss, and P. Jonsson, "Software Reuse : Architecture, Process, and Organization for Business Success," Addison Wesley Longman, 1997.

[8] G. Booch, J. Rumbaugh, and I. Jacobson, "The Unified Modeling Language User Guide," ISBN: 0-201-57168-4, Addison Wesley, 1999.

[9] E. Addy, A. Mili, and S. Yacoub, "A Case Study in Software Reuse", accepted for Software Quality Journal, to appear in 2000.

[10] RSP Domain Engineering Guidebook, Reuse-Driven Development Environment (RDE) Product Line Project, June 1996, available at http://source.asset.com/Boeing/rde/rsp/ RSPtoc.htm

Performance Issues of Variability Design for Embedded System Product Lines

Oliver Lewis[1], Mike Mannion[2], William Buchanan[1]
Extended Abstract

[1] School of Computing
Napier University
219 Colinton Road
Edinburgh, UK, EH14 1DJ.
+44 131 455 4432
Lewis@dcs.napier.ac.uk
b.buchanan@dcs.napier.ac.uk

[2] Dept of Computing
Glasgow Caledonian University
Cowcaddens Road
Glasgow, UK, G4 0BA
+44 141 331 3279
m.a.g.mannion@gcal.ac.uk

1 INTRODUCTION

In a software development process model (Figure 1) that is underpinned by reuse new products emerge from the integration of two separate activities:

- domain engineering to create reusable assets.
- systems engineering to build systems using those assets.

In this paper, these two activities are known as product line engineering. Figure 1 shows the relationship between domain engineering and systems engineering.

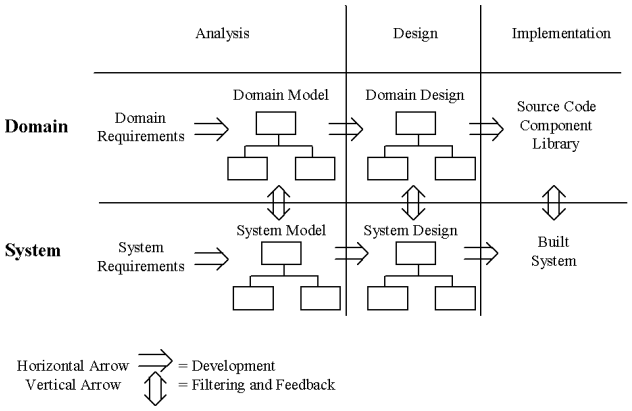


Figure 1. Product Line Engineering Process Model

During domain engineering first identify requirements from the combined requirements of existing systems and from domain expertise. These requirements form the input to domain analysis in which you generate a domain model. The domain model identifies the commonality and variability of existing and future systems. Use the domain model to generate a domain design i.e. a domain architecture and design components. The commonality and variability in the domain model must be reflected in the domain design. During domain implementation the architecture and components are built in the chosen implementation language.

During systems engineering, the analysis and design for a new system are generated from the domain workproducts by filtering out requirements and design solutions that do not trace to the system requirements. Each output from each systems engineering phase may be extended after filtering to include items for which a conscious decision has been made not to include at the domain level. For example, a system requirement that exists as a domain requirement and that was modelled during domain analysis, and designed during domain design, may not have been implemented during domain implementation for commercial reasons.

The feedback arrows in Figure 1 represent decisions to modify the domain workproducts as a result of systems engineering activities.

The degree of difficulty in generating a new system from domain workproducts is dependent on the amount of variability that is built into domain workproducts and the ease with which the workproducts can be customised to meet the needs of the new system. For embedded systems engineers there is an additional concern that the variability built into domain code components, is often achieved at the expense of performance and memory constraints. However, there is little empirical evidence to justify these claims.

Our hypothesis is that: when building a single system from a domain design, there will be a correlation between the number of points of variability in a domain code component which are included in the single system and the space and time overhead that single system will incur.

The focus of our research is to quantify the space and time overhead incurred by a domain code component containing variability mechanisms by comparing its performance to a code component that implements the same functionality but does not contain variability mechanisms.

2 EXPERIMENTAL DESIGN

We are evaluating our hypothesis by comparing the performances of systems having one-off designs with no variability and the same systems having product line designs with variability. Experiments to this effect are

currently in progress and results will be available for the workshop.

We have selected six common interface techniques based on the components of six popular architectural styles[1] (Shaw et al [1]). For each architectural style we have selected a case-study product line that would typically be designed using that style. For example the *pipe and filter* style will be used for a product line of feedback control and sampling systems for liquid-level regulators.

To attain the measurement for the space and time overhead for a particular architectural style, we must implement two systems. Each system pair will consist of two functionally identical systems. One system is built using a traditional single system development approach not incorporating variability and common interface constructs; the other is engineered from a product line and does incorporate variability. We will compare the performance of the two systems to attain a measurement of overhead.

For each case-study product line we have chosen approximately ten points of variability. For example, in a product line of liquid-level regulators the points of variability include different types of control loops, and different types of input and output devices.

To discover if there is a correlation between space and time overhead and the number of points of variability selected into a single system, we must compare the space and time overhead of systems drawn from a product line with different numbers of points of variability to their single system counterparts.

We propose that the following measurements be taken for each architectural style:

- one measurement of space overhead between a system implementing the minimum amount of variability (usually 1 point of variability) and its one-off counterpart.
- one measurement of space overhead between a system implementing the maximum amount of variability (~10 points of variability) and its one-off counterpart.
- at least two random intermediate readings of space overhead between minimum and maximum number of points of variability.
- at least one duplicate measurement of space overhead containing the same number of points of variability as an existing measurement but implementing different functionality.
- one measurement of time overhead between a system implementing the minimum amount of variability (usually 1 point of variability) and its one-off counterpart.
- one measurement of time overhead between a system implementing the maximum amount of variability (~10 points of variability) and its one-off counterpart.
- at least two random intermediate readings of time overhead between minimum and maximum number of points of variability.
- at least one duplicate measurement of time overhead containing the same number of points of variability as an existing measurement but implementing different functionality.

This set of 10 experiments is then repeated for each of the six architectural styles providing 60 measured results. The format of expected results is shown in Table 1.

	Pipe & Filter Time Overhead	**Pipe & Filter** Space Overhead	…	**OO Style** Space Overhead
Minimum	T_{11} ms	S_{11} Bytes	…	S_{61} Bytes
Intermediate$_1$	T_{12} ms	S_{12} Bytes	…	S_{62} Bytes
Intermediate$_2$	T_{13} ms	S_{13} Bytes	…	S_{63} Bytes
Duplicate	T_{14} ms	S_{14} Bytes	…	S_{64} Bytes
Maximum	T_{15} ms	S_{15} Bytes	…	S_{65} Bytes

Table 1. Format of Results

To help evaluate the space and time overhead, each pair of systems shall be run from within a test harness. The test harness shall invoke a system, emulate its I/O, time its execution and measure its memory usage.

The system clock shall be used to measure the execution time. To eliminate any inaccuracies due to the differences in resolution of the unit of measurement and the system clock, we must run a number of cumulative experiment cycles. For example, the unit of measurement of the clock may be 1/100 second. A typical cycle time may be of the order of 1/1000 second. Sufficient measurement cycles must be taken to eliminate inaccuracies (10000 cycles was more than enough in our experiments in [2]).

In many real-time experiments that measure execution time, there will be slight differences in the absolute values recorded. These can be attributed to background tasks being performed by the operating system. To reduce the effect of these differences an average value from multiple test-runs shall be taken (e.g. 100 test runs of 10000 cycles).

Once our experiments are complete, we can analyse the values of the data in Table 1 to discover any correlation between the overhead and the number of implemented points of variability. Plotting graphs of space and then time overhead against the number of points of variability can show the gradient of correlation. Figure 2 shows the format of the graph for the time overhead for different numbers of points of variability for the Pipe and Filter architectural style.

[1] Architectural Styles: Pipes and Filters, Implicit Invocation, Layered Systems, Repositories, Distributed Processes, Main/Subroutine Organisations, Object-Oriented Organisation.

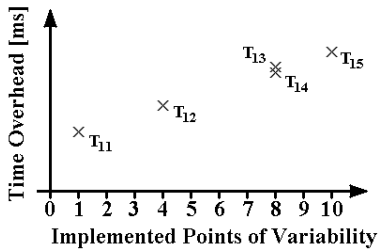


Figure 2. Example Graph for Time Overhead of Pipe & Filter Style

To confirm a correlation we can first estimate the best-fitting equation for the measured points. In circumstances where the plot shows some linearity, a linear regression can be used for the estimation. Situations where the results do not correlate approximately linearly, may require further analysis may including taking a log of the values to make them more linear, bootstrapping or even taking more measurements.

Consider fitting N data points (x_i, y_i) to a straight-line $y = a + bx$. In our case N corresponds to the number of measurements for overhead, x_i is the number of points of variability and y_i is the measured overhead. Equation 1 and Equation 2 show how we can calculate the coefficients (a and b) for the best fitting straight-line equation using a linear regression (taken from Burden et al [3]).

$$a = \frac{\left(\sum_{i=1}^{N} x_i^2\right)\left(\sum_{i=1}^{N} y_i\right) - \left(\sum_{i=1}^{N} x_i\right)\left(\sum_{i=1}^{N} x_i y_i\right)}{N\left(\sum_{i=1}^{N} x_i^2\right) - \left(\sum_{i=1}^{N} x_i\right)^2}$$

Equation 1. Finding coefficient a

$$b = \frac{N\left(\sum_{i=1}^{N} x_i y_i\right) - \left(\sum_{i=1}^{N} x_i\right)\left(\sum_{i=1}^{N} y_i\right)}{N\left(\sum_{i=1}^{N} x_i^2\right) - \left(\sum_{i=1}^{N} x_i\right)^2}$$

Equation 2. Finding coefficient b

To establish more confidence in the gradient a of the straight-line equation as an indication of the degree of correlation between any overhead and the number of implemented points of variability, we can calculate a residual error.

$$R^2 = \frac{\sum_{i=1}^{N}\left(\frac{x(y_i) - x_i}{x(y_i)}\right)^2}{N}$$

Equation 3. Residual error

The smaller the residual error is, the greater the likelihood that our measured data lie on the estimated straight-line equation. If we can show with a degree of confidence, that our measured points can be modelled using the equation, then we can use the gradient of that equation to indicate the trend of the overhead. A positive value for the gradient a will indicate (to the certainty of the residual error) that a correlation exists between space and time overhead and the number of implemented points of variability. A residual error of less than 0.05 is accepted as an indication that an equation correlates exactly to the points it was derived from (James [4]). As we are just trying to indicate a trend, rather than find an exact equation, a residual error that is less than 0.2 is acceptable. Equation 3 shows how to calculate the residual R^2.

3 CONCLUSIONS

The aim of our research is to quantify the space and time overhead that may be introduced when variability is incorporated into a single system. In this paper we have described a set of experiments which we believe will contribute to that problem. This data can inform embedded systems engineers about the behaviour of overhead they might expect in a single system solution built from a product line model.

REFERENCES

[1] M.Shaw, D. Garlan. Software Architecture, Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River, New Jersey, USA, 1996, ISBN 0-13-182957-2.

[2] O. Lewis, M. Mannion, B. Keepence. Performance Concerns of Polymorphism in Modelling Domain Variability in Real-time Systems. IEEE Symposium on Engineering of Computer-Based Systems. Nashville, Tennessee, USA, Pages 240-246, March 1999.

[3] R. L. Burden, J. D. Faires. Numerical Analysis. Fifth Edition. PWS Publishing Company, 20 Park Plaza, Boston, USA, 1993, ISBN 0-534-93219-3.

[4] G. James. Modern Engineering Mathematics. Addison-Wesley, Reading, Massachusetts, USA, 1992, ISBN 0-201-18054-5.

Athena: A Software Product Line Architecture for Meter Data Processing and Control

Daniel J. Paulish
Michael L. Greenberg
Siemens Corporate Research, Inc.
755 College Road East
Princeton, NJ 08540 USA
+1 609 734 6579
dpaulish@scr.siemens.com

ABSTRACT
One of the few known certainties when embarking upon the design of a new software system product line architecture is that the design and its implementation will likely change over time as market requirements, technologies, hardware, and business factors change. Some of these influencing factors impact the entire system, and some directly contradict other factors. In order to avoid major potential rework, these factors must be addressed from the beginning of high-level design. This paper describes our experience with applying the technique of global analysis to plan better software projects by designing product line architectures that anticipate change. The purpose of global analysis is to analyze the factors that influence the architecture and to develop strategies for accommodating these factors in the architecture design. These influencing factors fall within three categories: organizational, technological, and product. The paper describes our experience with applying global analysis to the design of a meter data processing and control central station platform.

Keywords
Software architecture, high-level design, web-based GUI, Internet, product line architecture.

1 BACKGROUND

In 1999, we were asked to contribute to the design of a software architecture (code-named Athena) platform for meter data acquisition and processing central stations. A platform-based product line architecture was required since multiple application packages were envisioned in order to support the future business needs. In addition, existing product architectures were being evaluated for supporting the envisioned application packages.

Historically, these existing product platforms were often modified and tailored towards customer specific requirements within project engineering centers located throughout the world. As a result, proliferation of platforms and products was occurring so that it was difficult to bring new features back into the baseline product. Furthermore, there was a strong trend towards power distribution industry deregulation that was believed would cause requirements discontinuities as well as new business opportunities.

A meter data processing central station collects meter data via telephone lines from electric, gas, and water meters. The meter data is stored and processed. The type of processing depends on the type of consumer using the resource and its contractual agreement with the supplier of the resource. Thus, many different types of application software must run on the Athena platform. For example, the processing software for commercial consumers using electricity would be much different than residential water consumers. Furthermore, control functions are provided, for example, commands are sent out to high power utilization equipment when load must be shed during high demand periods. Tariff agreements are specified between energy consumers and providers such that the price of energy varies depending on the time of day, week, and year. Athena performs calculations on the meter data collected that is typically processed and sent to a utility's billing system.

Five initial application packages were planned for eventual implementation on the Athena platform. These ranged from meter data acquisition through billing determinant calculation to load management control and payment systems implementation. The application requirements were quite diverse and imposed a high degree of flexibility

upon the product line architecture design. The first two applications were directly supported by the architecture since their development was initiated shortly after the architecture was designed and reviewed. At the time of the architecture design, marketing requirements specifications existed for the first two applications but not for the other three applications.

A high-level design team was formed consisting of five engineers with a mixture of domain and architecture design expertise. A chief architect was appointed and the team began the architecture design with analyzing the marketing requirements, developing the conceptual architecture, investigating applicable development technologies, and some simple prototyping.

2 GLOBAL ANALYSIS

Early on during the high-level design, a global analysis [1] was completed for Athena. The global analysis considered factors that would influence the design grouped into categories of organizational, technological, and product influences. Analysis of the influencing factors resulted in a set of design strategies that have been used to guide both the product line architecture design and implementation of the application packages.

Organizational Influencing Factors

Organizational factors such as schedule and budget apply only to the product currently being designed. Other organizational factors such as organizational attitudes, culture, development site(s) location, and software development process can impact every product developed by an organization.

An example of an organizational influencing factor for Athena was that the technical skills necessary to implement the application packages were in short supply since prior products had been Unix-based with local user interfaces and marketing required new products to be Windows-based with web-based user interfaces. The resulting strategy to address this influence was to bootstrap and exploit expertise located at multiple development sites and to invest in training courses early in the development. Also, a second level of design specification documentation was developed at a lower level than the high-level design. This system design specification concentrated on describing the interfaces between major subsystems of the architecture, so that it was easier to parcel out a subsystem development to a remote software engineering site.

Another organizational factor was that company management wanted to get the product to the market as quickly as possible. Since the market was rapidly changing, it was viewed as critical to quickly get some limited features of the product to potential users so that their feedback could be solicited. Our strategy to address this factor was to develop the product incrementally such that scheduled release dates were met even if some features were missing from the release. Thus for Athena, project schedule took priority over functionality. A build plan was developed for each engineering release identifying the sequence for adding functionality. The project functionality and schedule were baselined after each engineering release. We found that a 6-8 week development cycle for each engineering release worked well for the development team to provide a reasonable set of features that could be tested and evaluated.

Technological Influencing Factors

Technological factors limit design choices by the hardware, software, architecture technology, and standards that are currently available. But technology changes over time and products must adapt, so the architecture should be designed with flexibility in mind.

An example of a technological influencing factor was that a distributed object broker was necessary for meeting the scalability and availability requirements within a distributed hardware configuration. The strategy selected to address this factor was to use Microsoft COM throughout the system development.

Another technological factor was that we knew that our database system would change over time. Marketing specified that Oracle 8 be used for Athena. But, we knew that new database versions would become available and that certain customers would likely prefer vendors other than Oracle. Thus, we designed a layer in the architecture so that we could isolate and encapsulate the database for anticipating that these requirements would change in the future.

Product Influencing Factors

Product factors include features of a product as well as qualities like performance, dependability, security, and cost. These factors may be different in future versions of the product, so the architecture should be designed to support the anticipated changes.

An example of a product influencing factor was that to support a product line architecture, the Athena GUI must be able to accommodate many different types of users for different applications. The strategy selected to address this factor was to implement the GUI as a web-based GUI, so that additional flexibility could be achieved as new applications are added and location independence could be achieved for the various user populations.

Another factor was the anticipated performance of the system. Athena was intended for industrial and commercial applications where thousands of meters would be handled. It was never specified to address market requirements where meter data for millions of consumers would be required. However, we knew that a scalable distributed platform would be necessary to meet these potential unknown market performance requirements. Furthermore, the primary purpose of Athena is to perform

the calculations on meter data before they are sent to a billing system. Again, we anticipated that a scalable distributed platform was necessary to meet unknown calculation time requirements.

As is the case for any product line architecture, but especially one that will be sold into a changing market resulting from deregulation, a high degree of flexibility is necessary as an overall design goal in order to have a chance at anticipating meeting new unspecified requirements. Unfortunately, this means that the architecture may be considered "overdesigned" for the simpler nearer term applications, but the flexibility will be necessary to extend the life of the product line. Within an organization with limited software development resources, application packages will be mainly developed sequentially and thus the product line architecture will necessarily be required to live for many years.

3 PRODUCT LINE DESIGN STRATEGIES

Design strategies determine the priorities and constraints of the architecture and help identify potential risks associated with the implementation of the software system. As a result of the Athena global analysis, 24 design strategies were identified that we believed could address the influencing factors. From these 24 design strategies, six major conclusions were derived and used as guiding principles for the Athena architecture design and resulting development.

These summary product line design strategies are:

- Reuse the current data acquisition system. The architecture design shows the new data processing system loosely coupled to the existing acquisition system with well-defined interfaces and separate data storage (Figure 1). This saves development time, since code for handling communication protocols and meter setup need not be redeveloped or ported. Basic system utilities such as the message/alarming system are used by all subsystems.
- Reuse 3rd party software wherever possible. A design strategy was followed to attempt to use 3rd party tools whenever possible. Furthermore, most of these tools came from Microsoft that helped reduce tool selection decision and training time. An innovative approach was implemented to use Excel as a computation engine.
- Replace the functionality of the current product with the new product. The current product's functionality was used to determine the basic requirements and features of the new product. This helped simplify requirements definition and testing. Innovative new features were added to the new product to distinguish it from the current product.
- Design Athena as a software product. New software technologies and business models were evolving at the time we started designing Athena. By developing a purely software product, newly emerging business models (e.g., ecommerce, application service providers) could be investigated and potentially offered as solutions to a deregulated market with new requirements.
- Web-based GUI. All interactions with Athena are designed to be performed using a web-based GUI. This has been implemented within a three-tiered architecture. GUI development effort and the process of incremental releases are simplified. From a marketing point of view, the web-based GUI enables location independent access to the system and a high degree of flexibility to network and scale the system using intranets or the Internet. The three-tiered architecture provides a structure for adding future applications as business objects.
- Multisite development. The lack of sufficient technical skills within a single location was an influencing factor that was addressed by setting up a multisite development at four sites within three countries. This put constraints on the design so that components could be more easily distributed for development at multiple locations, and the development environment and tooling was set up for multiple locations.

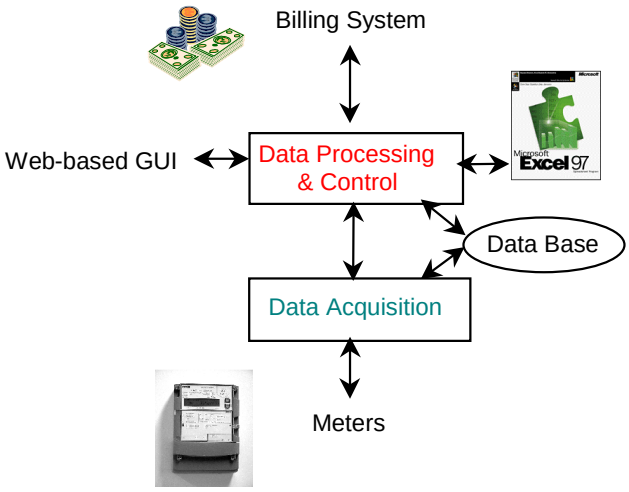


Figure 1. Athena conceptual architecture.

4 RESULTING ARCHITECTURE

The Athena high-level software architecture was designed using our four views approach - conceptual, module, execution, and code [1]. This design approach decreases the complexity of the implementation and improves understanding, reuse, and reconfiguration (Figure 2). The architecture was reviewed using the Architecture Tradeoff Analysis Method (ATAM) [2] which is a structured analysis technique that evaluates a software architecture with respect to multiple desired qualities (e.g., survivability, modifiability, performance, security), developed at the Software Engineering Institute (SEI).

We have also analyzed the architecture to provide inputs to the project's development cost and schedule estimation [3].

With our architecture-centered software project planning approach (ACSPP) (Figure 3), the software architecture design document is a primary input to the top-down and bottom-up project schedule and effort planning processes. From this an incremental development build plan is generated such that the product functionality is built up feature by feature within engineering releases that are system tested until the functionality and quality are adequate for beta testing with prospective customers.

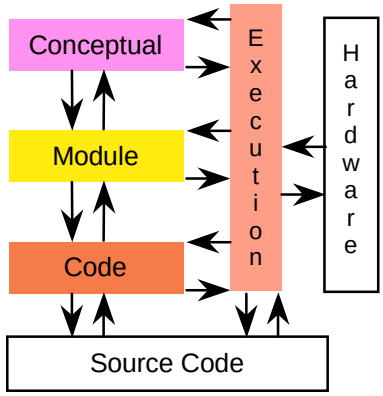


Figure 2. Four views of software architecture.

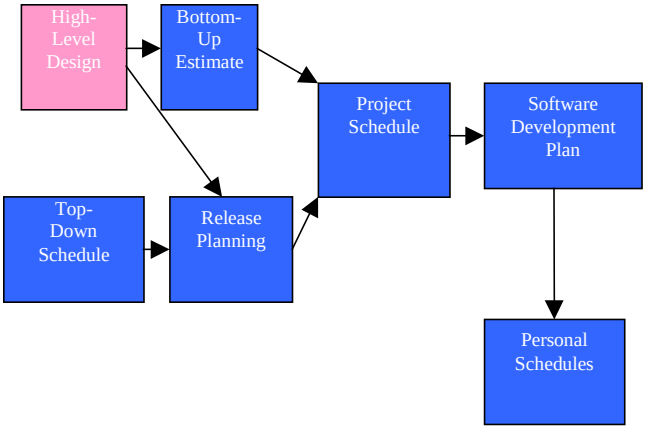


Figure 3. Architecture-centered software project planning.

The software architecture of Athena is based on a three-tiered model (user interface tier, business logic tier, and database tier), such that new metering applications can be easily added in the future at the middle or business logic tier (Figure 4). The user interface tier consists of a set of web pages, a web server, and a web browser for interaction with the user. The business logic tier is a group of subsystems that defines the business logic. The database tier contains the database interface, database tables, and database procedures. The business logic subsystems use the database interface or call database procedures to obtain and manipulate data in the database tables.

Customer accounts are managed through the consumer tree subsystem design, where the relationships among master accounts, accounts, contracts, and consumers are described within a tree structure (Figure 5). Each node in the tree contains active elements such as meter proxies, calculations, reports, and tariff agreements. Scheduled events are maintained at each node such that a daily schedule is automatically generated and loaded to the meter data acquisition subsystem from the consumer tree subsystem.

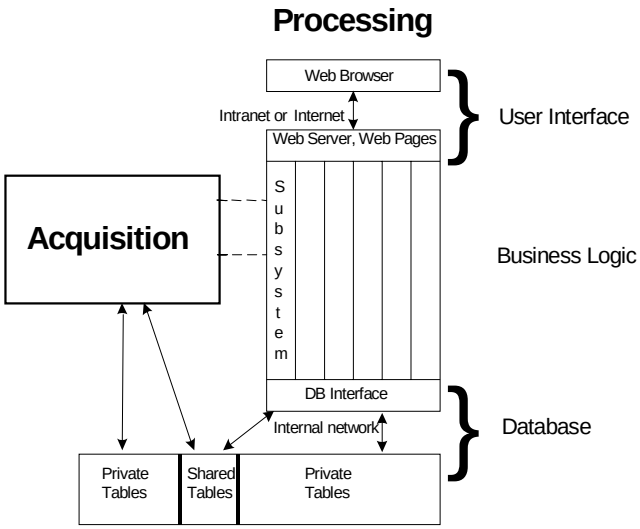


Figure 4. Three-tiered architecture.

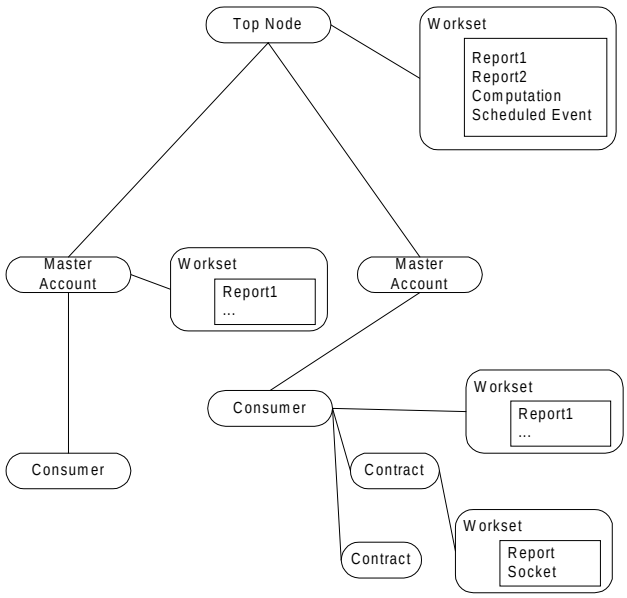


Figure 5. Consumer tree.

The web-based GUI (top tier) provides marketing, implementation, and cost advantages. User capability is simplified and empowered through the use of templates. Operators of Athena can manage the system (depending on their security profile) from a web browser on any client computer connected to the Internet or their intranet. The architecture is designed for scalability since multiple users can access the system simultaneously, with customer data segmentation and checkin/checkout capabilities for avoiding conflicts.

Microsoft's Excel is used as a general-purpose load profile computation engine. This provides both power and ease of use since most users are already experienced with using Excel spreadsheets for calculations. The relationships among meter proxies, calculations, tariff agreements, and reports are managed using a mapping description notation that allows simple to complex calculation schemes for generating load profiles and billing determinants.

5 LESSONS LEARNED

At the current point of development, we are positive about our experience with implementing the first two application packages using the Athena software product line architecture. The development of a system design specification was viewed by some team members as an unnecessary step that delayed the start of the application packages implementation. However, new development team members working on both the current and new application packages have successfully used this specification. The system design specification has been modified and updated as detailed designs were completed and requirements evolved. It has been critical for partitioning work packages across the four development sites located within Europe and the U.S. We have observed that integration of the various subsystems has gone remarkably smoothly when the subsystem leaders are brought together in one location.

We've had good experiences with our approach to incremental development. By publishing the URL for the test system, all team members and their management can watch the progress of the development as new features are continually added. This was a big morale boost for the team, since everyone was aware of the rapid progress that was being made after the high-level design phase was completed and development began. The first engineering release was an implementation of a vertical slice through the architecture. This helped validate the architecture and gave the development team the confidence and understanding of the architecture to be able to implement the future engineering releases.

Since we put priority on meeting scheduled release dates and traded off functionality and quality as necessary, the development team successfully achieved every release date. This helped build up the credibility of the development team with management, since they knew that a new set of functionality would be ready for validation testing by the dates that were planned and committed to at the beginning of the baselined engineering release cycle. Fortunately, our quality remained relatively high throughout the development, so the tradeoff between meeting schedule and quality was never seriously challenged.

In the beginning, project meetings were held monthly rotating among the development sites. Currently meetings are held mainly for major subsystem integrations or when training and/or detailed design work is necessary to get someone started on a new development task. We have weekly teleconference meetings to track schedule status and to bring up common problems that the developers should be aware of. In addition to subsystem leaders, we have a chief architect responsible for overall technical decisions.

Despite our best efforts at communicating among the four development sites and our emphasis on design documentation with well-defined interfaces, multisite development is clearly more difficult than single site development. This is a result of occasional miscommunications that are caused by different vacations and holidays in the three countries, time zone differences, and occasional network or computer outages. For example, if questions arise for colleagues in Europe during their evening hours while the U.S.-based teams are working, they likely will need to wait until the next day before they can be resolved. To compensate for the unexpected, team members often use the home telephone numbers of their colleagues in the other countries, and the system is rebuilt almost every day in multiple locations using the latest checked in source code. We have also invested in team building and multicultural training for the development team members.

The Athena software product line architecture is designed to be very flexible and expandable to handle a wide variety of applications. This is a primary design requirement, since the power distribution industry is rapidly changing as a result of worldwide deregulation.

ACKNOWLEDGEMENTS

We wish to acknowledge the contribution of the other members of the Athena high-level design team, namely Bill Sherman, Paul Bruschi, Henk LaRoi, and Sascha Lukic and the support of our management team, namely Uli Syre, Peter Hess, Michael Sommer, Tom Murphy, and Ali Inan.

REFERENCES

1. Hofmeister, C., Nord, R., and Soni, D., *Applied Software Architecture*, Addison-Wesley, 2000.

2. Kazman, R., Barbacci, M., Klein, M., Carriere, S., and Woods, S., "Experience with Performing Architecture Tradeoff Analysis", *Proceedings of the 21st International Conference on Software Engineering*, New York, ACM Press, 1999, 54-63.

3. Paulish, D., Nord, R., and Soni, D., "Experience with Architecture-Centered Software Project Planning", *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, New York, ACM Press, 1996, 126-129.

Applied technology for designing a PL architecture of a pilot training sytem

W. El Kaim, S. Cherki
LCAT
Thomson-CSF/LCR
Domaine de Corbeville
91404 Orsay cedex, France
{elkaim,cherki}@lcr.thomson-csf.com

P. Josset, F. Paris, J.-C. Ollagnon
LCAT
Thomson-CSF/TT&S
1, rue du Général de Gaulle
ZA Les Beaux Soleils - BP 226 Osny
95523 Cergy Pontoise cedex, France
{josset,paris,ollagnon}@tts.thomson-csf.com

ABSTRACT
This paper reports on the product-line experiment leaded by Thomson-CSF in the scope of the ongoing PRAISE Esprit project. It focuses on the design and representation of a product-line architecture with UML. The product-line experiment addresses the simulation for ground vehicle pilot training domain. Lessons learned are drawn from this experiment on process, modeling techniques, and UML notation to represent product-line architectures.

Keywords
Product-line approach, software architecture, product-line architecture, UML.

1 INTRODUCTION

Domain-specific and architecture-centric product-line is now recognized as a promising approach to deal with software intensive systems development and evolution [7]. However, industrial companies lack today methodological framework and supporting tools to make a software development product-line approach realizable in industrial settings. The ongoing European PRAISE project[1] is therefore pursued by Thomson-CSF, Robert Bosch GmbH, Ericsson, and the European Software Institute to provide an integrated and validated methodological support to product-line approach [20].

PRAISE project is divided into five work packages (see Figure 1). Work packages 1 and 2 are dedicated to the definition of the baseline technology and have resulted in process, benefit assessment model, and requirements traceability and architecture methods. In the scope of work package 3, Robert Bosch GmbH and Thomson-CSF are currently leading real large-scale industrial experiments in a coordinate way to validate and consolidate previous technology.

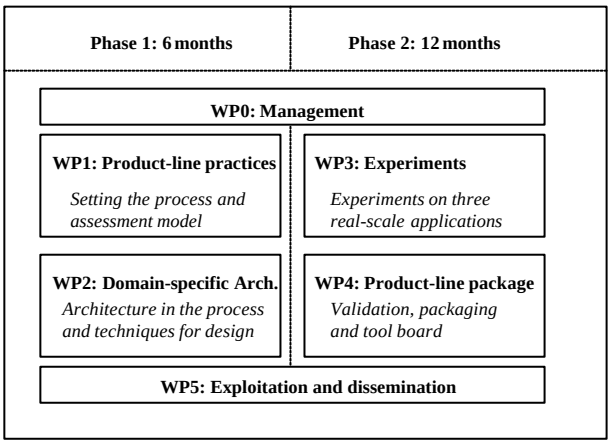


Figure 1: PRAISE project in work packages.

More precisely, the focus is made on domain engineering [4] in both experiments and the same models have been selected from the baseline technology for experimentation. Moreover, a different domain is investigated in each company to get more sound and credible lessons learned. The domain addressed in Robert Bosch GmbH experiment is car periphery supervision whereas the one addressed by Thomson-CSF experiment is simulation for ground vehicles pilot training. Considering that domain data are present in quite different forms in each domain, technology is customized accordingly in each experiment, thus providing results covering a broader area of problems. This paper reports specifically on the product-line experiment

[1] PRAISE project is partly founded by the European Commission under ESPRIT project 28651.

leaded by Thomson-CSF.

In practice, technology related to product-line architecture design and representation has been already proposed [15] and is currently being experimented. It has included identification of architectural styles and views relevant to the product-line domain, the domain architecture modeling, the specification of patterns and the establishment of traceability between the domain architecture model and the domain requirements model. Lessons learned can thus be provided in terms of process, modeling, notation and tools used to perform these activities following a product-line approach. This will be the focus of this paper.

This paper is divided into five sections. In section 2, we describe the Thomson-CSF team organization put in place for handling this experiment. In section 3, we present the domain chosen for the experimentation, the ground vehicle pilot training system. In section 4, the process of the product-line architecture building experiment is described. In sections 4 and 5, experiments leaded on software architecture reengineering and product-line architecture modeling, are respectively described in terms of their process, modeling results and lessons learned. And finally, the last section gathers the whole lessons learned and gives the experiment perspectives envisaged in the future.

2 THOMSON-CSF EXPERIMENT ORGANISATION

PRAISE project is running inside the LCAT, a common research laboratory of Alcatel and Thomson-CSF [7]. LCAT unit working on product-line technology is divided into one technical team and several domain experiment teams. One domain experiment team is composed of experts and engineers from an industrial business unit of Thomson-CSF or Alcatel and an experiment leader from LCAT. This latter organizes experiment work and relates the business unit needs and results to LCAT technical team technology. Thomson Training and Simulation (TT&S) is the business unit involved in the PRAISE project.

TT&S is a pilot business unit in Thomson-CSF regarding software engineering practices and has reached level 3 of the CMM [3]. In order to start up the LCAT project, a preliminary product-line experiment has been leaded by Thomson-CSF/LCR in collaboration with TT&S. It has involved two persons at full time and one TT&S expert at 20% for one year. Following a full bottom-up approach of domain engineering, it has allowed us to get domain data and evaluate human and time resources needed in LCAT project domain experiments.

3 DOMAIN DESCRIPTION

The domain description was realized during the product-line scoping phase, as described in [23]. As input of this phase, we have considered the **ground vehicle pilot training domain**.

In practice, the product-line domain has been first delimited in terms of:

- a short informal description of the domain addressed,
- a list of systems included,
- and a list of coarse-grained variability.

The related activity has been realized fast and merely with coarse-grained selections performed by business unit experts, not requiring any analysis of the domain.

The selections performed on this input domain to define the product-line domain are mainly motivated by making the product-line domain fit TT&S organization with regards to market needs and development teams. For example, focus has been made on some ground vehicles both civilian and military, thus allowing TT&S both to extend its business to a civilian market and to leave some domain features no more required in military market. Also, TT&S development teams being built according to specific competencies, focus was made on real-time simulation and environment simulation competencies.

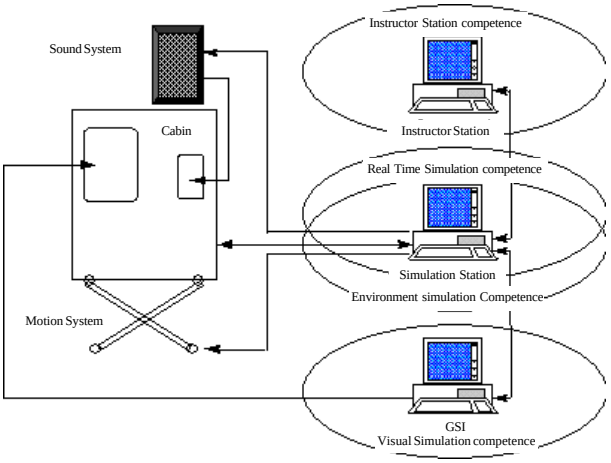


Figure 2: TT&S competencies related to ground vehicles pilot training domain

Figure 2 illustrates the main hardware elements of a ground vehicle pilot training system, with arrows denoting communications between these elements and ovals denoting TT&S software teams organized by competencies. In TT&S systems, real-time simulation and environment simulation competencies correspond to a software component (designated later with SIM) of the system; previous selections defining the product-line domain therefore leaded us to consider not a whole system as the product, but rather SIM software component of the system.

Several systems have been then selected by TT&S expert as relevant to the product-line domain thus delimited: three

legacy ones and two future ones to be developed.

Finally, a list of coarse-grained variability has been selected by TT&S expert, considering coarse-grained evolution of existing systems and anticipating future systems one: this list includes variability related to trainee vehicle model which may be a tank model or a truck model, and variability as optional ability to test the availability of SIM devices at run-time.

These selections have allowed TT&S to favor product-line process adoption, getting rid as much as possible of its market opportunities and its business competencies. They have also allowed product-line experiment practitioners to deal with a "reasonable" number of domain data and variation points during product-line experiment and to access easily domain data.

4 PRODUCT-LINE ARCHITECTURE DESIGN PROCESS

According to [4], a domain engineering process phase named domain design includes the product-line architecture design (see Figure 3). In this paper, a product-line architecture is "*a generic architecture that applies to a set of products grouped into a product-line and from which the software architecture of each product can be derived*" and a software architecture is "*a collection of subsystems and the relationships between them. These subsystems are built using various stakeholders' points of view and must be related to stakeholders' requirements expressing rationales*". The latter definition is inspired from [2], [9], [19], and [22].

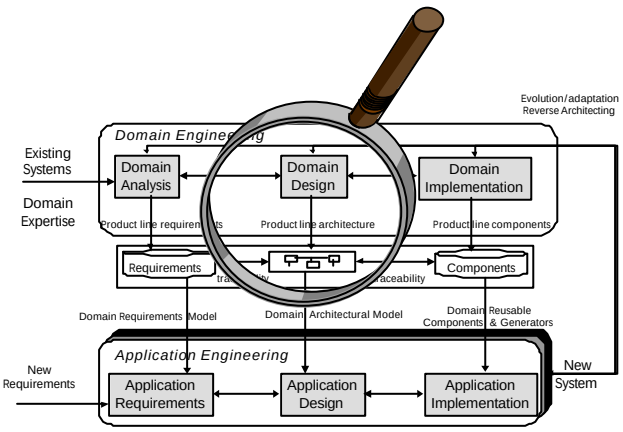


Figure 3: Product-line process extracted from [4].

The product-line architecture design process has been defined in terms of activities and tasks, represented in Figure 4 activities are modeled as rectangles with round corners, tasks as items, data flows as arrows, activity practitioners as faces, and activities inputs as databases.

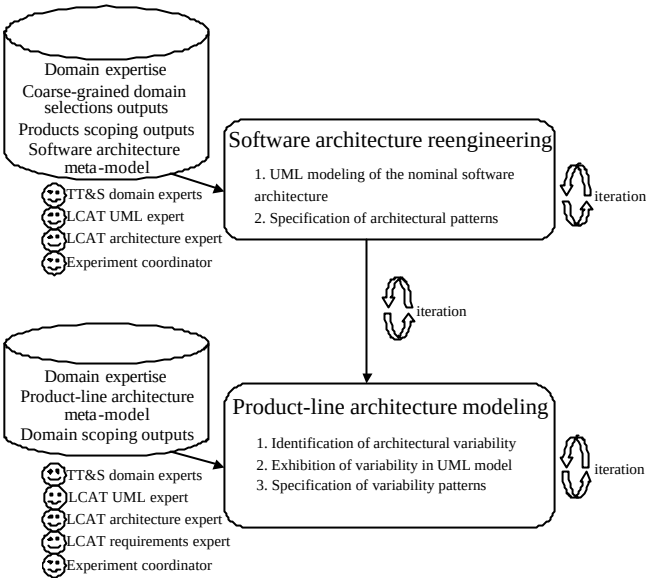


Figure 4: Product-line architecture design process in Thomson-CSF experiment.

Some of the activities inputs (represented on databases in Figure 4), as coarse-grained domain selections outputs, products scoping outputs, and domain scoping outputs, have been produced during a previous phase of our product-line experiment called product-line scoping. Product-line scoping phase is part of domain analysis phase in the product-line process (see Figure 3). Product-line scoping activities are:

- Coarse-grained domain selections. It produces an informal definition of the domain in terms of a textual description of the domain delimited, a list of legacy and future products to integrate in the product-line and a list of coarse-grained variability to handle within the product-line.
- Products scoping. It produces tables matching characteristics of the products integrated in the product-line, including development referential, requirements, legacy architectures, technology and terminology.
- Domain scoping. It produces a domain context model, a domain requirements model and their traceability.

In the following section, we describe each task of the software architecture reengineering and the lessons learned. The same is done for product line architecture modeling activity in section 5.

5 SOFTWARE ARCHITECTURE REENGINEERING

Software architecture reengineering consists in organizing

domain data related to software architecture in order to favor architectural commonality and variability extraction. Indeed, domain data are organized according to a kind of V cycle development in legacy products using a data model. We strongly believe that object technology can be used to design software architectures [17], and furthermore product-line architectures [13], [15], [16]. Reengineering existing domain data with object technology has been therefore a natural step before dealing with architectural variability.

Tasks performed during software architecture reengineering are UML modeling of the nominal software architecture, and specification of architectural patterns. They are presented in the following subsections.

5.1 UML Modeling of the Nominal Software Architecture

This task has two main goals:

1. Formalize an object model of the nominal software architecture. **A nominal software architecture is the software architecture of the most representative product in the product-line domain.**
2. Test the adequacy of UML with regards to software architecture representation in the product-line approach.

Indeed, a product-line project previously leaded on the same domain has pointed out that the software architecture is actually expressed in a specific textual form. It is thus **not easily understandable by a non TT&S business expert** and **almost impossible to be managed automatically in a reuse approach** since it is only documentation. Moreover, an important objective of PRAISE project is to pay special attention to the validation of UML as standard notation to represent software architecture and furthermore product-line architecture.

In practice, architectural components and their relationships have been first selected from existing products to identify the nominal software architecture. Then, they have been modeled with UML in Rationale Rose™ 4.0. The UML diagrams produced have allowed to highlight the services provided by architectural components to each others. It has been however **necessary to structure them because they contain many domain data coming from different levels of the product life-cycle**. To do it, we have defined a model of architectural views, after having looked at those proposed in the literature.

5.1.1 Related work on architectural views

Works have been done in software architectural views and the way to relate them. ITU-T with the Open Distributed Processing initiative defined several viewpoints [12]. ODP viewpoints are not offering coherency between views, and are difficult to use because the views are too abstract.

The "4+1" view model from [14] is a little bit too conceptual and sometimes difficult to apply at the architectural level. Nevertheless, it proposes a very attractive contribution: relating views using a coherency mechanism based on scenarios.

Gacek [9] proposes a model with several views, tempting to further separate concerns in function of stakeholders' identified needs. The drawback of this model relies on the fact that assets are described at different abstraction levels in the chosen views.

Siemens [11] proposes views based on descriptions which are much code oriented and describe assets at a low level of abstraction.

5.1.2 Thomson-CSF selected architectural views

Matching the nominal software architecture of our domain to the advantages and liabilities of each architectural views model previously described, we have defined our own model of architectural views [6]. It is composed of:

1. **Business view**. Real world business entities and their relationships. The real world business entities are grouped by subject area which relationships are collaborations. This view extends and refines, if necessary, the information model obtained at the end of the analysis phase. This view includes business collaborations, business context, and business patterns.
2. **Subsystem view**. Design entities and their relationships. The upper design entities are grouped by subsystem which relationships are collaborations. This view includes subsystem collaborations, process, interfaces, information, and subsystem patterns.
3. **Technology view**. Deployment entities and their relationships. This view includes deployment on the infrastructure of the application and technology context.

This architectural views model has been represented with UML notation and implemented in Rational Rose™ 98i. It has been used then to structure the nominal software architecture of our product-line.

UML notation supports well this views model through the use of UML extension mechanisms like stereotypes. The number and semantics of stereotypes additionally defined by a UML user is not constrained in UML. This can lead to the definition of too many new stereotypes not clearly documented, which makes a UML model no more readable and sometimes no more coherent. **It is therefore important to control the definition of new stereotypes to keep coherency and readability of the UML model.**

Rational Rose™ 98i supports globally the implementation of our views model. It has however several liabilities: **it is model oriented and not view**

oriented (it imposes the use of "4+1" view model from [14]), **it does not fully comply with the UML standard 1.3** (UML collaborations and subsystems are not implemented) **and it is workstation oriented** (for example, sharing Rose scripts requires local initialization file modifications).

5.1.3 Thomson-CSF selected architectural views examples

In the ground vehicles pilot training domain, several pilot training systems have already been developed or are to be developed in TT&S. These systems are composed of hardware and software elements. The product of our product-line is then a software element named SIM of the whole pilot training system.

To describe locally important architectural elements we have systematized the use of architectural collaborations. In an architectural collaboration, each architectural element plays a specific role in the collaboration. For example, Figure 5 illustrates a UML collaboration pertaining to the business view of the SIM nominal architecture. The *Exercise* subject area plays the role of exercise execution manager and calculator, including subject areas like *trainee pilot automatic assessment* and *trainee guidance*. The *Simulation* subject area plays the role of trainee environment parameters calculator, including subject areas like *trainee vehicle dynamics* and *trainee vehicle movement*. The *System management* subject area plays the role of system components communication manager, including subject areas like *motion hardware platform interface*.

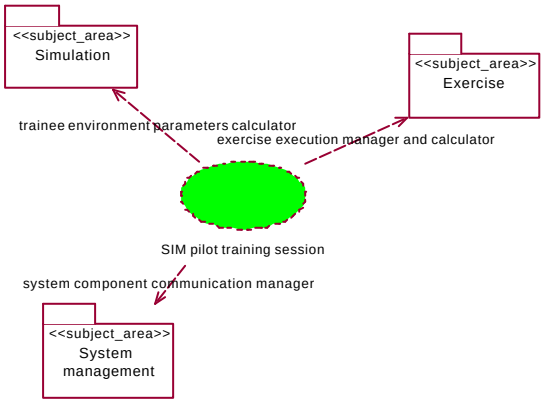


Figure 5: UML model of a business collaboration in the product-line domain.

The systematic use of collaborations has contributed to better understand and represent the nominal software architecture.

5.2 Specification of Architectural Patterns

This task has two main goals:

1. Capitalize important design solutions and their rationale to favor their use and reuse.
2. Emphasize some important design solutions encapsulated in the software architecture UML model, giving a deeper description of them.

In the business unit, **many important design solutions and their rationale are implicit and difficult to get, although valuable; it is therefore worth writing patterns** since patterns allow to collect design knowledge from experience [10].

In practice, we have first defined a textual pattern template, extending some of those defined in the literature [1], [5], [10], [21] (see Figure 6). The pattern template has been implemented in Microsoft Word™ 97 and used to specify several architectural patterns in the subsystem view (currently 5).

Identification of patterns as recurring solutions in the domain has been facilitated by the domain expertise actually present in the TT&S business unit (TT&S has reached level 3 of CMM [3]). Moreover, products scoping done during domain analysis phase has allowed to highlight the architectural style of each product-line product. **Writing patterns has been a more difficult task**. Indeed, it has required to abstract solutions from code to architectural level, using several iterations. Therefore, it has been **impossible for non TT&S business experts to write the patterns because of their deficiency on domain knowledge**. On the other hand, as **business units experts are not used to write patterns,** it has been difficult for them to select the relevant information to fill the pattern template.

We have also exhibited patterns as frameworks in the UML model of the nominal software architecture. The Word files containing patterns specification have then been respectively connected to the pattern related frameworks in the Rose™ model. In practice, we have defined the stereotype *"framework"* to characterize a UML package containing a pattern related framework model.

In Figure 7, the UML diagram represents a part of the subsystem view. In this diagram, the collaboration *Motion execution* has two entities connected to it: the subsystem *Simulation_LifeCycle* plays the roles of *Task* and *EventTask*, and the subsystem *Motion* plays the roles of *Strategy* and *EventStrategy*. This collaboration is realized by the framework *TaskEvent*, reported in Figure 8.

Proposed Template	Alexander	GOF	PLOP	AGCS
Reference	[Alexander 77]	[Gamma & al. 95]	[Coplien & al. 95]	[Rising 99]
General Section				
Name	X	X	X	X
Author				X
Date				X
Source				
Anti-pattern				
Intent		X	*Abstract*	
Also know as		X	X	X
Keywords				X
Example Section				
Problem	X	X	X	X
Solution	X	X	X	X
Problem Section				
Problem Description	X	X	X	X
Context of applicability	X	X	X	X
Forces	X		X	X
Solution Section				
Resulting Context	X		X	X
Description	X			X
Participants	X	X	X	
Collaborations	X	X	X	*Sketch*
Rationale				X
Consequences		X	X	
Known Uses		X	X	X
Variations				
See Also		X	X	X
Technical Section				
Implementation		X	X	
Code and Usage		X		
COTS				
External Documentation				
References			X	X
Glossary				

Figure 6: Comparison of patterns templates.

Exhibition of patterns as frameworks in the UML model of the nominal software architecture lets us envisage automatic refinement of collaborations using UML framework models. This has already been done in the Catalysis approach [8]. **However, we have identified some limitations due to UML tool.** Indeed, a subsystem inherits from a class and a package, as described in UML 1.3 metamodel. So a subsystem could be connected to a collaboration using a realization link. But UML subsystem and collaboration are not implemented in Rational Rose™ 98i. So stereotyped UML packages are used to represent them and then dependency is used instead of realization link (see Figure 7). To achieve automatic refinement of collaboration using its related framework, Rose™ scripts should be developed.

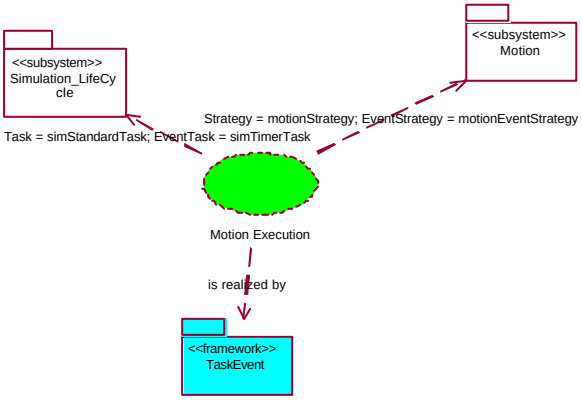


Figure 7: Motion execution subsystem collaboration.

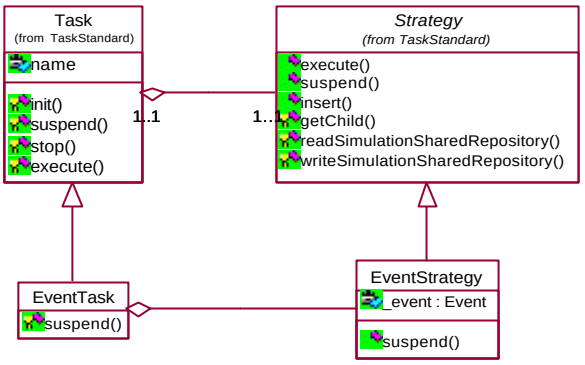


Figure 8: The TaskEvent framework in UML.

6 PRODUCT-LINE ARCHITECTURE MODELING

Product-line architecture modeling consists in adding the domain variability in the nominal software architecture and the decision related to this variability for architecture derivation. Tasks performed during product-line architecture modeling are identification of architectural variability, exhibition of variability in UML model, and specification of variability patterns. They are presented in the following subsections.

6.1 Identification of Architectural Variability

This task has two main goals:

1. Identify the projection of requirements variability on the nominal software architecture. Requirements variability is an output of the domain scoping activity done during domain analysis phase.
2. Identify the projection of products architectural variability on the nominal software architecture. Products architectural variability is an output of the products scoping activity done during domain analysis phase.

The projection of requirements variability on the nominal software architecture has been difficult to perform. Indeed, the impact of an evolution of requirement on architecture is not easily delimited. Therefore, we have only considered variability related to high level requirements in a first iteration of this task. Then, we have worked to refine variability expressed in the more detailed requirements in order to delimit more precisely all the impacts on the product-line architecture. **This task is still considered by us as an open issue.**

6.2 Exhibition of Variability in UML Model

The main goal of this task is to represent the variability identified in the previous task (see section 6.1) in the UML model. Within each architectural view, we have represented the variability following Praise generic solutions described

in [15] and LCAT recent work on variability management and representation in UML described in [6].

Product-line variability management is mainly done at the architectural level using collaborations, frameworks and variant realizations between them. The UML notation used is illustrated in Figure 9.

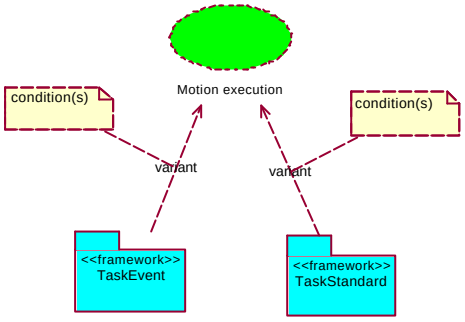


Figure 9: Representing variability in the UML model

Collaborations associated to variants are tagged "variation point", so that they can be easily retrieved by scripts. Each framework representing a realization of a variant is related to a collaboration class using a UML dependency relationships stereotyped <<variant>>.

Notes can be associated to dependency link to describe conditions for framework selection. We intend to write framework selection conditions using the UML constraint language called OCL instead of notes.

This description is sometimes inappropriate for two main reasons:

1. **Variability representation is not enough visible in the model.**
2. **The variability is buried in frameworks and patterns. This leads to merge the description of variability and the way of handling it.** If the way of handling variability is not clearly described, application engineers may adapt or alter the assets in such a way that it has not been anticipated; it then leads to product-line erosion, as stated in [18].

That's why, we have decided to use more recent work [6] in order to treat the variability issue in separating these three concerns:

1. Expression of what can vary in the product-line architecture and how it can vary.
2. Explicit display of places in the UML model where variation can occur, called variation point and represented by hotspot.
3. The decision to take when facing variants selection, called the decision model and related to the variation point.

Figure 10 gives an example of using hot spots to exhibit variability in the UML model.

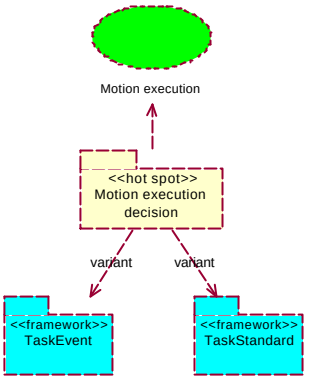


Figure 10: Managing variability with UML hot spots.

6.3 Specification of Variability Patterns

The main goal of this task is to document the variability management in the product-line architecture.

Sometimes, additional architectural parts must be incorporated in the product line architecture in order to guide the derivation of the final product architecture. These additional architectural parts are modeled using patterns, and are called variability patterns. These patterns are used when the variation leads to adaptation of the architectural entities implied in a complex collaboration.

For example, due to economic context, companies use more and more COTS from subcontractors. These subcontractors have their own development policies and the COTS do not match completely the interfaces required by the in house company. The challenge is to use in house software and subcontractor software in an application with respect to the in house application design. In the vehicle simulation domain, it happens that vehicle behavior models are developed by subcontractors. That occurs in case of too specific behavior models (e.g. aquatic vehicle model) or in case of cooperation projects. The specific vehicle behavior model has its own internal states which are different from simulation states. Some functionalities may be lacking. As code is not available, it is not possible to modify the model software. Nevertheless, it is necessary to adapt the whole application to integrate this model without modifying all the application.

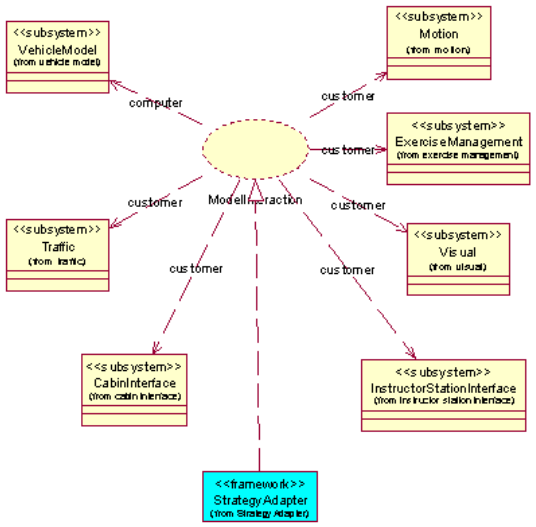


Figure 11: Variability framework realizing the model interaction collaboration.

So, in order to manage this major variability feature among products, we have used a Strategy Adapter framework (see Figure 11) and its related StrategyAdapter pattern. The variability pattern StrategyAdapter documents the solution systematically implemented when adapting locally the specific model interfaces to standard model interfaces.

The major drawback of variability patterns is that the rationale and the way of resolving variability questions are hidden in the variability pattern. No clear presentation of the decision model is made.

7 CONCLUSION AND OPEN PROBLEMS

We have proposed in this document a non exhaustive description of the lessons learned in the product-line experiment leaded by Thomson-CSF in the scope of the ongoing PRAISE Esprit project. We have also described the empirical process we have used, and the modeling techniques and notations we have created in order to represent a product-line architecture.

7.1 Architectural views

The three architectural views defined (business, subsystem and technology) are sufficient to represent the product-line architecture in the domain addressed. Developing an organization-specific architecture, it is very likely that some of the existing architectural views will be useful with adaptation(s) or combination(s). Although these architectural views should be considered as an exhaustive set, they can also be considered as a starting point to extend the proposed set of views in the model.

In this way, considering the Praise project results, we have introduced the architectural perspective notion in the LCAT laboratory [6]. An architectural perspective defines new assets from architectural views assets, related to specific stakeholders needs. Each perspective demonstrates the concern of a stakeholder. Adding architectural perspectives to the basic architectural views model can be seen as a generalization of the "+1" view of [14].

7.2 Requirement and architecture traceability: Bottom-up vs. top-down approach

Building a product-line can be done in two ways:

1. Using a top-down approach: The product line is built from scratch. Domain engineering assets are created and then reused in application engineering.
2. Using a bottom-up approach: The product line is based on legacy products. Domain engineering assets are built from existing application assets.

These approaches are not exclusive, but using both of them to build the product-line can lead to assets having different granularity levels. When trying to relate them, some major difficulties arise. This is then a major problem since the product-line approach is enforcing the traceability between assets. So how to enforce the same level of granularity between assets during product-line construction?

7.3 Peopleware

Engineers must be aware of the objectives of product line architecture. Systematic reuse and explicit description and documentation of each asset is sometimes difficult to understand for people developing one shot products. Extensive use of guidelines and examples are needed to improve engineers learning curve.

Software engineers are sometimes very disturbed by the separation of concerns. "Why creating several diagrams, if I can put all the information in just one. It's better with one drawing, it's more synthesized". But, UML seems to be a good way of sharing a mental vision of the system.

The term variability is also very confusing if badly used. Each application manages variability, due to the application logic. It must be stated clearly that we are only interested in the product line variability, i.e. the variability needed to decline specific products from a product family. That's also a reason, why, it's sometimes difficult to make people do commonality description first and then variability. They are used to manipulate both at the same time in their day to day work.

Finally, engineers are used to generate code. It is then difficult to make them understand that after the architecture phase, people have to work in order to build components and then generate code.

7.4 Tools

Rational Rose™, does not fully support the current official norm (UML 1.3) and that situation leads to:

- Use extensively UML extensibility mechanisms (stereotypes, constraints and tagged values).
- Use extensively scripting in order to implement our process.

Rose™ must be used in the product-line context as a dynamic browser and not only as static diagrams drawer. Engineers are not used to navigate in such tools. **Formation on tools is therefore necessary to acquire the good way of using them**.

Interoperability between tools is not easy, and we are now very committed to Doors™ / Rose™ couple. We have also to emphasize the fact that the platform price/seat is very expensive and costs more than 20 000$ per user ! This is to be put in relation with the fact that actual tools are not delivering all expected features needed. So, how to promote product-line technology in our business units if actual tools are not mature enough and do not fully support it?

ACKNOWLEDGEMENTS

We would like to thank for the technical discussions all of our LCAT colleagues and PRAISE project partners.

REFERENCES

[1] Alexander, C. The timeless way of building. New York: Oxford University Press, 1979.

[2] Bass, L., Clements, P., and Kazman, R. Software Architecture in Practice. Addison Wesley, 1998.

[3] Bate, R., and al. A Systems Engineering Capability Maturity Model. Technical Report CMU/SEI-95-MM-003, Software Engineering Institute, Pittsburgh, PA 15213, November 1995.

[4] Bristow, D., Bulat, B., and Burton, R. Product-line process development. STARS (Software Technology for Adaptable, Reliable Systems) project, 1995. On-line at <http://www.asset.com/stars>.

[5] Coplien, J., Advanced C++ programming styles and idioms. Addison Wesley, 1992.

[6] Coriat, M., and El Kaim, W. Software System Product Line Architecture: the Daisy model. Submitted to *First Software Product-Line Conference*, Denver, Colorado, August 2000.

[7] Donnan, G., and Jourdan, J. Software architectures, product-lines and frameworks. Alcatel Telecommunications Review, 1st Quarter 1999.

[8] D'Souza, D. and Wills, W., "Objects, Components and Framework with UML: The Catalysis Approach", Addison-Wesley, 1999.

[9] Gacek, C., Abd-Allah, A., Clark, B., and Boehm, B. On the Definition of Software System Architecture. In *ICSE 17th, First International Workshop on Architectures for Software Systems*, Seattle, Wa, April 1995.

[10] Gamma, E., Johnson, R., Helm, R., and Vlissides, J. Design Patterns: Elements of reusable object-oriented software. Addison Wesley, 1995.

[11] Hofmeister, C., Nord, R., and Soni, D. Applied Software Architecture. Addison Wesley, 1999.

[12] Information Technology. Open Distributed Processing. ITU-T Recommendations X. 902 and X. 903, ISO IEC 10746-2:1996 and 10746-3:1996.

[13] Jourdan, J., Lalanda, Ph.. "Product-line and object-oriented technology ?". In *third SEI Product Line Practice Workshop* , December '98, Pittsburgh, USA.

[14] Kruchten, P. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42—50, November 1995.

[15] Lalanda, P. Product-line software architecture. PRAISE project deliverable number 2.2, March 1999. On-line at <http://www.esi.es/Projects/Reuse/Praise>.

[16] Lalanda, P., and al. First International Workshop on Object technology for Product-Line Architectures. In *ECOOP'99 Workshop Reader*, LNCS 1743, 1999.

[17] Lalanda, P., and Cherki, S. Object Oriented Methods and Software Architecture. In *ECOOP'98, Workshop on Object Oriented Software Architecture*, Brussels, Belgium, July 1998.

[18] Leishman, D.A. Solution Customization. IBM Systems Journal, Vol. 38 (1), pp. 76-97, 1999.

[19] Perry, D., and Wolf, A. Foundations for the study of Software Architecture. *Software Engineering Notes*, 17(4):40—52, 1992.

[20] PRAISE. Product-line Realization and Assessment in Industrial SEttings. Workplan, 1998. On-line at <http://www.esi.es/Projects/Reuse/Praise>.

[21] Rising, L. Pattern Mining. Handbook of Object Technology, Chap. 38. Ed. Saaba Zamir, CRC Press, 1999. On-line at <http://www.agcs.com/patterns>

[22] Shaw, M., and Garlan, D. Software Architecture: Perspective on an Emerging Discipline. Prentice Hall, 1996.

[23] Vinga-Martins, R. and Süßlin, S. Requirements traceability. PRAISE project deliverable number 2.3, March 1999.

A product line experience in the domain of fund management

Tullio Vernazza[§], Stefano De Panfilis[¥], Paolo Predonzani[§], Giancarlo Succi[£]

[§] Dipartimento di Informatica, Sistemistica e Telematica, Università di Genova, Genova, Italy.

[¥] Engineering Ingegneria Informatica S.p.A., Roma, Italy.

[£] Department of Electrical and Computer Engineering, The University of Alberta, Edmonton, Canada.

ABSTRACT

Product lines offer significant business and technical advantages to software companies that produce a range of similar products customized for different users. These advantages derive from the strict relationships between the products. Product line development can benefit from domain analysis and engineering techniques. The paper reports the experience of product line development in the domain of fund management, an important part of the domain of banking systems.

1 INTRODUCTION

Fund management is becoming a strategic part of banking systems. Recent changes in the law and regulations has greatly increased the relevance and opportunities of fund management. Many banks are approaching this business and are willing to offer a fund management service. The production of a product line in this domain has become a compelling need for many IT producers. The paper describes DOMINARE, a project aimed at the production of a fund management product line with Domain Analysis and Engineering techniques.

The need for product lines often arises when there is a possibility to deploy similar systems in different environments. The similarity of the systems offers great potentiality for reuse across the product line, with a consequent reduction in development cost and time. It also allows a better exploitation of the synergies and compatibility between the products.

DOMINARE's approach to product lines is based on Domain Analysis and Engineering (DA&E). DOMINARE has performed DA&E on the domain of fund management. The test-bed for the discussed techniques has been the GLOBAL FUND product line. The product line comprises a package product, reflecting all the major characteristics of the domain, and several installations tailored on specific customer's needs.

The identification of the common parts across the product line has been the starting point of the work. Commonalities evidence the cohesion of the product line and highlight the potentiality of reuse. However, the products of the product line also differ for several features. The differences are due to the peculiarities of the customer requirement and of the deployment environment. The management of variability, i.e., of the differences between products, has shown to be a major issue of product lines and a critical requirement in the development of GLOBAL FUND.

DOMINARE is a European ESSI Project. It has been undertaken by Engineering Ingegneria Informatica S.p.A., a major Italian software company. The underlying DA&E methodology, named Sherlock, has been developed by DIST Università di Genova. The CASE tool used for DA&E is a customized version of System Architect 2001 by Popkin Software Inc.

The paper is structured as follows: section 2 presents previous work in product lines and DA&E; section 3 presents the project's approach to product lines; section 4 describes the product line's commonality; section 5 provides insight in the product line's variability; finally section 6 draws the conclusions.

2 STATE OF THE ART

A product line is a set of products produced by one company according to a coherent strategic line. Several authors have addressed the benefits and problems that derive from this definition.

Bass et al. highlight the potentiality of reuse across a product line [2]. Reuse pervades all development assets, including architectures, experience, solutions, and code. They consider the organizational implications of product lines, discussing the need for separate groups for managing core assets (the product line's commonalities) and products. They also advocate a component-based approach to product line development.

Product lines are often associated to domain analysis [5, 7]. Several approaches to domain analysis exist. Arango gives an overview of several domain analysis methodologies and provides a common process, which summarizes the commonalities of the methodologies [1].

Baumol et al. analyze product line production from an economic perspective [3]. Their focus is on economies of scale and scope, as well as on the equilibrium in markets. They consider different cases of monopoly, oligopoly, and competition and analyze under which conditions and with what consequences new firms can enter the market.

In parallel with the interest in the common aspects of product lines, there is also a effort to analyze and formalize variability. The Proteus project puts a strong emphasis on variability in domains, as a means to support an evolutionary development of software [4].

The differentiation between the product line's products can result from variability in the environment where the products are deployed. Predonzani et al. analyze the implications of variability in business processes and its effects of software systems [6].

3 A DOMAIN-BASED APPROACH TO PRODUCT LINES

Fund management is an increasingly important part of the domain of banking systems. Engineering Ingegneria Informatica has undertaken the development of a product line to address the demand of fund management products. The product line is named GLOBAL FUND and builds upon the long experience of Engineering Ingegneria Informatica in the domain of banking systems. The product line is made of a package product and a series of customized products.

The installed base of the product line consists entirely of the customized products. This evidences that the need for customization is fundamental in the considered market. The reason is that the combination of customer requirements and deployment environments makes each product practically unique.

Despite the dominance of the customized products, the role of the package product in the product line is fundamental. The package product is meant to support the effective and efficient development of new customized products. The package product represents a generalization of the domain and comprises the major functionalities of the product line.

The DOMINARE project has enforced the relationship between the package product and the customizations. The package product has been made more flexible to accommodate the variety of situations that may occur. This has been achieved using DA&E techniques. More specifically, the approach has been based on the identification of commonality and variability in the domain, and on the embedding of such concepts in the product line's products. The rationale has been the following:

- The identification of commonality is relevant as it highlights the cohesion of the product line. The very idea of reuse of experience and code is based on the assumption that a common part exists between the products. The explicit identification of such part is a factorization and classification process that most DA&E approaches comprise.
- The identification of the variability highlights the differences between the products. Differences are, in many cases, the most characterizing aspects of products. As such, they should not be considered as less relevant than commonalities. This aspect is a fundamental feature of Sherlock, while it is frequently overlooked by other DA&E techniques.

The identification of commonality and variability has been based on the analysis of past, present and future products. For past and present products, requirements, design documents, documentation, etc. have been available for the analysis. For future product, the development plans, the study of market and technology trends, and the contribution of domain experts have been an input for the analysis.

The connection between commonality and variability has been ensured by Sherlock's concept of "variation point". A variation point is a common feature of the product line that is (or can possibly be) implemented in

different ways in different products. A variation point belongs to the commonalities – as all product share it – but its implementation is product-dependent. Variation points can reflect business or technical variability. A variant is a specific implementation of a variation point. Variants belong to the differences. The relationship between variation points and variants is depicted in Figure 1.

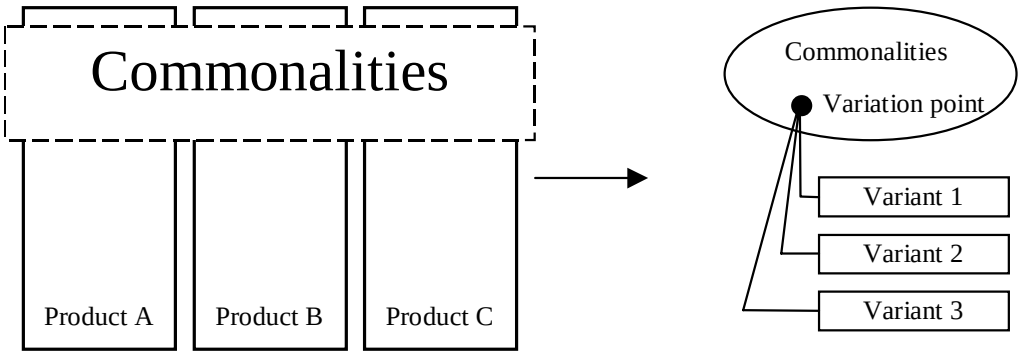


Figure 1: Variation points and variants

Commonality and variability (with a particular focus on the latter) have proven crucial for the management of the product line. The advantages they have provided are the following:

- Commonality allows reuse of knowledge and code. Reuse reduces the development effort and time. Variability points out where the code most needs to be flexible.
- Knowledge is shared better across the developers of the product line. The formalization of commonality and variability evidences the relationships between the several products and points out where synergies can be exploited.
- Alternatives (the variants) can be identified and evaluated at the beginning of a project. The cost of a thorough, up-front identification of variability is usually bearable and small. This allows to make feasible plans about the variability to implement in the products.

The following two sections provide insight on the specific product line's commonality and variability encountered in the project.

4 PRODUCT LINE COMMONALITY

The problem of product line commonalities has been addressed, at the level of requirements, through the adoption of use cases. Use cases are adequate to represent the requirements of a product. In DOMINARE, they have been used to represent also the requirements of the domain, considered as a collection of products. The term "domain use case" has been used to indicate a use case that is a generalization of several "product use cases" (Figure 2). From a different perspective, product use cases are extensions – or specializations – of the domain use cases.

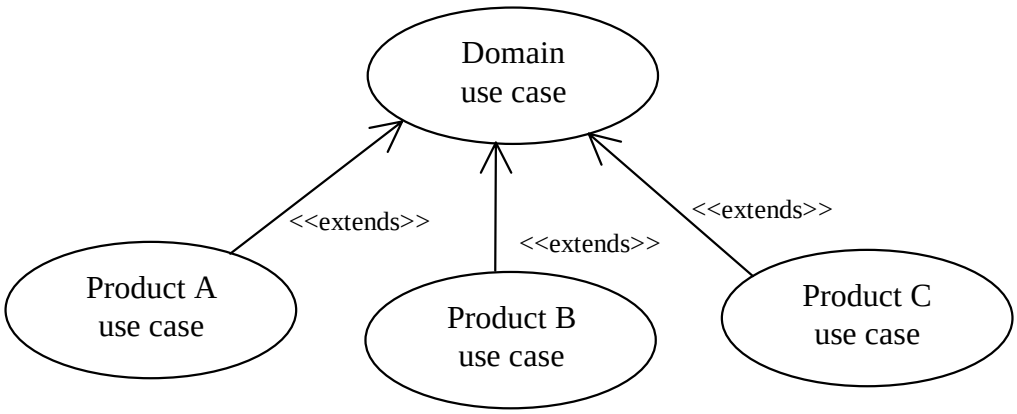


Figure 2: Domain use case as generalization of product use cases

Before discussing the captured domain use cases, for a better understanding of the product line, a set of frequently used terms needs to be introduced:

- Fund. A fund can be a pension fund (intended to provide a retirement income to, generally, the employees of a firm) or investment funds (intended to be a generic investment).
- Fund administrator. The fund administrator owns and manages the fund. It's main task is to determine the fund's portfolio composition.
- Subscriber. The subscriber signs a subscription to a fund.
- Bank. The bank manages the subscribers and the subscriptions for a group of funds.
- System owner. The system owner runs the information system for the support of fund management activities.

The sources of information for the identification of the use cases have been many: requirement and analysis documents, product documentation, market surveys, etc. The domain use cases that are presented here are a small selection of the identified ones. They represent the core of the product line's common part (Figure 3, Figure 4).

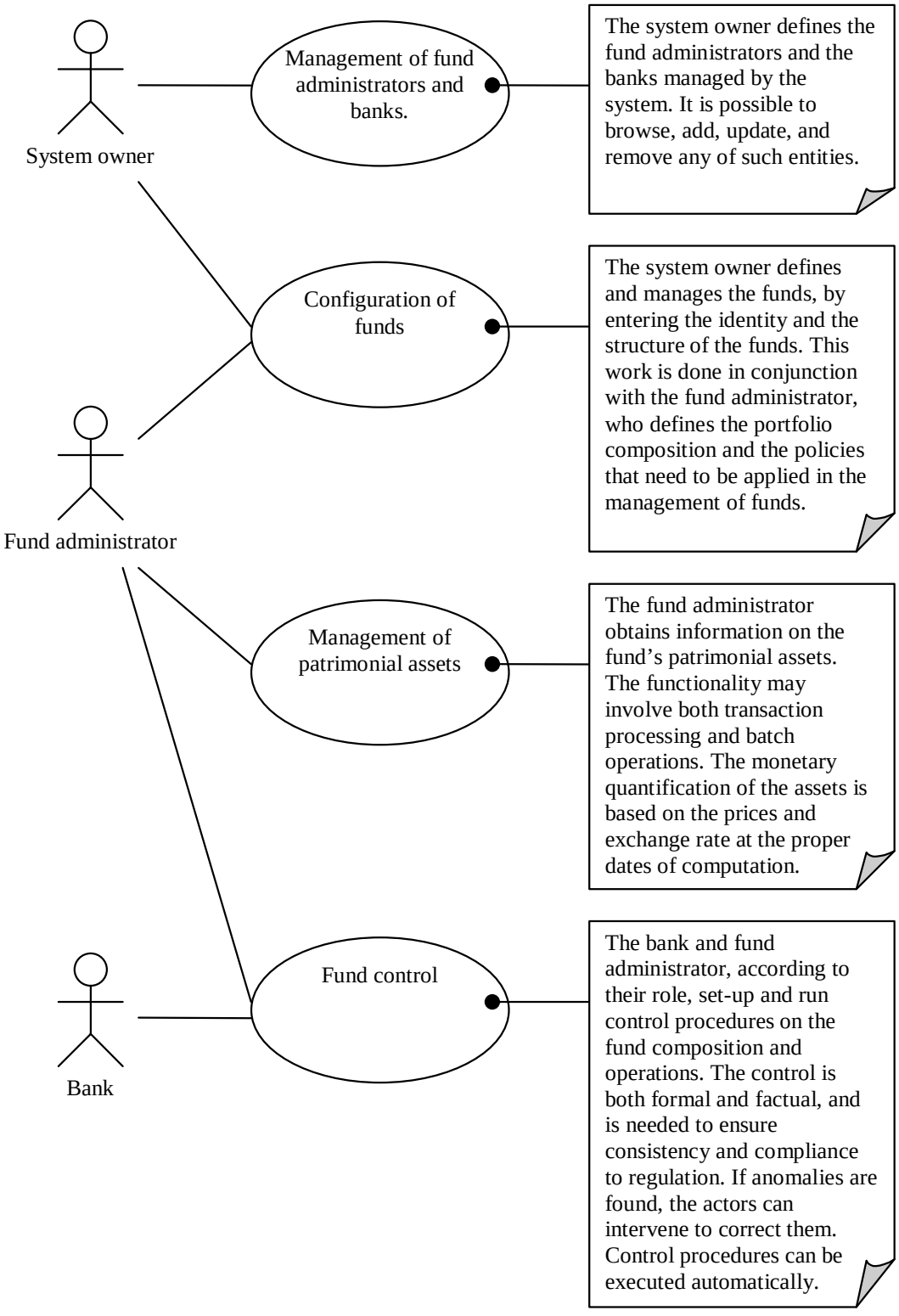


Figure 3: Domain use cases (1)

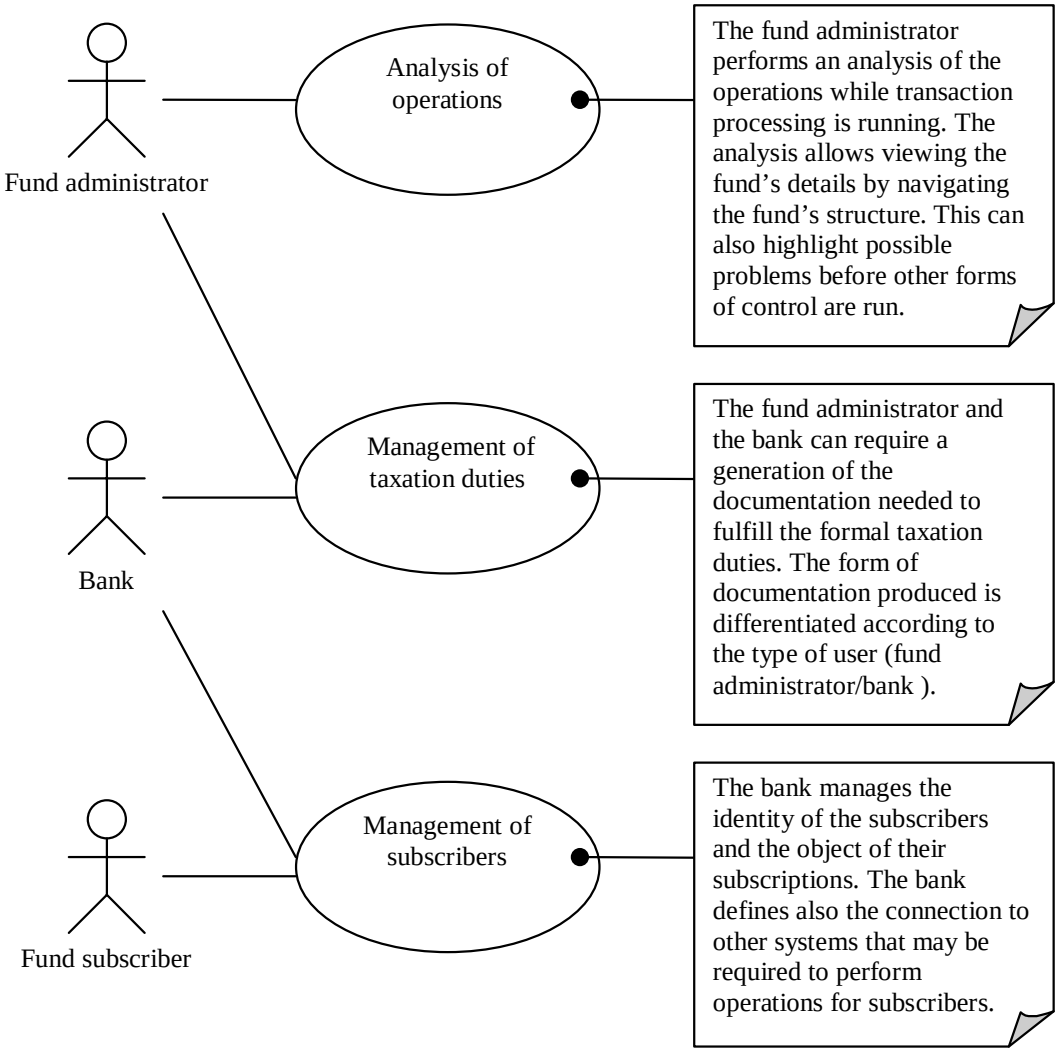


Figure 4: Domain use cases (1)

5 PRODUCT LINE VARIABILITY

The sources of variability in the product line are manifold. Many of them reflect the influence of external factors, such as the differences in the deployment environment or in the adopted regulations. The formalization of variability is based on variation points and variants. So far, 85 variation points have been identified in the product line. Here, a selection of variation points provides a high-level understanding of the product line's variability. A first group of variation points gives a characterization of the funds (Table 1).

Table 1

VP_1: type of fund • $V_{1,1}$: Open pension fund • $V_{1,2}$: Closed pension fund • $V_{1,3}$: Open common investment fund • $V_{1,4}$: Closed common investment fund
VP_2: Financial composition of fund's portfolio: • About 20 variants have been identified, including shares, bonds, funds, and other types of investment. Any of these variants can coexists, as long as this is compatible with the regulation.

VP_3: Fund structures: • $V_{3,1}$: Simple (unstructured) • $V_{3,2}$: Multi-division: structured in simpler funds

A second group shows the dimensions in which the system can grow to accommodate multiple actor instances (fund administrator and banks) and multiple computation dates (Table 2).

Table 2

VP_4: Number of fund administrators supported by the system: • $V_{4,1}$: Single administrator • $V_{4,2}$: Multiple administrators (with separate rules, constraints, and preferences possible for each administrator)
VP_5: Relationship between funds and reference computation dates: • $V_{5,1}$: Single date for all funds • $V_{5,2}$: Multiple dates for different funds.
VP_6: Number of institutions (banks) concurrently managed by the system: • $V_{6,1}$: Single bank (possibly having the system on the bank's site) • $V_{6,2}$: Multiple banks (each accessing the system as a service on a shared site)

A third group deals with regulation and currency issues (Table 3).

Table 3

VP_7: Conformity to regulation (Variants reported as categories): • $V_{7,1}$: Legislation • $V_{7,2}$: Supervision institutions • $V_{7,3}$: Treasury's decrees
VP_8: Currency for transactions, records and computations: • $V_{8,1}$: Euro • $V_{8,2}$: National currency • $V_{8,3}$: Other currencies
VP_9: Management of the transition period for the Euro currency (this variation point is subordinated to the implementation of $V_{8,1}$) • $V_{9,1}$: Gradual approach • $V_{9,1}$: Big-bang approach

The last group, comprising one variation point, defines an important aspect of data consistency between the fund management system and external systems (Table 4).

Table 4

VP_{10}: Computation of derived data
• $V_{10,1}$: Internal computation
• $V_{10,2}$: Import from external systems

At the current state of development, most of the product line's variants have been implemented. Some of the variation points are coming to a stable point, in the sense that no new variants are discovered. This is especially true for variation points that express variability in functionality (second and fourth group).

Other variation points are still undergoing adjustments and changes. This is the case, e.g., of VP_7 (Conformity to regulation). For these variation points, special attention has been devoted to provide the maximum flexibility in the product line. For instance, for the mentioned variation point VP_7, a specific module has been designed to encapsulate the variability of changing regulations.

6 CONCLUSIONS

The paper has analyzed an application of product lines to the domain of fund management. The approach has been based on DA&E techniques, focussing on the exploitation of the product line's commonality and variability. The project has demonstrated the applicability of product lines in domains where two factors are met:

(a) A core of functionalities is common to the whole domain. This is a prerequisite for reuse across the product line.

(b) The individual products are customized on the customer's needs. Customization of products is a competitive advantage but can also be a development challenge for the variability it introduces.

The analysis has evidenced that both factors need to be properly managed through a thorough analysis of commonality and variability. More specifically, the management of variability appears to be a fundamental factor that determines how much a product line can grow and accommodate diverse requirements.

REFERENCES

[1] Arango, G., "Domain Analysis Methods" in *Software Reusability*, editors: W. Schaefer, R. Prieto-Diaz, and M. Matsumoto, Ellis Horwood, New York, 1994.

[2] Bass, L., P. Clements, R. Kazman, *Software Architectures in Practice*, Addison Wesley, MA, 1998.

[3] Baumol, W.J., J.C. Panzar, and R.D. Willig, *Contestable Markets and The Theory of Industrial Structure*, Harcourt Brace Jovanovich, Inc., 1982.

[4] CAP SSP, CAP Gemini Innovation, CAP Sesa Telecom, Hewlett Packard, Intecs, Matra Marconi Space, Sintef, University of Lancaster, "Domain Analysis Method," Deliverable D3.2B, PROTEUS ESPRIT project 6086, 1994.

[5] Poulin, J.S., "Software Architectures, Product Lines, and DSSAs: Choosing the Appropriate Level of Abstraction", *8th Workshop on Institutionalizing Software Reuse*, Columbus, Ohio, 1997.

[6] Predonzani, P., G. Succi, T. Vernazza, "Reflecting Business Process Variability in Information Systems", in *Proceedings of CAiSE'99 workshop Software Architectures for Business Process Management* (SABPM'99), Heidelberg, Germany, June 14-15, 1999.

[7] Simos, M.A., "Lateral Domains: Beyond Product-Line Thinking", *8th Workshop on Institutionalizing Software Reuse*, Columbus, Ohio, 1997.

Domain analysis and product-line scoping: a Thomson-CSF product-line case study

S. Cherki & W. El Kaim,
LCAT
Thomson-CSF/LCR
Domaine de Corbeville
91404 Orsay cedex, France
{elkaim,cherki}@lcr.thomson-csf.com

P. Josset & F. Paris
LCAT
Thomson-CSF/TT&S
1, rue du Général de Gaulle
ZA Les Beaux Soleils - BP 226 Osny
95523 Cergy Pontoise cedex, France
{josset,paris }@tts.thomson-csf.com

ABSTRACT
This paper reports on the product-line experiment leaded by Thomson-CSF in the scope of the ongoing European project PRAISE, focusing on experimental results regarding the domain analysis phase of the product-line process. The experiment product-line addresses a real domain which is simulation for ground vehicle pilot training.

Keywords
Product-line approach, domain analysis, product-line requirement, UML.

1 INTRODUCTION

Domain-specific and architecture-centric product-line is now recognized as a promising approach to deal with software intensive systems development and evolution [2]. However, industrial companies lack today methodological framework and supporting tools to make a software development product-line approach realizable in industrial settings. The ongoing European PRAISE project[1] is therefore pursued by Thomson-CSF, Robert Bosch GmbH, Ericsson, and the European Software Institute to provide an integrated and validated methodological support to product-line approach [7].

PRAISE project is divided into five work packages (see Figure 1). Work packages 1 and 2 are dedicated to the definition of the baseline technology and have resulted in process, benefit assessment model, and requirements traceability and architecture methods. In the scope of work package 3, Robert Bosch GmbH and Thomson-CSF are currently leading real large-scale industrial experiments in a coordinate way to validate and consolidate previous technology.

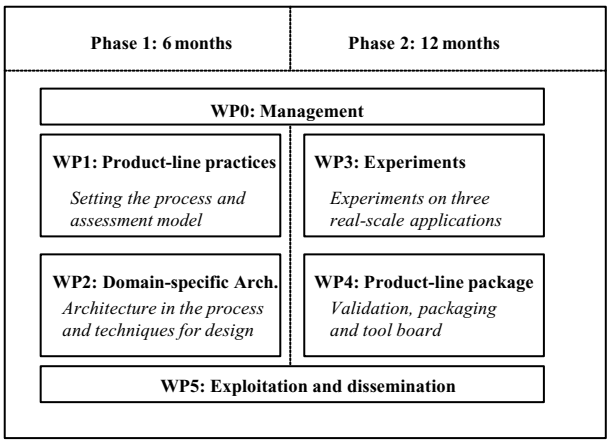


Figure 1: PRAISE project in work packages.

More precisely, the focus is made on domain engineering **Error! Reference source not found.** in both experiments and the same models have been selected from the baseline technology for experimentation. Moreover, a different domain is investigated in each company to get more sound and credible lessons learned. The domain addressed in Robert Bosch GmbH experiment is car periphery supervision whereas the one addressed by Thomson-CSF experiment is simulation for ground vehicles pilot training. Considering that domain data are present in quite different forms in each domain, technology is customized accordingly in each experiment, thus providing results covering a broader area of problems. This paper reports

[1] PRAISE project is partly founded by the European Commission under ESPRIT project 28651.

specifically on the product-line experiment leaded by Thomson-CSF.

In practice, technology related to domain analysis phase [8] and product-line scoping phase [9] has been already experimented. It has included products scoping, domain context modeling, domain requirements modeling, domain features modeling, and domain traceability modeling. Lessons learned can thus be provided in terms of process, modeling, notation and tools used to perform these activities following a product-line approach. This will be the focus of this paper.

This paper is divided into seven sections. The first one is dedicated to the presentation of Thomson-CSF organization allowing to understand team work and coordination. In the second one, the experiment process is described relating product-line scoping to domain analysis phase. In sections 3, 4, 5, and 6, experiments leaded on domain definition, products scoping, domain scoping and features modeling are respectively described in terms of their process, modeling results and lessons learned. And finally, last section gathers the whole lessons learned and gives the experiment perspectives envisaged in the future.

2 THOMSON-CSF BACKGROUND

PRAISE project is running inside the LCAT, a common research laboratory of Alcatel and Thomson-CSF [2]. LCR/LCAT unit working on product-line technology is divided into one technical team and several domain experiment teams. One domain experiment team is composed of experts and engineers from an industrial business unit of Thomson-CSF and an experiment leader. This latter organizes experiment work and relates the business unit needs and results and LCAT technical team technology. Team involved in Thomson-CSF experiment of PRAISE project is TT&S domain experiment team.

TT&S is a pilot business unit in Thomson-CSF regarding software engineering practices and has reached level 3 of the CMM [1] In order to start up the LCAT project, a preliminary product-line experiment has been leaded by Thomson-CSF/LCR in collaboration with TT&S and TT&M. It has involved two persons at full time and one TT&S expert at 20% for one year. It has allowed us to get domain data and evaluate human and time resources needed in LCAT project domain experiments.

3 DOMAIN ANALYSIS AND PRODUCT-LINE SCOPING PROCESS

In this paper, a domain is understood as "*an area of process or knowledge driven by business requirements and characterized by a set of concepts and terminology understood by stakeholders in that area*".

According to [8], domain analysis is "*the domain engineering activity in which domain knowledge is studied and formalized as a domain definition and a domain specification*". Domain definition is then defined as "*an informal description of the scope, extent, and justification for a domain*" and domain specification as "*a specification of a standardized application engineering process and product family for a domain*".

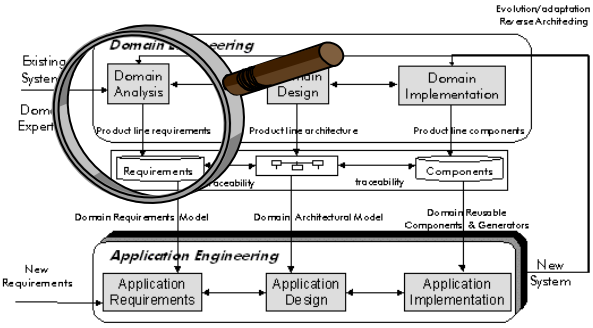


Figure 2: Product line process extracted from [8].

At the beginning of this experiment, it has been necessary to position product-line scoping within the product-line process, and more particularly with regards to domain analysis phase of this process.

In term of process, a first step of product-line scoping can be matched to domain definition step of domain analysis. A second step can be considered as part of domain specification step of domain analysis.

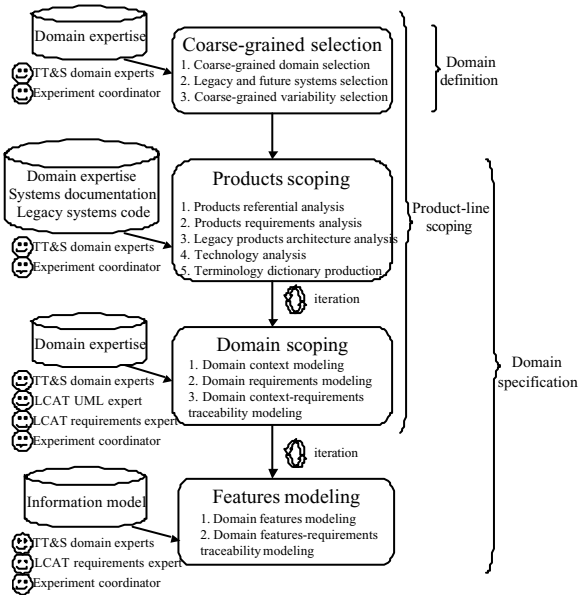


Figure 3: Domain analysis process part in Thomson-CSF experiment.

Our experiment part of domain analysis phase reported in

this paper can be described in terms of activities and tasks, as reported in Figure 3. In this figure, activities are modeled as rectangles with round corners, tasks as items, data flows as arrows, activity practitioners as faces, and domain analysis inputs as databases.

In the remainder of this paper, we will describe each activity.

4 COARSE-GRAINED SELECTIONS FOR DOMAIN DEFINITION

The main objective of the product-line scoping phase is to delimit the product-line domain [9]. As input of this phase, we have considered ground vehicle pilot training domain.

In practice, the product-line domain has been first delimited in terms of:

- a short informal description of the domain addressed,
- a list of systems included,
- and a list of coarse-grained variability.

The related activity has been realized fast and merely with coarse-grained selections performed by business unit experts, not requiring any analysis of the domain.

The selections performed on this input domain to define the product-line domain are mainly **motivated by making the product-line domain fit TT&S organization with regards to market needs and development teams**. For example, focus has been made on some ground vehicles both civilian and military, thus allowing TT&S both to extend its business to a civilian market and to leave some domain features no more required in military market. Also, TT&S development teams being built according to specific competencies, focus was made on real-time simulation and environment simulation competencies.

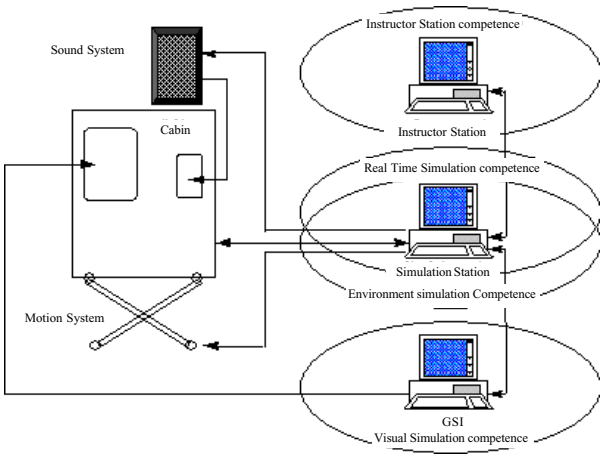


Figure 4: TT&S competencies related to ground vehicles pilot training domain

Figure 4 illustrates the main hardware elements of a ground vehicle pilot training system, with arrows denoting communications between these elements and ovals denoting TT&S software teams organized by competencies. In TT&S systems, real-time simulation and environment simulation competencies correspond to a software component (designated later with SIM) of the system; previous selections defining the product-line domain therefore leaded us to consider not a whole system as the product, but rather SIM software component of the system.

Several systems have been then selected by TT&S expert as relevant to the product-line domain thus delimited: three legacy ones and two future ones to be developed.

Finally, a list of coarse-grained variability has been selected by TT&S expert, considering coarse-grained evolution of existing systems and anticipating future systems one: this list includes variability related to trainee vehicle model which may be a tank model or a truck model, and variability as optional ability to test the availability of SIM devices at run-time.

These selections have allowed TT&S **to favor product-line process adoption, getting rid as much as possible of its market opportunities and its business competencies**. They have also allowed product-line experiment practitioners **to deal with a "reasonable" number of domain data and variation points during product-line experiment and to access easily domain data**.

Results obtained with the first step of product-line scoping has allowed us to start up domain analysis. **However, they were not enough explicit to give a meaningful view of the product-line domain to people who are non business unit experts.**

Then, the second step of product-line scoping has been performed to get a more detailed and formal scope. It means that the product-line domain has been further delimited with more detailed and formal models, like domain context model and domain requirements model proposed in [9]. The related activities have been realized with longer analysis and specification phases involving product-line experts and business unit experts.

The second step of product-line scoping is divided into two activities: products scoping and domain scoping. Products scoping aims at matching products characteristics, exhibiting variability between products. Then, domain scoping objective is to build a domain context model and a domain requirements model and their traceability as stated in [9]. In practice, we have performed products scoping and domain scoping successively by iteration. Experiment related to them is described in the following sections 5 and 6.

5 PRODUCTS SCOPING

Inputs are domain expertise on TT&S business and systems documentation and code.

Outputs are information tables matching different kind of products features and exhibiting their variability.

Techniques used mainly consist in:

- Reading of legacy systems documentation (and punctually systems code) by experiment coordinator to acquire domain data.
- Filling of information tables by experiment coordinator to store different kinds of domain data and the rationale of their variability between systems.
- Interviewing domain experts to validate domain data previously acquired and to get both future systems prospective data and implicit rationale.

Tasks performed during products scoping are products referential analysis, products requirements analysis, legacy products architecture analysis, technology analysis, and terminology dictionary production. They are presented in the following subsections.

5.1 Products referential analysis

This task has two main goals:

1. Mining the type of decomposition used during systems development and the different categories of product requirements and product architecture elements.
2. Comparing categories between different systems.

Mining development referential used in legacy systems, we have found that TT&S uses a customization of DOD-STD-2167A standard [10] and Thomson-CSF MIST methodology, which may be viewed as a combination of functional decomposition and V development cycle. We have also found that the way of using this referential can differ between different systems. For example, Figure 5 shows that *subsystem* level is or is not present in a system. In this figure, CSCI, HWCI and CSC are acronyms designating respectively Computer Software Configuration Item, Hardware Configuration Item, and Computer Software Component.

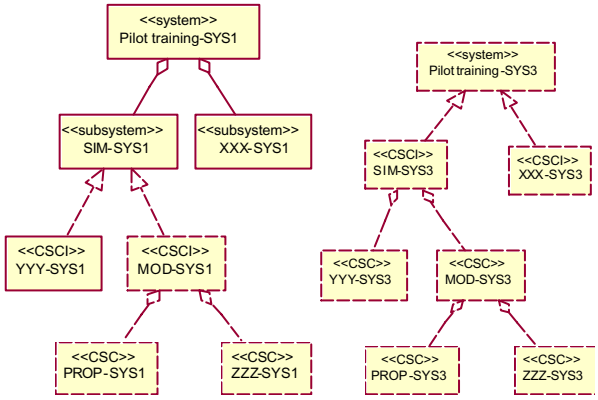


Figure 5: Comparison of referential use between systems

We have then used previous information to identify the different categories of product requirements and product architecture elements in systems documentation.

For example, the categories of SYS1 requirements have been extracted from SIM-SYS1 SSS document: it includes states/modes[2] and system capabilities. Also, the categories of SYS3 requirements have been extracted from SIM-SYS3 SRS document: it includes states/modes[2], software capabilities and their related detailed requirements, and requirements on external interfaces. Comparing these categories between different systems, we have defined a correspondence between them and selected some of them as inputs to products requirements analysis. Thus, system capabilities of system SYS1 and software capabilities of system SYS3 have been matched to constitute a category called *capability*.

Likewise, the categories of SYS1 architecture elements have been extracted from SIM-SYS1 SSDD document: it includes CSCI, HWCI and interfaces. Also, the categories of SYS3 architecture elements have been extracted from SIM-SYS3 SDD document: it includes CSC and CSC interfaces. Comparing these categories between different systems, we have defined a correspondence between them and selected some of them as inputs to legacy products architecture analysis. Thus, CSCI of system SYS1 and CSC of system SYS3 have been matched to constitute a category called *product components*.

---

[2] In Thomson-CSF system engineering methodology MIST, "states" are elements which specify geographical and operational environment in which system is brought into play (e.g. ground, sea, war, peace, operation, storage, maintenance, …), and "modes" are elements which specify system exploitation type in a given state (e.g. assessment mode, nominal mode).

5.2 Products requirements analysis

This task has two main goals:

1. Mining product requirements which have been specified in existing systems and those which must be met in future systems.
2. Matching product requirements between different products, and exhibiting their variability.

Product requirements have been mined according to categories selected previously (see section 5.1). In practice, it consists in collecting requirements in SIM requirements documentation of each existing or future system and organizing them into predefined tables. These tables have been built in such a way to match SIM requirements of each system, adding a column with a term renaming them in a common way, a column with variability description and a column with variability rationale. One table is related to one requirements category.

5.3 Legacy products architecture analysis

This task has two main goals:

1. Mining product architectures which have been specified in existing systems.
2. Matching product architectures between different products, and exhibiting their variability.

Product architectures have been mined according to categories selected previously (see section 5.1). The same process as for requirements has been followed to mine architectural elements (see section 5.2).

Another category selected (see section 5.1) is *interfaces between product components*. Considering that these interfaces are used to exchange data types (i.e. they are described in a data model form), matching interfaces has been a very laborious work, requiring to collect and match almost 500 data types.

Finally, global data communication mechanisms on the one hand, and global control communication mechanisms on the other hand, have been matched between different products. It leaded to identify an architectural style commonly used in every product. This architectural style is named *shared repository pattern*. It has been described under an architectural pattern form [2].

5.4 Technology analysis

This task has two main goals:

1. Mining technology used in existing and future systems.
2. Matching technology between different products, and exhibiting their variability.

Technology used has been identified with analysis of existing systems documents and interviews of TT&S experts. Thus, technology impacting SIM in existing or future systems has been organized in a predefined table.

This technology is both related to SIM infrastructure, including for example operational system and hardware units, and to system infrastructure, including for example hardware connections of SIM with other system components. Technology table has been built in such a way to match technology of each system, adding a column with a term renaming them in a common way, a column with variability description, and a column with variability rationale.

5.5 Terminology dictionary production

The goal of this task is to get and store the meaning of terms and acronyms used by practitioners of the domain.

Terminology mining has first consisted in collecting terms definitions in text corpus of existing systems documents, and acronyms definitions in glossaries of existing systems documents. Then two tables have been built with data collected for each existing system. First one is for terms and second one for acronyms.

It is important to notice that an acronym may be used to designate several entities according to its context of using, its context being for example defined by a specific life cycle step. In this case, we associated every entity designated by acronym clarifying its own context with the keyword *context* and splitting two entities by "|" symbol. Figure 6 gives an example of this case.

Acronym	Description
IS	Instructor Station software component *context* system architecture specification | Instructor Station and auto instruction station interface *context* SIM software architecture specification

Figure 6: Acronym designating several entities according to context of using.

Matching terminology between different systems to get the domain terminology can lead to add system identifier in context when it is necessary.

5.6 Products scoping lessons learned

Products scoping is a very laborious activity but necessary for product-line practitioners to get domain data. In practice, it has required a real TT&S domain expert involvement in domain expertise transfer (10% of one man year) and a real other practitioner involvement in getting domain data (20% of one man year).

At the beginning of the experiment, two scenarios were a priori envisaged to perform products scoping:

- Make TT&S domain expert perform products

scoping.

- Make TT&S domain expert transfer domain expertise towards one (or several) other product-line experiment practitioner who does not know the domain and make this other practitioner perform products scoping.

Second scenario has been selected because it has entailed more interactions between TT&S domain experts and LCAT technical experts, which has **favored knowledge transfer between them**: TT&S domain experts have got a deeper overview of the product-line technology, and LCAT technical experts have got a deeper overview of the domain.

Notation used to perform products scoping is mainly constituted of informal text and matching tables. In the scope of experiment, it appears sufficient to be used as inputs for domain scoping. However, traceability issues between products models and domain models can be already predicted because of different notation used. Moreover, products models have to be enhanced with regards to their completeness and their structure in order to fit in a real product-line configuration.

It is important to notice that thanks to CMM level of TT&S, the quality and liability of systems documentation and people expertise was sufficient in such a way that we did not need to go deeper into code analysis.

Products scoping activity comes before domain scoping one. It may be performed exhaustively but then it is a very long activity which outputs may not be exhaustively used by domain scoping. Therefore, it is important **to perform both activities by several iterations to focus on products scoping outputs which are useful for domain scoping**.

6 DOMAIN SCOPING

Inputs are domain expertise on TT&S business, products scoping outputs.

Outputs are domain context model, domain requirements model and domain context-requirements traceability model.

Techniques used mainly consist in PRAISE techniques proposed in [9], that is the templates of domain context model, domain requirements model and domain context-requirements traceability model and UML notation associated to them. These techniques have been however customized in some cases.

Tasks performed during domain scoping are domain context modeling, domain requirements modeling, and domain context-requirements traceability modeling. They are presented in the following subsections.

6.1 Domain context modeling

The goal of this task is to define the product-line boundaries.

In practice, we have built:

- a structure diagram to locate product-line domain with regards to domains encompassing it
- a context diagram to model relationships between product-line domain and peer domains.

UML notation has been used to represent these diagrams, and Rational Rose version 98i has been used to implement them.

As first step of product-line scoping (see section 4) has leaded to focus on SIM software component rather than on the whole system, we have merged domain (problems) and product SIM (solutions) to locate the product-line domain in the structure and the context diagrams.

In the structure diagram, UML packages have been used to represent domains; UML package hierarchy has been used to represent encompassing hierarchy among domains; and UML dependencies have been used to represent peer domain relationships. In practice, **UML notation and its implementation in Rose were adequate to model the structure diagram**.

In the context diagram, UML class have been used to denote the product-line domain; UML actors to denote peer domains; UML interfaces, UML associations and UML dependencies to denote relationships between the product-line domain and peer domains; and UML class diagrams to denote context diagram and actors diagram. This **UML notation and its implementation in Rose were adequate to model the context diagram, not considering its variability**. UML interfaces allow to model peer domains connections in a such a way that it is possible to extend current domain context diagram with scenarios between the product-line domain and peer domains using a combination of UML use cases diagrams and UML sequence diagrams as notation. The first results shows that **due to the size and complexity, it is not possible to produce exhaustively all the scenarios**.

As variability notation, we have used **efficiently UML cardinality "0..1" to denote an optional peer domain** as external actor of SIM. We have also used UML inheritance between actors with the non standard UML convention (coming from [2]) of rolling up several inheritance to one triangle and "{mandatory}" mark to denote a peer domain exclusive alternatives, exactly one of them having to be selected for a specific product SIM. This notation is **efficient but not standard**. Finally, we have used UML inheritance between interfaces to denote some optional services in an interface. However, some variability like **exclusive alternative enrichment of two interfaces has not been modeled with UML in a satisfactory way**. It is

of course possible to use OCL, but this language is not interpreted in Rose.

Figure 7 shows a UML class diagram denoting a simplified part of the context diagram produced. It includes the two first variability notations mentioned above. First one is used to denote that motion hardware component is optional as external actor because motion simulation may not be required by the client. Second one is used to denote that motion hardware is either a MVT1 or a MVT2 motion hardware system.

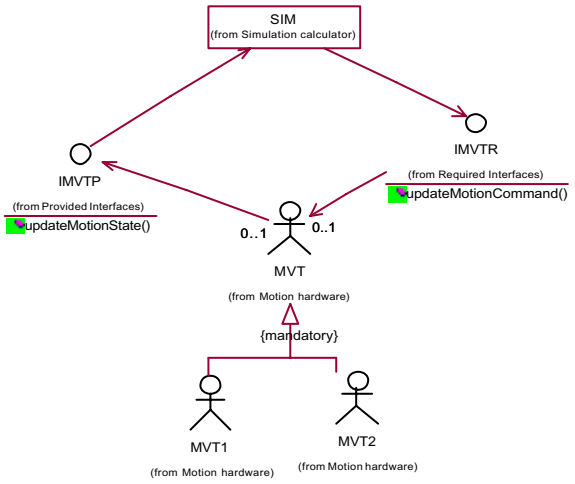


Figure 7: Part of the context diagram.

Identification and specification of UML actors and interfaces take products scoping outputs as input with a focus on categories *system components* and *external interfaces requirements*. Identification of UML actors consists in considering other system components than SIM, including software components and hardware components. Identification of UML interfaces does not only consist in collecting matching external interfaces of SIM component.

Indeed, in the domain studied, interfaces between components are specified under data types exchanges and there is no object model of it. To match UML interfaces with existing interfaces, a group of data types provided by SIM to an external component has been therefore **reengineered** into a service required by SIM and conversely, a group of data types provided by an external component to SIM has been **reengineered** into a service provided by SIM.

Two UML interfaces are distinguished for required and provided services regarding one peer domain. 13 UML actors and 22 UML interfaces have thus been specified in context diagrams. The diagram obtained is business oriented. We have then completed it with technology information in a separate view.

6.2 Domain requirements modeling

The goal of this task is to represent the product-line detailed requirements: identification, variability and traceability.

In practice, we have built a domain requirements template taking into account several pre requisites:

- It must support several activities which are design, representation, visualization, storage, traceability, evolution of product-line detailed requirements.
- It must be implemented in existing and available tool.
- It must be used to take in existing sources from the domain.

A tabular template stands for a good candidate. Indeed, it actually constitutes a support for previous activities and it is moreover implemented in several existing tools as Microsoft Excel 97 and QSS Doors 4.1.

Each line of this tabular template was dedicated to one product-line detailed requirement, and each column to one product-line detailed requirement attribute. Some attributes were mainly defined for requirements identification (identifier, wording, atomic level[3]), other for requirements traceability (related HLR[4], impacted requirements), other for requirements variability (variability type, variability overview, related coarse-grained variability), and other for traceability to products requirements (presence in product 1, presence in product 2,…).

Microsoft Excel 97 has been first selected to implement this template because of its easiness of use, its availability at the beginning of the experiment, and its compatibility with other documentation tools allowing to recover automatically existing data. However, Excel showed itself mainly inadequate with regards to traceability visualization and evolution of the product-line detailed requirements. It is moreover true if traceability involves elements implemented in other tools like Rational Rose version 98i. To summarize, **Excel is the best candidate to take in and edit domain existing sources but is not sufficient**

[3] Atomic means that the requirement cannot be further decomposed into other ones.

[4] HLR is an acronym designating high level requirement. It is defined to structure the product-line detailed requirements, grouping several of them to a more abstract one. For example, *trainee vehicle model* and *testability* are high level requirements (HLR) within our domain.

regarding requirements traceability. Therefore, the domain requirements template has been also implemented in QSS Doors 4.1 which was supposed to offer traceability services lacking in Excel.

Product-line detailed requirements have been designed and represented within the product-line domain using the previous template. More than 500 detailed requirements have thus been represented and structured by nearly 30 high level requirements (HLR). This task was first reduced to collect matching capabilities and their related detailed requirements from products scoping. However, the following issues were raised:

- Only attributes "requirement identifier", "requirement wording" and "related HLR" could be directly extracted from these data: other requirements attributes were implicit. Each missing attribute has been therefore produced by TT&S domain expert. To explicit variability attributes, the list of coarse-grained variability obtained with first step of product-line scoping (see section 4) has been used.
- Also, these attributes are product-specific: it means that one requirement may have different requirement identifier, requirement wording and one high level requirement (HLR) may have different high level requirement identifier in each product addressing it. Each product-specific requirement identifier has been therefore renamed in a generic one, which is independent from products. Likewise, each product-specific requirement wording has been reformulated into a generic one corresponding to the most meaningful informal formulation.
- Moreover, the set of requirements extracted from products scoping is incomplete: it means that many requirements remained implicit, especially non functional ones. New requirements have been therefore designed by TT&S domain expert. Production of new detailed requirements has been performed by high level requirement (HLR).

With regards to variability, detailed requirements have been first divided into common ones and variable ones. Common means that no variability is encapsulated in the requirement and that the requirement is met by every product. Among variable requirements, two kinds were identified as being optional and having alternatives. Being optional means that the requirement does not encapsulate variability and is not met by every product. For example, the requirement "*Command visualization of tanks from traffic environment*" is optional since it is met by tanks simulators but not by trucks simulators. Having alternatives means that variability is encapsulated within the requirement. For example, the requirement "*Fill cabin equipment repetition page of instructor station*" has alternatives since the list of equipment is different according to trainee vehicle model coarse-grained variability. To characterize a requirement with regards to variability, value "common", "optional", or "variable" has been first assigned as variability type attribute. Then, to characterize a requirement having alternatives, its variability parameter has been described as variability description attribute. Figure 8 illustrates a part of variability attributes related to the detailed requirements.

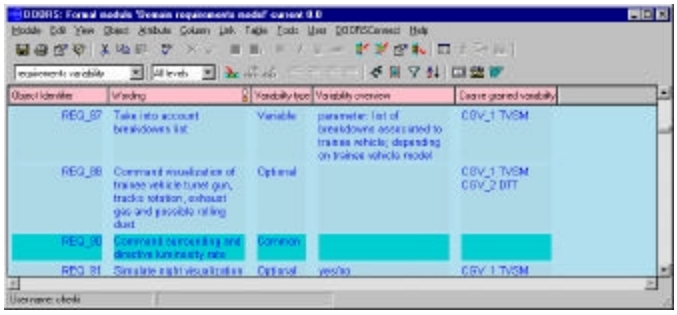


Figure 8: Variability attributes of the detailed requirements

As stated before, the domain requirements model was first implemented in Excel. Then, this model was automatically imported from Excel to Doors, thanks to the Doors importation service. In the Doors formal module thus created, three views were created, focusing on requirements traceability attributes (see Figure 9), requirements variability attributes and products traceability attributes. **Doors is efficient to import Excel domain requirements model and to provide a more manageable visualization of domain requirements with its views services**.

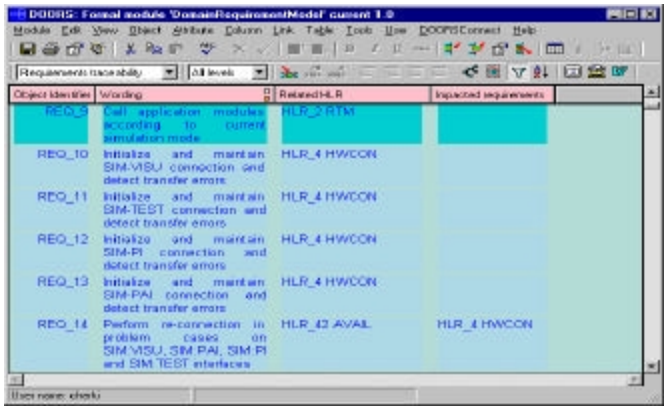


Figure 9: Doors "requirements traceability" view of domain requirements model.

In the Doors formal module, traceability within product-line requirements is implemented using textual attributes. It

cannot be actually managed. To solve this problem, a Doors script has been implemented; it uses traceability attributes values from Doors formal module to create a Doors link module. The Doors link module thus created implements traceability in a more manageable way.

Doors is efficient to get a more manageable traceability between domain requirements with its dedicated link modules and its script language DXL. The lack of some links visualization and search services requires Doors customization including DXL script development. **On the one hand it is therefore possible to have these services. On the other hand, it induces additional costs.**

6.3 Domain context-requirements traceability modeling

The goal of this task is to represent relationships between domain context model and domain requirements model.

In practice, we have established links between detailed requirements from domain requirements model and SIM interfaces from domain context model: for example, the detailed requirements which wordings are "Fill assessment page", "exercise final mark", "passed duration" and "for every criteria, mistakes number and mark" have been linked to the operation "updateAssessmentResults()" of SIM required interface regarding instructor station actor.

In term of representation and implementation of these links, we encountered difficulties, including that domain requirements model is implemented in Doors whereas domain context model is implemented in Rose. We then used tool DOORS Rose Link version 2.0 to create a Doors view of Rose domain context model. **DOORS Rose Link is efficient with regards to domain context model importation** with the objective of establishing context-requirements traceability. However, it **is not flexible enough with regards to selection of elements to import**: elements not involved in traceability are also imported from Rose to Doors. This must be managed within Doors using Doors filtering services. Moreover, importation of UML connectors, including aggregation, inheritance and dependency, are not supported. This is a major drawback when establishing requirements-architecture traceability.

Links between domain detailed requirements and SIM interfaces could be represented and implemented following two methods.

1. The first one consists in using link creation services of Doors.
2. The second one consists in adding context traceability columns in Excel tab containing domain requirements model and importing it in Doors as ever done for requirements traceability.

Both methods have been experimented. **Links creation is feasible and easy to perform in Doors** (often a mere drag and drop or use of contextual menu). However, **links are not constrained enough in Doors with regards to modification**: object which is only involved in links as a target (Doors links are oriented) can be deleted without impacting the link itself or at least indicating the deletion.

7 FEATURES MODELING

Inputs are domain expertise on TT&S business, domain scoping outputs and technology defined in [9].

Outputs are domain features model and domain features-requirements traceability model.

Techniques used mainly consist in PRAISE techniques proposed in [9], that is the templates of domain features model and domain features-requirements traceability model. UML notation proposed in [9] has been however replaced by a Doors one.

Tasks performed during features modeling are domain features modeling, and domain features-requirements traceability modeling. They are presented in the following subsections.

7.1 Domain features modeling

The goal of this task is to structure the product-line requirements variability in such a way to assist decision during application requirements phase of application engineering (see Figure 3).

In practice, we have built a domain features template and used it to represent product-line requirements variability as a tree according to FODA technology [4] (see Figure 10).

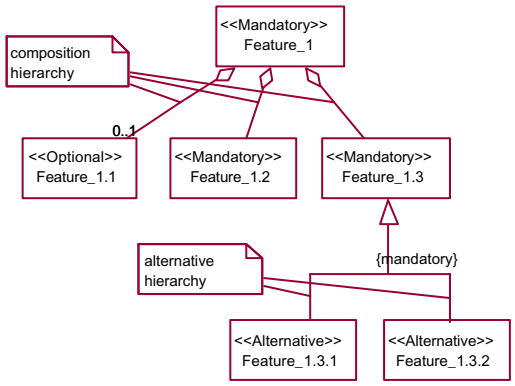


Figure 10: Features attributes and relationships according to FODA.

FODA notations have been used to represent this tree and QSS Doors version 4.1 has been used to implement it. The selection of Doors was made to favor further traceability to domain requirements model which was implemented in this tool.

Domain features have been designed with detailed

requirements variability as input. More precisely, a focus was made on domain detailed requirements which variability type attribute has value "variable" or "optional". Domain features obtained have been then structured according to a coarse-grained information model. More than 120 domain features have thus been designed, represented and implemented within the product-line domain using the template previously implemented in Doors.

To establish traceability between domain features, we have transformed the domain features tree previously designed into a domain features graph, thus representing variability collaborations. FODA notations have been used to represent this graph: it means that we have put inclusive and exclusive links between features, as specified in [9] and [4]. Contrary to domain requirements traceability, the links are not already represented under the form of Doors attributes. Therefore, links between domain features must be made by hand.

FODA notations implemented in Doors showed itself **adequate for domain features design, representation and visualization**. More time should be spent to experiment more parameterization in features to conclude about notation and tool adequacy about it.

However, **visualization of domain features traceability is not enough supported up to now** since Doors does not allow to have a global overview of these links. This is **damageable since domain features model must be used as input for application requirements derivation**. More work must then be done to explore Doors customization required for application requirements derivation.

7.2 Domain features-requirements traceability modeling

The goal of this task is to represent relationships between domain features model and domain requirements model.

In practice, we have established links between detailed requirements from domain requirements model and some features from domain features model.

The same statements regarding traceability as those previously done can be repeated here.

8 CONCLUSION AND PERSPECTIVES

A first lesson learned from this experiment is that working with emerging technologies as input required to follow a pragmatic approach. Indeed, emerging technology is not easily accessible and has not covered every technical problem yet. It must therefore be overcome by pragmatism.

Moreover, emerging technology is not always broadly supported by existing notations and tools. Some non standard notation conventions and tools customizations must therefore be developed. This requires a lot of time and cannot be done exhaustively. As solution, PRAISE project involved a tools providers board, providing them with requirements for tools supporting product-line technology.

With regards to our industrial partner, introducing new technology as product-line is facilitated by the fact that it implies reuse of business knowledge ever acquired by industrials. On the other hand, it is ambitious because it requires a real and durable investment from industrial partner since product-line technology shows itself effective after a long time.

REFERENCES

[1] R. Bate and al.. A Systems Engineering Capability Maturity Model. Technical Report CMU/SEI-95-MM-003, Software Engineering Institute, Pittsburgh, PA 15213, November 1995

[2] G. Donnan and J. Jourdan. Software architectures, product-lines and frameworks. Alcatel Telecommunications Review, 1st Quarter 1999.

[3] M. Fowler and K. Scott. UML distilled. Applying the standard object modeling language. Addison-Wesley, 1997.

[4] K. Kang and S. Cohen and J. Hess and W. Novak and A. Peterson. Features-Oriented Domain Analysis (FODA) Feasibility Study. Technical report CMU/SEI-90-TR-21, ESD-90-TR-222. Software Engineering Institute, November 1990.

[5] P. Lalanda. Shared repository pattern. In *Proc. 5th Annual Conference on the Pattern Languages of Programs*, Monticello, USA, August 1998.

[6] P. Lalanda. Product-line software architecture. PRAISE project deliverable number 2.2, March 1999. See http://www.esi.es/Projects/Reuse/Praise.

[7] PRAISE. Product-line Realization and Assessment in Industrial SEttings. IT RTD PROJECT. Project Program, 1998. See http://www.esi.es/Projects/Reuse/Praise.

[8] Software Productivity Consortium (SPC). Reuse-driven Software Processes Guidebook. Technical report SPC-92019-CMC v.02.00.03, Software Productivity Consortium Services Corporation, November 1993.

[9] R. Vinga-Martins and S. Süßlin. Requirements traceability. PRAISE project deliverable number 2.3, March 1999. See http://www.esi.es/Projects/Reuse/Praise.

[10] U.S. Department of Defense. *Military Standard: Defense System Software Development*. DOD-STD-2167A. Washington, D.C., February 1988.

Moving toward software product lines in a small software firm: a case study

Tullio Vernazza
Università di Genova
Via Opera Pia 13
16145 Genova, Italia
+39 010 3532793
tullio@dist.unige.it

Paolo Galfione
RE.SI.CO.
Via F. S. Orologio 6
35129 Padova, Italia
+39 049 774566…
paolo@ospiti.it

Andrea Valerio
COCLEA
via Magazol 32
38068 Rovereto, Italia
+39 0464 490201
Andrea.Valerio@coclea.it

Giancarlo Succi
University of Alberta
Edmonton, AB
Canada T6G 2G7
+1 780 492 7228
Giancarlo.Succi@ee.ualberta.ca

Paolo Predonzani
Università di Genova
via Opera Pia 13
16145 Genova, Italia
+39 010 3532793
predo@dist.unige.it

ABSTRACT
Product line engineering aims to take to the software development process the benefits of manufacturing processes where reuse and standardization lead to reduced costs, decreased time-to-market and improved product quality. The challenge is to capture, formalise and reuse past and present expertise in new projects, improving the software process and the products delivered to the customer.

This paper present a case study regarding the introduction of domain analysis and object-oriented frameworks in a small software firm with the purpose to set-up a development environment based on product lines. The goal of the DOOM project was to evaluate the impact and the benefits of the introduction of a domain engineering approach in a specific domain, laying the groundwork for the definition of a corporate reuse program toward the introduction of a product line. In this paper we present the results achieved with the experiment and the lessons learnt.

Keywords
Domain analysis, reuse, software process improvement

1 INTRODUCTION

Information technologies are continuously and rapidly evolving; software firms are facing the chance to develop products and services that meet new customer requests. These products embed an ever growing knowledge, presenting complex functionalities with simple and intuitive user interfaces. Firms are specializing their production into application domains where they are capable to exploit their internal knowledge following the rapid organizational, technical and market changes.

Software firms are facing the challenge to move from a project-by-project approach to a knowledge-centric environment where new projects reuse and exploit past and present expertise. The underlying idea is to structure the software process into product lines, exploiting the reuse potential of components and architectures. Domain engineering aims to foster this new way of organizing the software process. Domain analysis, a core aspect of domain engineering, is a methodology that support the identification, collection and organization of the artifacts used in software development in a particular domain so to make them available for the development of other products in the same or in different domains. The considered artifacts are not necessarily the traditional deliverables of the software development process, such as code. They are all the products and the by-products generated during software development, including requirements, designs, code, test cases, pictures and even scratch ideas, comments, notes, and so on.

Domains are areas of core product expertise for which economies of scope can be realized. Economies of scope occur when building a reusable resource and then re-using it in several products yield more income than recreating the resource ‘from scratch’ in each product. The economy stems from leveraging expertise and reducing learning - major cost drivers in product development. It is dependent on a strategy that pursues market share and product differentiation, while working on stable and reusable product architectures. Flexibility and customization is obtained through variations.

Domain analysis can be defined as: “a process by which information used in developing software systems is identified, captured and organized with the purpose of making it reusable when creating new systems” [5]. During software development, different information is produced,

and the delivered software products are only part of this heap of data. Domain analysis elaborates all these information aiming to exploit and reuse most of them in present and future software development projects. Following this view, domain analysis fosters software reuse in the sense that it supports the identification and definition of information and components that can be reused in applications and in contexts different from the ones for which they were originally conceived.

The emphasis in domain analysis has moved in the years from the analysis of code to the analysis of every kind of information produced in the software process. The goal is to identify and define high level reusable artifacts, such as frameworks and architectures, with the goal to maximize the benefits coming from reuse. In this view, the domain analysis process became a fundamental part of a global software engineering process. The purpose is to produce new application reusing past components, frameworks and information and aggregating them following the model proposed by general domain architecture [5], laying the foundations for the introduction of product lines in the software development process.

Real benefits directly linked to the introduction of domain analysis and its methods are rather difficult to measure because domain analysis is a (support) cross-process that affects all the production process. Moreover, these benefits can be revealed mostly in the medium-long term, not really upon successful completion of the experimentation, but when more projects will be completed adopting it. In these last years, these concepts have been exploited and different methodologies for domain analysis have been proposed [1].

Software engineering stimulates the software firms to structure their development environment following sound principles, achieving the control over the process and exploiting the benefits other firms have from industrial development, such as in the hardware sector. Domain analysis is the first necessary step required for starting this process.

In this article we present an experiment aimed to introduce domain analysis in a software firm in order to evaluate the effects that it has on the development process. Domain analysis takes to the identification of the enterprise knowledge while its formalization is done with object-oriented techniques. Frameworks and reusable components are developed to feed the new projects, laying the groundwork for the introduction of a corporate reuse program.

Chapter two presents the starting scenario for the experiment and the motivations that led to it. The experiment phases and the technical aspects are described in chapter three. Chapter four investigates the results achieved and the lessons learnt, while chapter five presents the conclusions and the future work.

2 STARTING SCENARIO FOR THE EXPERIMENT

RESICO is a software firm located in Italy; its core business is the development of software applications in the sector of public services, in particular concerning the management of rest houses. The software process is based on the Microsoft development environment and technologies; since it was born, a strategic goal of RESICO has been to ‘stay on the technology edge’, adopting the most modern technologies that the market offers.

The conceivement of the DOOM process improvement experiment (PIE) was originated by the analysis of the status of the development process of RESICO. This internal assessment put into evidence several weaknesses that were mostly due to the lack of a real formalisation of the process itself. Among these aspects, the introduction of a well-defined development process and operative procedures which best exploit the commonalities among RESICO’s applications could have a high pay-off. Moreover, experience made in past projects is not formalised and all the acquired know-how is hard to reuse.

This was one of the leading motivations that originated the DOOM project: the company considered the introduction of domain analysis a necessary step to improve its internal organization and productivity. DOOM was an experiment for understanding the impact of a product line based approach instead of the traditional acting on a project-by-project basis. We expected in the medium term a contribution to reduce the time-to-market and development costs, due to the deployment of reusable components and architectures, and the grown ability to efficiently cope with rapid market evolution and changed customer requests. From a technical point of view, the goals of DOOM were:

- to gain a deeper knowledge of the application domain concerning software products developed by RESICO (in particular for the control management domain);
- to increment the effectiveness and the efficiency of software reuse practices;
- to improve the analysis and design of software products;
- to improve the quality of the software development process (leading to improved product quality such as robustness, reliability, usability, modularity, structure);
- to improve interoperability and maintainability of software applications.

The commercial strategy that guided the definition of the DOOM experiment considered the increase of the company prestige behind its clients (leaded by the publicising of the adoptions of extremely modern techniques such as domain analysis). Moreover, it aimed to enlarge the market share and to set-up the basis for a durable expansion program, built on a core control management application customised for several customers (exploiting the network externalities effects generated).

The DOOM experiment was launched and strongly committed by the top management that saw in the project an important step in the company improvement plan. The

personnel were aware of the purpose of DOOM and it shared the decision of the management.

3 THE DOOM PROCESS IMPROVEMENT EXPERIMENT

The main goal of this experiment was to introduce a method for formalising the knowledge owned by the organization in the control management domain. The specific improvement action regarded the introduction of domain analysis and object-oriented frameworks. This should result in better structuring present and future applications in the domain, laying the groundwork for the effective introduction of a reuse-centred software process. The DOOM PIE originated from the need to tackle with the weak aspects that come out from an internal assessment of RESICO's software development practices.

The baseline we chose for the experiment was the production of a software application that supports control and quality management in goods and services production. This baseline was well suited for the experimentation of domain analysis: it was a typical project in the control management domain and a real case to experiment and evaluate how the domain knowledge, once formalised and structured in the form of reusable components and architecture, could be exploited in the development of new applications.

Domain analysis has been considered in the DOOM project from a wide perspective: all artifacts generated in the software life cycle have been in the focus of the project. The artifacts considered were not the traditional deliverables of the software development process, such as code. They were all the products and the by-products generated during software development, including requirements, designs, code, test cases, pictures and even scratch ideas, comments, notes, etc.

The phases of the experiment

DOOM was organized into four main phases:

1. Domain analysis study, selection and adaptation: in this first phase we dealt with the identification of a suitable domain analysis methodology to apply in RESICO, its customisation for the integration in the current development environment and the identification of an adequate tool supporting the domain analysis methodology chosen. We analysed different methodologies, considering their main characteristics and the compatibility with the DOOM goals and the RESICO development environment [1]. We decided to move along the indications of PROTEUS [2] and FODA [3], following the main lines of SHERLOCK [4]. We found that FODA has an interesting approach based on user-visible features, while PROTEUS has a robust object-oriented approach. The two approaches complement each other: FODA simplified the analysis of applications and of the domain, in particular regarding user needs and requirements, while PROTEUS enabled the design and implementation of architectures and components following a natural way of working basing on objects. This allowed building up a coherent methodology that well integrates in the RESICO environment. Then we searched for a tool supporting the domain analysis activities and the production of object oriented components. We did not found in the market tools whose declared purpose was to support domain analysis. We decided to take a different approach and we considered our specific requirement, i.e. to support modelling and design of components, and the constraints imposed by the current environment in RESICO. These considerations led us to purchase and adopt a modelling tool in the Visual Studio package, Visual Modeler, that well integrates in the Microsoft environment. It allows to produce and document object-oriented models, it supports UML notation (with the exception of Use Cases) and it helps in the coding phase.
2. Domain analysis application: this phase included the formulation of the domain models and the development of the object-oriented frameworks. The first step was the collection of the knowledge (present and future requirements, too) about the 'control management' domain. This information was classified according to the importance and the type of data. Afterwards, the analysis of the information conducted to the production of the domain models. In the second step, the reusable object-oriented frameworks were designed. They were derived from the domain models defined in the previous step, and they captured the generic features and functionalities in the domain in consideration. Besides, they have variability points where variants will be hooked. During the project the personnel had the important opportunity to abstract from the daily stress caused by direct production and deadlines and they were allowed to focus the attention on the technical aspects of the experiment. The PIE team agrees in considering this a very stimulating and motivating experience.
3. Framework specialization and application in the baseline: in this phase the frameworks were specialized and applied to the baseline, i.e. implemented considering the specific context of the baseline project. The adaptation was based on the specialization and/or derivation of the frameworks produced at the previous step filling out variability points, exploiting object-oriented techniques.
4. Analysis of the experiment and final evaluation: this phase aimed to evaluate the results achieved with the PIE, allowing RESICO management to understand if the adoption of domain analysis was effectively advantageous. In order to evaluate the effectiveness and the real improvement of the domain analysis in the development process, a measurement program has been planned and performed. Two types of metrics have been identified: process metrics, i.e. quantitative measures of effort related to the experiment activities, and product metrics, such the number of errors. A qualitative assessment of the impact and benefits obtained with the introduction of domain analysis completed the evaluation phase.

A dissemination activity was performed during the whole project execution with the goal to communicate to the personnel the relevant information about DOOM and to internally transfer the experience grown by the PIE team. Besides, external dissemination activities aimed to present DOOM results and lessons learnt to the wider community.

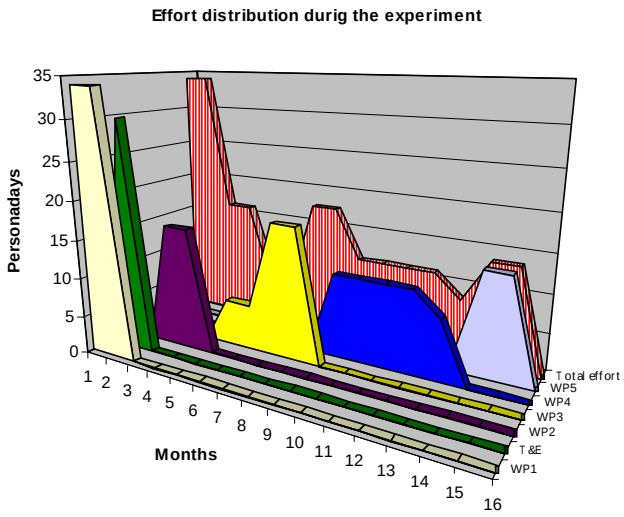
4 RESULTS ACHIEVED AND LESSONS LEARNT

At the end of the experiment we reported a general satisfaction for the achieved results; the personnel directly involved in the PIE have shown a great interest and appreciation for the DOOM project.

From the software engineering point of view, the most significant results of the DOOM project have been:

- the personnel gained a deeper understanding of the 'control management' domain, due to the greater level of formalisation of user requirements that was necessary in the first steps of the domain analysis activity in order to characterise the domain and define its boundaries; the formalisation of the knowledge and experience concerning the applications led to the identification of the common features characterising them, and to an initial categorisation of the many variation points and variants that distinguish the different control management applications;
- focusing the attention in the definition of domain models contributed to shift most of the life cycle effort to the analysis, design and planning activities; this led to a robust and modular skeleton of the application produced in the baseline project and a clear definition of the components interfaces;
- the quality of the software process has increased because the reuse-centred development promoted by domain analysis and based on the aggregation and specialization of well tested components, effectively resulted in a higher quality of the products.

Figure 1: Effort distribution during the experiment



We measured the effort the personnel directly involved in the experiment spent in every activity of the project [see figure 1], but this information are not directly comparable with past data regarding single project due to the different work break-down structure and organisation introduced by domain analysis.

We revealed that domain analysis is an effort-consuming activity if compared to the traditional analysis activity because a domain (a family of applications) rather than a single application is analyzed. Return on the investments are expected only in the medium term and over multiple projects [see figure 2].

Some weak points also arise, mainly concerning the difficulties in a complete and clear evaluation of results connected, in particular considering business impact and in measuring significant and useful metrics for comparing DOOM with past experiences.

An important lesson we learnt is that the introduction of a new technique or method in the software development process has to be carefully planned and the impact should be accurately estimated in order to minimize it. This lesson is probably well known to all the organizations, but rather often in the information technology field it is difficult to precisely evaluate the trade-off between the need to introduce new technologies and methods and the need to have a stable and productive software process. Domain analysis introduces uncertainty, a kind of 'we do need more' syndrome, with the risk to introduce too many changes in the software process at the same time. It is necessary to focus, bound and plan well domain analysis introduction, moving small steps towards the target and maintaining the control over the changes made.

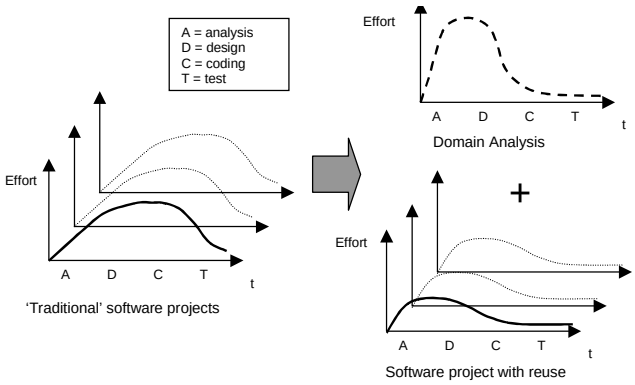


Figure 2: The effect of domain analysis

Domain analysis impacts the entire software process, i.e. the whole organization. It spreads its influence from requirement management and engineering to product design and implementation, introducing a way of looking at the software life cycle focused on product families and domains. We revealed that, while the initial investments are quite relevant, the potential return and benefits will come only in the medium-long period (in the same way as for reuse).

Another lesson we learnt regards the specific domain analysis methodology we chose to adopt. We based it on object-oriented analysis and modeling because in our

opinion object-orientation lays the groundwork for an effective analysis of software applications and for the definition of reusable components and domain architectures. However, every organization has to chose the best methodology from the different ones that are available in the market considering its internal structure, development environment and technoloy.

5 CONCLUSIONS AND FUTURE WORK

This paper presented the experience made in the DOOM project, a process improvement experiment made in a software firm and concerning the introduction of domain analysis.

The experiment showed satisfactory results in general; the introduction of domain analysis and object-oriented models and frameworks had positive effects on the software process.

Domain analysis requires a change in the way people usually work, shifting the focus from single applications to an application domain. We noticed some difficulties in carrying out the activities of domain analysis and modeling, but it is difficult to understand if this is due to the novelty of the technique or to its complexity. The adoption of a modeling tool was lead by an analysis of the tools available in the market, considering the specific context and needs of RESICO. We faced two issues related to the tool:

firstly, Visual Modeler does not support the collection of user requirements with Use Cases, and this is a serious limit in the first phases of the software life cycle (if we want to adopt the Use Cases model), when we must capture and formalize user requirements and the functionalities of past and present software applications. We experimented the development of Use Cases both with drawing tools and modeling tools; the main drawback is that these models were not linked to the successive phases of the development process, and they required to be maintained 'by-hands'.

Secondly, the tools currently in the market do not support the evolution and maintenance of the artifacts produced. Considering Visual Modeler, the analysis of the software system produces a design diagram based on object oriented methodologies (in the same way as reality is abstracted into ER-diagrams when working with relational databases). Through a semi-automatic mechanism, the design diagrams can be translated into Visual Basic code that can then be compiled and executed. When the diagram has to be changed (due to evolution, maintenance or correction issues) there is not a mechanism for tracing these modifications and, perhaps more important, it is impossible to reflect them into the code in a semi-automatic way. It could be very useful, in particular for maintenance purposes, to have the story of the changes done and the decisions that led to the current solutions. Working at a higher level of abstraction than code, this requires tracing and documenting the evolution of the design rather than the modification to the pure code itself. We have to remember that among the goals of using object orientation and domain analysis there are also the capability to transfer into flexible object-component architectures the variability (see below for more details), encapsulating component behavior through interface standardization. These are particularly true in a software process that considers interation and cycles, not the traditional waterfall model. Reusable assets are the result of several refinement steps and evolution and then they are subject to evolution and maintenance.

These aspects will be deepened inside RESICO in order to study possible alternative solutions, also in consideration of the increased experience in object orientation and domain analysis.

Considering the business objectives we planned to reach at the beginning of the experiment, we can say from a qualitatively point of view that we reached them and we reported a positive feedback from all the people involved in the experiment (from top management, too).

We found it difficult to quantitatively judge the success of the experiment because of the difficulties in measuring and comparing data with past projects and considering that domain analysis is a cross-process activity that is very tightly coupled with the software development process and it affects all the projects of the organization. Product lines and reuse introduce additional complexity in the software process: we revealed that the impact that domain analysis has on the software process and the organization as a whole is quite relevant and a successful introduction requires accurate planning and a general training and organizational support. Domain analysis can alone lead to measurable benefits (in the medium and long term), but it is best viewed as a step in an improvement movement toward the concept of software factory exploiting reuse and production through composition of assets.

Considering the satisfactory results achieved, we are integrating domain analysis in the standard software development process and in the quality procedures (as required by the ISO 9001 certification). In the future, RESICO intends to go on with the improvement program and to promote further process improvement actions, experimenting innovative technologies and software engineering methods to pursue a reduction of the time-to-market and development costs and to be able to efficiently cope with rapid market evolution and changed customer requests. The next steps will be targeted to the introduction of a complete reuse policy, exploiting the experience we made in the DOOM project toward a software process centered around a compositional model based on software reuse.

6 REFERENCES

[1] Guillermo Arango, *Domain Analysis Methods*, in Software Reusability, ed. W. Schaeffer, R. Prieto-Diaz and M. Matsumoto, Ellis Horwood, New York, 1993.

[2] Hewlett Packard, Matra Marconi Space and CAP Gemini Innovation, *Domain Analysis Method*, Deliverable D3.2B, PROTEUS ESPRIT project 6086, 1994.

[3] J. Hess, S. Cohen, K. Kang, S. Peterson, W. Novak, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical

Report CU/SEI-90-TR-21, Software Engineering Institute, November 1990.
[4] A. Valerio, Assembling successful software products through domain engineering and market driven varints, PhD thesis, University of Genova, 1998.
[5] R. Prieto-Diaz, *Domain Analysis: an Introduction*, in ACM SIGSOFT - Software Engineering Notes, vol. 15, no. 2 (47-54), April 1990.

New product line approaches

A Product Line Process for the Production of Platform Software at Bosch

John MacGregor
Corporate Research and Development
Robert Bosch GmbH
Theodor Heuss Allee 70
60486 Frankfurt, Germany
+49 69 7909 532
john.macgregor@bosch.com

ABSTRACT
Product development based on the Product-Line Approach promises substantial improvement in productivity through architecture-based reuse. Robert Bosch GmbH, along with Thomson-CSF (France) and the European Software Institute (Spain) is participating in a joint research project funded by the European Union to validate the approach in industrial settings. The industrial partners; Bosch and Thomson, sought out suitable development areas within their companies and started to apply the approach experimentally. Among the topics examined were product line scoping, domain analysis, feature modelling and product line architecture.

The focus of this paper, however, is on the processes necessary to operate and maintain the product line.

The product examined was a platform, which in turn, part of other, quite unrelated product lines. The fact the applications could execute individually or simultaneously on the platform significantly affected both the architecture and development process of the target product line as well as the production processes of its internal customers.

KEYWORDS
Product Line Process, Organization

1 INTRODUCTION

Technical

This paper focuses on a domain called Car Periphery Systems (CPS), which combines all products which use, or could use, radar or ultra-sonic sensors to detect objects in the immediate vicinity of or approaching a vehicle.

There are many conceivable applications of this technology, but for expository purposes only 3 applications will be mentioned here:

Stop and Go (S&G): maintains a predetermined minimum distance to vehicles to the front by accelerating and decelerating appropriately. This is a specialization of automatic cruise control (ACC) for low-speed situations such as traffic jams.

Parking Assistance (PA): displays the distance to the nearest obstacle to the front, back or to the side (either side) when the vehicle is driving slowly and sounds an alarm when a predetermined minimum distance is reached. It is directed at areas that are normally obscured to the driver.

Pre-Crash (PC): scans objects that intrude certain distance and direction envelopes while the car is underway. The relative velocity of the object is calculated after it penetrates the outer envelope. When the object penetrates the inner envelope, the sensitivity of the airbag acceleration sensors (which release the airbag) are adjusted to react to impacts that would normally be ignored as normal driving impacts. This allows the airbag to release earlier, which in turn allows a softer activation of the airbag.

As can be seen in Table 1, these 3 applications vary significantly in their information requirements with respect to range, direction and characteristics of the objects monitored.. They also differ with respect to the number of objects to be monitored, sample frequency and sensitivity. There are corresponding differences in their requirements with respect to the number, type and location of the sensors.

Note also that these applications are sometimes mutually

	Range	Information		
		Distance / Displacement	**Speed / Direction**	**Acceleration**
S&G	Front .2m – 7m	Y/Y	Y/N	N
PA	Front, Back & Side .25m – 2m	Y/Y	N/N	N
PC	Front .6m – 1.5m	Y/Y	Y/Y	Y

Table 1: Demand Comparison for Selected Applications

exclusive. Stop and Go, for example, is irrelevant during a parking maneuver.

There is one simple major constraint. There is a limit on the number of sensors that can be mounted on the vehicle. All applications that could conceivably be installed on a vehicle cannot have their own set of sensors. Over and above that, it is simply sensible and economic to share the sensors.

This consideration led to the definition of a sensor platform, that would deliver peripheral object distance / direction / speed / acceleration information to client applications as needed.

Organizational

The sensor products mentioned are usually developed by separate development units. Although these products are usually developed using proven methodologies, each product conceivably represents a separate product line. The development of a common sensor platform is currently, at most, the subject of research or prototypical development. This platform also represents a product line, entirely with internal customers, though.

As such, there could be a series of sensor product lines; one for each end application domain. Given that, even though the requirements on the platform all have thematic similarities, signal processing capacity constraints dictate that the services supplied by the sensor platform be tailored to each application. The sensor platform +will therefore be end-product-specific *as long as only one end-application is operating in the vehicle.*

The sensor products are sold to automotive manufacturers and in the following sense, these customers are also the platform's customers, albeit once removed. In the medium-to-long term, it is certainly possible, when not likely, that they may want to install multiple applications in one vehicle type.

Each domain, the applications and the platform, has its own base of expertise. It takes a long time to acquire the technical knowledge, the context knowledge (regulatory constraints around the world, for example) and the skills necessary to develop the applications. Duplicating this knowledge in a company would not only be protracted, when possible considering the labor market, it is expensive and inefficient.

Because each development unit is organized to focus on its end-product spectrum it is neither practicable nor sensible for a single development unit to undertake all platform development or conversely for platform development to undertake the development of all applications.

At this point in time it is not clear whether the automotive manufacturers will coordinate their requests for integrated CPS systems internally or whether they will prefer to continue dealing with the individual development units. Internally, the customer requests must be distributed to the individual development units in order to assess the impact of each end application and the end applications' requirements on the platform must be gathered again to be assessed by the platform group.

Praise –validation of product line based reuse in an industrial setting

The product line approach is being applied experimentally in industrial settings in the EU project PRAISE. The project consists of three phases. In the first phase, the base methodology was defined. In the second phase this base methodology was applied to realistic development projects by the industrial partners. In this case, it was applied to the sensor platform development. In the third phase, the validated base technology will be packaged in a form suitable for use by practitioners.

To date, the local experiment has derived important platform requirements from the end application requirements, produced a feature model and a first architectural outline for the platform.

The production process definition began at the same time as the architecture was being developed. This paper documents the considerations made and the findings so far.

Terminology

It is important to differentiate between the terms that will be used in this report as situation it describes has a plethora of applications.

In the context of ***PRODUCTS***: The **Platform Application** manages the sensors and provides distance information. The **End Applications** involve feature sets that are seen by the end customer. Stop and Go, Parking Assistance and Pre-Crash are all end applications. The **End System, or System** is the integrated package comprising the platform and end applications that are installed in a single vehicle. The system software may all run on one controller, or may

be distributed.

In the context of ***PRODUCT LINE***: There are the **Domain and Application Processes** which produce reusable assets and commercial products respectively. All of the products above have domain and application processes. (The end system does not have a domain process because it is merely the integration of the other products and its form is dictated by the platform). For example, the platform domain process develops platform assets. The system application process develops sensor product packages that are installed in vehicles.

From the ***Platform Group*** perspective, the **internal customers** are the development units that produce the end applications and integrate the system. The **customers** are direct customers that install the sensor systems in their vehicles.

2 SCOPE

Product Line Process for a Sensor Platform

The original goal was to define the platform processes and the initial assumption was that only one end application would run on a platform. The requirements for the platform had to be derived from the existing products' requirements. This, in turn, led to the consideration of how the platform would evolve and to the realization that the architecture and processes of the end applications had to be understood and considered as well, especially when multiple end applications were involved.

Focus: End System and Platform

In order to properly understand the platform processes and its delimitation, the system application process (for systems with multiple end applications), actually an external process, had to be examined in detail.

Focus: System, not Hardware-Software-System Development

Although a deeper level of abstraction will be addressed later, the current study addressed only the process for developing the whole system. Actually, perhaps in contrast to other software domains, there are 3 separate aspects to the development: hardware, software and algorithm. A significant part of the development effort is related to analyzing the physics underlying the applications and developing appropriate measurement and interpretation strategies.

Focus: Steps, not Methods

At the level of detail reached during our project, it was sufficient to describe **what** was to be done and in which order, not the **how**.

Similarly, at this level, roles and entry/exit criteria were also unnecessary.

3 APPROACH

The study defined the Platform product line process. It synthesized the constraints inherent in the technological and organizational environment with standard company development processes to produce a process suitable for architecture-based reuse according to the product line approach.

Based on Company Standards and Best Practices

As far as possible, the processes defined should resemble processes that are being used in the development units.

There are company-wide standards for process planning and monitoring and for quality assurance. It was a goal, insofar as they were relevant to the processes under consideration, to incorporate the best practices already established in the development units

Operational, not Strategic or Tactical

There is no lack of processes in this undertaking. Product line encompasses organizational processes (product planning and marketing, for example), development, maintenance and technical processes to define an architecture and develop assets.

Although there is a tendency to think of product line top-down, the organizational impact, and the operational feasibility, (bottom-up from the produce line perspective,) had to be established at an early stage in order to define the domain context.

The focus of this paper is on the processes necessary to operate and maintain a product line. That is, there is a process to define the architecture and develop the assets: domain scoping, requirements analysis, feature modeling, architecture design, and so forth. It results in a product line infrastructure and there is only one cycle in the lifetime of a product line. This process is only a process in the sense that it is repeated over many product lines. There is the process of producing products using the assets. This process cycles once per product, that is, many times in the lifetime of a product line. This paper addresses this latter process.

Steady-State
It is conceivable first, that in the short term, multiple applications would not be installed on the platform.

Secondly, it is also conceivable that the infrastructure may not be complete as the architecture is first used.

Be that as it may, the study assumed a mature infrastructure and multiple applications in order to define the process for the long-term. Should intermediate states exist during the implementation of the process, then the definition should serve as a template and a target which allows a quicker transition to the steady state of the process.

Framework, Not Kit
Packaging to the internal customers was also a major consideration. On the one hand, it was possible to lay the structure and interfaces of the platform bare. With this scenario, the end-applications would have been free to customize the platform according to their needs, inheriting predefined classes or overloading methods as necessary; in effect using the platform as a toolkit.

On the other hand, it was possible to offer the platform as a fixed set of functionality, that covered a certain application sphere. The platform would not be modifiable, and with severely limited parameterability (i.e. limited to dropping functionality)

Some applications require information in addition to the sensor information, such as the state of the gear selector (in reverse for parking assistance) or the absolute vehicle speed (relative to the ground, not to surrounding objects that might also be moving). The idea of expanding the definition of the platform to include this additional information was considered, but rejected.

In the end, it was decided that the platform would be offered as a closed system, not as a toolkit, after it became clear that multiple applications would run on the platform. The potential that modifications undertaken by one application group would adversely affect other applications could not be eliminated or controlled.

4 END SYSTEM PROCESS

The nature of the platform is that one end customer, and possibly a number of intermediate customers (development units) share a single platform. This makes itself evident through a more elaborate process to coordinate and arbitrate the requirements for the platform. Ultimately, all requirements from all stakeholders must be clear before work can begin on the platform, and therefore, somewhat contradictorily, on any of the end applications.

This constellation dictates an iterative process requirements / product-concept process and in turn appropriate formalization. In effect, the planning processes of the platform's direct customers are formally drawn into the platform's planning process to a greater extent than for a typical producer/consumer relationship.

Figure 1. illustrates the end system process. The viewpoint taken here is that of the platform process. That is,

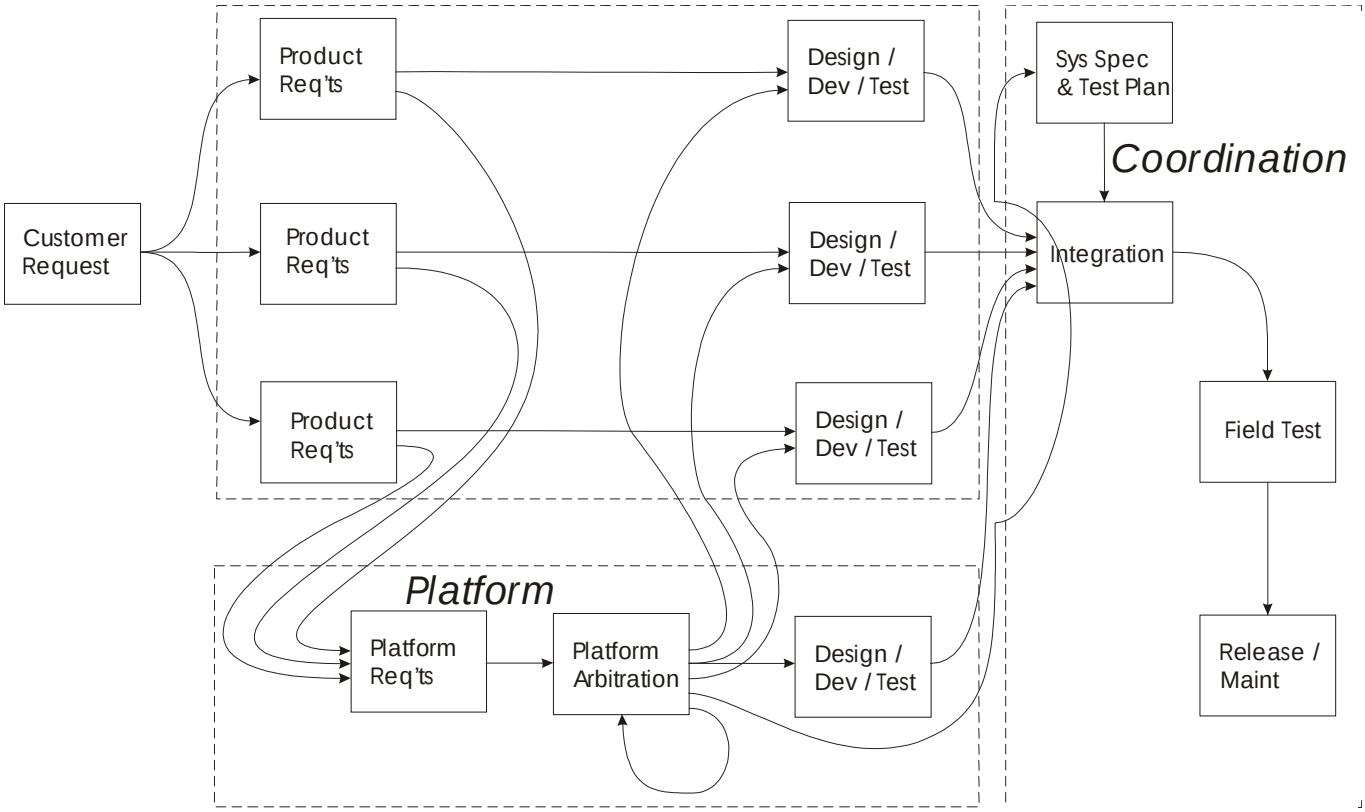


Figure 1: End System Process

extraneous details in the end-application processes have been omitted. As long as the end applications respect the interfaces and synchronization points defined with respect to the platform, the actual nature of their processes is not particularly relevant.

3.1 System Process

The process diagram is idealized in that its presentation suggests that the parallel steps execute in parallel. In reality, it is neither possible to require the customer to state his complete intentions with respect to sensor applications simultaneously, let alone deliver the complete requirements for each application nor to expect that all participating end application development units would take the same amount of time to process those requirements. This means that all requirements on the platform will not be ready at the same time.

Given the distributed nature of the end applications, and the fact that specialists are required to develop each of them, it is only sensible to develop the end applications in the corresponding development units. The customer requests must be coordinated and the end applications must be integrated after they are developed. The coordination activities were therefore simply assigned to a coordinating group for expository purposes.

Customer Request
A new product request may come from a specific customer or may be generated internally. The request, when it affects more than one end-application, may be coordinated by the customer and delivered separately to the different development units or coordinated by an internal organization.

The nature of the initial request may be open-ended or specific, incomplete or complete. When the request is deemed to be complete, it is reviewed by all participating development units for completeness and feasibility. After approval, the system requirements are split for / tailored to the respective development units and distributed.

End Product Requirements Analysis
The end applications determine the features necessary to fulfill the requirements and develop the concepts for their individual products.

Each concept consists of a specification of the (product-level) algorithm to be used to fulfill the product requirements, the choice of sensor profile and information over special operating conditions.. When complete, the sensor-related parts of the end application concept are bundled and sent to the platform group.

Platform Requirements Analysis
The platform group gathers all sensor platform requirements for a particular undertaking and analyses them when they are complete.

Similarly to the end application groups, the platform group must synthesize an overall concept for the instance of the platform being considered. Again, this involves developing the appropriate algorithms to accomplish the sensing task, but it also includes identifying conflicting and overlapping requirements from the various applications.

This step produces a list of conflicts between the applications that must be resolved.

Platform Arbitration
In the case where conflicts occur, alternative schemes and trade-offs must be found so that the platform requirements are feasible. Since this cannot be done in a vacuum, all stakeholders in the initiative must actively participate in the arbitration process.

Regardless of whether the problems are solved by negotiation or edict, this step produces systems and sensor platform concepts that are accepted by all participants.

The overall platform concept must then be tailored to the requirements of the individual end applications. In the end, the platform group delivers “made to measure” solutions to each end application.

Final System Specification & Test Plan
After the system concept has been finalized, the corresponding system specification must be prepared and presented to the customer.

The system specification can then be used to develop an integration plan, an integration test plan and field test plan.

Design / Development / Test
At the overview level, the design, development and test phases are similar between the application developers and the platform developers. Each develops a design and a test plan for its application area, develops the development and test environments, develops the its application and tests it until a satisfactory quality has been established.

The platform group may have the additional task of providing prototype platforms for use as the basis for application development.

The process mechanisms will be discussed in detail in a subsequent section.

Integration
The functionality of the platform is verified independently using the platform’s test environment. End applications are added to the platform and are tested for basic functionality in the integration test environment until all applications are present. The test suite based on the system specification is performed under artificial, but demanding, conditions.

Field Test
The system is installed in a vehicle and tested as defined previously.

Release / Maintenance
The system subjected to an acceptance test by the customer,

released for series production. Maintenance occurs as necessary.

5 PLATFORM PROCESS

The considerations of production and maintenance are critical to understanding the nature and extent of the processes in a running product-line.

Assets are not created, or at least not adapted for generality in time-critical situations such as production or customer projects. rather in special projects that are isolated from normal production activities: hence the domain process. The converse of this is that while the product-line application process is running, assets already exist and the domain development process is in maintenance mode. That is, the activities in the domain process during this time are generally limited to fixing bugs in the assets, adding functionality to the assets or adding assets related to domain evolution.

A special consideration comes from the fact the domain under consideration is electronic control units. Typically, once the control units are in the field or in series production at a customer installation, the possibilities to fix bugs are very limited and expensive. Therefore, practically all bugs are identified and fixed during the integration, field test and customer acceptance testing phases. This means especially in the case of the platform, product maintenance occurs practically only until the end system, has been accepted by the customer. In other words, the maintenance phase is subordinate to the development process of the overriding product.

In summary, while a platform product-line is in its operative phase, domain asset development is vestigial, domain asset maintenance is predominant among domain activities and application maintenance only occurs practically while the end applications are developed, integrated and tested.

Platform Process Overview

Figure 2 illustrates the basic processing elements and information flows in the Platform Process. Note that both the domain and application aspects are, perhaps, typical for a product-line process.

The platform application process, outside of maintenance, illustrated in the lower half of Figure 2 is contained in the box labeled “Design/Dev/Test” in Figure 1. That is, after the system process has determined all the tradeoffs between the end-applications, developed a concept for the entire system and derived the requirements on the platform, the preconditions necessary to develop an instantiation of the platform product line have been achieved.

The illustration presents an operative, rather than a logical, view of the process. The two tiers therefore have management and productive aspects which will be described in greater detail in subsequent sections.

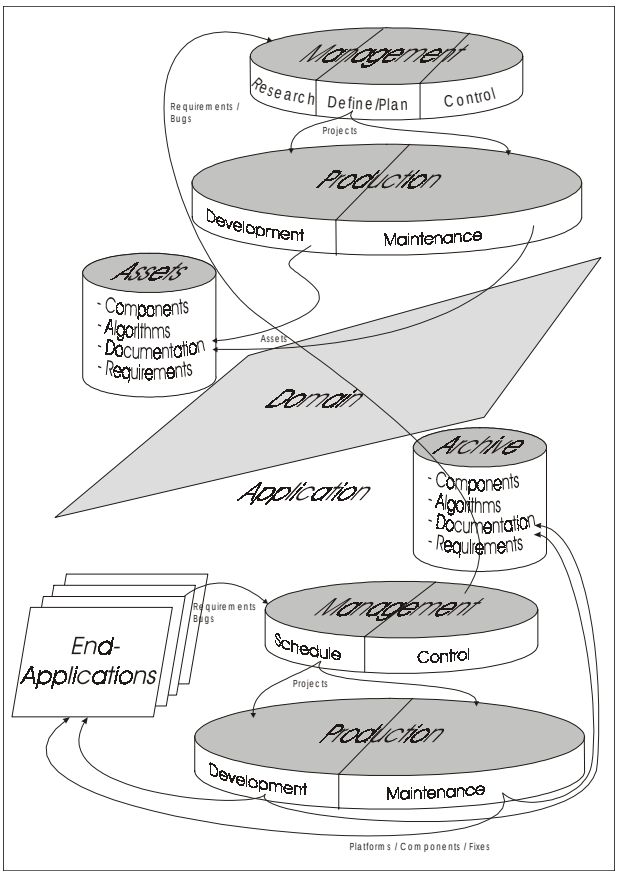


Figure 2: The Platform Process

It is perhaps easier to start with the application process, as everything in the domain-engineering process should just exists to support it.

The Platform Application Process

The logical relationships between the Application Process activities are illustrated in Figure 3. It is expected that the hardware basis of the platform will remain stable, or at least, will not be developed as part of the (customer project) application process. At this level of abstraction, each step contains both algorithmic and software development portions. Their sequencing is neglected.

Platform Application *Management Process*

The Platform Application Management Process is concerned with the planning and control of the tasks necessary to produce the platform. Planning involves the definition, estimation, risk analysis and scheduling of the tasks. Control involves continually assessing the risk of projects that are running and monitoring their progress with respect to the forecast schedule, expenditures and quality.

Assessment

After the arbitration process has decided the functional characteristics of the platform, the asset repository is searched for similar platforms. The candidates are identified based on the scheme the end-applications use to

specify their requirements; that is displacement/velocity/acceleration, sensitivity, timeliness and sequencing.

The amount of rework necessary for the candidates is compared to find the best basis for development project being considered. The amount of work necessary to produce a new platform from scratch is also calculated, if needed.

Scheduling
A project manager is assigned to the project. Depending on the size of the project, the project manager or a project team plans the project with respect to:

- Task Structure
  The nature and extent of the steps necessary for software and algorithm development are defined.. The nature and extent of intermediate products, their versioning and storage are defined. Finally, the personnel and resources necessary for the performance of the tasks are defined
- Capacity
  The personnel and resource availability is compared to the task structure.
- Milestones
  The sequence of the tasks, accounting for possible resource constraints, are defined.
- Risk Identification
  Risks are identified and plans for their recognition and redress are defined. At least the following aspects are considered:
  ↘ New Development
  ↘ Product or process changes
  ↘ Conformance to safety standards / requirements
  ↘ Changes in / enhancements to customer requirements

The project plan defines checkpoints and expected values to control the progress of the project with respect to expenditures, quality, and task completion.

The resources available are compared with the resources needed and the priority of the task. The resources are assigned to the project for a particular timeframe.

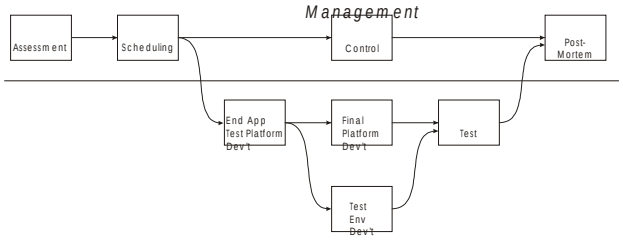


Figure 3: Platform Application Process

Control
The control phase lasts as long as the project runs. A control committee composed of the platform project leader, representatives of the end application projects and the upper management of the platform group meet regularly to review the project progress.

The predicted expenditures, quality and task completion dates are compared to the actual values. The list of potential risks is reviewed to see if any of the foreseen events have occurred and the current situation is assessed to ascertain if any unforeseen risks have appeared.

Should the situation warrant, the possible response alternatives are investigated by a team led by the project leader and their impacts are assessed. Measures with larger impacts are approved by the control committee.

Post-Mortem
In the post-mortem phase, the performance of the project is assessed and critical factors (to the success or failure) of the project are identified.

The appropriate requirements, design and test documents are archived along with the source code, components, development and test environments.

Should the product, its components or any part of the associated records have a remarkable character or should the project's course have deviated significantly from the predicted course, the facts are gathered and a report is sent to the domain group.

Platform Application Production Process
At the current level of abstraction, the application production process contains a set of steps that are largely logically independent of another. It is not necessary that all end application development platforms be finished before work can start on the final platform, if indeed, they are necessary at all in the project under scrutiny.

Similarly, there is an implicit parallelism at the next deeper level of abstraction. That is, each step contains algorithmic, software, and possibly hardware, aspects. The algorithm must be defined before the software can be completely developed, but which must be started first is a scheduling matter.

All components are defined in UML and the software is written in code frames that are generated from the UML.

All development phases have a design phase, a coding phase and a test phase.

In the design phase, the concrete implementation of the platform in question is planned. Alternatives within the defined system concept are explored. The phase ends after a successful review among the platform architects and any affected end application architects. The design the review

minutes and supporting documents are archived.

The test plan is developed in the design phase and is again reviewed, as necessary, by the platform architects and any affected platform architects

The coding phase contains code reviews. It ends when the programmer delivers his code for independent testing.

Testing is performed independently of the development group. There is a change management system with a change control board to oversee the error management process.

End Application Test Platform Development
When required, platforms are developed or adapted to allow individual end applications to develop without having to wait for the development of a platform that satisfies the requirements of all applications. In contrast to the final platform, these platforms are customized or pruned to the requirements of the specific end-application being developed.

Final Platform Development
The integrated platform needed to support all end-applications simultaneously is developed in this phase.

Final Platform Test Environment Development
Appropriate measures are taken to build a test bed if it is needed to simulate the demands and load of the various end applications to be supported by the platform.

Test
The tests planned in the design stage are performed on the product using the test bed developed for that purpose. The test results are archived to provide audit trails required by some customers and by some safety standards or safety laws.

Platform Application Maintenance Process
As mentioned previously, the maintenance phase of the platform application occurs mainly while the system is being integrated and tested. As such, it would be the change management process of the system integration and release process performed by the coordination group which is illustrated in Figure 1.

From the platform product-line perspective, the only aspect of interest is that the error reports provide some indication of the reliability of the assets. As these errors are found after the release of the corresponding platform, they are the quality indicators of the platform application development process and therefore part of its control process.

The errors are therefore classified by asset and platform and the results are integrated into the asset base.

The following steps are involved in identifying and fixing a system defect:

Investigation
The initial problem report is investigated and the descriptions of the conditions under which the problem occurs and the nature of the problem are checked for completeness.

Estimation
Alternatives are identified to solve the problem. The steps required to implement the best alternative are defined and the corresponding expenditures, impacts and risks are estimated.

Scheduling
At this point at the latest, the criticality of the problem is specified. Irrelevant problems are rejected and relevant problems are assigned a priority. Depending on the priority of the problem, it is scheduled either to be fixed in the current release or in the corresponding asset.

Depending on the size and priority of the problem, when the problem is designated to be fixed in the current release, it is either fixed immediately or grouped with other similar problems and fixed in batch.

A project manager is assigned and the task is planned with respect to activities, capacity, milestones, risks, and expected progress as defined in the production scheduling description above.

Development, Test, Release
The problem is fixed. The platform is tested in its test environment. Each of the end applications is tested through repeating the integration test, except where the problem is unambiguously trivial.

The Platform Domain Process
After the product-line process is fully introduced, the asset base must merely be maintained. The activities in the domain process during this time are generally limited to fixing bugs in the assets, adding functionality to the assets or adding assets related to domain evolution.

In comparison to the major structure of the application section, the production process has been dropped, the maintenance process has been kept and the control process is dedicated to controlling the maintenance activities.

The overall platform domain process is illustrated in Figure 4

Platform Domain Management Process
Research
Research is responsible for investigating the need for change in the assets. As such, it is mainly a marketing

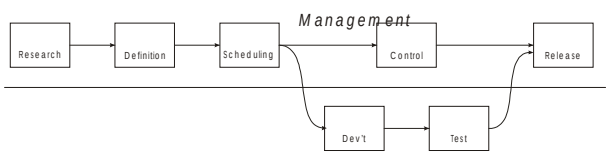


Figure 4: Platform Domain Process

activity. There are two basic modes: pro-active and reactive.

The proactive activities involve searching for trends in areas extraneous to the platform group. These include monitoring:

- Technological Trends
  These include improvement in hardware components, algorithmic improvements and improvements in software technology.
- Requirements Trends
  This done through interviewing the end applications' planning department and conducting interviews with external customers.

The reactive activities involve assessing the information flowing back from the application process. The sources of this information include:

- Platform Bug Reports (from the end applications or from the field)
- Problem Reports from the platform application development group
- Project Post-Mortems
  In this respect the research group is responsible for identifying and promoting the best development practices from the platform development projects.

Research is responsible for analyzing the information, identifying similarities and trends, defining the requirements, considering the profitability aspects of the enhancement and assigning a priority to it.

Definition
After a need for a change in the asset base has been established, the alternatives for achieving this change are identified. The best alternative is selected and a task definition is prepared that specifies the steps necessary.

Scheduling
A project manager is assigned and the task is planned with respect to activities, capacity, milestones, risks, and expected progress as defined in the application production scheduling description above.

Control
The progress of the project and the state of the environment is monitored for risks as described in the application production control description above.

Release
The enhancement is placed in the asset base, along with the appropriate documentation. If appropriate, the platform application group is notified of its existence.

Platform Domain Maintenance Process
Development
The task is performed and the asset is enhanced or a new asset is created.

Test
The platform is tested in its test environment. Each of the end applications is tested through repeating the integration test, except where the problem is unambiguously trivial.

6 SUMMARY AND CONCLUSIONS

After making suitable restrictions and the relevant factors we have modeled the production process for our product line in the following steps:

- Investigate current practice in development unit
- Assume steady state
- As a first cut, concentrate on the steps involved at the system level rather than hardware or software process.
- Define the target process's relationships to its overriding process.
- Define the application process
- Define the domain process

In general, production process definition is intimately related to the products and organizations involved. Aspects that may differentiate our approach from others include:

- Consideration of the project management aspects
- Consideration of the influence and subsequent modeling of the development processes of the target product line's customers
- Consideration of the asset maintenance aspects.

Insights gained from this exercise include:

- The assets are primarily in the maintenance phase during production
- The algorithmic feasibility aspects force an arbitration mechanism when multiple applications run on a platform

7 INFORMATION AND QUESTIONS

For more information, contact the author John MacGregor (john.macgregor@bosch.com).

ACKNOWLEDGEMENTS

I would like to thank my colleagues, the EU and my colleagues at PRAISE .for their support and help.

REFERENCES

[Clements, 1999] Clements, Paul A Framework for Software Product Line Practice *SEI Technical Report*, 2 (Winter 1992)..

[Jacobson, 1997] Jacobson, I., Griss, M., and Jonsson, P.: Software Reuse – Architecture, Process and Organization for Business Success. Addison Wesley Longman, 1997. ISBN 0-201-92476-5.

[Karlsson, 1995] Karlsson, Evan-André, Software Reuse – A Holistic Approach. Wiley, 1995, ISBN 0-471-95819-0

[Weiss, 1999] Weiss, David, Lai, Chi Tau Robert, Software Product-Line Engineering: A Family-Based Software Development Process. Addison Wesley Longman, 1999, ISBN 0-201-69438-7

A Framework for Software Product Line Practice

Paul C. Clements
Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213
+1 512 453 1471
clements@sei.cmu.edu

Linda M. Northrop
Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213
+1 412 268 7638
lmn@sei.cmu.edu

ABSTRACT
Software product lines are emerging as a new development paradigm for software-intensive systems, but before an organization can successfully adopt the product line paradigm, there are organizational as well as technical hurdles to be surmounted. This paper introduces a framework for product line practice that consists of a set of practice areas in which competence must be gained before a successful product line capability can be achieved.

Keywords
Software product lines, software reuse, core assets, framework for product line practice

1 INTRODUCTION

Software product lines are emerging as a new and important software development paradigm. Companies are finding that the practice of building sets of related systems from common assets can yield remarkable quantitative improvements in productivity, time to market, product quality, and customer satisfaction. Organizations that acquire, as opposed to build, software systems are finding that commissioning a set of related systems as a commonly-developed product line yields economies in delivery time, cost, simplified training, and streamlined acquisition. But along with the gains come risks. Although the technical issues in product lines are formidable, they are but one part of the entire picture. Organizational and management issues constitute obstacles that are at least as critical to overcome, and may in fact add more risk because they are less obvious.

At its essence, fielding a product line involves (1) *development or acquisition of core assets*, which are software, document, process, and management artifacts engineered to be re-used, and (2) *development or acquisition of products* using those re-usable core assets. These two activities can occur in either order (new products are built from core assets, or core assets are extracted from existing products). Often, products and core assets are built in concert with each other. Core asset development has been traditionally called domain engineering. Product development from core assets is often called application engineering. The entire effort is staffed, orchestrated, tracked, and coordinated by management. Figure 1-1 illustrates this triad of essential activities. The iteration symbol at the center represents the decision processes that coordinate the activities. The bi-directional arrows indicate not only that core assets are used to develop products, but that revisions to or even new core assets might, and most often do, evolve out of product development. The diagram is neutral about which part of the effort is launched first. In some contexts, already-existing products are mined for generic assets—a requirements specification, an architecture, software components, etc.—that are then migrated into the product line's asset base. In other cases, the core assets may be developed or procured for later use in production of products.

Organizations that have succeeded with product lines vary widely in the nature of their products, their market or mission, their organizational structure, their culture and policies, their software process maturity, and the maturity and extent of their legacy artifacts. Nevertheless, there are universal essential activities and practices that emerge, having to do with the ability to construct new products from a set of core assets while working under the constraints of various organizational contexts and starting points.

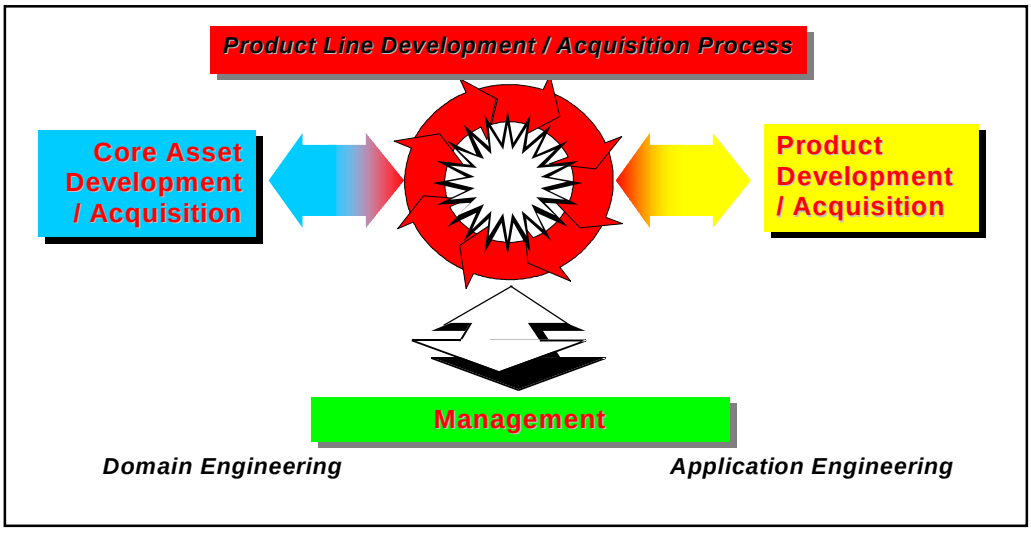


Figure 1-1: Essential Product Line Activities

The Software Engineering Institute has developed a framework[1] for product line development and/or acquisition. It describes these essential activities and practices, in both the technical and organizational areas. These elements and practices are those in which an organization must be competent before it can expect to successfully field a product line of software or software-intensive systems. The audience for the framework includes members of an organization who are in a position to make or influence decisions regarding the adoption of product line practices as well as those who are already involved in a product line effort. The goals of the framework are

- to identify the foundational concepts underlying software product lines and the essential activities to consider before creating or acquiring a product line
- to identify practice areas that an organization creating or acquiring software product lines must master
- to define practices in each practice area, where current knowledge is sufficient to do so
- to provide guidance to an organization about how to move to a product line approach for software

An organization using this framework should be able to understand the state of its product line capabilities by (a) understanding the set of essential practice areas, (b) assessing how practices in those areas differ from their conventional forms for single product development, and (c) comparing that set of practices to its existing skill set. As such, this framework can serve as the basis for a technology and process improvement plan aimed at achieving product line development or acquisition goals. There is no one correct set of practices for every organization, but this document contains practices that we have seen work successfully in practice.

This paper outlines the practice areas described in the framework, which we believe are essential for an organization that aspires to be successful in product line practice.

[1] The framework may be found on the web at http://www.sei.cmu.edu/plp.

2 PRODUCT LINE PRACTICE AREAS

As a way of organizing the practice areas for easier reference, we divide them loosely into three categories: *software engineering practices*, *technical management practices*, and *organizational management practices*.

Software Engineering Practice Areas

Software engineering practices are those practices necessary to apply the appropriate technology to create and evolve both core assets and products.

Understanding Relevant Domains

Domains are areas of expertise that can be applied to the creation of a product line. Domain knowledge is characterized by a set of concepts and terminology understood by practitioners in that area of expertise. It also includes an understanding of recurring problems and known solutions within the domain. Knowledge from several domains is usually required to build a single product (e.g., knowledge of user interfaces, database management systems, and networking are needed to

create a distributed banking application; this expertise is in addition to the fundamental financial knowledge needed for banking). The motivation to apply such domain knowledge in a product line context usually begins with the recognition of an opportunity to satisfy business goals by exploiting the reusability of an organization's expertise. The journey from the initial recognition to the creation of a set of reusable core assets requires an ability to understand and package the relevant domain information for use and reuse across the entire product line. The practice of understanding the relevant domains therefore encompasses the following responsibilities

- identifying the areas of expertise—domains—that are useful for building the product line
- identifying the recurring problems and known solutions within these domains
- capturing and representing this information in ways that allow it to be communicated to the product line stakeholders, and used and reused across the entire product line

Achieving this understanding is a fundamental prerequisite to the task of succeeding at large-grained systematic reuse. It is the basis for identifying opportunities to automate the creation of the product line. It also allows an organization to reason about the properties of the product line, the capabilities to be provided by the core assets, the range of variability in the products, and the processes, methods, and tools to be employed in the creation of the product line.

Mining Existing Assets

Product lines are seldom built as "green field efforts" beginning with a clean slate. Rather, systems in an organization's inventory are often used as the first members of the software family. These legacy systems therefore serve as the raw material for populating the new product line's core asset base with software and software-related artifacts. These certainly include code, but also test plans and cases and scaffolding, processes, schedules and budget estimates, and documentation of all varieties. This practice area describes the steps necessary to judiciously identify salvageable artifacts in an organization's legacy inventory and the steps necessary to efficiently re-engineer them for use in the new context of the product line.

Architecture Exploration and Definition

The software architecture for the product line may be the most important of all of the reusable core assets. The architecture must apply to all of the products in the product line, and enable every product to meet its (possibly differing) behavioral, performance, and other quality attribute requirements. The architecture typically accomplishes this by featuring built-in variation points, at which different components may be plugged in or at which different quality attributes (such as, for example, security) may be tuned for individual members of the family. This practice area describes the drivers that serve as requirements for a product line architecture, and steps and approaches for crafting such an architecture from the product line's business and technical goals.

Architecture Evaluation

Evaluations of the architecture both of the product line and specific products is a low cost risk reduction method for determining whether the systems in the product line will achieve the business and quality goals desired for them. A product line is intended to achieve particular business goals. Individual products in the product line are intended to achieve other business goals – presumably in harmony with the goals for the product line. Since the product line success hinges on the suitability or fitness of the architecture, its evaluation is an essential part of the product line construction, especially now that effective methods such as the Software Architecture Analysis Method [SAAM][2]and the Architecture Tradeoff Analysis Method[SM] [ATAM[SM]][3] are emerging. More approaches to analysis may be found in [11].

COTS Utilization

The software core assets for a product line include an architecture and a set of components that are consistent with the architecture. These assets may be derived from a number of different sources such as leveraging existing assets, new development, and the utilization of COTS (commercial off the shelf) or NDI (non-developmental item) items. COTS products exist independently of a specific product line, and they are developed by a

---

[2] http://www.sei.cmu.edu/architecture/sw_architecture.html

[3] Architecture Tradeoff Analysis Method[SM] and ATAM[SM] are registered service marks of Carnegie Mellon University. http://www.sei.cmu.edu/architecture/sw_architecture.html

commercial organization for commercial purposes. COTS products are available for purchase or licensing to the general public through a catalogue or price list. NDI products are relevant primarily within the context of government acquisitions. They have previously been developed by somebody else, and are available for other authorized projects, though not necessarily to the general public, or through a price list.

In traditional software development, each component is built from scratch to meet requirements. The economies of scale that accrue from product line development are further multiplied when COTS components are used. During the last decade, the explosion of middleware technologies and standards has enabled the use of COTS components on a large scale in system development to cut costs, leverage the advantages of a common architecture, and enable large scale reuse. However, when COTS products are employed, there is far less control over the ways in which the components fit into the architecture, and the evolution of the components. The utilization of COTS components in a product line system introduces a new set of issues, concerns, and tradeoffs. Because of the central importance of the architecture for product lines, and because of the need to fit a potentially large family of systems, a product line solution using COTS products needs to be generalized, involving general purpose integration mechanisms that span a number of potential products. As a result, the range of potentially qualifying products may be reduced, and the use of wrappers and middleware needs to focus on generalized solutions, rather than some of the more opportunistic solutions that may be appropriate for single product systems. The evolution strategy for updates, incorporation of new releases into product lines and need for rigorous configuration management all become more critical for a product line approach.

Software System Integration

Software systems integration refers to the practice of combining individual software components into an integrated whole. Software is integrated when components are combined into subsystems or when subsystems are combined into products. In the extreme, software system integration can be reduced to a step taking zero time when generating products in product lines where the variability is known a priori. This is called "system generation." In this case, developers produce an integrated parameterized template of a generic system with formal parameters. They then generate final products by supplying the actual parameters specific to the individual product requirements. The purpose of this practice area is to give a range of options for combining components to produce a system, and to show how software systems integration works for product lines.

Other Software Engineering Practice Areas

These practice areas will not be discussed here because of spaced limitations.

- Component Development
- Testing
- Requirements, Elicitation, Analysis, and Management

Technical Management Practice Areas

Technical management practices are those management practices necessary to engineer the creation/acquisition and evolution of the core assets and the products. Technical management practices are carried out in the core asset and product development/acquisition parts of the figure below. Technical management includes those practices that would be carried out by a systems builder regardless of whether or not software was part of the product.

Data Collection, Metrics, and Tracking

A measurement program is essential for managing a product line initiative and making decisions about success, failure, and continued funding. Adopting a product line approach often requires wholesale change throughout the organization, and it is essential that management be confident that the often painful changes are in fact yielding positive results; otherwise, there will be great temptation to abandon the effort and revert to the conventional one-at-a-time paradigm. Measurement provides the quantitative data necessary to make informed decisions to guide the product line program. A comprehensive set of metrics is essential for obtaining insight into the management and technical issues that are most important for achieving the business goals, improving assets, products, and processes, and determining the return on investment.

Product Line Scoping

Product line scoping defines the bounds for systems that will constitute the products in the product line. Scoping also establishes commonality and bounds variability for these systems that will then be developed from a core set of assets. The objective of product line scoping is to define the set of products for which commonality is sufficient to achieve economies of production and for which the ability to produce distinct systems is not compromised but rather is enhanced. In addition, the scope must satisfy business considerations such as the following:

- prevailing or predicted market drivers
- the nature of competing efforts
- other business goals that involve embarking on a product line approach, such as merging a set of similar but currently independent, product development projects

The scope defines the space of products that will be developed from core assets. It identifies the commonality that members share and the ways in which they vary from each other. The scope not only includes an initial set of products, but also an assumed set of products that have not yet been built or completely defined, but whose possibility is being considered or planned. In most cases, product line scoping must continue after the initial scope has been defined because new market opportunities may arise, and new opportunities for strategic reuse and merging of projects may make themselves known. Achieving the appropriate scope is critical. If the scope is too large, then the core assets must be engineered to be hopelessly generic, and the effort will fail. If the scope is too small, then only a few products can ever be built from the core assets, and the investment will not pay off.

Configuration Management

For every development effort of any size, whether a product or not, CM is an essential practice. For a product line, all constituent products are managed with a single unified CM context, whereas conventionally, each product with all its versions may be managed separately. Product line CM must account for the fact that assets are produced by one project and are used in parallel by several others. A primary goal of the product line CM is to allow the rapid reconstruction of any version of any product, which may have occurred using various versions of the core assets. CM must also support the process of merging results either because new versions of assets are included in a product or product specific results are introduced into the asset base. Finally, since introducing changes may affect multiple versions of multiple products, it is essential that the CM system delivers sound data for an impact analysis to help understand what impact a proposed change will have.

Technical Risk Management

Technical risk management is the practice of managing risks within a project. A complete risk management program should provide processes, methods and tools to identify and assess what could go wrong (the risks), determine what to do about the risks, and implement actions to deal with the risks. There are well-established approaches and guidelines for performing risk management that should be used as a starting point [1], [2], [3], and [10]. These approaches provide practices for risk identification, analysis, planning, tracking, and control. The SEI Continuous Risk Management (CRM) paradigm [4] is representative.

For risk management targeted to product lines, the general practices will be consistent with the standard approaches. However, specific practices will require more coordination because of the range of organizational groups that are involved in product line activities and the fact that product lines often represent a different way of doing business. The most significant difference between traditional and product line risk management programs is likely to be in the scope of the program. Traditionally, risk is managed on a project basis. In a product line approach the intimate relationship between product development and core asset development will require coordination of risk management among several projects or functions within the organization. This would necessitate special mechanisms to ensure coordination of risk activities.

Other Technical Management Practice Areas

These practice areas will not be discussed due to space limitations.

- Process Modeling and Implementation
- Planning and Tracking
- Make/Buy/Mine/Outsource Analysis

- Tool Support

Organizational Management Practice Areas

Organizational management practices are those practices necessary for the orchestration of the entire product line effort.

Achieving the Right Organizational Structure

Organizational structure refers to how the organization forms groups to carry out the various responsibilities inherent in a product line effort. All organizations have a structure, if only implicitly, that defines roles and responsibilities appropriate to the organization's primary activities. Particular organizational structures are chosen to accommodate enterprise goals and directives, culture, nature of business, and available talent. The organizational structure reflects the division of roles, responsibility, and authority.

A traditional (non-product line oriented) organization primarily manages software at the project level. Individual projects tend to be fully responsible for technical decisions affecting their products. Specialization might require each project to have its own development group, with no sharing of personnel among projects. The role of the organization's management is usually to support the projects, gather and allocate resources, and provide high-level oversight. In a product line context, however, the development and acquisition of core assets and products dictate different organizational structure that is not project-centric. In addition to identifying the right organizational structure for product lines, management must be concerned with identifying organizational charter and boundaries; identifying functional groupings; allocating and assigning resources; monitoring organizational effectiveness; improving organizational operations; establishing inter-organizational relationships; and managing organizational transition.

Operations

When an organization makes the decision to move to a product line approach for acquiring or developing software, it must decide on the day-to-day operational steps that must be taken for the product line to thrive. Who builds the core assets? How do the product-building groups send feedback about the assets' suitability? What are the roles and responsibilities for maintaining the product line? How will the architecture be developed and maintained?

The decisions regarding these and several other key questions establish the basis for a product line. As these decisions become operational, the organization establishes a process for fielding the product line. The definition, development, and maintenance of this process require creation of an operational concept in order to do the following:

- Describe the characteristics of the process for fielding the product line from an operational perspective. (Included in fielding are product line development or acquisition and product line sustainment throughout its life.)
- Facilitate understanding among stakeholders of the goals of this process. Stakeholders for the product line include developers and users of the products of the process.
- Form an overall basis for long-term planning for the product line and provide guidance for the development of specific product line outputs such as a Developers' Guide, Business Plan, Architecture, and other assets.
- Describe the organization fielding the product line and using its products.
- Define the role acquisition will play and solidify the general acquisition approach. Acquisition will include procurement strategies for asset development, product development, and/or needed contractual products and services.

As the product line is fielded, the operational concept provides a baseline when the organization considers alternatives in their approach as changing conditions warrant. The operational concept for a product line should be documented as a Concept of Operations (CONOPS). The CONOPS documents the decisions that define the approach and the organizational structure needed to put the approach into operation.

Training

Training is an important element of both the initial product line adoption strategy and the longer-term product line evolution strategy. The initial training occurs in the context of the organizational and cultural changes needed to support a product line approach; the follow-on training occurs in the context of the evolution of the product line or lines. It is management's responsibility to ensure that training is an integral part of the

organization's product line strategy. This section of the framework focuses on the training practices that need to be instituted by management to ensure that the organizational units responsible for creating, fielding, and evolving the product line have properly trained personnel.

Launching and Institutionalizing a Product Line

Launching and Institutionalizing a Product Line is concerned with how to introduce product line practices into an organization and change that organization's "sea course" so that it becomes an effective product line organization. Institutionalizing can be viewed as a special case of a technology change project. Technology change projects are highly dependent upon the context of the organization. Thus, it is usually inappropriate to prescribe an invariant sequence of steps to execute the project. Furthermore, successful technology change projects not only account for the specific technology involved but also account for the non-technical or human aspects of change.

Other Organizational Management Practice Areas

These practice areas will not be discussed here because of space limitations:

- Building and Communicating a Business Case
- Funding
- Market Analysis
- Training
- Customer Interface Management
- Developing and Implementing an Acquisition Strategy
- Technology Forecasting
- Organizational Risk Management

REFERENCES

1. Boehm, B.; IEEE Tutorial on Software Risk Management, IEEE Computer Society Press, Piscataway, NJ, 1989.
2. Carr, Marvin; Konda, Suresh; Monarch, Ira; Ulrich, Carol; & Walker, Clay. *Taxonomy-Based Risk Identification* (CMU/SEI-93-TR-6, ADA266992). Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University, 1993
3. Charette, R.; *Software Engineering Risk Analysis and Management*, McGraw-Hill, New York, NY, 1989.
4. Dorofee, A.; Walker, J.; Alberts, C.; Higuera, R.; Murphy, R.; Williams, R.*; Continuous Risk Management Guidebook*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.
5. Fairley, R.; "Risk Management for Software Projects." IEEE Software, May 1994: 57-67.
6. Gallagher, Brian P.; Alberts, Christopher J.; Barbour Richard E.; *Software Acquisition Risk Management Key Process Area (KPA) – A Guidebook* (CMU/SEI-96-HB-002). Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University, 1996.
7. Kirkpatrick, Robert J.; Walker, Julie A.; & Firth, Robert. "Software Development Risk Management: An SEI Appraisal." Software Engineering Institute Technical Review '92 (CMU/SEI-92-REV). Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University.
8. Myers, C.; Maher, J.; Deimel, B.; *Managing Technological Change*, Course materials. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992.
9. Radice, R.; Garcia, S.; "An Integrated Approach to Software Process Improvement." Tutorial presented at the Software Technology Conference, April 1994, Salt Lake City Utah.
10. Sisti, F., and Joseph, S.*; Software Risk Evaluation Method Version 1.0*, [CMU/SEI-94-TR-19, ADA 290697], Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1994. Available <http://www.sei.cmu.edu/publications/documents/94.reports/94.tr.019.html>
11. Zhao, "Bibliography on Software Architecture Analysis", Software Engineering Notes, vol. 24, no. 3, May 1999, pp. 61-62.

Product Line Process Framework: The Wheels process

Michel Coriat
Thomson-CSF / LCAT[1]
Central Research Laboratories
Domaine de Corbeville
91404 Orsay Cedex, FRANCE
+33 1 69 33 07 76
michel.coriat@lcr.thomson-csf.com

Frédéric Waeber
Alcatel / LCAT[1]
Corporate Research Center
Route de Nozay
91461 Marcoussis Cedex, FRANCE
+33 1 69 33 00 00
{frederic.waeber, sophie.veret}@alcatel.fr

ABSTRACT
The paper presents a process framework, called Wheels, for product-line development. This framework is called Wheels. Wheels is a tailorable process offering the ability and means to adapt to several situations (business, project management,…). Wheels is extensible and can be increased incrementally along feedback during usage. Wheels establishes relationships to ISO or IEEE standard documentation (SSS, SSDD, SRS) and CMM. To achieve these characteristics, Wheels offers three basic properties: formalization of the process using an UML [14] meta-model to increase the process understanding, usage of a matrix model to implement the tailoring, definition of a full handbook describing the process patterns.

The proposed approach is a part of an effort undertaken at the common laboratory of Alcatel and Thomson-CSF, the LCAT, for product line development. At the LCAT we are investigating a method called SPLIT (Software Product Line Integrated Technology) which is a global framework for engineering product line of software intensive systems. Wheels is a part of SPLIT.

Keywords
Product line process, flexibility, tailoring, extensibility, formalization, meta-model, UML.

1 INTRODUCTION

The product-line approach is driven by the PC&C (Produce, Consume & Customize) principle (figure 1). One process (domain engineering) aims at producing assets, such as requirements, product-line architecture, or product-line software components. The other one (application engineering) consumes and customizes assets produced

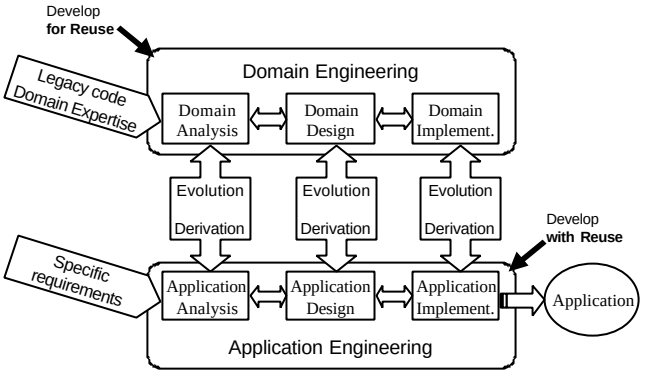


Figure 1 – The product line approach (inspired form STARS [1])

during domain engineering for application development.

Domain engineering objective is to develop assets for reuse, and application engineering is to develop with reuse. The product-line approach turns reuse from opportunistic to strategic (or systematic) reuse.

Flexibility is the key for product-line development in the sense that we have to study commonality and variability of assets. We must manage assets not only to support future variants and make assets feasible but also resilient to change.

To achieve these objectives we have adopted a process-driven and a model-oriented approach of product-line development.

Current processes such as V cycle, spiral model, waterfall process reached their limits for the product-line approach, firstly because of their rigidity (lack of capability to adapt itself to more and more demanding projects needs), and secondly because of their dogmatism (fixed procedural process, and mandatory stages).

In the product-line approach, the process is much more important since it assumes the coherence and the efficiency

[1] LCAT is the Alcatel/Thomson-CSF Common Research Laboratory

between the different items in Processes Engineering (Domain and Application) and guides the reusability of the work products produced during the process items. Moreover, the process allows the different project teams to work with each other in a same rationale, that's to assure the legitimacy of the product line approach to the organization (customer, manager or developer levels). Everyone uses the same language.

The proposed approach is a part of an effort undertaken at the common laboratory of Alcatel and Thomson-CSF, the LCAT, for product line development. We are investigating a method called SPLIT (Software Product Line Integrated Technology) [4] which is a global framework for engineering product line of software intensive systems. The Wheels approach is a part of the SPLIT method and focuses on the product line process.

The paper is organized as follow: the following part focus on motivation and rationale that has guided the Wheels design. The next section describes elements constituting the Wheels process (Wheels meta-model, applicability matrix, process template). The conclusion gives a brief positioning of Wheels relating to existing significant processes.

2 MOTIVATION

Since the product-line approach is defined by two different processes its applicability is more complex to apply in organizations than it is for classical applications development. Therefore, we choose four principles to integrate the notion of flexibility into the product-line process, and so, once it is set up, it is far easier for the users.

We choose to call the product-line process "Wheels model" because it drives work products creation and derivation. Our approach for the product line process development is inspired by [7] and aims at:

1. Offering pragmatic issue of process usability: rather a process handbook with a meta-model description than a full guide book;
2. Offering a tailorable process, with the ability and means to adapt the Product Line Process (PLP) to several situations (business, product line context) using an applicability matrix;
3. Giving an extensible PLP that can be increased incrementally along feedback during usage;
4. Establishing relationships to ISO or IEEE standard documentation (SSS, SSDD, SRS) and CMM.

Moreover, the Wheels process is based on two rolling bearing [15]: stakeholders and company objectives.

A stakeholder is someone who is involved into the product-line approach (e.g., customers, end-users, managers, architects, analysts). We'll give a non exhaustive list further on this paper. Hence, they are important in product-line process because their profiles guide the potential decisions as for the product-line design. Since we incorporate everyone in all design levels (requirements, architecture and components), the Wheels process presents two advantages:

- better participation of each member which feels more concerned and increase of the team motivation,
- better acceptance of the Wheels process because all actors (from the senior manager to the analyst or the designer) are concerned by the process implementation in their organization.

A software process must allows to respect all or a part of the company objectives. That's why our Product Line Process embodies the business units objectives within its structure: this process introduces two notions, variability and derivation, which permit to carry out an application more optimal (return on investment more quickly).

The purpose of the derivation is to create a specific application from work products (assets) produced in the domain engineering [8]. The generalization of this adjustment represents the knowledge variability.

3 SPLIT/WHEELS: THE PRODUCT-LINE PROCESS

Wheels is an iterative and incremental development process. These two properties have proved to be very crucial on the employees motivation and innovation which are directly proportional [16] with the project progress: short milestones combined with a well-defined handbook is the key of the success of the process.

According to [2], a process is a well-defined way to achieve one or more products and embodies several decision-making stages which allow engineers to know what they must do or what they should have done.

1. the Wheels model defines the software development process: guiding stakeholders to produce assets (reusable artifacts) according to identified objectives (by customers, company, …) in the product line. An artifact is any entity manipulated during product line approach (e.g., documentation, code source, domain models). The Wheels model formalizes the product line process, giving an explicit meta-model, defining guidelines in the form of process patterns.
2. people which use a method consider a process not as a real work but as extra work. Therefore, the Wheels model applies as much tailoring and as much pragmatic (simple but not simpler) as possible in order for the stakeholders to consider it as essential in their job. In fact, our product line process is completely overlapping inside the product line approach (STARS two-lifecycle model). The process relies on also applicability mechanisms between their different elements according to rules and/or constraints chosen

or imposed,

3. the Wheels model can be augmented incrementally along domain trials feedback and during usage, adding instantiations of the meta-model and increasing the applicability matrix.
4. the Wheels model proposes and advises to map the product line process with the standard documentation from IEEE standard (SSS, SSDD, SRS).

The product-line process formalization

The Wheels model is based on three basic assumptions to formalize the product-line:

- using an UML meta-model to increase the process understanding,
- using a matrix model to implement the tailoring,
- define a full handbook describing the process patterns.

The Wheels model defines all the elements of the product line process:

- Process: A generalization of the two sub-processes: Domain Engineering in which all of reusable elements are created (assets) and Application Engineering in which a new product is built, as far as possible, by derivation of the assets produced in Domain Engineering.
- Artifact: Any entity manipulated during software engineering. This includes any software, generation procedures, documentation, domain models, and process descriptions.
- Asset: An artifact that has been certified as being valuable to the organization and is reusable throughout the projects.

The difference between an artifact and an asset is a conceptual view; an artifact is created during the Application Engineering and may be generalized to an asset, while an asset is created during the Domain Engineering and derived during the Application Engineering.

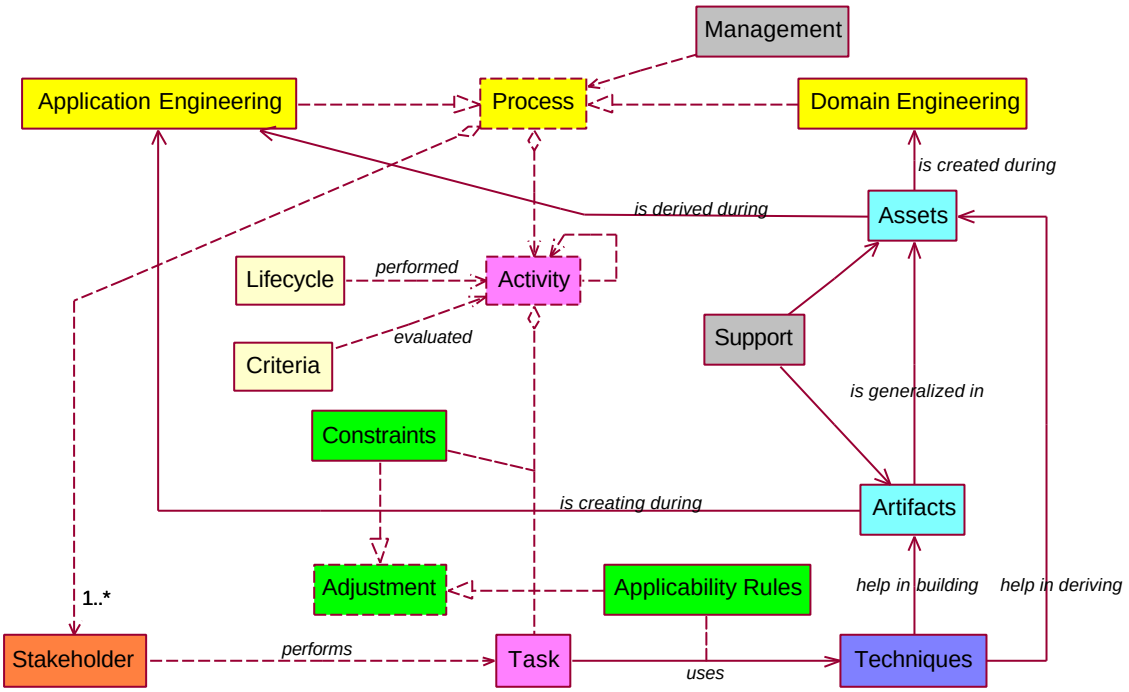


Figure 2 – PLPF - Product Line Process meta-model

The Wheels model does not offer a single and rigid method but offers a model describing all of process items which make up the product line approach [6]. To represent them, we defined the product-line process meta-model (figure 2) which presented these items composing the product-line process framework.

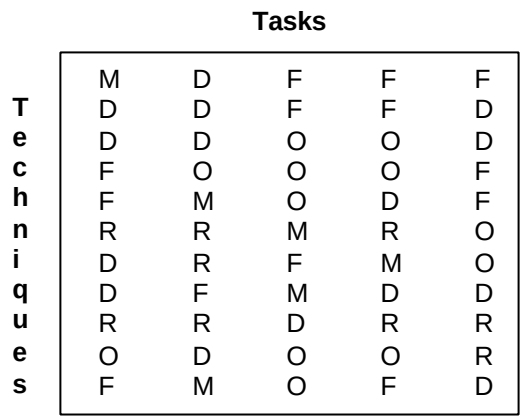
- Stakeholder: An individual, team, or organization who is concerned in some way or the other by the product which is generated by the product line approach. They are defined in the sequel.

- Activity: is a process item based on well-defined inputs are produced. There are two kinds of activities: the main activities which are composed of another activities (*e.g.*, domain analysis), and the activities which are further divided into a set of tasks (*e.g.*, product line scoping). An activity yields a well-defined artefact based on responsibility for the stakeholder and embodies other items (tasks).
- Task: A low-level set of activities where one or more artifacts or assets are produced and finished. A task is indivisible and is associated with techniques.
- Technique: A specific way to realize a task or a set of tasks.
- Adjustment: is a tailored concept, which allows to choose one or more software unit fulfilling a software item and using a rationale. There are two kinds of adjustment: constraints between Activities & Tasks, and applicability rules between Tasks & Techniques.
- Applicability rule: A business or technical condition allowing to choose a technique rather than another to realize a task.
- Constraint: A business or technical condition allowing to choose one or more tasks to realize an activity. A constraint may exist to a stakeholder profile, a specific view, a project progress, …
- Management: The majority of the Key Process Area (KPA) from CMM (Capability Maturity Model) as project management, risk management, quality assurance, schedule, subcontracting management, …
- Lifecycle: An iterative and an incremental approach of the product line process to perform an activity.
- Support: One of the KPAs (Key Process Area) called configuration management applied to a reusable context and so product line approach. Reuse configuration has an impact not only on the process but also on the work products as artifacts and assets.

To improve the flexibility of our product line process, these elements (we also say meta-types) are linked to each other by several applicability rules or constraints. The Wheels model proposes a way to increase the tailoring inside the process using a matrix model. This matrix allows to determine relevant elements to carry out the element parent (e.g. such activities to realize such main activity).

However, we have only defined the matrix between tasks and techniques. This part of process is certainly the less stable (techniques evolve highly).

One or more techniques are used to accomplish a task. The Key is how to choose the appropriate techniques to fulfill the tasks. This is accomplished by the use of a matrix (figure 3).



Applicability levels between tasks and techniques: **M**andatory, **R**ecommenced, **O**ptional, **D**iscouraged, **F**orbidden.

Figure 3 – The matrix model between tasks and techniques

Our product-line approach is stakeholder-oriented. Table 1 shows stakeholders who should be affected by the product line. Its aims at having a clear understanding of a potential re-organization of a software department or company. This table explains also the roles of these stakeholders. There is a difference between roles and persons: a person can have several roles (e.g. architect and designer) in a small project and a role may hold by several persons (e.g. analysts, testers) in a big project.

Stakeholders	Roles
Analyst	Takes user requirements and develops a domain analysis model with users customers and end-users) and requirement analyst.
Architect	Produces a consistent and documented software architecture that demonstrates the requirements by building tradeoffs on the software architecture to be built. Proposes and justifies technical options which will be used. Manages the development activities related to this project.
Customer	Places requirements on application systems. Emphasizes on durability of the software application. This translates in the following qualities: compatibility, scalability, reliability, maintainability, interoperability, use of industry standards, and use of well-proven technology. Pays, so wants to track the budget. Tracks the schedule (software on time). Assesses risks.

	Manages the red points. Is involved when deciding on priorities, features, and rollout plans. Defines strategic alliances with COTS providers.
Designer	Designs a subsystem or a category of classes, and manages interfaces to other subsystems. Directs implementation.
Developer	Develops documented and annotated classes, methods, codes in accordance with design decisions.
Domain Expert	Gives expertise on the business (including the future evolutions), and on the existing products (e.g., internal structures).
End-User	Uses the application when installed. Is a source of usability information when developing the software. Emphasizes on the following qualities: performance, dependability, use of existing systems, and usability.
Installer	Does the rollout.
Maintainer	Manages required modifications: error and improvement.
Marketer	Represents customers in the organization. Analyses the properties of described and prescribed systems, and indicates which features are competitive. Plans the future time to release of the products on the market (time to market). Says what slots the future products will fill in the market.
Project Manager	Plans and estimates the resources, the schedule, the budget of the development process. Decides technical choices. Returns feedback to marketer. Manages the roll-out. Tracks the costs, the schedule, the deliverables, the risks, and the red flags.
Requirement Analyst	Interviews end-users, examines existing requirement documents, studies documents that describe the scope of the domain, and derives specifications.
Senior Manager	Plans and estimates the quality and the evolution of the development process.
Technical Expert	Gives expertise on the technology (e.g., well proven and state-of-the-art technology and techniques), and on the existing solutions (e.g., properties and structures).
Tester	Develops and executes test cases for each development phase in order to validate the functions and the qualities.

Table 1 – List of stakeholders and their role

To setup a product line in an organization, the process template can be easily extended to select the relevant elements and to instantiate the rules. Moreover, the process assumes on the one hand the legacy system (standard documentation, software components, project or risks management, and so on), and on the other hand the traceability between the different artifacts which have to be produced.

The process pattern contains all the information which allow engineers to build in part or the whole process: description, purpose, involved stakeholders, informal specifications, required artifacts or assets in input, and imposed artifacts or assets in output. Therefore, a manager is able to describe a new process as a function of their own mains objectives, and to help an organization carrying out a product-line approach (tradeoffs between activities, stakeholders, assets, and anything else).

So, the Wheels model proposes a complete process template. Table 2 shows all items composing it:

Items	Descriptions
Synopsis	General definition and description of the process item. The purpose served by developing process item.
Stakeholders	Role of the persons who are involved in building the process item.
Timing	The step in a process where the process item is normally produced.
Informal specification	Accurate description of the process item give advice and guidance.
Inputs/Outputs	List of input and output software data of the process item.
start/final points	Adjustments which allow to decide when a process item must start and when it achieves.
List of activities/tasks	List of commented activities/tasks of the process item, and presentation of associated techniques.
Example	Illustration of a process item.
Reference	Citations, books, articles describing in further detail the process item. Connection to system engineering.

Table 2 – Process template

The Wheels Product Line Process Framework

We presented in the previous paragraph the advantages to develop a formalized process which clarifies the product-line understanding offering a pragmatic issue to a company (to manage their teams and to emphasize on their objectives).

Since process flexibility must allow a company to decide to go for a product-line approach (notions of constraints and applicability rules), we have to develop the tailoring (e.g. allow a business units to make easily a custom-made product line process because of our handbook and the principle of derivation) and the extensibility (add process elements inside the meta-model and the matrix model).

The Wheels model introduces the tailoring into several levels. A main activity is composed of activities and they must be all achieved in order to create this main activity. On the other hand, even if an activity may be composed of several tasks (cf. figure 2), only few tasks may achieve it. Indeed, depending on the organization context and/or of the stakeholders, it may be pertinent either to not carry out such and such task, or to realize them in no particular order. For example, there are at least two ways to create the "Product Line Scoping" activity. First of all, this activity is entrusted to a beginner analyst without knowledge of the domain. So our process will require that all of the tasks have to be created (cf. table 3). Secondly, this activity is entrusted to a domain expert and so our process won't require all of the tasks (e.g. the "Domain Definition" task).

Processes	Domain Engineering, Application Engineering
Main activities	Domain Analysis, Domain Design, Domain Implementation, Application Analysis, Application Design, Application Implementation
Activities	Product Line Scoping, Requirements Engineering, Component Design, Implementation, Legacy Architecture recovery, Building architecture views, …
Tasks	Domain Definition, Domain Scoping, Domain Modeling, Technological Context, Capabilities, Subsystem, Architectural Guidelines, Building Interface, Contextual Dependency, …
Techniques:	Use case modeling, feature modeling, documentation, static diagrams, views mining and modeling, …

Table 3 – a part of the Product Line Process Framework

The second level of tailoring is between the tasks and the techniques which allow to realize them. We have defined applicability rules between tasks and techniques (figure 2) by use of a process matrix (figure 3). The matrix shows how to choose the appropriate techniques to fulfil the tasks according to the stakeholder profile:

	Tasks	Domain Definition	Domain Scoping	Capabilities	Subsystem	Architectural Guidelines	Building Interface
Techniques	Use Case Modeling	O	R	M	O	F	O
	Feature Modeling	O	M	O	R	F	O
	Documentation	O	O	O	R	M	R
	Collaboration Modeling	F	O	F	D	F	M
	Style mining and	F	R	R	F	M	F
	Decision Modeling	F	O	M	F	M	R
	Traceability Support	O	M	M	O	M	M

Applicability levels between tasks and techniques: Mandatory, Recommended, Optional, Discouraged, Forbidden.

Figure 3 – Wheels Process matrix

Thus, in according to the product-line process framework (PLPF) meta-model, the Wheels model listed all of the activities, tasks, techniques, constraints, applicability rules, and so on, necessary for the product line approach. These different items, specially the constraints and the applicability rules, may be imposed (e.g. by the marketer) or chosen (e.g. by the architect).

The process template is the last point which emphasizes the tailoring. Indeed, the activities of the Product Line Process Framework are described by an accurate formalism. In this way, they are easily understood by all of the stakeholders.

First of all, the Wheels model has to guide the stakeholders. Since they must know what they have to do (output as assets) and from what (inputs), we apply a handbook which guides the users during all of the product line phases and activities.

This guide is dynamic. It is created from the instantiation of the UML meta-model of the PLPF (figure 2). Indeed, our idea is to generate the guide sheets from the UML models as a source code. This technique shares in the extension advantage (process formalization with UML meta-model) and the flexibility advantage (everyone can regenerate any time a guide sheet if the model evolves).

Figure 4 shows an example of a guide sheet of a product line process element: The Domain Modeling task which composed the Product-Line Scoping activity inside the Domain Analysis phase.

Name	Domain Modeling task
Synopsis	The *domain modelling* task defines the features of the product line domain. This goal is to build the features model asset which describes the structure of the selected domain. Indeed domain modelling analyses existing product and the market strategy inside the scope of the product line in order to identify the features of the product-line domain, and to organise them. To produce the features of the product line, a analysis on the existing products (in yours company and in your concurrent) and on the market strategy (to take into account the future standard and the market evolution).
Stakeholders	Domain expert, marketer, software manager.
Timing	
Informal specification	The *domain modelling* task is very important because it is the task which makes the link between the product-line scoping activity and the requirements engineering activity. The features model created during this task defines and describes what features of product-line domain are mandatory, optional or alternative. Therefore the derivation activity will select in the features model the features that you need. The both *domain scoping* and *domain modelling* tasks are extremely dependent because all modifications done in the *domain scoping* task impact directly the *domain modelling* task. Indeed if the scope of the product-line domain changes the features model may also change (*e.g.*,, some features may become outside of the product line domain). Moreover, the *domain modelling* task impacts the *requirements engineering* activity, because the features model will be only closed with the achievement of the *requirements engineering* activity. The *domain modelling* task defines the features of the product line follows into three steps that are: - Describe the features of existing product in the product-line domain. - Study the market strategy in the product-line domain to avoid to forget new features. - Define with selecting the features that will be implemented in the product line domain.
Inputs/Outputs	- The input of the *domain modelling* task is the domain model. - The output of the *domain modelling* task is the context model.
start/final points	To be defined
List of tasks	Not applicable
Example	To be defined
Reference	FODACom, http://www.pisa.intecs.it/reuse/FODACom.htm Kang, K.C. and Cohen, S.G. and Hess, J.A. and Novak, W.E. and Peterson, A.S., Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical report, CMU/SEI-90-TR-21, ESC-TR-90-222, Software Engineering Institute, Carnegie Mellon, University, Pittsburg, Pennsylvania, November 1990

Figure 4 – Example of a Process Pattern

4 CONCLUSION

Other current processes have been already built which could be applied on a product line approach: RUP [13] proposes full guidelines along a project but is not a flexible process and do not take into account the product line problematic. OPEN [11] (Object-Oriented Process, Environment and Notation) formalizes its process using a meta-model but does not define a full process template and is not product line oriented. ODM [10] (Organization Domain Modeling) describes only the Domain Engineering process. RSP [12] (Reuse-driven System Processes) and FAST [5] (Family-oriented Abstraction, Specification and Translation) are both processes oriented towards reusability but do not provide quite tailoring and extensibility.

We propose a process integrating all of the best practices to carry out a product line into an organization. The Wheels process model provides a full formalization of a product line process through the meta-model, the applicability model (the matrix), the stakeholder-driven approach and its full process pattern description. Moreover, the Wheels model proposes several applicability mechanisms between the different elements of the process according to rules and/or constraints chosen or imposed, which allow to have a tailored process. The Wheels model fulfils the notion of the extensibility through adding any process elements (e.g. activities, tasks, techniques) into the meta-model and the matrix model.

Table 4 summarizes the best practices which characterize and guarantee the process flexibility.

The Wheels model is currently assessed in the context of LCAT domain experiments. These experiments cover real business cases of Alcatel and Thomson-CSF.

Best practices	Explanations
Formalization	All of the PLP elements (activities, tasks,..) are identified in an UML meta-model to increase the process understanding. Moreover, a full handbook describes all of the process.
Tailoring	Constraints and applicability rules have been defined between process elements by use of a matrix. This matrix allows to choose such-and-such elements according to the company objectives and the stakeholders profile.
Extensibility	Any other elements can be added to the process, first of all into the meta-model and secondly into the matrix.
Standardization	Establishing relationships to IEEE standard documentation (SSS, SSDD, SRS, …) and CMM assessment.
Full lifecycle	All of the PLFP activities are defined into an iterative an incremental process for business and technological issues.
Pragmatism	The PLFP applies a whole of pragmatic collection of rules or constraints and a full handbook.
Deliverables	They are exactly described within the process. A deliverable template has been chosen and, a guide sheet explains what and when must be done.
Metrics	The PLPF defined appropriate metrics, standards and tests to assess its maturity.
Management	Some full guidelines have been set for project management and quality assurance.
Legacy	The PLPF preserves the business units knowledge (domain expertise, legacy system).
Technique	The PLPF separates the process and its representation even if we defined the process with a notation in this paper (in this case UML notation).

Table 4 – Best practices of the PLPF

REFERENCES

1. P. Allen, and S. Frost, “All for One and One for All”, Application Development Advisor, July/august 1998, pages 26-32
2. Scott W. Ambler, “Process Patterns: Bu ilding Large-Scale Systems Using Object Technology”, Cambridge University Press, 1998
3. David C. Gross, and al., “An Instance of the Air Vehicle Training Systems Domain: A STARS Demonstration Project”, 1994 SCS Simulation MultiConference, San Diego, April 11-15, 1994
4. M. Coriat, J. Jourdan and F. Boisbourdin, “The SPLIT method: Building product lines for software intensive systems”, SPLC1, August 2000
5. David M. Weiss and Chi Tau Robert Lai, “Software Product-Line Engineering: a Family-Based Software Development Process”, Addison-Wesley, 1999
6. D. Firesmith and B. Henderson-Sellers, “Improvements to the OPEN Process Metamodel”, Joop, November/December 1999
7. B. Henderson-Sellers and S. J. Mellor, “Tailoring Process-Focused OO Methods”, Joop, July/August 1999
8. IBM Object-Oriented Technology Center, "Developing Object-Oriented Software: An Experience-Based Approach", Prentice Hall, 1997
9. E.-A. Karlsson, “Software Reuse: a Holistic Approach”, John Wiley & Sons, 1996
10. Mark Simos, "Organization Domain Modeling (ODM) guidebook", STARS report VC-A025/001/00, Version 2.0, June 1996
11. Ian Graham, Brian Henderson-Sellers, Houman Younessi, “The Open Process Specification (Open Series)”, Addison-Wesley, October 1997
12. R. McCabe and G. Campbell, "Reuse-Driven Software Processes Guidebook", STARS report SPC-92019-CMC, Version 02.00.03, November 1993
13. Ph. Kruchten, “The Rational Unified Process – an introduction”, Addison-Wesley, 1999
14. J. Rumbaugh and I. Jacobson and G. Booch, "The Unified Modeling Language User Guide", Addison Wesley, February 1999
15. Barry Boehm, and Alexander Egyed, and Julie Kwan, and Dan Port, and Archita Shah, and Ray Madachy, “Using the WinWin Spiral Model: a case study”, Computer, July 1998
16. George Yamamura, “Process Improvement Satisfies Employees”, IEEE software, September/October 1999

Analysis of the Essential Requirements for a Domain Analysis Tool

Giancarlo Succi
University of Calgary
2500 University Dr. NW
Calgary, AB, Canada
T2N 1N4
(403) 220-8357
Giancarlo.Succi@enel.ucalgary.ca

Jason Yip
University of Calgary
2500 University Dr. NW
Calgary, AB, Canada
T2N 1N4
(403) 220-4927
j.c.yip@computer.org

Eric Liu
University of Calgary
2500 University Dr. NW
Calgary, AB, Canada
T2N 1N4
(403) 220-4927
liue@enel.ucalgary.ca

Abstract

The overall domain analysis process is a large and complex process involving a web of many activities. An effective domain analysis tool must address the need to track many different types of information and dependencies. Investigating other fields such as requirements traceability, tool integration, and critiquing systems has helped to derive several requirements for a domain analysis tool which are then matched against existing domain analysis tools. The Holmes domain analysis tool tries to learn from this analysis of domain analysis tool requirements and existing domain analysis tools.

Keywords

Domain analysis, automated domain analysis support, traceability, tool integration, critiquing systems, open architecture, JavaSpaces, tuple space, adapters

1. Introduction

An effective way to implement systematic reuse is through domain analysis, "the process of identifying and organizing knowledge about some class of problems – the problem domain – to support the description and solution of those problems" (Arango, 1991).

There have been several attempts to categorize domain analysis approaches (Wartik and Prieto-Diaz, 1991; Arango, 1994). In particular, Arango shows that domain analysis approaches are essentially based on coordinating a sequence of steps that start from the analysis of the market and end with the design of actual software systems (1994). This suggests that domain analysis is a large and complex process involving a myriad of interrelated activities.

Intuitively, higher levels of complexity lead to more difficult comprehension and a higher incidence of errors. The complexity of the domain analysis process thus implies that a support tool is needed. (Dionisi Vici *et al.*, 1998) confirms this, saying that domain analysis practice "must be supported by tools helping to manage complexity."

This paper analyses the scope of activities covered by a domain analysis tool and how such activities relate to other fields such as requirements traceability, tool integration, and critiquing systems. This analysis is used to derive some possible requirements for a domain analysis support tool and then discusses the extent to which existing tools satisfy these requirements. Finally, the paper shows how Holmes (Succi *et al.*, 1999), a domain analysis tool for the Sherlock methodology, addresses these issues with an architecture based on tuple spaces (Gelernter, 1985).

2. Requirements of a Domain Analysis Tool

Arango describes a domain as a collection of either similar programs or similar applications (1994). Domain analysis is then either the process of creating models to reason and predict aspects about a set of problems or the process of organizing descriptions of common components and architectures in a set of applications. Whether about similar problems or applications, Arango shows that domain analysis methods can be decomposed into many, interrelated activities.

A domain analysis tool has to track different types of information related to the different activities involved in a particular domain analysis methodology. Since these activities are interrelated, the tool also has to track the dependencies between different types of data. According to (Arango, 1994), domain analysis often involves many different knowledge contributors (i.e., analysts, domain experts, developers, etc.). This suggests that a suitable system should also support multiple users simultaneously. Therefore, a domain analysis support system needs to maintain both data and change consistency between many different activities. Furthermore, it is difficult to comprehend the relationships between multiple, interrelated activities without having semantic support.

For the remainder of this section, we will examine these aspects of a domain analysis tool: traceability, tool integration, and semantic support.

The problem of data and change consistency is also known as the problem of **traceability**. Traceability concerns have already been observed and analyzed in the field of **requirements engineering**. (Gotel, 1994) defines traceability in requirements as the ability to describe and follow the life of a requirement in both a forward and backward direction through the whole system's life cycle.

Traceability support has already been recognized as being valuable for tool support for domain analysis methods. For example, (Griss *et al.*, 1998) includes "Support for Traceability" as one of the key concepts for tool support for the FeatuRSEB domain analysis method.

(Domges and Pohl, 1998) describes seven key capabilities of existing requirements traceability environments. These capabilities are predefined and customizable data types, predefined and user-definable queries including filtering and sorting, comprehensive configuration management and change tracking, trace analysis, various presentation formats, teamwork support, and interfaces to existing 3rd party software.

These capabilities seem to apply also to domain analysis tools with a few exceptions. Domain analysis data types are likely to be predefined for a particular methodology and probably do not need to be customizable. Also having various presentation formats is not as essential as the other capabilities for domain analysis support.

According to (Krut, 1993), the purpose of a domain analysis support tool "should be to offer an integrated environment for collecting and retrieving the domain model and architectures". For each specific domain analysis activity, such as modeling or developing a domain vocabulary, there is often already an existing tool that can support it. It is preferable for users to use existing

tools rather than having to use a newly developed tool: users are already familiar with existing tools and specialized tools are more likely to have superior features. In addition, less time is needed to develop the domain analysis support system, since the only effort needed is the integration of a tool into the overall system. An effective domain analysis tool should therefore be able to relatively easily interface to existing 3rd party applications.

Interfacing with third party software has already been analyzed in the area of **tool integration**. (Gautier *et al.*, 1995) identifies two aspects of software tool integration: tool-to-framework, where tools communicate indirectly through the integration framework and tool-to-tool, where tools communicate directly with each other. (Wasserman, 1989) describes four different dimensions of integration: user interface, data, control, and process integration. User interface refers to a common "look-and-feel," data refers to the sharing of information, control refers to direct tool-to-tool communication, and process refers to tool activation based on a particular process model.

For the purpose of maintaining data and change consistency, the most important integration dimension is data. Data integration can be achieved through a shared repository or by direct data transfers. The other integration dimensions may be desirable for a domain analysis tool, but not essential.

Even only considering data integration on its own, tool integration is difficult to achieve because "in general, domain information is stored in a great variety of data sources, using different data models, access mechanisms, and platforms." (Braga *et. al.*, 1999) In this light, tool-to-framework integration seems more appropriate as it removes the problems of requiring combinatorial adapter interfaces between multiple 3rd party tools.

(Bayer *et al.*, 1999) recognizes that domain analysis tool support is necessary to "control information about a domain." The complexity of the domain analysis process and the difficulty for the domain analysts to deal with it suggest that some form of assistance would be extremely valuable and should focus on the semantic relationships between data rather than just syntactic checking.

The concept of **design critics** suggests a suitable approach for semantic assistance. Design critics are intelligent mechanisms that analyze a design and provide feedback to assist the designer in improving it (Robbins, 1998). According to (Fischer *et al.*, 1993), design critics should be embedded in the environment and actively but non-disruptively alert designers of potential problems suggesting potential solutions, if possible. (Robbins and Redmiles, 1998) describes the critiquing approach as different from traditional software analysis approaches in that the focus is on the designer's cognitive needs. Traditional approaches attempt to prove correctness of a completed or nearly-completed system. Critiquing approaches, on the other hand, pessimistically detect potential problems in partially specified systems.

The P3, domain-specific component generator (Batory *et. al.*, 2000) uses a tool called a design wizard which is somewhat similar to design critics except that it is targeted toward optimizing data structures rather than higher-level design advice.

The previous analysis suggests that a domain analysis tool should support the following activities:

- Queries on dependency links: filtering, sorting, and other relationships
- Link consistency management: ensures that link traces make sense
- Change consistency management between different activities: ensures that all changes are propagated correctly
- Simultaneous multiple users: allows more than one user on the system at once
- Semantic assistance: warns user of potential problems
- Data integration with COTS tools: allows COTS tools to communicate with the framework

3. Current Domain Analysis Tools

As mentioned earlier, there have been several surveys on domain analysis methodologies (Wartik and Prieto-Diaz, 1991; Arango, 1993). An investigation of existing domain analysis tools (Yip and Succi, 1999) showed that some tools are focused on particular activities within domain analysis, mostly modeling and architecture or component development (Krut, 1993; Prosperity Heights Software, 1999; Loral Defense Systems, 1996) and domain-specific component generators like P3 (Batory et. al., 2000). Other tools attempt to cover a broader range of activities. These tools include (Bayer *et al.*, 1999), (Braga *et. al.*, 1999), (Frakes et. al., 1998), (Terry et. al., 1995), and (Tracz and Coglianese, 1995). In this paper, we are mostly interested in the complications caused by the multi-activity nature of domain analysis and therefore the following analysis focuses on the second group of tools.

DADSE (Hayes-Roth *et al.*, 1992; Terry *et al.*, 1995) is a support environment for DICAM (Distributed Intelligent Control and Management) application development. Its key features include a blackboard architecture, the ArTek ADL (architecture description language), a query-capable persistent memory repository, knowledge-based design assistants, and the DADSE launcher. The design assistants can be registered to respond to event patterns posted by tools to the blackboard and are able to automate certain tasks. The configurable DADSE launcher consists of a tool registry, used to "enroll" tools, and tool activators.

DARE-COTS (Frakes *et al.*, 1998) uses COTS and freeware tools to support various domain analysis activities. The overall environment centers on the Domain Book, currently implemented using Microsoft Word in outline mode.

Diversity/CDA (Bayer *et al.*, 1999) is a support tool produced in the context of the IESE PuLSE methodology. This tool uses a 3-tier architecture, consisting of a client layer, a conceptual data schema layer, and a physical database. Each unit within a client is implemented as a separate JavaBean and change propagation between views uses InfoBus, the Java implementation of a software data bus. The architecture allows pluggable workproducts through the use of a general tool framework and well-defined tool interfaces. Other key features include domain model instantiation using a decision model, traceability using extensible link types, and link inspectors to browse links.

DOMAIN (Tracz and Coglianese, 1995) was designed to support the DSSA (Domain Specific Software Architecture) approach to domain analysis. The tool uses the Chimera hyperweb server

to link objects, viewers, views, anchors, and links. A Browser is used to navigate the hyperweb. There is also a Domain Launcher that is used to invoke the various DOMAIN editors (hypertext, dictionary, reference requirement, scenario, and thing).

Odyssey (Braga et. al., 1999) supports a domain engineering method that is called Odyssey-DE. There is a hypermedia interface to models, patterns and components specified by various tools. An information agent tool helps with navigation of the hypermedia web. A key feature is the use of a mediation layer to provide "a uniform representation and manipulation mechanism" for domain information.

Table 1 shows how the existing tools listed in the previous section target the six proposed requirements.

Table 1: How Existing Tools Realize the Proposed Requirements

Tool	Allows queries	Link consistency management	Change consistency management	Multi-user support	Semantic support	Tool integration support
DADSE	Directly on repository contents	No explicit links		Simple configuration management with file locking	Knowledge-based design assistants	Tcl/Tk scripting for tools not needing access to system data
DARE-COTS		Manual; links represented in "Domain Book"				Loose manual integration; all tools are COTS
Diversity/ CDA		Textual and graphical link browsers	Uses InfoBus for change propagation	Long-term locking for multi-user consistency		Supports COTS tools implementing InfoBus interface(s)
DOMAIN	Customizable hyperweb browser display	Chimera hyperweb browser				Must supply Chimera-supported interface
Odyssey		Information agent tool; hypermedia web				Mediation layer

Some tools have support for querying. In particular, DADSE allows queries directly on the contents of its persistent memory repository and DOMAIN allows customization of its hyperweb browser display using filtering. Link consistency is supported by all of the tools to varying degrees. Both Diversity/CDA and DOMAIN use a browser to navigate links while DARE-COTS has the manual method of simply representing links in its "Domain Book". Diversity/CDA also has change consistency support using InfoBus. Both DADSE and Diversity/CDA provide multi-user support in the form of file locking. DADSE also provides semantic support with its knowledge-based design assistants. Tool integration support is provided by all the tools examined. DARE-COTS has very loose integration while the other

tools are tighter, using scripting (DADSE) or requiring implementation of an interface (Diversity/CDA, DOMAIN, and Odyssey).

Altogether, the strongest area of support seems to be link consistency management and tool integration support, with each tool having different levels and forms of integration support. The weakest area of support seems to be in change consistency management and semantic support.

4. Holmes Architecture Experience

Holmes is a tool system built to support Sherlock.

Sherlock is a domain analysis and engineering methodology with five main phases (Domain Definition, Domain Characterization, Domain Scoping, Domain Modeling, Domain Framework Development), each with multiple activities. The goal of Holmes is to support the web of activities in Sherlock while remaining open to easier future evolution.

Figure 1 shows the overall architecture of Holmes, which centres on the use of JavaSpaces (Sun, 1998) as a shared blackboard of objects. This is similar to DADSE's use of blackboard architecture. This type of architecture was chosen because it more simply meets several of the identified essential requirements, namely change consistency management, multi-user support, and tool integration. This architecture also allows for the possibility for tools to be applets to allow a Web-based structure. A more detailed description of the Holmes architecture follows.

As shown in Figure 1, tools connect to the JavaSpace to communicate data state changes. Data repositories used for more permanent state storage also connect to the JavaSpace to wait for requests for data retrieval as well as monitor posted data state changes. This architecture supports multiple users in a very simple way since each user's client can anonymously connect to the space and then post and listen to appropriate data state changes. This style is also appropriate for use on the Web. The tools, in this case, would be applets and would locate the appropriately named JavaSpace using Jini lookup.

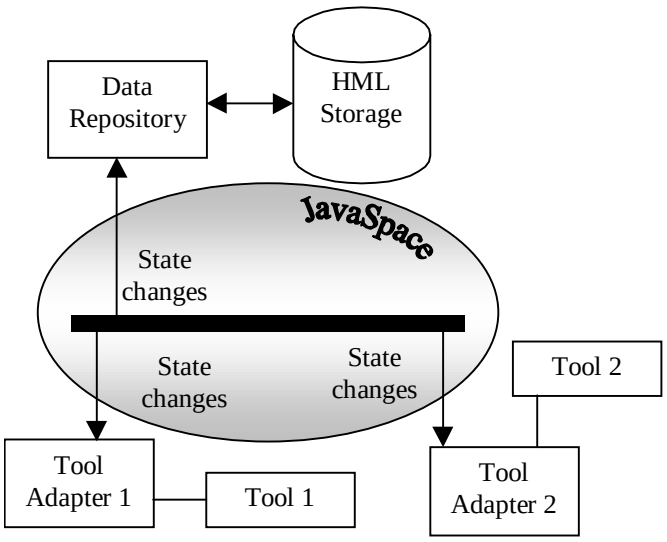


Figure 1: Holmes architecture

Each domain analysis activity requires different data types; therefore event queues for each data type exist in the JavaSpace. Tools interested in changes to a certain type of data listen for new events on such queues. When a change occurs in that particular type of data, a tool posts this change on the event queue. All listening tools are notified, and each can update their local state accordingly. This is similar to Diversity/CDA's use of InfoBus to propagate changes. Figure 2 shows a simplified version of the event queue.

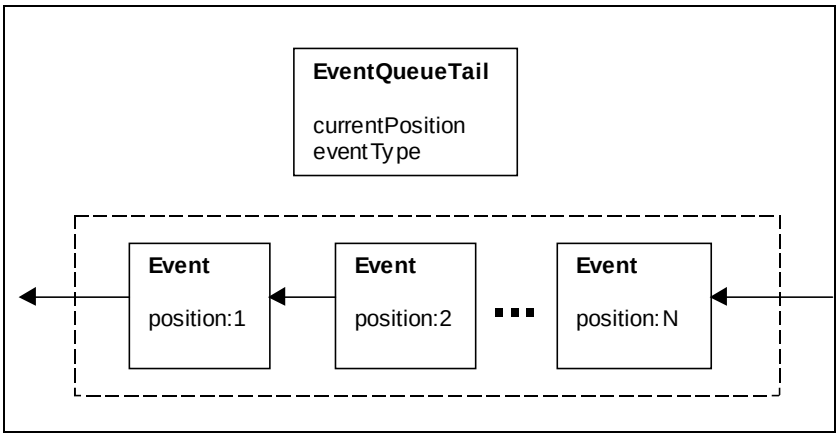


Figure 2: Event Queue

Tool integration with this architecture is accomplished using tool adapters that are being developed for each Sherlock domain analysis activity. These adapters handle the interaction with JavaSpaces and the event queues. As shown in Figure 1, tools simply have to communicate with the tool adapters to receive and transmit data state information. For instance, a user may be interested in using Emacs for code editing during the domain framework development (DFD) phase of Sherlock. A DFD adapter keeps track of the state of the shared source files by attaching itself to the appropriate event queues in the JavaSpace. When a user edits a file, the adapter asks Emacs to load it via an Emacs Lisp (Elisp) function call. The user can now make code changes within Emacs. Once editing is done and changes are to be committed, the user invokes another Elisp function to notify the adapter. The adapter will then handle the posting of the code changes to the appropriate event queue(s).

The choice of how Holmes data is statically represented is also related to the tool integration requirement. Holmes data is stored using the Holmes Markup Language (HML), which is essentially XML (W3C, 1998) with a custom Document Type Declaration (DTD). The advantage of an XML-based format is that the data can be viewed in a human-readable form using a text or XML viewer. This maintains independence of the data from the particular tool that manipulates it. The human-readable structure also provides the capability of building DTD translators to convert from HML to other XML-based languages. Although it does not completely eliminate the need to build adapters, in this case DTD translators, the effort is somewhat standardized and reduced. Obviously the advantages of this approach relies on the growing popularity of XML as a data format, especially for Product Data Management systems (Gould, 2000) and UML CASE tools (OMG, 1998).

The tool adaptors also assist with integrating a design critiquing system for semantic support. Similar to the knowledge-based design assistants in DADSE, the critiquing system can base its responses on the data objects that appear in the JavaSpace independent of the tool that placed the data objects. The advantage of this approach is that the critiquing system becomes independent of the different tool interfaces.

The critiquing system uses the Prolog (Sterling, 1986) language to write specific critiquing rules. Prolog was chosen as the scripting language of choice since Prolog clauses are very well suited to the description of relationships. Prolog scripts are also used to support dependency link querying. A tool or user can apply Prolog scripts that then query the system for the dependencies of a particular piece of activity data.

The mapping of Holmes features to the derived requirements for domain analysis tools is shown in Table 2:

Table 2: How Holmes Satisifies Identified Requirements

Feature	Allows queries	Change consistency management	Link consistency management	Multi-user support	Tool integration support	Semantic support
JavaSpaces		X		X	X	
Data repositories		X			X	
Event queue		X				
Tool adaptors					X	
Holmes markup language					X	
Critiquing system			X			X
Prolog scripting	X					X

5. Conclusion and Future Work

Our proposed requirements for domain analysis tools are the ability to maintain change and link consistency, query dependency links, simultaneously support multiple users, support tool integration, and provide semantic support. The currently existing tools support these identified requirements in varying degrees. The weakest area seems to be in maintaining change consistency and providing semantic support.

Our approach with the Holmes domain analysis tool has been to combine the various ideas of existing tools with novel implementations. The experience has shown that the key ideas of existing tools are not mutually exclusive and that there are avenues of improvement that have been neglected or have not yet been investigated.

Our initial examination of requirements traceability tools has brought up capabilities that do not yet exist in current domain analysis tools, namely customizable data types, trace analysis, and various presentation formats. These capabilities should be examined more closely to assess their usefulness and criticality in developing domain analysis tools.

Overall, more work will be done on developing the default Holmes tools, visualizing links, and adding capability to the critiquing system.

Finally, using XML as a data format leads to the question of whether there are any essential data types within domain analysis. Perhaps a standard or common DTD can be generated for domain analysis data. This would increase interoperability between different domain analysis tools as well as just the activity-specific tools.

Acknowledgements

This research has been partly supported by the Canadian National Science and Engineering Research Council, the Government of Alberta, and by the University of Calgary.

References

Arango, G. and R. Prieto-Diaz (1991) "Domain Analysis and Software Systems Modeling", *Domain Analysis Concepts and Research Directions*, IEEE Computer Society Press

Arango, G. (1994) "Domain analysis methods", *Software Reusability*, Ellis Horwood

Bayer, J., D. Muthig and T. Widen (1999) "Support for Domain and Variant Engineering: DIVERSITY/CDA" submitted to *Automated Software Engineering '99*

Batory, D., G. Chen, E. Robertson, and T. Wang (2000) "Design Wizards and Visual Programming Environments for GenVoca Generators", to appear in *IEEE Transactions on Software Engineering*, URL: ftp://ftp.cs.utexas.edu/pub/predator/ieee-tse-99.ps

Braga, R., C. Werner, and M. Mattoso (1999) "Odyssey: A Reuse Environment based on Domain Models", *Proceedings of the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering & Technology*

Dionisi Vici, A., N. Argentieri, A. Mansour, M. d' Alessandro, and J. Favaro (1998) "FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry", *Proceedings of the Fifth International Conference on Software Reuse*

Domges, R. and K. Pohl (1998) "Adapting Traceability Environments to Project-Specific Needs", *Communications of the ACM*, **41**(12)

Fischer, G., K. Nakakoji, J. Ostwald, G. Stahl, and T. Sumner (1993) "Embedding Computer-Based Critics in the Contexts of Design", *Conference proceedings on Human factors in computing systems*

Frakes, W., R. Prieto-Diaz and C. Fox (1998) "DARE: Domain analysis and reuse environment", *Annals of Software Engineering*, **5**(1998)

Gautier, B., C. Loftus, E. Sheratt, and L. Thomas (1995) "Tool Integration: Experiences and Directions", *Proceedings of the 1995 International Conference on Software Engineering*

Gelernter, D. (1985) "Generative communication in Linda", *ACM Transactions on Programming Languages and Systems*, **7**(1)

Gotel, O., and A. Finkelstein. (1994) "An analysis of the requirements traceability problem", *Proceedings of the 1994 International Conference on Requirements Engineering*

Gould, J. (2000) "PDM/EDM/ERP/SCM... Where Will It All End?", *Desktop Engineering*, **5**(4), URL: http://www.deskeng.com/articles/00/Feb/pdmedm/index.htm

Griss, M., J. Favaro, and M. d'Alessandro (1998) "Integrating Feature Modeling with the RSEB", *Proceedings of the Fifth International Conference on Software Reuse*

Hayes-Roth, F., L. Erman, A. Terry, and B. Hayes-Roth (1992) “Domain-Specific Software Architectures: Distributed Intelligent Control and Management (DICAM) Applications and Development Support Environment”, *Software Technology Conference: Proceedings of a Workshop*

Krut, R. (1993) “Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology”, Technical Report CMU/SEI-93-TR-11, Software Engineering Institute, Carnegie Mellon University

Loral Defense Systems (1996) “User Manual: ELPA Domain Generation Environment (EDGE) Version 2.0”, Technical Report STARS-PA 19-S001/002/00

OMG (1998) “XMI Revised Submission to the SMIF RFP”, OMG document ad/98-10-05, URL: http://www.omg.org/cgi-bin/doc?ad/98-10-05

Predonzani, P., G. Succi, and T. Vernazza (1999) *A Domain Oriented Approach to Software Production*, Artech House Publisher Inc.

Prosperity Heights Software (2000) “Metaprogramming Text Processor”, URL: http://www.domain-specific.com/MTP/index.html

Robbins, J. (1998) “Design Critiquing Systems”, Technical Report UCI-98-41, University of California, Irvine

Robbins, J. and D. Redmiles (1998) “Software Architecture Critics in the Argo Design Environment”, *Knowledge-Based Systems*, **11**(1)

Sterling, L. and E. Shapiro (1986) *The Art of Prolog: Advanced Programming Techniques*, MIT Press

Succi, G., A. Eberlein, J. Yip, K. Luc, M. Nguy, and Y. Tan (1999) “The Design of Holmes: a Tool for Domain Analysis and Engineering”, *Proceedings of the 1999 IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*

Sun Microsystems (1998) “JavaSpaces™ Specification, Revision 1.0 Beta”, URL: http://java.sun.com/products/javaspaces/specs

Terry, A., T. Dabija, T. Barnes, and A. Teklemariam (1995) “DADSE 2.3 User Manual”, Teknowledge Federal Systems

Tracz, W. and L. Coglianese (1995) “DOMAIN (DOmain Model All Integrated): A DSSA Domain Analysis Tool”, Technical Report ADAGE-LOR-94-13

Wartik, S. and R. Prieto-Diaz (1992) “Criteria for Comparing Reuse-Oriented Domain Analysis Approaches”, *International Journal of Software Engineering and Knowledge Engineering*, **2**(3)

Wasserman, A. (1989) “Tool Integration in Software Engineering Environments”, *Lecture Notes in Computer Science 467 – Proceedings of Software Engineering Environments, International Workshop on Environments*

W3C (1998) “Extensible Markup Language (XML) 1.0”, W3C Recommendation, REC-xml-19980210, URL: http://www.w3.org/TR/REC-xml

Yip, J. and G. Succi (1999) “Domain Analysis & Engineering (DA&E) Support Tools Feature List and Comparison”, Technical Report, University of Calgary

Embedded Systems Product Lines

Jorge L. Díaz-Herrera [†] ***and Vijay K. Madisetti*** [††]

[†]Department of Computer Science, Southern Polytechnic State University
1100 South Marietta Parkway, Marietta, GA 30060-2896 jdiaz@spsu.edu

[††]School of Electrical and Computer Engineering, Georgia Institute of Technology
Atlanta, GA 30332-0250 **vkm@ece.gatech.edu**

ABSTRACT

Embedded electronics products, ranging from PDAs, PCS phones, information appliances (IA), automotive computing systems, to complex radar systems, consist of application-specific hardware and application-specific software optimized for size, weight, power, and performance considerations. In most embedded systems, 80-90% of the functionality is provided in the software, with the hardware providing the interface and computing resources. The software functionality can be quite complex with multiple crosscutting aspects (such as performance, reliability, safety, and other embedded systems requirements.) and diverse and stringent interface requirements (e.g., networking in terms of wireless, wired or satellite links). Developing software for such systems can be likened to creating, composing, and conducting an orchestra or symphony, as opposed to playing an individual instrument, and requires a new and powerful methodology. As part of the Yamacraw Embedded Systems (YES) program[1], we are developing a systematic methodology for facilitating rapid and efficient software development for embedded real-time systems. This report presents some initial approaches and describes our product-line oriented, reuse-driven YES methodology drawing upon best practice from industry and academia.

1 Embedded Systems Development

Embedded systems applications range from smart phones, network computers, personal digital assistants (PDAs) to enterprise web servers. These systems, and more specifically systems-on-a-chip (SoC), are characterized by the need for interaction with their environment, requirements for meeting time-driven, power, and weight constraints, need for low cost and efficient implementation in hardware and software, in addition to high levels of reliability and availability. Designing embedded electronics products involves a diversity of creative professionals including systems, hardware, and software engineers involved in *conceptualizing, creating, implementing, testing, and manufacturing* the products. This process requires many steps and iterations, each with specialized tools. A simplified view of this integrated process is shown in the flow chart on the right, highlighting the major steps in going from a set of needs to a product/solution that involves both computer hardware and software.

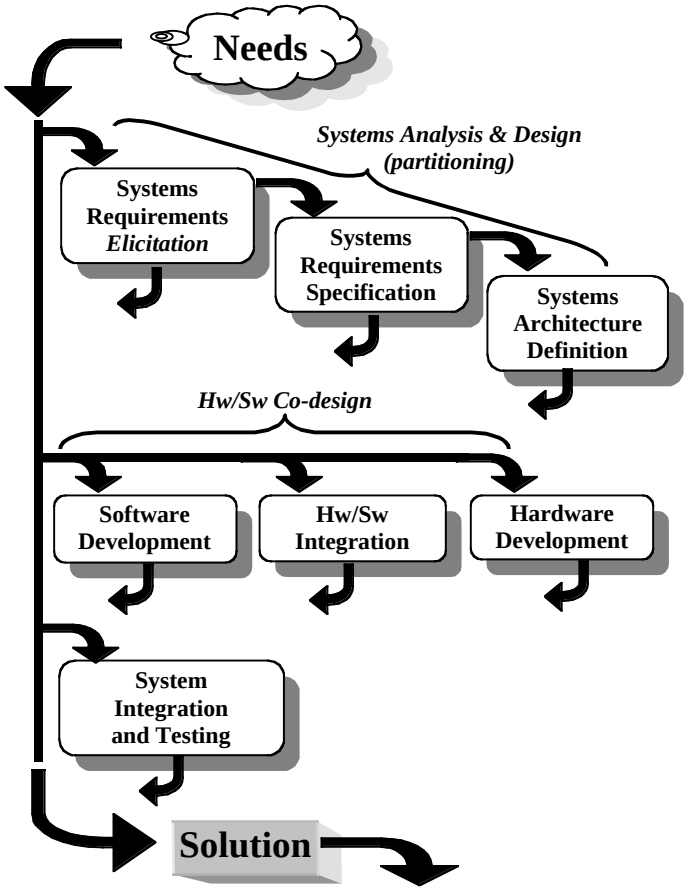


Figure 1: System Development Process

The first three steps are part of *systems engineering*, an area that deals with high-level structures such as the overall system architecture,

[1] http://www.yamacraw.org

and the tradeoffs of partitioning functionality into hardware or software. *Co-design* then follows, where hardware and software engineers are concerned with implementation details and the integration of hardware and software; these steps are usually performed in parallel. Software development includes application and system software. Hardware development provides the computer system architecture and the physical devices. The Hw/Sw integration is a complex task, and includes operating systems, device drives, APIs and the software interfacing the hardware [Madisetti1998].

Embedded system designs are increasingly more software-driven, as more and more of the functionality is placed in software, thus software engineers are being required to perform a greater design and implementation job. This puts much of the cost of designing embedded systems in their software development. Embedded systems applications are highly device dependent requiring that each time a new system is built, routine functionality is custom-written repeatedly from scratch for the hardware being controlled. Such practices have a number of drawbacks and shortcomings as already noted [Brownsword 96; Brown 98].

While traditional software industry has had some measure of success in the past decade (due perhaps to the development of modern programming methodologies and practices), little progress has been observed in the area of development methodologies for embedded systems. For example, only recently, has there been some discussions of the use of object technology for real-time systems; product development/system design life cycle is fundamentally waterfall, and top-down driven. Current approaches are not entirely satisfactory in all aspects of embedded systems development. With increasing pressure to reduce time-to-market, we need to look for ways to accelerate this process. The YES effort is one of the largest and possibly the most comprehensive effort in the world devoted to a systematic study and research in the area of embedded systems design.

We need to move our focus from engineering single systems to engineering systems of systems through a product line approach. This will allow us to come up with the "right" solution quickly, thus dramatically shortening time-to-market while improving quality. The goal is to produce quality software products consistently and predictably by moving toward an asset-supported development process and by adopting a component-based development strategy. The quality of products is maintained through rigorous application of analysis and prediction, continuous improvement and refinement through designed experimentation and feedback from observations [Withey 96]. This move may assist in finding the correct direction toward the more encompassing aim of attaining industrial-strength software engineering, a necessary condition that naturally leads to systematic reuse [Diaz-H 95].

One of the factors that hold reuse back in the embedded systems domain is the known fact of lack of firm standards in many areas. For example, development environments are fragmented, with little interoperability between different levels of tools and poor portability across environments. Industry-adopted connectivity standards between development environments involving mixing languages, cross-platform compilers, and multilingual debuggers do not exist[2]. There is a need for such standards simplifying interfaces between environments and across embedded systems components. In addition, most embedded systems methods focus on developing single systems rather than families of systems. Furthermore, as mentioned earlier, *embedded systems have to optimize the form factor and real-time requirements of the electronics product – including its area and power consumption, volume, and cost – a set of factors often ignored in mainstream software practice*, thus rendering opportunistic reuse ineffective at best. . [Madisetti 1998].

In summary, our goal is then to reduce time-to-market, while increasing quality, by providing a comprehensive embedded systems methodology supported by a rich set of reusable assets.

[2] A noticeable exception is the Ada ISO standard which provides platform independent and paradigm independent technology for embedded systems development, but Ada technology has not been widely accepted by industry.

2 Yamacraw Embedded Systems Methodology: a Model-Based Approach

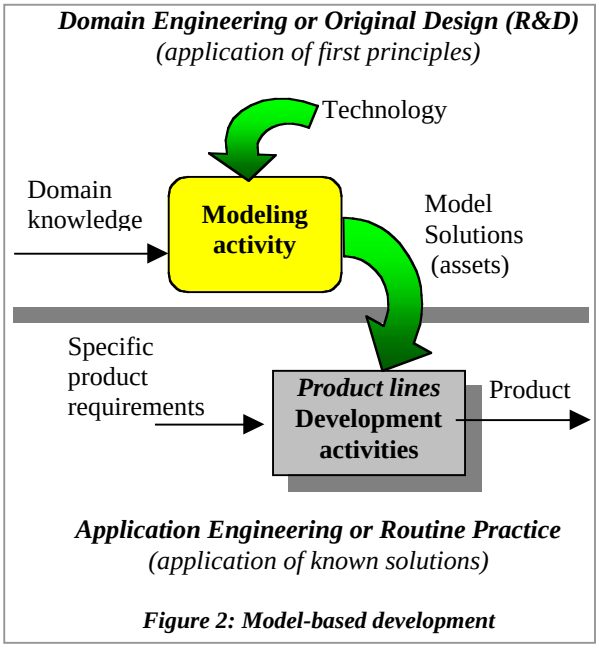


Figure 2: Model-based development

We contend that the causal relationship between building *model solutions* (or assets) and constructing actual *products* from these models provides a candidate basis for industrial-strength embedded software construction. This duality is commonly found in more traditional engineering where a distinction is made between original design and routine practice [Díaz-H. 97; Paul96]. (See Figure 2.) Within this framework, products are created by instantiating models and by integrating prefabricated artifacts, whereby the developing process becomes more a routine practice activity of mapping from needs to solutions rather than a synthesis activity of building from scratch. That is***, setting problems in terms of known solutions, not building products from first principles repeatedly***. In the software realm, the term *Domain Engineering (DE)* is used to refer to a development-for-reuse process to create software assets. This process is realized by creating new solutions using first principles, and sometimes by inventing completely new technology. This usually requires careful technical and economic analyses and goes through all development phases. The complementary term of *Application Engineering (AE)* refers to a development-with-reuse process of producing specific systems by the routine application of prefabricated assets to solve reoccurring problems in a domain. This generally requires original design effort only once. Model-based Software Engineering is an approach in this direction [Withey 94].

We take this approach, discussed below, to directly support the systematic construction of products in the domain of embedded systems for the specific Yamacraw product lines of personal, enterprise, and home applications of embedded electronics products. The reusable assets include software components as well as hardware blocks or "cores."

Fundamentally, reuse (certainly not a new idea, although ASIC design for reuse is new) has the effect of reducing variance and increasing homogeneity, key for any discipline that has moved into industrialization[3]. For this, we must identify the common components that implement functionality typically present in the applications in a domain, and particularly, the way that these may vary from one product to another. Furthermore, those crosscutting aspects would be tackled in a uniform way, thus generating more reusable standard solutions. The successful implementation of this approach leads to systematic software reuse for a specific set of products, i.e., a product line. A common definition of product line is "*a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market or mission. 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.*" [Cohen 95 & 99].

In addition to quality requirements, reusable assets must be more general, and hence adaptable, not designed for a single technology, that is "portable" and independent of specific simulators, capable of being independently verified in different environments (or chips), and fully documented. A critical aspect is the ability to design "reusable" components from specification to silicon with the understanding of their need for adaptation when reused, a more difficult task for hardware cores. Furthermore, we must be able

[3] Statistical process control, for example, has worked successfully in environments characterized by low variance.

to capture the design information in a consistent and "standard" form, in such a way that components can be integrated by engineers other than the original designers.

The Yamacraw embedded software design project encompasses a multi-disciplinary team of diverse researchers grouped into five groups labeled SOW1..5 as follows:

- SOW 1.0 Requirements & Specification Methodologies for Embedded Systems
- SOW 2.0 Smart Compilers for Embedded Systems

} *modeling*

- SOW 3.0 Personal Embedded Computing Environments
- SOW 4.0 Networked & Enterprise Embedded Applications
- SOW 5.0 Home Computing Applications

} *product development*

The framework presented in Figure 1 above is quite generic. In YES, we have broken the domain modeling activity into two converging sets of activities, namely *representation* and *co-design/optimization* (see Figure 3). The development activities correspond to the various Yamacraw product lines. A third element, and one that represents the interface between domain engineering and application engineering, is the repository.

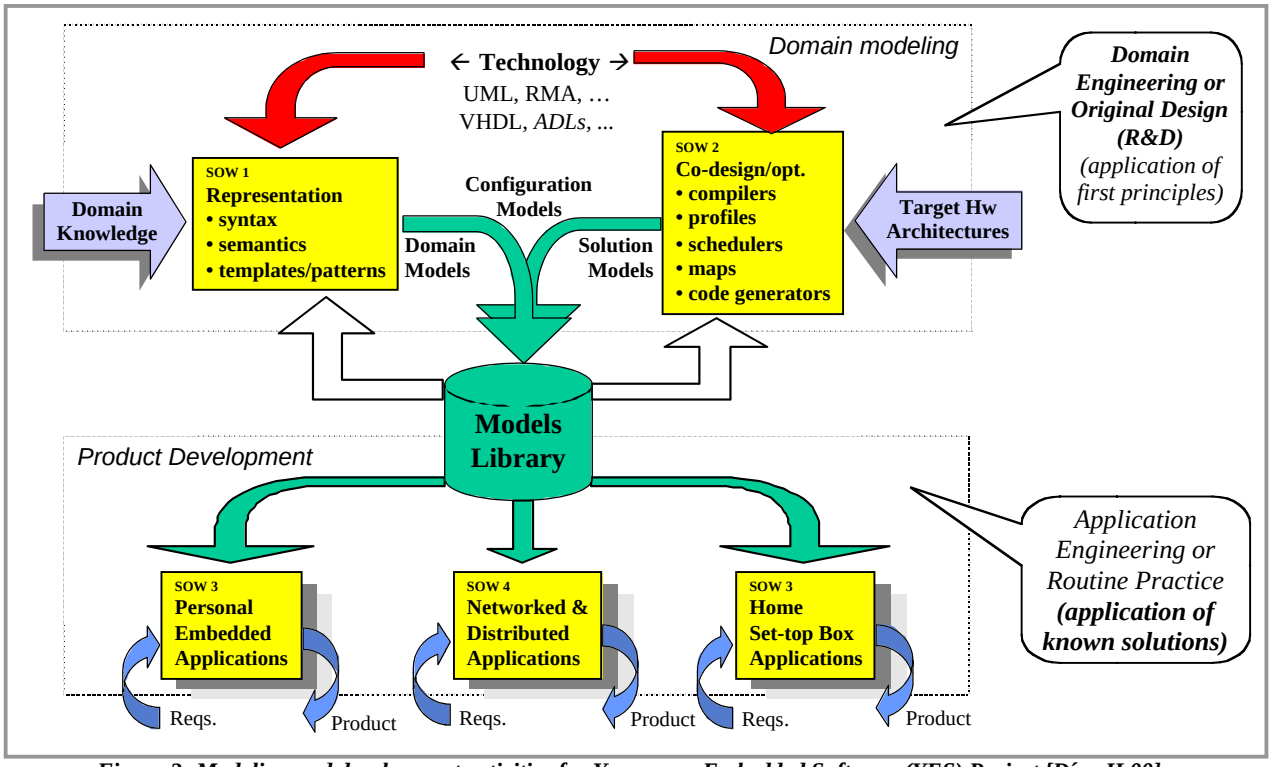


Figure 3: Modeling and development activities for Yamacraw Embedded Software (YES) Project [Díaz-H 00]

The modeling activities define the **solution space** consisting of the "reusable assets;" these take the form of generic components[4], with all their possible combinations. The assets are stored in a suitable repository, or models library, and serve as templates for the generation of the various work-products during a product development. Generative development can be applied, whereby that the application engineer states requirements in abstract terms and the generator produces the desired system or component. There are three parts to the solution space, namely domain models, solution models, and

[4] We use the term *components* in its most widely form to include requirements, software and hardware architectures as well as blocks (both hardware and software code modules) and their test harnesses.

configuration models. A domain model formalizes knowledge about existing systems and requirements for new systems. *Solution models* represent both software and hardware architectures (i.e., components and their interfaces) suitable for solving typical problems in the domain. A *configuration model* maps between the domain model and solution models in terms of product construction rules, which translate capabilities into implementation components. These rules describe legal *feature* combinations, default settings, etc. For example, certain combinations of features may be not allowed; also, if a product does specify certain features, some reasonable defaults may be assumed and other defaults can be computed based on some other features. The configuration model isolates abstract requirements into specific configurations of components in a product line architecture.

A fundamental problem we are facing is the inability of representing all the design information in an appropriate modeling notation, one that is suitable at various levels of abstraction and that can present the design from various view points.

Unified Modeling Language (UML) Extensions: We are conducting research to develop new and powerful extensions to UML to develop a methodology to specify and describe embedded software systems and their underlying models of computation and communication. The extensions, which will constitute our System-Level Description Language, include:

- Extensions to include behavior of analog interfaces within the embedded specification
- Extensions to include real-time related behavior within the specification
- Extensions to include size, weight, area, and power (SWAP) constraints early on the design process.
- Extensions to include hardware/software interfaces and capabilities to develop an executable specification capable of expressing the concurrency in the real-system being described.

Embedded Software Product Lines: Our target market areas are for the development of applications for the personal, enterprise and home-based embedded computing systems. The main objective of this activity is to develop common (semantic) design pattern libraries for embedded applications. A library of building blocks will be developed as part of this research that will allow an embedded software developer to rapidly customize them for various applications within the target market areas using the tools and languages provided with the YESDESK.

We have identified a number of tasks to be performed as part of this activity as follows. To address the question of what components are needed requires special, broad analyses of the problem domain to identify the components "present" in the products of a given domain, a process known as **domain analysis**. The goal here is to produce or to acquire a description of the products that will constitute the product line together with an analysis common and optional features. Components, or blocks, exist at various levels ranging from domain independent IP (intellectual property) such as processors, standard interfaces, common algorithms, to domain-specific blocks (e.g., multi-media, DSP, communication IP, etc.), to application specific components (those intended for a single system).

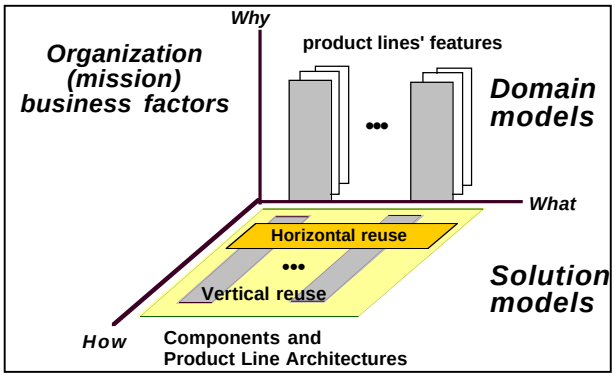


Figure 4: Domain and Solution models

Components are designed to fit a common architecture of the products in a domain to exploit implicit commonality with explicit control of variability. This requires a consolidation of understanding of the software systems in a domain which in turn leads to design for commonality and control of variability of the cross-cutting aspects found in the application in the domain. This understanding is documented in *domain models* representing the common features of the software applications, or product lines, in that particular domain. The synthesis focus is one of control of variability within a product line and the design of

commonality across product lines, which results in the definition of *product lines architectures* and *components designs* documented as *solution models*. Reuse thus actually occurs both within a product line and across product lines, a notion discovered earlier and associated with the concepts of horizontal and vertical reuse. These notions of vertical and horizontal reuse are widely recognized today, and have been formally incorporated in important software construction technology such as CORBA [Siegel 98]. The top layers of the CORBA architecture specify standard objects that can be shared by applications across domains; they are known as the Horizontal CORBA Facilities. Standard objects that are reusable within products in a given product line are referred to as the Vertical (Domain) CORBA Facilities.

Designing the architecture for the product line involves answering questions such as:

- what kinds of components are needed? What are the relevant architectural building blocks, such as Styles, Patterns, and Frameworks, that can be applied towards meeting the product and production constraints?
- Development of model libraries for analog, RF, and digital building blocks: These libraries will facilitate rapid evaluation of various design and implementation alternatives.
- High level descriptions of the features of each product line documented as use-case models and object models. What are the commonalties and variations among the products that will constitute the product line? What are their behavioral and quality requirements? What features do the market and technology forecasts say will be beneficial in the future?
- Will generic components be produced internally or purchased on the open market? What off-the-shelf components should be used?
- How can they be connected? What kind of middleware or component model will be used? What interfaces component categories will have?
- how they will accommodate the requirements, etc.

Inventory of Pre-existing Assets:

- What legacy components could/should be reused?
- What are the software and organizational assets available at the outset of the product line effort?
- Are there libraries, frameworks, algorithms, tools, and components that should be utilized?

We also need to define the software engineering process and support tools for building products from the core assets, and their storage and retrieval. This involves the use of domain-specific languages, code generators and component libraries, and it is supported by component composition technology such as CORBA, DCOM, JavaBeans, ActiveX, etc.

- What standards apply to the products in the product line? What is the underlying infrastructure? What are the time-to-market or time-to-initial-operating-capability requirements?
- Will the product line be built from the top-down or bottom-up? (I.e., starting with a set of core domain-wide assets and spinning off products from those, or starting with a set of products and generalizing their components to produce the product line assets)
- Will products be automatically generated from the assets or will they be assembled?
- What is the mechanism for cataloging, indexing, and retrieving assets? What is the configuration management policy?

Figure 1 represents the canonical, generic design process for embedded systems. Increases in complexity and pressures to shorten time-to-market indicate that the development/product design life cycle become one of incremental development combining bottom-up development with top-down specifications, with increasing iteration between hardware and software designers.

The impact of our research in this area can be significant. It can greatly increase the productivity of the embedded software effort with our new specification languages and tools that draw from the library of templates that will be developed. For the first time, we are able to include analog behavior, in addition to information on constraints on the design, resulting in first pass success in code development [Chonlameth 99]. For example, SoC's specifications are required for both hardware and software blocks, and they

must completely describe the behavior of the system as a whole. The various components are typically specified using different specific technology as illustrated by Figure 5 [Keating and Bricaud 1999].

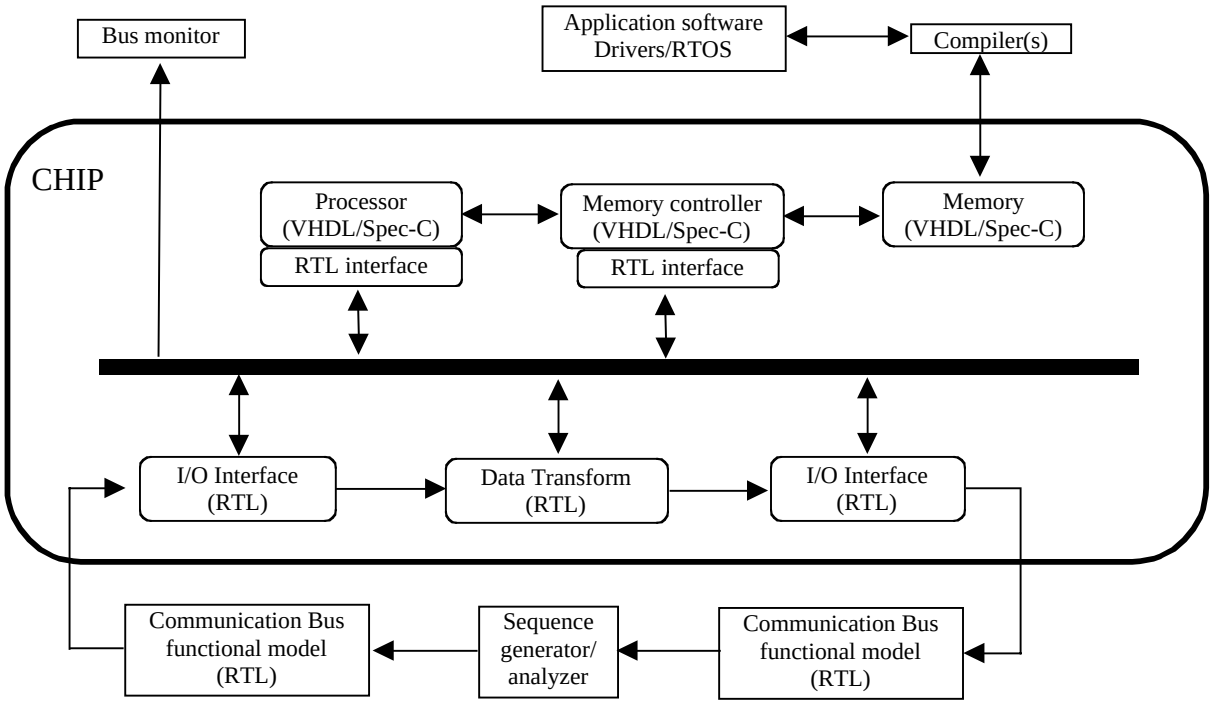


Figure 5: SoC Components specifications from [Keating and Bricaud 1999]

Smart Compilers for Embedded Systems. Most of the detailed code development for embedded applications is done manually and is thus error-prone and also costly and time consuming. Using the leverage provided by the UML-YES extensions in describing the embedded application in a way that provides visibility into its function and interface requirements, we can organize the solution mapping as follows:

- High level mapping of embedded applications on to processing architectures --- Our research is focussed on developing tools that can extract the concurrency from an embedded specification to assist with its efficient mapping onto parallel/multiprocessor embedded systems.
- Development of advanced simulation infrastructure for system-on-chip (SoC) and system-on-package (SoP) application that allow efficiency co-simulation of analog, digital, and RF components that constitute embedded applications.
- Smart compilers for embedded DSP and telecom applications: We are developing new intermediate representations (IR) for compilers that allow capture of those characteristics in DSP and telecom applications that will facilitate their efficient compilation on to modern system-on-chip DSP platforms. The focus is on developing tools, unlike current research that focuses on mainframe applications and their compilation, for embedded application compilation, taking into advantage a library-based approach. The compilers are expected to be able to rapidly be customized for various processor architectures, ensuring high reuse and reduced cost.
- Real-time operating systems: We are developing new specifications for real-time operating systems that are characterized by hardware support for essential features.

The development of a new and efficiency specification for embedded software as in UML-YES, requires support for converting this specification (in an automated manner) to its implementation. For the first time, we are focussing on both these aspects - the specification and its implementation through the process of “smart compilation.” The smart compilation process will synthesize (or provide extensive

support for this process) of efficient embedded systems quickly for an UML-YES specification. This will greatly improve the productivity in developing embedded systems, possibly by an order of magnitude.

3 Summary

The YES effort, one of the largest of its kind, is characterized by its 2-3 year focus of research that will rapidly change the way embedded software in SoC is specified, analyzed, synthesized, and implemented in the near future. Some of the fundamental problems we are addressing include heterogeneous design representation using a common notation (UML-YES), reuse-driven business model for SoC, sources of reusable IP for SoC, Reuse for ASIC development is not common, however, most designs are not unique but modifications/improvements of existing designs (new features, better performance, etc.) or integration of existing blocks into a new larger design.

4 Bibliography

[Brownsword 96] Lisa Brownsword and Paul Clements "A Case Study in Successful Product Line Management" CMU/SEI-96-TR-016. Pittsburgh, PA: Carnegie Mellon University, 1996.

[Brown 98] A. W. Brown and K. C. Wallmau "The Current State of CBSE." IEEE Software, 1998.

[Cohen 95] S. Cohen, Friedman, Martin, Solderitsch, and Webster. "Product Line Identification for ESC-Hanscom." CMU/SEI-95-SR-024. Pittsburgh, PA: Carnegie Mellon University.

[Cohen 99] S. Cohen. "Guidelines for developing a product Line Concept of Operations." CMU/SEI-99-TR-008. Pittsburgh, PA: SEI, Carnegie Mellon University.

[Díaz-H. 95] J. L. Diaz-Herrera, S. Coehn, and J. Withey, Institutionalizing Systematic Reuse: A Model-Based Approach, in Proceedings of the Seventh Workshop on Institutionalizing Software Reuse, 1995.

[Díaz-H. 97] J. L. Díaz-Herrera, "Integrating Architectures, Frameworks, and Patterns: a Model-Based Approach." OPPSLA 97 workshop #26, Object technology, Architectures, and Domain Analysis – What is the Connection? – Is there a Connection?

[Díaz-H 00] J. L. Díaz-Herrera and V. K. Madisetti "The Yamacraw Embedded Software (YES) Methodology: A Technical Analysis." Yamacraw (YES) Technical Report CSIP-TR-00-01, 1/31/2000.

[Keating & Bricaud 1999] M. Keating and P. Bricaud. Reuse methodology Manual, for system-on-a-chip designs. Kluwer, Boston: 1999.

[Madisetti 98] V. Madisetti, "An Embedded Software Research Program: A Proposal." CSIP TR-98-03, 30 August 1998. URL www.ece.gatech.edu/~vkm/TR/

Chonlameth 99 Chonlameth A. and V. K. Madisetti, "Constraint-Based Codesign (CBC) of Embedded Systems: The UML Approach." CSIP TR-99-01, 12 December 1999. www.ece.gatech.edu/~vkm/TR/

[Pahl 96] Pahl, G. and Beitz, W. Engineering Design: a Systematic Approach. Springer-Verlag: Berlin, 1996.

[Siegel 98] Siegel, . "OMG Overview: CORBA and the OMA in enterprise Computing" Communications of the ACM, vol. 41, no. 10, 1998: 37-43.

[Withey 94] Withey, J. " Implementing MBSE in Your Organization: An Approach to Domain Engineering (CMU/SEI-94-TR-1). Pittsburgh, PA: Carnegie Mellon University, 1994.

[Withey 96] Withey, J. "Investment Analysis of Software Assets for Product Lines" (CMU/SEI-96-TR-010). Pittsburgh, Pa.: Carnegie Mellon University, 1996.

Helping Small and Medium-Sized Enterprises In Moving Towards Software Product Lines

Dirk Muthig and Joachim Bayer

Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
{Dirk.Muthig, Joachim.Bayer}@iese.fhg.de

ABSTRACT

The KobrA method developed at Fraunhofer IESE is an approach for component-based product line development. It is based on our experience gained from many industrial projects. In this paper, we present that KobrA's approach of building software product lines is scaleable to small and medium-sized enterprises (SMEs). Therefore, we analyze the typical SME context and the constraints under which these companies usually operate. The result is a set of requirements for product line approaches that are designed for being successful also in the context of SMEs. Finally, we show that the KobrA method does fulfil these requirements and, thus, helps SMEs in moving towards software product lines.

1 INTRODUCTION

Small and medium-sized enterprises (SMEs) operate under specific constraints. Independent of whether they develop market-driven or customer-specific products, their economical success does typically not enable them to do long-term planning. In other words, most of them cannot afford to hire additional people, do strategic development, and thus become a little bit more independent of fast changing markets. Independence of the market means that a company does more than only following the market and fulfilling the already existing requirements but also having an impact on the market itself by predicting, exploring, and thus defining the needs of the future. If no additional resources, however, can be hired, the only way out of the daily fight of surviving in the market is to use the existing resources more efficiently and, thus, free some resources that can be used in a more strategic fashion.

Today, many approaches exist that promise to improve software development practices (i.e., to make the development more efficient) and also promise to improve the situation of SMEs, too. Unfortunately, none of these approaches has really proved what it promises, especially in the context of SMEs where – as described above - only little resources can be spent on the systematic introduction of new practices or on improvements in general.

We claim that an approach that is supposed to be successful also (or especially) in the context of SMEs must take into account the specific needs and problems of companies of this smaller size. We believe that most of the problems in the context of SMEs can be traced back to the fact that many of these approaches have not considered SME-specific issues. That is, one kind of the approaches address only isolated problems and, thus, improve only one aspect of software development, which will – step by step – return to the lower maturity level of all other aspects during the daily work. The other kinds of approaches are big endeavors that are usually not scalable down to pieces manageable by SMEs.

Product line engineering approaches are usually such big endeavors that promise to improve efficiency of software development and are typically not manageable by SMEs. Their idea is to systematically exploit commonalities of systems, which are in the same or closely related application domains, and which will be developed within the same organization.

In this paper, we motivate the KobrA approach as an approach for introducing and exploiting product lines in the context of SMEs. The KobrA approach has been developed at IESE and integrates our experience gained from projects applying product line approaches in general[1] and also transferring product line concepts into SMEs [1].

In the next section, we start with an analysis of SME-specific issues, characteristics, and needs with respect to software product lines. The analysis leads in the third section to general requirements for a product line approach that is applicable within SMEs. Finally, it is shown that the KobrA approach fulfils the requirements identified before and, thus, is an approach for component-based product line development that is also applicable in

[1] These projects have been performed in the context of PuLSE™ (Product Line Software Engineering), which is a registered trademark of the Fraunhofer IESE.

the context of SMEs.

2 ANALYSIS OF THE CURRENT SITUATION

In this section, we, first, analyze the status of product line approaches and why these approaches are not applied in the context of SMEs. Then, we analyze the market for product lines with respect to SMEs. In the next section, we then summarize the results and list the issues that are not tackled by the existing product line approaches but that must be resolved to successfully introduce any one of them.

2.1 State of the Art

In the early days of domain engineering, people tried to completely analyze, model, and exploit theoretical domains. These approaches were not accepted in industry because the necessary effort was too big and the systems eventually be developed covered only subsets of the theoretically defined domains.

As a consequence, people moved from domain engineering to product line engineering. The main difference between these two approaches is the way of defining and bounding the domains of interest. In product line engineering, a domain is defined by what is covered by the set of systems, which must be developed and which are seen as a single system family. Hence, the effort spent on domain analysis is only spent on analyzing things that are really needed. But the effort that must be invested in introducing these approaches are still large because the approaches still assume that the whole organization must be transitioned into an organization able to manage product lines.

Hence, product line approaches are big endeavors, which can only hardly be scaled down, and thus they are typically not manageable by SMEs.

No approach has ever explicitly tried to be make product lines exploitable in the context of SMEs and, thus, there is no experience reported that product lines in general are not manageable by these kinds of organization.

For us, the reason that none of the existing product line approaches aimed at the SME market is that developers of these approaches questioned at least on of the following assumptions.

1. There are SMEs that (plan to) develop different systems in the same application domain whose commonalities can be exploited.
2. There are SMEs that can benefit from the introduction of product line engineering.
3. There are SMEs that have the capability needed for building and managing product lines.
4. There are SMEs that have the resources with respect to quantity and quality needed for building and managing product lines.

2.2 State of the Practice

In this subsection, the state-of-the-practice is analyzed according to the four assumptions listed above, which kept people away from developing a product line approach for SMEs.

A company that develops software either develops systems for single customers individually or in a market-driven way.

The former case, a company develops systems individually for single customers, fits ideally to the product line concept: similar systems typically in the same application area are (sequentially) developed within a single organization. The application domain the SME focuses on is typically selected when the company is founded. It is the domain in which initially know-how exists and first potential customers have been identified.

In the other case, a company develops one or more market-driven systems, which are systems that are produced for the market in the hope to sell many copies, it seems (at the first sight), that per definition no product line exists in these environments. Unfortunately most of the SMEs count themselves to this type of organization.

However, a study concerning the state-of-the-practice within SMEs reports on characteristics common to all regarded SMEs [2]. The software systems these companies develop must all be adaptable to customers' and users' needs and the main triggers for most of their projects are technology changes (e.g., move systems to a new operating system), requests for new features from the market, or request for customer-specific adaptations. That is, these organizations spent most of their resources on tailoring their systems to the needs of individual customers or enhance the system by features that are newly required by customers.

Product line engineering supports both activities: individual adaptations (i.e., the development of variants) and planning for integration of features only needed. Hence, also SMEs building so-called market-driven system have a product line - even when they have never looked at their system that way.

The result of this analysis is that independent of what kind of systems SMEs built, in most cases the systems can be regarded as a product line. So, there are SMEs that develop different systems in the same application domain whose commonalities can be exploited.

The next concern is the question whether SMEs can really benefit from regarding their systems as a product line. Real benefits mean that they profit more from the product line approach than they would do from any other approach that improves their maturity in general.

As described above, the solutions and improvements product line engineering promises match the problems

most SMEs face. Beside this improvement of the development of the systems these companies build anyway, the product line idea supports them in systematically increasing their target market. More adaptations and variants can be realized with the same, available effort, as well as parts of their systems can be separated and marketed as third-party components in different but related application domains. Additionally, the increased number of delivered products makes the company more visible and thus simplifies the search for potential future customers. In general, there are SMEs that can benefit from the introduction of product line engineering.

The next concern is the question whether SMEs have the capability to build a product line, to exploit, and to manage it. Our project experience tells us that SMEs are very creative in finding alternative ways of selling their products in different contexts. They are also very flexible due to the fact that they are used to adapt very quickly to changed markets or to new but related opportunities. They also have a good knowledge about their domain and the needed variants in that domain. This is a consequence of working closely with their customers and continuously collecting diverse requirements. In general, SMEs have good domain knowledge and many ideas of potential products.

Their problem is the lack of mature capabilities needed for exploiting their existing knowledge. Typically, they do only little strategic planning, they do not have well-defined processes and products, they document too little of their knowledge explicitly, and their project management is rather weak. Hence, SMEs have the domain knowledge and the mentality needed for building and exploiting a product line but they lack overall in capabilities needed to realize it technically.

Last but not least, there is the question whether SMEs have quantitative and qualitative sufficient resources needed for building and managing product lines. In general, each SME has at least some people that are very experienced in the domain and in the built applications. Similar to an SME's organization as a whole, the people there have deficits concerning capabilities needed to realize product lines, such as systematic development, or talking and writing about domains and systems at a more general, abstract level, etc.

The more important issue is the quantity of resources: their number of resources is very limited but there are usually also some resources available. Unfortunately, their best and most experienced people are the most needed for both the daily customer business and the construction of a product line. So, even SMEs lack in their technical capabilities, they do have, however, the people needed to do product line engineering.

The essence of the analysis is that SMEs have product lines, they can benefit from the introduction of product line engineering, they do have the knowledge to build product lines, and they miss a good understanding of their product line, as well as technical capabilities needed for the realization.

3 PRODUCT LINE APPROACH FOR SMES

In this section, we describe the requirements, which are derived from the analysis above, for a product line approach applicable in SMEs.

- The approach must take care of the immaturity of its target environment. That is, the approach must also aim at the introduction of fundamental software engineering capabilities (e.g., writing documentation) in conjunction with the introduction of product line concepts.
- The approach must provide the possibility to introduce new techniques and concepts step by step. Thereby, it is important that each step results in some visible improvement for the SME.
- The approach must be able to step in the work in progress because SMEs cannot stop their current work. So, they have to continue working on the existing products and cannot start freely to build a product line. Hence, the approach must integrate (parts of) existing products into a planned product line.
- The approach must be centered around the evolution of product lines because SMEs do typically not plan for a set of variants that must be build but they continuously change their product portfolio by adding new variants or changing the existing ones.
- The approach must rely on standard notations and tools. A SME, typically, cannot risk going into a different direction than the main stream and, thus, relying on exotic solutions, which include relying on special, unusual tools, education, etc.

Beside these more technical requirements for an approach, it is important to note the introduction of the approach must be guided by external people that are experienced in working with SMEs, in the particular approach, as well as in building product lines in general. The guiding consultants bring in alternative viewpoints on an SME's products by frequently recalling the underlying product line ideas and pointing to existing exploitation possibilities.

4 THE KOBRA METHOD

At Fraunhofer IESE, PuLSE has been developed as a methodology for product line engineering, which is customizable to its application environment [3]. PuLSE has been applied successfully in various different contexts for different purposes. However, in projects with SMEs without pre-existing processes and well-defined products,

the introduction of PuLSE turned out to be problematic. Therefore, the KobrA method has been developed as a "ready-to-use", object-oriented customization of the PuLSE method [4].

The KobrA method represents a synthesis of several advanced software engineering technologies, including product line development, components-based development, frameworks, architecture-centric inspections, quality modeling and process modeling. These have been integrated in KobrA with the basic goal of providing a systematic approach to the development of high-quality, component-based application frameworks.

The KobrA method is also applicable in SMEs because it is designed to fulfil the requirements for such an approach stated above.

- As a consequence of integrating best practices from a large and diverse set of software engineering technologies, KobrA already contains processes supporting the complete software development life cycle. Hence, it also provides processes for the processes that may be missing in immature organizations.
- In the center of KobrA is a tree of components that describe either a generic framework or a particular application. Each component is described by a set of models, mainly UML models [6]. The types of model that belong to the model set are well-defined but it is not necessary to build all types of models all the time. Therefore, KobrA provides guidelines to decide which models should be built under which circumstances. That means, KobrA is scalable. Additional processes for project management or quality assurance can also be added when needed and when the necessary resources are available.
- In KobrA, all products are organized around, and oriented towards, the description of individual components. This means that, components (and the products that describe them) can easily be separated from the environment in which they were developed and reused independently. This characteristic allows introducing KobrA initially for single components. Components that are already existing and thus KobrA is integrated component by component with existing systems.
- The idea of modeling variabilities within KobrA is related to Gomaa's idea of integrating variabilities in object-oriented models [5]. That means, that variabilities can also be added to models that were build before without variability. In KobrA, the component tree describing a single application is extended by integrating variant parts and, thus, is transitioned into a generic framework that embodies several product variants.
- KobrA describes components in terms of a mixture of textual and UML-based (graphical) models. So, it is conform to the most popular modeling notation, which is supported by many commercial modeling tools. Additionally, industrial project members of the KobrA project develop a workbench to naturally support the KobrA method.

In this paper we have shown that software product lines are an issue for small and medium-sized enterprises and that the KobrA is a suitable approach for introducing and exploiting product lines in the context of SMEs.

REFERENCES

[1] Knauber, P., Muthig, D., Schmid, K., Widen, T., *Applying Product Line Concepts in Small- and Medium-Sized Companies*, submitted to: IEEE Software Special Issue on Software Engineering in-the-Small

[2] Kamsties, E.; Hoermann, K.; Schlich, M., *Requirements Engineering in Small and Medium Enterprises. State-of-the-Practice, Problems, Solutions, and Technology Transfer*, Conference on European Industrial Requirements Engineering 1998, pp.40-50, Industrial Program Papers, London 1998

[3] Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., and DeBaud, J.-M. *PuLSE: A methodology to develop software product lines*, In Proceedings of the Symposium on Software Reusability (SSR'99), May 1999.

[4] Atkinson, C., Bayer, J., Muthig, D., *Component-Based Product Line Development: The KobrA Approach*, accepted by the First International Software Product Line Conference, Denver, 2000

[5] Gomaa, H., Kerschberg, L., Sugumaran, V., Bosch, C., Tavakoli, I., and O'Hara. L. *A knowledge-based software engineering environment for reusable software requirements and architectures*. Automated Software Engineering, 3(3,4), pp. 285–307, August 1996.

[6] Unified Modeling Language (UML) Resource Center. http://www.rational.com/uml/. December 1999.

Product Line Viewpoint and Validation Models

Nadar Nada, L. Luqi
Naval Postgraduate School
C.S. Dept. Code CS/ 833 Dyer Rd.
Monterey, CA. 93943 USA
+1 831 656 4075
nnada,luqi@cs.nps.navy.mil

Khaled Jaber
Case Western Reserve Univ.
C.S. Dept./10900 Euclid Ave.
Cleveland, OH. 44106 USA
+1 860 2149
jaber@lucent.com

David Rine
George Mason University
C.S. Dept. MS 4A5
Fairfax, VA 22030
+1 703 993 1546
drine@gmu.edu

ABSTRACT
A product line is a group of systems sharing a common, managed set of features that satisfy specific needs of a selected market or mission. In the product line approach, management, system developers, and a reuse team are interested in some views of the product line. In this paper a model is defined to present product lines, its derived products, and common assets used in these product lines. The model is used to convey views of interest to different stakeholders: management, system developers, and a reuse team in the product line approach. Its purpose is to capture information and present this information about organizations' product lines, and make it visible to the stakeholders inside and outside organizations. Management can use the model when producing new products of a product line, negotiating with customers, and assessing the benefits of adopting the product line approach. Product line developers can use the model when developing products of a product line. A reuse team can use the model through asset identifications, ensuring a successful use of asset base in and across product lines, and assessing the level of reuse.

Keywords
Product line, Product line architecture, COTS, Organizational components, Stakeholders, and System-unique components.

1 INTRODUCTION

Organizations that develop similar products are adopting the product line or product family approach to deploy systems faster, at a low cost, and a high quality. Systems are produced in a product line using common architecture and assets that are used across products. Organizations reuse common assets, integrated assets, etc. that would otherwise have to be needlessly repeated for each system.

Each stakeholder, i.e. management, systems developers, and reuse team is interested in a particular view of the product line. Management, for example, might be interested in viewing products of a product line to estimate time and schedules. Systems developers might be interested in a view of a product line looking for common assets. The reuse team might be interested in a view of a product line to assess the level of reuse in a product line. These are some of the interesting views.

We are presenting a product line viewpoint model that shows different views of the product line, its derived products, and common assets used. Also we are showing how the model conveys particular views interesting to management, systems developers, and reuse team.

Section 2 describes the product line concept. Section 3 describes the product line model. Section 4 describes views captured by the model. Section 5 is an empirical model for product line validation. Section 6 represents a repository support. Section 7 is the conclusion.

2 PRODUCT LINE CONCEPT

A product line is defined as a group of products sharing a common, managed set of features that satisfy specific needs of a selected market or mission [1, 4]. Products in the product line are engineered through customization from base requirements and standard product line architectures, and integration of common components rather than using system-unique software [2].

The product line architecture is one of the important assets shared by the systems in a product line. It provides the structure for building systems in the product line. All products are based on the product line architecture.

Product line assets are used across products in the product line. Product line assets depend on the solutions common to the products in a product line. Reusing these solutions reduces or eliminates work that otherwise would be required to build each product [3].

In the product line development, a dual life-cycle model can be used in which domain engineering is the process used to create domain artifacts useful across the entire

Table 1 The Viewpoint and Attribute Template.

Viewpoint Template

Reference	The viewpoint name
Attributes	Attributes providing view point information
Tasks	A reference to a set of event scenarios describing how viewers interact with the product line and their tasks
Sub-views	The names of sub-viewpoints

Attributes Template

View Entities	**Attributes**
Product line	Name, owner, intended market.
Product	Name, contact person, customer(s).
Product line architecture release	Contact person, release number, number of times reused, development time, number of staff, used architectural style, inter-component used communication mechanisms, operating systems(s)and platform(s).
Product release	Customers, release number, contact person, development time, development cost, when developed, number of staff, status, operating system(s) and platform(s).
COTS component release	Name, vendor, release number, contact person, cost, number of times reused, operating system(s) and platform(s).
Organizational component release	Name, release number, contact person, developed internally or externally, development cost, number of times reused, development time, number of staff, operating system(s) and platform(s).
System-unique component release	Name; release number, contact person, development cost, development time, and number of staff, operating system(s) and platform(s).

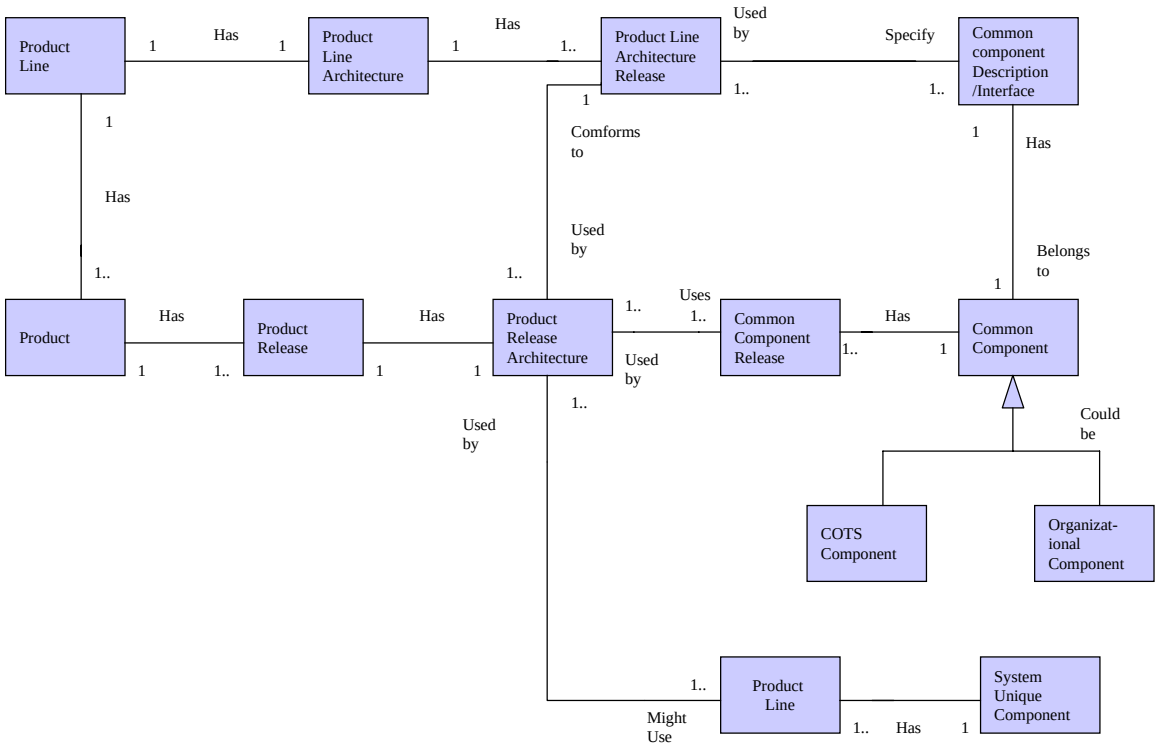


Figure 1 Product Line Views Model

product line, and application engineering is the process used to produce a single product by adapting the domain-wide assets [1].

3 PRODUCT LINE VIEWS MODEL

A product line model that shows different views of a product line, its derived products, and common assets used is presented in this section. It defines entities and relationships between these entities to present product lines. It presents different ways to viewing a product line keeping in mind enhancement, modification, other models, other entities and relationships. Figure 1 depicts the model. The following sections describe the product line views-point model.

3.1. Product Line Overview

A product line is defined as a group of products sharing a common, managed set of features that satisfy specific needs of a selected market or mission [1, 4]. A product line has a group of products associated with it; it has a 1:M relation with its products. A product line has a common architecture associated with it; it has a 1:1 relation with its architecture.

3.2. Product Line Architecture

Product line architecture provides the structural elements and their interfaces by which the system is composed out of the product line [18]. Products are customized using the product line architecture. Product line architecture might evolve during the product line life cycle. New releases of the product line architecture could be seen and this is due to change in customers' requirements, new technologies, design fixes, etc. It has a 1:M relationship with its releases. The early releases of product line architecture specify the common components used in the product line architecture; they could specify the functionality needed by these components and might specify their interfaces. An M: N relationship is established between product line architecture release and common component description/interface. After common components are developed, later releases of product line architecture might refer directly to common component releases. A product line architecture release is used by many products' releases; it has a 1:M relationship with their architectures.

3. 3. Products

Products in a product line are engineered through customization from base requirements, standard product line architectures and integration of common components, and might use system unique components. Each product is associated with its releases. Each product release has architecture associated with it called product release architecture. Product has a 1:M relationship with its releases, whereas, product release has a 1:1 relation with its architecture.

3. 4. Product Release Architecture

Product release architecture is derived from the product line architecture release and must conform to the product line architecture release. It uses many common components described by the product line architecture release; for each common component used, it uses one of the releases of that component. In addition, it might use many system-unique components; for each system-unique component used, it uses a release of that component.

3.5. Components

Components are the building blocks of products in a product line and are classified into two categories: a common component and system-unique component. A common component is used across products of a product line and could be a commercial-off-the-shelf (COTS) component or an organizational component. Organizational components refer to common components developed by the product line organization. They could be developed internally by the organization owning the product line or externally by a different organization within the business unit of which the organization is a part. A system-unique component is used in specific products. Both types of components, common and system-unique, could have releases associated with them and have a 1:M relationship with their releases. They are used in many product releases and have an M: N relationship with product architecture release.

3. 6. Viewpoint Attributes

Entities in the viewpoints have some interesting attributes. Table 1 represents the viewpoint and attributes template. Organizations that adopt the product line approach might be interested in other attributes; these attributes can be added to the table. The attributes listed in table 1 are used to support the views described at section 4 in this paper.

4 PRODUCT LINE VIEWPOINT MODEL

In the product line approach, product Lines share several different views that are interesting to management, system developers, and a reuse team. Other interesting views might be possible.

4.1. Management View

Management of an organization that adopts the product line approach has authority, vision, and leadership. It manages the development of products in a product line. They manage staffing, training, cost, directions, and schedules through the product line cycle. They have a clear vision about the direction of a product line. They interact with customers and make business decisions.

Management in the product line approach can be interested in the products derived from a product line, customers of these products, and customer contact persons. Also they can be interested in cost, contact persons, time intervals, and staffing for products and assets used in these products.

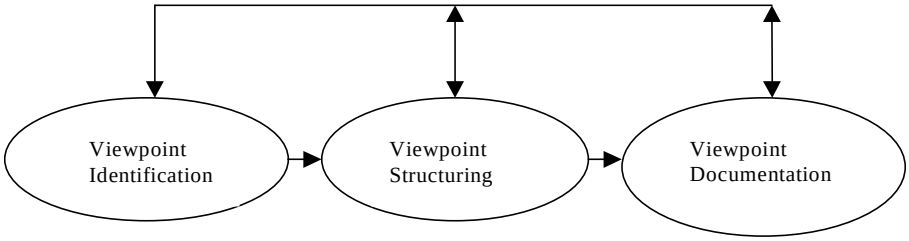


Figure 2. Viewpoint Development Phases

Table 2. Experimental Model Phases for Product Line Validation

Phase	Function	Data Collection Methods
1. Adoption	Assessment	Survey, Legacy
2. Planning & Management	Measurment & Control	Survey, Legacy
3. Utilization	Monitring	Case Study, Project Monitor
4. Expansion	Adaptation	Case Study, Survey, Legacy

This data is supported by the mode. Management can use this data when producing a new product of a product line, negotiating with customers, and assessing the benefits of adopting the product line approach.

The structural of management view and its relationships presented by the model answers questions related to what are the products of a product line and assets used in these products. Attributes used in model's entities answer questions related to who is the customer, contact person, time interval, cost, staffing, etc., of products in a product line.

4.2. **Reuse Team View**

A reuse team of an organization that adopts a product line approach supports reuse across product lines. They support reuse of components through asset identification. With systems developers they ensure successful use of asset bases in and across product lines. They assess the reuse level across product lines. Reuse team can be interested in viewing product lines, their derived products, and reusable assets (product line architectures and components) used in a product line. They can also be interested in the number of times an asset is reused, and the type of components used in a product line.

The structural of reuse team view and its relations presented by the model shows products of a product line and assets used in these products. Attributes used in the model's entities answer question related to the type of components used, number of times an asset is reused**.**

The reuse team can use this information through asset identification, ensuring a successful use of asset base in and across product lines, and assessing the level of reuse.

4.3. **Systems Developers View**

System developers in the product line approach are also interested in viewing product lines, their derived products, the product line architecture, its evolution, assets used and their evolution, the operating system(s) and platform(s) are used, components types, their interfaces.

The structural of system developers view and its relations presented by the model shows the products derived in a product line, the product line architecture, its evolution, components used and their evolution. Attributes used in the model's entities answer questions related the contact person of an asset, components interface, component type, operating system(s) and platform(s).

4.4. **Viewpoints Development**

We used the method called VORD [17] for the development of viewpoints. Also, this method is principally intended for requirements discovery and analysis, it includes steps that help to translate this analysis into a viewpoint. We considered only the first three stages of the VORD method concerned with viewpoint identification, structuring, and documentation.

a- ***Viewpoint Identification*** involves discovering

stakeholder viewpoint and identifying the specific attributes, tasks, and sub-viewpoints.

b- ***Viewpoint Structuring*** involves grouping related viewpoints into a hierarchy. Common viewpoints are provided at higher levels in the hierarchy and are inherited by lower-level viewpoints.

c- Viewpoints documentation involves refining the description of the identified viewpoints.

Viewpoints and attributes information in VORD are collected using standard forms. The form used for viewpoint information (the viewpoint template) and attributes information (attributes template) are shown in Table 1.

The viewpoints and attributes templates, as well as the viewpoint hierarchy diagrams are developed during the three phases shown in Table 1. The templates are used to structure the information collected, and in general a template cannot be completely filled in during single activity.

5 EMPIRICAL MODEL FOR PRODUCT LINE VALIDATION

In this section an experimental integrated model for product line pilot project planning, measurement, and assessment is presented. This section discusses how qualitative and quantitative process and product line goals are established based on customer and business needs. The process of flow-down of goals to the level of processes and the experimental pilot model is described. Table 2. presents the empirical and engineering model phases for product line validation.

5.1. Making the Product Line Adoption Decision

Product line adoption is defined in the context of an organization rationale to agree, sponsor, commit, or allocate resources for initiating a product line plan or project. Product line utilization is defined in the context of an organization as the creation of assets with the specific "intention" to be reused as well as the utilization of assets that had been specifically created with the "intention" of being reused. Product line management is defined in the context of an organization that manages the creation, utilization, and evolution (i.e., maintenance) of reusable assets. The application of software reuse technologies to planned products (both new and existing) and planned product lines is an indicator that software reuse adoption is strongly correlated with organizational opportunities.

Most software development organizations operate according to marketing and finance strategies. An organization wishing to improve its financial status may look for new or extended opportunities in software product markets. Product line is one possible approach that may be used to leverage decreased time to such markets with decreased effort and increased product quality.

So the first step is to make the product line adoption decision based on some empirically validated software reuse reference model (RRM) [Nada 97]. This in turn will lead to a set of decisions balancing market opportunities with market risks. This step will also identify reuse opportunities, reuse objectives, costs, constraints, and options.

For adoption decision organizations conduct an analytical study to decide either to adopt certain product line process or technology or not. This study collects both qualitative and quantitative benchmark data on the product line approach.

The adoption phase includes several steps to evaluate the technical and organizational aspects of the introduced product line process or technology.

5.1.1 Organization context

Organization context describes the environment in which the organization exists or existed when it launched the product line effort. The following lists common factors that are used in the adoption phase to evaluate the existing environment before applying the product line approach. The following factors will be used to record and evaluate the context environment of organizations adopted the product line approach. Also it used by organizations exploring the transition to the product line approach.

Process or technology objective. To adopt the product line approach; the objective of developing product lines needs to be addressed and defined. This includes defining the scope of the product line, how long the organization has been building product lines, and the product line life cycle.

Costs/benefits. Organizations that already adopted these processes or technologies should have data related to the costs and benefits of this adopting. Organizations that are thinking to adopt a software reuse approach might not have data about the cost of adopting this technology, but the benefits of software reuse approach should be defined. Cost varies based on the size and the number of products in the organization, the technical experience, organization structure needed, skills and training, and tools.

Commonalties and variabilities. Organizations exploring the transition to software reuse approach should identify which products can be considered and what their commonalties and variabilities

Common architecture. Organizations exploring software reuse approach should consider the feasibility of common architecture for their products. Also the style of the architecture might be defined, e.g. layered architecture, client server architecture, etc.

Assets used. In software reuse development approach products are assembled using common set of assets and might use system unique assets. Assets could be domain

models, communication protocol descriptions, user interface descriptions, code components, type of common components that developed internally or by using Off-The-Shelf "COTS" components, application generators, domain knowledge, test plans and procedures, requirement descriptions, performance models, metrics, etc. Organizations adopted the software reuse approach records the common assets used in their products. Organizations exploring the transition to the product line approach should define what are the common assets exist.

Level of reuse. One of the benefits of adopting software reuse is increasing the level of software assets reuse in organizations. Organizations adopting reuse approach should have or find other organizations data related to the percentage of reuse achieved in adopting the this approach. Also the type of reuse used, for example, horizontal reuse or vertical reuse. Horizontal reuse represents wide domain width reuse, i.e. a component that can be used in many applications. Vertical reuse represents a narrow domain width reuse, i.e. a component that can be used in one application.

Organization structure. The organization's structure for developing one-at-a-time systems might not be suitable to product line development. Adopting a product line approach has an impact on organization structure. This factor defines the impact of the new structure needed to adopt the product line approach. The impact might be low, medium, or high.

Process. Process used in developing one-at-a-time systems will not be suitable to the product line development. As part of adopting reuse technology, existing process might be modified and new processes need to be in place, e.g. customer interface process, software development processes, etc. This factor defines the impact on the organization processes by adopting new approach, what type of the processes need to be changed, and what type of new processes needed.

Training. Transitioning to new processes or technology requires skilled personnel to achieve a successful transitioning. This factor defines the type of training needed, e.g. in house training, external consultant, etc. Also it defines who needs training, e.g. management, systems developers, etc.

Tools. This factor defines which tools are needed in software development, e.g. tools to assemble products, configuration management tools, tools to record the progress of the product line development, etc.

Software reuse assessment is the main function of this phase. Historical methods are used to collect data, e.g., survey and/or legacy

5.2. Product Line Planning

Organizations use this phase as a plan for the transition to product line software development approach. Organizations can use this phase to record, evaluate, and assess the planning for the product line approach. Organizations intending to adopt software reuse use this phase to put the software reuse in practice.

The following include the implementation plan for software reuse approach; a list of common factors is described in this section as part of the planning phase.

Management Support. Building software products is not just an engineering agenda, it precipitates changes in personnel, personnel management, incentives, customer interface, scheduling, budgeting, and a whole host of management practices. It is a new vigorously and actively supports the transition, the effort will fail. Software reuse strategy means that organizations and managers have less direct control over their product developments and increased dependency on other organizations to understand their requirements and provide acceptable solutions. Giving up this control and the necessary dollars to support product line technology and application development may be difficult. Organizations adopted the software reuse approach should record their experience of the management support, evaluate, and assess that support.

Cultural change. The software reuse concepts should be defined and understood by people of organizations adopting this new approach. A particular attitude that had to be overcome was the one-at-a-time mentality of building a system for its own sake rather than as a contributing effort to the organization's strategic goal of fielding and building up a base set of core assets. Software reuse terminology should be defined and understood across organization.

Organization structure. Adopting new technology or process has an impact on the organizational structure. For example organizations develop product line has a structure different than organizations develop one-at-a-time systems. Some organizations has a product line structure where a marketers group relate product line capabilities to prospective customers; relate customer needs to asset and application developers. A core assets group develops architecture and other assets for product line. An application group deliver systems to customer. There are different players in the product line approach and they should have different skills to launch the product line approach. Transitioning to the product line approach requires the organization's structure and players in the product line approach to be defined.

Training and processes. Transitioning to software reuse involve education and training on the part of management and technicians. Managers need to support the business motivation and strategy of the software reuse approach. They need to understand and role of the infrastructure technologies, understand how to monitor progress and

identify potential problems within their area of the program. Different type of training might be needed; Formal training, on-the-job mentoring from external consultants, etc.

New processes are needed to develop a product line is different from processes used in developing one-at-a-time systems. These processes might be customer interface processes, development process, resource ownership processes, etc.

Training and processes changes should be defined in the transition to the product line approach.

New technologies. Technologies allow organizations to stay a competitive edge. Some of the technologies, for example, used in the production of product line are domain engineering and application engineering. Domain engineering used to create artifacts useful across the entire product line. Application engineering is used to produce a single product by adopting the domain-wide assets. Other technologies, for example, using CORBA, COM, etc. These technologies need to be defined in the transitioning to software reuse development approach.

Tools support: Using tools to support the new development approach increase organizations' productivity. Some organizations use tools that are used to assemble products together. Others use tool to capture domain knowledge, etc. These type of tools used needs to be defined in the transition phase.

Software reuse measurement is the main function of this phase. Historical methods are used to collect data, e.g., survey and/or legacy.

5.3. Utilization and Management

Product line utilization is defined in the context of an organization as the creation of assets with the specific "intention" to be reused and the utilization of assets that had been specifically created with the "intention" of being reused.

The next step is to decide upon the levels of the RRM utilization and management and to look closely at any significant changes or impacts on both top and middle management. This step includes the assessment of an organization's willingness to adopt the RRM, the implementation levels, and the incremental investment strategies.

5.3.1 The Product Line Utilization.

Asset Utilization The objective of processes in this family is to utilize existing assets in software development and evolution (i.e., maintenance) activities. The processes for this family consist of developing or selecting criteria for asset identification, modifying or tailoring selected asset(s), and integrating the selected asset in the system under development or evolution

This step is the actual production phase by applying evolutionary approach (Boehm Spiral Life-Cycle Model) to the reuse plan implementation. Our early research results have shown that software development organizations at a high success (capability) level usually carry out several pilot (experimental) projects to help them in the construction of a prototype repository, component model definition, components classification scheme definition, domain model, common architecture, and product-line as follows:

I. Develop a prototype (pilot project)

II. Learn and evaluate of risk versus opportunities

(including assessment of effort, quality, schedule, tools, and procedures)

III. Expand prototype to a safer version of product line with the necessary adjustment

Repeat step (II) and (II) until you achieve a stable product line version.

This approach to the successful learning and evolving the RRM within an organization is like the Boehm Spiral Life-Cycle Model [8] applied to the RRM implementation plan.

5.3.2 Product Line Management

Reuse management is defined in the context of an organization that manages the creation, utilization, and evolution (i.e., maintenance) of reusable assets.

Asset Management and Control: The objective of processes in this family is to develop and organize collection(s) of quality reusable assets, define and develop services and capabilities to access these assets (i.e., for asset utilization processes), and establish, support, and enact a broker role for asset developers (i.e., from asset creation) and asset consumers (i.e., from asset utilization).

The reuse management and control is based on the classic plan, enact, and learn cycle. The plan, enact, learn cycle in the reuse management idiom is based on the following principles as described in the STARS CFRP [11].

Software reuse monitoring is the main function of this phase. Observational and historical methods are used to collect data, e.g., survey, case study, historical analyze and/or legacy

5.4. Product Line Expansion

In this phase, organizations look for new product opportunities and asses the customer needs and reuse future plan.

Determining and evolving the future objectives, strategy, and scope of a reuse program, resulting in selection of a set of suitable domains and products lines in which to apply reuse within an organization. Planning, establishing, monitoring and evaluating Reuse engineering idiom (asset

creation, asset management, and asset utilization) projects addressing the selected domains and product lines. Looking for new market opportunities, market analyze, and assess the future financial plans.

Software reuse adaptation is the main function of this phase. Observational and historical methods are used to collect data, e.g., survey, case study, historical analysis and/or legacy.

6 REPOSITORY SUPPORT

Organizations adopting the product line approach can use a repository to implement the model. The repository supporting the product line approach can capture the entities and their related attributes, and the relationships between these entities to covey the model's views. A web-based repository is a good choice to implement the model. It provides and easy access for many users internally or externally to organizations developing product lines. The Web-based repository can model the entities, some of their related attributes, and relationships as Hyper-text links to present a complete picture of the entire product line.

7 CONCLUSIONS

Organizations that produce similar systems are moving towards implementing the product line approach. Products in the product line approach are engineered through customization from base requirements and product line architectures, integration of common components and system-unique components.

The model described in this paper is intended to capture a view of the product line, its derived products, and assets used in the product line. The model is defined to present views interested to management, system developers, and a reuse team in the product line approach.

REFERENCES

1. Bass, L., Clements, P., Cohen, S., Northrop, L., and Withey, J., "Product Line Practice Workshop Report", June 1997, http://www.sei.cmu.edu/about/website/indexes/siteIndex/siteIndexTRnum.html.
2. Cohen, S., Fridman, S., Martin, L., Poyer, T., Solderitsch, N., and Webster, R., "Concept of Operations for the ESC Product Line Approach", Sept. 1996.
3. Brown, A., and Wallnau, K., "Engineering of Component-Based Systems", Proceedings of the 2nd IEEE International Conference on Engineering of Complex Systems, 1996, IEEE Computer Society Press 1996.
4. Brownswod, L., and Clements, P., "A Case Study in Successful Product Line Development", Oct. 1996, http://www.sei.cmu.edu/about/website/indexes/siteIndex/siteIndexTRnum.html
5. Clements, P., "Report of the Reuse and Product Lines Working Group of WISR8", Aug. 1997, http://www.sei.cmu.edu/about/website/indexes/siteIndex/siteIndexTRnum.html
6. Fraks, W., "Success Factors of Systematic Reuse", IEEE software, Sept. 1994.
7. N. Nada, Software Reuse-Oriented Functional Framework, Ph.D. Dissertation, George Mason University, fall 1997.
8. Perry, D., "generic Architecture Descriptions for Product Lines", http://www.bell-labs.com/usr/dep
9. D. Rine and R. Sonnemann, "Investments in Reusable Software: A Study of Software Reuse Investment Success Factors", The Journal of Systems and Software, Vol. 41, pp. 17-32, 1998.
10. D. Rine and N. Nada URL-http://www.gmu.edu/depts/survey.
11. Shaw, M., and Garlan, D., "Software Architecture", Prentice-Hall, Inc., 1996.
12. Software Technology for Adaptable, Reliable Systems (STARS), "STARS Conceptual Framework for Reuse Process (CFRP)", CDRL A018, Oct. 1993
13. Sommerville, I., Software Engineering, 5th Edition, Addison-Wesley, New York, (1996).
14. The Software Evolution and Reuse Consortium, "Solutions for Software Evolution and Reuse", SER Deliverable SER-D2-A, 1995.
15. Withey, J., "Investment Analysis of Software Assets for Product Lines", Nov. 1996, http://www.sei.cmu.edu/about/website/indexes/siteIndex/siteIndexTRnum.html
16. M. Zelkowitz, "Experimental Models for Validating Technology", IEEE Computer, May 1998.
17. Kotonya and Summerville, Requirements Engineering, Wiley, 1992
18. G. Bootch, J. Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide", Addison Wesley, 1999.

An XML-based Approach to Product Line Engineering

Fred Waskiewicz
MCC
3500 West Balcones Center Drive
Austin, TX 78759 USA
+1 512 338 3604
wask@mcc.com

Douglas Stuart
MCC
3500 West Balcones Center Drive
Austin, TX 78759 USA
+1 512 338 3478
stuart@mcc.com

ABSTRACT

The engineering effort necessary to develop the software architecture for a product line (a family of software products) requires several inter-related steps as the architecture evolves. A method, either formal or informal, and, ideally, a tool should guide each step. In order for information regarding the architecture to be passed from one step to the next (either between tools or between human and tool), a mechanism is needed to capture all relevant information derived in each step. This paper describes how product line markup languages, derived from the *eXtensible Markup Language (XML),* were used to capture product line development information - specifically, information regarding requirements and rationale capture, scenario definition, architecture design and links between the artifacts of product line development. The paper also introduces a prototype research tool suite employing these languages that supports the automation of the product line engineering effort.

Keywords

Architecture description, artifact linking, product line engineering, rationale capture, requirements, scenarios, tool support, XML-based languages.

1 INTRODUCTION

As a means of context setting, this section offers the briefest of introductions to the notion of a product line and advances the rationale for taking an architecture-based, software engineering approach to product line development.

Product Lines

The Software Engineering Institute (SEI) defines a *product line* as "a group of products sharing a common, managed set of features that satisfy the needs of a selected market or mission area" [12]. The business case for adopting a product line approach to software development is that, by developing a group of products into a planned, shared domain as a family, an organization can reduce software development time through increased productivity and reuse of business information and technical solutions. Examples of reused business information include the business case for the application and/or system to be built, use cases / scenarios, and requirements derived from them. Technical solutions that are candidates for reuse include development processes and decisions, designs and their rationale, test suites and an implementation repository (an *asset base*) of software artifacts (e.g., components).

A Software Engineering Approach to Product Line Development

If reuse is to be gained, techniques are required to identify *commonality* and to manage *variability*. Commonality may be thought of as requirements applicable to all architectural elements of a product line's reference architecture[1], while variability may be thought of as those requirements peculiar to an instance of the product line. Both concepts infer that an organization will exercise proven software engineering practices in order that commonality of business requirements and technical solutions can be adequately identified. The concepts also infer that variability can be consistently achieved without violating the underlying reference architecture.

Applying software engineering techniques to product line development requires a method - product line engineering (section 2) – and requires a means of representing relevant product line development information (section 3), a role effectively played by XML (section 4). These techniques are greatly enhanced through tool support (section 5).

Architecture-based Approach to Product Line Development

An architecture-based approach assumes that architecture is central to successful product line development. It is the architecture that:

- captures the commonality among the products within the product line
- reflects the degree of variability spanned by the

[1] an adaptable structure that is specifically intended to be instantiated in multiple target systems [5]

product line

- is the starting point for achieving system qualities and for allocating functionality within a system and
- serves as the basis for generating applications within the domain by providing a template that can be instantiated with product line assets.

Thus, not only is a disciplined approach to product line development required, but also an approach which focuses on architecture.

2 PRODUCT LINE ENGINEERING

Having advanced the case for an architecture-based, software engineering approach to product line development, this section introduces the steps involved. This discussion is based upon an applied research effort at Microelectronics and Computer Technology Corporation (MCC) [6] (an Austin, Texas based research consortium) that has, in turn, leveraged work in academia and industrial research bodies.

The Goals of Product Line Engineering

MCC has adopted the SEI's view that software architecture forms the backbone for building successful software-intensive systems and that a system's quality attributes are largely permitted or precluded by its architecture [13]. Given this emphasis on architecture, product line engineering is a disciplined effort that should yield the definition and representation of product-line software architectures. The facets of this discipline should include the evaluation of architectures for product line suitability; architecture extraction; and evaluation of architecture conformance to requirements.

MCC's Approach

MCC's approach to the product line development process is based upon the premise that there are similarities, yet key distinctions, between product line development and single application development. While both efforts must be guided by techniques that manage the progression of software development through its entire lifecycle, a significant difference lies in the separation of the development process into two levels. The first, *domain engineering*, focuses on establishing an *asset base* (an implementation reuse repository) with which the products in the product line can be created, while the second, *application engineering*, focuses on creating an individual application using the assets of the product line.

Domain engineering requirements activities are aimed at identifying the boundaries of a group of products that can be developed more effectively as a product line, not at identifying the requirements of a single product or series of independent products. Specification entails domain modeling - defining the characteristics of the group of products so that the resulting domain model can be used to define the assets needed to build products that span the domain. Design becomes product line architecting - creating a reference architecture that will serve as the basis for all of the products in the product line, and that will identify the software assets that will make up products conforming to that reference architecture. Implementation becomes asset base creation, configuration and composition. The assets needed to create applications in the product line are acquired and used to populate the product line asset base, and a mechanism for composing applications using the asset base is established. Finally, testing entails activity not only at the application level but also at the product line reference architecture level. Results of application testing are used to evaluate the architecture artifacts defined at the domain level.

Application engineering focuses on composing an application from product line assets. Each *application instance* is created by using and reusing artifacts and processes developed during domain engineering. The requirements for an application instance are expressed in the vocabulary of the product line domain model and product line requirements.

3 EXPRESSING PRODUCT LINE DEVELOPMENT INFORMATION

The importance of conveying development information among the progression of product line development steps cannot be over-emphasized, especially if automation of that effort is to be undertaken. Given that the product line reference architecture is at the core of this effort, it is this need to convey information that demands a means of expressing and representing that architecture and the rationale and requirements that went into its design. That need also demands the ability to navigate between the artifacts (the products of each product line engineering step) of the architecture. That is, the necessity to link to and exchange architectural information, either between automated processes and/or humans, as product line development progresses.

In its investigations of product line development, MCC identified the following specific needs.

- The capability of expressing requirements in the form of scenarios.
- The ability to capture design rationale.
- The ability to adequately describe software architectures in terms of architectural elements.
- The ability to link among architecture artifacts and documents.

Languages were required to express this information, and, ideally, a standardized meta-language was needed to develop these languages. Fortunately, such a meta-language exists: *XML*, the *eXtensible Markup Language* [3], developed by the World Wide Web Consortium [15].

4 APPLYING XML

With a firm foundation for an architecture-based, software engineering approach well in place, this section, the heart of this paper, describes how XML can be applied to a product line engineering effort. It describes product line markup languages, derivatives of XML, that were defined to support specific steps identified by MCC.

XML Background
XML was developed with the ambitious goal of providing a means of specifying content, not simply specifying presentation of content. XML describes a class of objects called XML documents, which are domain-specific collections of a series of entities. An XML document may consist of one or many "storage units" called *entities*. Entities have content (parsed or unparsed data) and most are identified by name. An *entity* can contain one or more logical *elements*, which, in turn, can have certain *attributes* (properties) that describe the way in which it is to be processed. The significance of XML is that it provides a formal syntax, but not the semantics, for describing the relationships between the entities, elements and attributes that make up an XML document. Thus, it provides a ubiquitous, "standard" meta-language for describing domain-specific information (such as describing computer software architectures).

MCC's assessment of XML's meta-language features [9] identified several advantages that made it an attractive foundation for developing the languages required to support MCC's approach to product line engineering. Among XML's strong points:

- The very nature of the XML's syntactical flexibility introduces a capable mechanism for architectural expressiveness and extensibility. That is, adhering to the syntax of XML's elements, the semantics can represent and be extended to meet the needs of most, if not all, domains.
- The *Document Type Declaration* (DTD) associated with an XML document introduces a mechanism for validation of an architectural specification. It also provides the means for extracting multiple "views" from a single document (e.g., descriptions of instances from a reference architecture) [4].
- XML provides for links with multiple locators (targets) into documents that may be represented external to the linked documents (e.g., establishing a relationship between a requirement and the architectural elements implementing that requirement.) XML's linking facilities are well suited for storing the elements of an architectural model into a repository; associating them with system requirements; and composing them into an implementation aimed at a solution at a later date. This requirements-to-implementation linking feature greatly assists in the construction of test suites.
- Finally, the emergence of commercially available tools supporting XML (e.g., parsers provided with the Java/XML package providing a uniform model interface) and the capability of web browsers to inspect XML documents.

Armed with this positive assessment of XML, MCC proceeded to develop XML-based languages that were capable of representing the artifacts captured in each phase of product line engineering. Those four languages are presented in the order in which they appear within product line engineering.

Domain Model Example
Due to restrictions on submission size, only the simplest domain model, akin to "Hello World" examples found elsewhere, will be used to illustrate the XML derivatives presented in this paper.

This example domain model presents a system providing field support for queries on customer and product information. The system is broken down into three subsystems providing customer support, security and communication. There are four major components, architectural elements encapsulating high-level functionality, defined within the customer support system.

- `RemoteClient`, the component residing on remote systems supporting access to the central server. It also marshals and unmarshals queries to and from that server.
- `ClientServer`, located at the central site and charged with directing remote queries to the appropriate processing component.
- `Customer_DB`, which processes queries on the database containing customer-related information.
- `Product_DB`, which processes queries on the product-related database.

It is the expectation of the authors that the generic model-related terms and examples used in the following language descriptions will be self-explanatory. Where necessary, elaboration is provided.

RCML – the Rationale Capture Markup Language
MCC's *Rationale Capture Markup Language (RCML)* describes product line engineering information related to identifying requirements and negotiating and resolving pertinent issues and their options. It also captures design rationale. RCML is influenced by work in academia on WinWin [1], a tool that assists in the capture, negotiation and coordination of requirements for large systems of any nature (software or otherwise). The requirements captured by RCML reflect the commonality and variability of the product line reference architecture. A Document Type Definition (DTD) has been developed that describes the RCML grammar.

Figure 1 offers a partial sample of RCML. The major elements within the XML document are `Project`, `ProjectUser`, `Artifacts` and `Taxonomy`. The `Project` being defined is represented as an XML document. A `ProjectUser` is a stakeholder, an individual (or organization) with significant involvement in the project. Attributes provide pertinent information regarding the user. WinWin defines five artifact types: *WinConditions*, *Issues*, *Options, Agreements* and *Taxonomy*. Each are specified in RCML by an `Artifact` element, enhanced by relevant information captured as attributes.

A `WinCondition` captures the stakeholders' goals and concerns with the `Project`. If a `WinCondition` is non-controversial, it is covered by a simple `Agreement` indicating that the condition was not contentious. Otherwise, an `Issue` is created to record the conflict. `Options` suggest alternative solutions, which address the `Issues`. If an `Option` resolves an `Issue`, that adoption is recorded by an `Agreement`. A `Taxonomy` represents the organization of domain information. For example, it may represent the documentation of all project-related information.

WinWin also provides the ability to define internal links between all of its `Artifacts`. Figure 1 illustrates a small subset of a RCML document.

```
<?xml version="1.0" ?>
<!--  DOCTYPE Project SYSTEM file:/C:/RCML.dtd -
->
<Project Name="Field_Support_System">
  <Users>
    <ProjectUser Name="Developer*"
      Directory="C:\Project\Developers"
      Role="Developer" Title="" Position=""
      Organization="My_Company" Owner="true" />
    <ProjectUser Name="Domain_Expert"
      Directory="C:\Project\Domain_Expert"
      Role="Domain_Expert" Title="" Position=""
      Organization="Project_Company" />
  </Users>
  <Artifacts>
    <Term identifier="Developer-TERM-1"
      owner="Developer" role="Developer"
      creationDate="01/05/00"
      creationTime="08:32"
      revisionDate="01/10/00"
      revisionTime="08:32"
      name="Composition"
      priority="MEDIUM"
      status="ACTIVE">
        <ArtifactBody>Identifying and assembling
        together the set of reusable components
        needed to support field access to client
        and product information.
        </ArtifactBody>
    </Term>
      <WinCondition  identifier="Developer-WINC-
      1" owner="Developer" role="Developer"
      creationDate="01/05/00"
      creationTime="15:31"
      revisionDate="01/10/00"
      revisionTime="12:52"
      name="Application_Diverisity"
      priority="MEDIUM"
      status="ACTIVE" state="UNCOVERED">
        <ArtifactBody>The application code will
        be in a portable language and the
        program will run on multiple platforms.
        </ArtifactBody>
      </WinCondition>
  </Artifacts>
  <Taxonomy>
    <TaxonomyItem Index="1" Title="Field Support
    System" />
    <TaxonomyItem     Index="2"     Title="Domain
    Requirements" />
    <TaxonomyItem    Index="3"    Title="Quality
    Requirements" />
    <TaxonomyItem                    Index="3.1"
    Title="Reusability" />
    <TaxonomyItem                    Index="3.2"
    Title="Composability" />
    <TaxonomyItem   Index="4"   Title="Functional
    Requirements" />
    <TaxonomyItem Index="5" Title="Deliverables"
    />
    <TaxonomyItem      Index="6"      Title="Risk
    Management" />
  </Taxonomy>
</Project>
```

Figure 1. A Partial RCML Specification

As seen, RCML captures project and product line development process requirements-related information through extensive use of XML elements and attributes. The reader is encouraged to study WinWin to fully appreciate the nature and level of the information captured.

SML – the Scenario Markup Language

The *Scenario Markup Language (SML)* developed by MCC describes product line engineering scenario information; specifically, a collection of usage scenarios that capture operational requirements (i.e., how a system will be used). More precisely, SML defines application-specific requirements, which reflect the variability of the product line reference architecture. SML is influenced by research conducted in academia on scenario specification [8] and UML use case specifications [10] and represents those views of the constituent parts of a scenario: *Agents, Objects* and *Actions*. A DTD has been developed that describes the SML grammar.

Figure 2 illustrates the principal elements of SML. The XML document, the `ScenariosDocument`, represents a collection of scenarios. XML elements represent each scenario as well as their constituent parts. An `Agent` is an actor within the system that performs an `Action` – some system-specific behavior. XML attributes describe the `Agent's name` and `type`. `Actions` are specified by attributes describing the `Action's` behavior, the `Agent` that serves as an actor in the action, links to that `Action`, and the effect of the action on object states. Actions may be aggregated into `ActionLists`. An `Object` is an abstraction of some real-world entity within a system. XML attributes describe the `Object's name` and `state`. Last, but certainly not least, `Scenarios` are specified in terms of their `Goals`, `Descriptions`, the sequences of `Actions` which comprise the `Scenario`, links and references to information regarding each specific `Action` and states affected by the `Action`.

```
<?xml version="1.0" ?>
```

```
<!DOCTYPE ScenariosDocument (View Source for
full doctype...)>
<ScenariosDocument>
  <Objects>
    <Object                    name="RemoteClient"
    state="logged_on,logged_off,accessing" />
    <Object                    name="ClientServer"
    state="active,inactive" />
  </Objects>
  <Agents>
    <Agent                     name="RemoteClient"
    type="clientType"/>
    <Agent                     name="ClientServer"
    type="serverType"/>
  </Agents>
  <Actions>
    <Action name="Remote_log_on" state="
     logged_on">
      <Description>Remote client log on.
      </Description>
      <AgentRef  href="#descendant(1,Agent,name,
      RemoteClient) xml-link="locator" />
      <InitialState>
        <ObjectState
        href="#descendant(1,Object,name,
        RemoteClient)"
        state="logged_off" xml-link="locator" />
      </InitialState>
      <FinalState>
        <ObjectState
        href="#descendant(1,Object,name,
        RemoteClient)"
        state="logged_on" xml-link="locator" />
      </FinalState>
    </Action>
    <Action                    name="Remote_query"
    state="accessing">
    <Description>Remotely  send  customer  or
    product query to server.</Description>
    </Action>
  <Scenarios>
    <Scenario name="Info_Query">
      <Goal>Remote  access  of  customer  and/or
      product information.
      </Goal>
      <Description>This scenario describes how a
      customer  representative  in  the  field
      remotely  accesses  customer  and  product
      information.
      </Description>
      <InitialState />
      <FinalState />
      <ActionSequence>
        <ActionRef
          href="#descendant(1,Action,name,Remote
          _log_on)"
          xml-link="locator"
        </ActionRef>
        <ActionRef
          href="#descendant(1,Action,name,Remote
          _query)"
          xml-link="locator"
        </ActionRef>
      </ActionSequence>
    </Scenario>
  </Scenarios>
</ScenariosDocument>
```

Figure 2. A Partial SML Specification

The reader should view this example as a validation of how XML can be extended to meet the requirements of existing scenario specification techniques.

The details of scenario specification found in this example are omitted due to space constraints imposed on this paper. However, note should be made of the usage of the `xml-link` attribute.

```
<ActionRef
  href="#descendant(1,Action,name,Remote_query)"
  xml-link="locator"
</ActionRef>
```

This is linking information interspersed with scenario specification. This attribute will be described in the next section.

LDML – the Link Definition Markup Language

The *Link Definition Markup Language (LDML)* describes the necessary links found between XML documents that capture product line analysis, design and implementation artifacts. It is based upon work being conducted at MCC on linking architectural elements [14]. MCC's work is, in turn, influenced by work at the World Wide Web Consortium [15] on designing hypertext links for XML [16] and XPointer [17], a language that provides for addressing into the internal structures of XML documents. In particular, XPointer provides for specific reference to elements, character strings, selections, and other parts of XML documents. A DTD has been developed that describes the LDML grammar.

The major elements of this XML document type, identified as a `LinkedDocuments` type, are `DocumentGroup`, `Document` and `Links`. This ability meets a key requirement of product line development – the need to navigate among the essential artifacts comprising a product line reference architecture.

Figure 3 offers a partial sample of LDML. `Documents` may be linked together and categorized within a `DocumentGroup`, which is described by attributes that reference its identifying name and qualifiers defining the type of link. `Documents` in turn have their own unique identifiers, references and qualifiers defining the type of link. `Links` provide link information that, for example, ties a design artifact back to its requirement(s) as captured in a business scenario. A `Link` may be grouped into a set of `Links`.

```
<?xml version="1.0" ?>
<!DOCTYPE LinkedDocuments (View Source for full
doctype)>
<LinkedDocuments>
  <DocumentGroup  name="default_documents"  xml-
  link="group">
    <Document             href="Field_Access.sml"
    name="Field_Access_Scenario"
    xml:link="document" />
    <Document    href="Field_Support_System.adml"
    name="Field_Support_System"
    xml:link="document" />
  </DocumentGroup>
  <Links name="default_links" title="Links">
      <Link             title="RemoteClient_Reqs"
      name="RemoteClient_Requirements"
      direction="both">
       <From xml-link="extended" inline="false">
          <Locator
          href="Field_Access_Scenario.sml#name(R
          emoteClient)" xml:link="locator" />
        </From>
        <To xml-link="extended" inline="false">
          <Locator                          href="
          Field_Support_System.adml#id(203981035
          274767)" xml:link="locator" />
        </To>
      </Link>
  </Links>
```

```
</LinkedDocuments>
```

Figure 3. A Partial LDML Specification

What is most important about this abbreviated example is the use of XML mechanisms, linking and the XPointer language, to tie together:

- the scenario requirements document (*Field_Access.sml*) with the architectural description XML document (*Field_Support_System.adml*) and
- specific requirements within the scenario document with the implementation of those requirements as found in the architectural description document.

In Figure 3, the `xml:link` attribute value "document" for the `Document` elements signifies that documents *Field_Access.sml* and *Field_Support_System.adml* are being linked. The `xml:link` attribute value "locator" associated with the `From` and `To` elements signifies that resources that are not XML documents are being linked.

ADML – the Architecture Description Markup Language

Research has yielded various *architecture description languages* (ADLs) [11] over the years to address the need for a descriptive language for software architectures. *ACME* [2], developed at Carnegie-Mellon University as an interchange language between ADLs, represents a distillation of the information provided by its predecessors, making it an attractive basis for an ADL for product line development. While ACME provides the semantics for adequately describing software architecture, it lacks status as a *de jure* or *de facto* standard, limiting its use as a representation language. Using ACME as a basis for architectural description and XML as a basis for the presentation of that information, MCC developed its *Architecture Description Markup Language (ADML)*. A DTD has been developed that describes the ADML grammar.

Using graphical artifacts derived from the ACME, XML elements define a top-level `design` (the XML document), `systems`, `components`, `representations` (refinements of architectural elements) and `connectors` (architectural design elements showing the interaction between components). The conventional concept of relationship has no relevance in this type of high-level design model; rather, the notion of attachment describes how `components` and `connectors` are related. One appealing feature of ACME is the provision for the definition of properties, name-value pairs that are not limited by ACME semantics (have no predefined meaning) and provide the primary mechanism for extensibility. That ACME feature has been cast to ADML `properties` that describe relevant architectural information such as dependencies among design components, performance attributes for the implementations of design components, requirements traceability, design rationale and so forth. A single property or a set or sequence of properties may be used to describe an architectural element. The `PropertyLiteralValue` element is used to specify the value of the name-value attribute pair associated with a property.

As architecturally-complete entities, systems can represent architectures of complete applications within a product line, or subsystems providing specific capability (e.g., a security subsystem or communications subsystem within a customer service system.) Components represent a system's computational and data elements. Connectors may exhibit role-like behavior that describes how a component participates in an interaction with another component.

The following figure offers a partial sample of ADML in which XML elements are employed to offer design information for the `Field_Support_System` example. Note that the design breaks the larger system down into sub-systems: the `Customer_Support` subsystem that implements the paper's example domain model, and a `Security_System` and `Communication_System` that provide infrastructure support.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE Design (View Source for full
doctype...)>
<Design identifier="Field_Support_System">
 <System name="Customer_Support_System">
  <SystemDescription>
   <SystemStructure>
    <Component name="ClientServer">
     <ComponentDescription />
    </Component>
    <Component name="Customer_DB">
     <ComponentDescription>
      <ComponentBody>
       <Property name="Asset Base Properties">
        <PropertyValue>
         <PropertySetValue
                name="Implementation">
          <PropertyValue>
           <PropertySequenceValue
                 name="Implementation object">
            <PropertySequenceElement
                position="1">
             <PropertyValue>
              <PropertyLiteralValue
        value="%ASSET_BASE%\source\ATM.java" />
             </PropertyValue>
            </PropertySequenceElement>
            <PropertySequenceElement
                position="2">
             <PropertyValue>
              <PropertyLiteralValue
                value="%TARGET%" />
             </PropertyValue>
            </PropertySequenceElement>
           </PropertySequenceValue>
          </PropertyValue>
         </PropertySetValue>
        </PropertyValue>
       </Property>
      </ComponentBody>
     </ComponentDescription>
    </Component>
    <Component name="Product_DB">
      <ComponentDescription />
    </Component>
    <Component name="RemoteClient">
     <ComponentDescription>
      <ComponentBody>
```

```
        <Property name="Dependency">
         <PropertyValue>
          <PropertySequenceValue>
           <PropertySequenceElement position="1">
            <PropertyValue>
             <PropertyLiteralValue
                value="ClientServer" />
            </PropertyValue>
           </PropertySequenceElement>
           <PropertySequenceElement position="2">
            <PropertyValue>
             <PropertyLiteralValue
                 value="Customer_DB" />
            </PropertyValue>
           </PropertySequenceElement>
           <PropertySequenceElement position="3">
            <PropertyValue>
             <PropertyLiteralValue
                value="Product_DB" />
            </PropertyValue>
           </PropertySequenceElement>
          </PropertySequenceValue>
         </PropertyValue>
        </Property>
        <Port name="Server_API">
         <PortDescription />
        </Port>
       </ComponentBody>
      </ComponentDescription>
     </Component>
     <Connector>
      <ConnectorDescription>
       <ConnectorBody>
        <Role name="Server_client">
         <RoleDescription />
        </Role>
       </ConnectorBody>
      </ConnectorDescription>
     </Connector>
     <Attachments>
      <Attachment PortID="PT01" RoleID="R01" />
     </Attachments>
    </SystemStructure>
   </SystemDescription>
  </System>
  <System name="Security_System">
   <SystemDescription />
  </System>
  <System name="Communication_System">
   <SystemDescription />
  </System>
</Design>
```

Figure 4. A Partial ADML Specification

A typical initial reaction by a reader is that the language is overly verbose, yet this apparent excess provides one of the major features of ADML: *flexibility* (*extensibility*), inherited from both XML and ACME. Guided by a syntax that, while prescriptive, is not excessively constraining, a designer can craft a complex ADML document that is both human and machine readable. For example, information related to a component's association with its asset base is specified as a complex property. Implementation information is specifically composed of a set of properties. Within the set, sequences define implementation information such as the location of source files and the location of the build environment. The set could easily contain other, more primitive types of information, such as strings or integers denoting performance criteria. A component's dependency on other component(s) is defined as a sequence whose elements identify the component(s) upon which the dependency is placed. In both cases, the `PropertyValue` element is used in ADML as a signal to the processor (either human or machine) that what follows may or may not conform to any predefined schema.

Using this specific example to clarify the above discussion, `PropertySequenceElement` attributes are used to define properties for component `Customer_DB` that specify the location of a *Java* [tm] file containing the source code implementing this component and the target directory in which the instance build image resides. The `RemoteClient` component requires (is dependent upon) the other three components in order to function as desired. The `PropertyLiteralValue` of elements of a `PropertySequenceValue` are used to describe this property.

Finally, connectors are specified through use of a `Connector` element which defines the role to be played by the component employing the connector to interact with another component and by an `Attachment` which specifies the association between that role and the port into the component being employed. Due to space constraints, representations were not incorporated in to this example.

5 TOOL SUPPORT

The previous sections advanced the case for an architecture-based software engineering approach to product line development; offered a specific technique – MCC's product line engineering approach; and showed how XML can play a key role in that effort. This section provides a cursory description of a prototype suite of tools developed at MCC which was constructed to assist in product line development. More germane to this paper, it also discusses how those tools employ the XML-based product line markup languages developed by MCC. The intent is *not* to provide in-depth descriptions of each tool but rather to provide a summary of each tool's purpose, to show how its functionality applies to product line engineering, and to show how the XML-derived languages are employed. The six tools to be discussed are WinWin-NT, the ScenarioManager, the LinkManager, VisualADML, the DependencyChecker and the CompositionTool.

Figure 5, depicting interaction among the tools in the suite, conveys the flow of architectural information as captured in MCC's product line markup languages.

WinWin-NT

Originally developed at the University of Southern California, WinWin [1] is a tool that captures product line engineering information related to identifying requirements and negotiating and resolving pertinent issues and their options. It also captures design rationale. MCC's version, WinWin-NT, is a modification that exports that information in RCML, using a Document Type Definition (DTD) to validate its output. Although

RCML employs the full complement of WinWin artifact types, three are of special interest: `conditions`, `issues` and `options`. Conditions represent requirements, the commonality of a reference architecture, while issues and their options represent the variability that may yield instances of a product line.

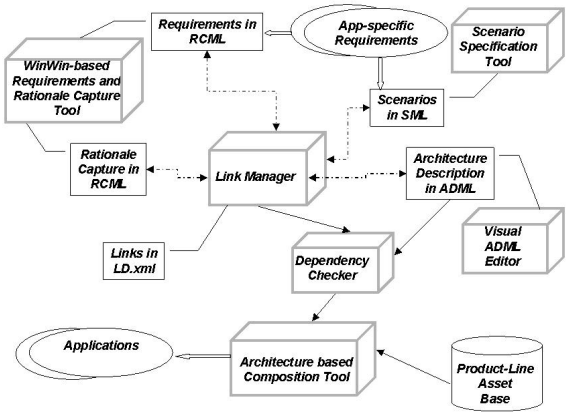


Figure 5. Prototype Tool Suite for Product Line Development

ScenarioManager
ScenarioManager is a graphical tool for creating, viewing, and editing an SML document, a scenarios document consisting of a list of scenarios and their constituent parts: *Agents*, *Objects* and *Actions*. The tool uses a DTD to validate its output. The value of the tool lies in its assistance in defining application-specific requirements, which represents the variability that may yield instances of a product line.

VisualADML
VisualADML is a graphical editor that supports the visual modeling of a product line reference architecture and creation of a textual ADML description of that architecture. A designer creates a reference architecture by identifying its high–level design and specifying the systems and subsystems within that design. Those systems are further decomposed into components whose interactions are defined by connectors and attachments, which describe the relationship between components and connectors. Editing assistance in the form of menus are provided that aid in the specification of component properties. The output of this tool is an ADML document that textually describes the design. A forthcoming version of the tool will use a DTD to validate this output.

LinkManager
LinkManager is an editing tool that permits designers to manually create and navigate between the necessary links found within XML documents that capture product line analysis, design and implementation artifacts. For example, experiments at MCC defined product line requirements in RCML and components implementing the solutions to those requirements in ADML. As both RCML and ADML are XML derivatives, it was possible to establish multi-way links. These links allowed validation of multiple instance designs through linking the artifacts (e.g., requirements) established within application design back to the original product line requirements. Automatic link creation would be ideal, but investigation at MCC indicates that current knowledge is not sufficient to meet that need. The product of this tool is a LDML document, which is validated by its DTD.

Another element of this tool, the Architecture Components Selector (ACS), provides a set of instance requirement artifacts by generating a list of product line architecture components. It accomplishes this by navigating the links between the reference architecture options captured in RCML and the issues defined for the instance.

DependencyChecker
The DependencyChecker is an automated tool designed to create a product line instance from its reference architecture. This is accomplished by using the output from the LinkManager's ACS – essentially, a list of components - as the basis for the instance architecture. Given this list of components, the DependencyChecker examines the ADML document created by VisualADML, the textual description of the reference architecture, for dependency-related properties for each of these components. The intent of this operation is to ensure that a complete instance architecture has been specified. The output from this tool is a modified ADML document reflecting the instance architecture.

An interesting feature of the tool is support of both direct and indirect dependency. A direct dependency infers that an architectural element explicitly requires the implementation of another. Indirect dependency, while still inferring an implementation of both, implies that they may be required by some other architectural element(s) while not necessarily interoperating with one another.

Figure 6a provides an example of an ADML specification of a direct dependency. A `RemoteClient` requires an implementation of the `ClientServer` component in order to function properly.

```
<ComponentDeclaration name="RemoteClient">
    <SequenceDeclaration name="Dependency">
        <LiteralDeclaration          type="String"
        value="ClientServer"/>
    </SequenceDeclaration>
</ComponentDeclaration>
```

Figure 6a. Specifying a Direct Dependency

Figure 6b provides an example of an ADML specification of an indirect dependency. A new component, `Product_Forecast`, is deemed necessary to the system design by the mutual dependence upon it by two *non-interacting* components,

Customer_DB and Product_DB.

```
<SystemDeclaration name="Customer_Support">
  <SetDeclaration name="Indirect Dependencies">
    <SequenceDeclaration
    name="Product_Forecast_users">
        <LiteralDeclaration          type="String"
        value="Customer_DB" />
        <LiteralDeclaration          type="String"
        value="Product_DB" />
    </SequenceDeclaration>
  </SetDeclaration>
</SystemDeclaration>
```

Figure 6b. Specifying an Indirect Dependency

CompositionTool

The CompositionTool composes the build environment for the implementation of the instance architecture. Using information contained with the modified ADML document produced by the DependencyChecker (the instance architecture), the CompositionTool creates a script file that moves source files representing component implementations from the *asset base*, its reuse repository, to target directories on the build platform where instance applications are constructed. The CompositionTool also adds command line or operating system directives to the script when specialized operations are required. Examples of these operations include modifications to configuration files or identification and potential generation of "glue code" necessary to integrate software components. The final step in creating the instance architecture executable(s) is to run a site-specific *makefile* on this build environment.

6 CONCLUSION

This paper has provided an assessment of effectiveness of the role that XML can play in product line development. Taking an architecture-based product line engineering approach, it has shown how product line markup languages derived from the XML meta-language can be used to capture artifacts produced during the product line development lifecycle. It has also illustrated how tools assisting in product line engineering tasks can employ these XML-derived languages.

The work described in this paper is part of on-going research at MCC on the application of an architecture-based approach to product line development. Areas of interest for possible future product line development research affecting the methodology, tool suite and the XML-based languages include:

- A more substantial validation effort regarding the artifacts exchanged between tools in the product line tool suite.
- Increased support of the concepts of commonality and variability with regard to creating instances of a product line reference architecture.
- Analysis of architectural "ilities" (e.g., interoperability, extensibility) and implementation characteristics (e.g., security, performance).
- Expansion of composition capabilities to represent "glue code" (composition-related information) within ADML.

It is worth noting that, as of publication date, the Open Group [7] is in the final stages of adopting MCC's ADML as a standard notation for describing their architectures.

7 INFORMATION AND QUESTIONS

For more information or questions, contact the authors or visit MCC's public web site at [6].

ACKNOWLEDGEMENTS

The authors wish to acknowledge the following team members for their significant technical contributions upon which this paper is based: Deborah Cobb, T.W. Cook, Charles Goyette, Steve Pruitt and Wonhee Sull.

DISCLAIMER

REFERENCES

1. Boehm, B. and Egyed, A., "WinWin Requirements Negotiation Processes: A Multi-Project Analysis." In *Proceedings of 5th International Conference on Software Processes*, June, 1998.
2. Garlan, D.; Monroe, R. and Wile, D. "ACME: An Architecture Description Interchange Language." In *Proceedings of CASCON 97*, November, 1997.
3. Garshol, L.M. "Introduction to XML." Available at http://www.stud.ifi.uio.no/~lmariusg/download/xml/xml_eng.html
4. IEEE, Architecture Working Group of the Software Engineering Standards Committee. "Draft Recommended Practice for Architectural Description." IEEE P1471/D4.1.
5. Lao-Tsu. "The Tao of the Software Architect." Available at http://www.sei.cmu.edu/architecture/essays.html
6. MCC, Information Systems Engineering Project Web Site. Available at http://www.mcc.com/projects/ise/
7. The Open Group Web Site. Available at http://www.opengroup.org/
8. Potts, C. "ScenIC Guidebook: Scenario-based Requirements Determination for Evolving Software Systems." Technical Report DRAFT, College of Computing, Georgia Tech, July, 1998.
9. Pruitt, S. et. al. "The Merit of XML as an Architecture Description Language Meta-Language." Position paper for WICSA, San Antonio, TX. February, 1999.

10. Quantrani, T. “Visual Modeling with Rational Rose and UML.” Addison-Wesley, 1998.

11. SEI Architecture Description Languages Web Site. Available at http://www.sei.cmu.edu/architecture/adl.html

12. SEI Product Line Systems Web Site. Available at http://www.sei.cmu.edu/activities/plp/product_line_overview.html

13. SEI Software Architecture and the Architecture Tradeoff Analysis Initiative Web Site. http://www.sei.cmu.edu/ata/ata_init.html

14. Stuart, D.; Sull, W. and Cook, T.W. “Dependency Navigation in Product Lines Using XML.” To appear in *Proceedings of Third International Workshop on Software Architectures for Product Families (IW-SAPF-3).*

15. World Wide Web Consortium (W3C) site. Available at http://www.w3c.org/

16. W3C, XML Linking Working Group Web Site. Available at http://www.oasis-open.org/cover/xml/WG1999.html#linkingWG

17. W3C, XML Pointer Language (XPointer) Working Draft. Available at http://www.w3.org/TR/xptr

Reusable Architectures for Software Product Lines

H. Gomaa
Department of Information and
Software Engineering
George Mason University
Fairfax, VA 22030-4444
hgomaa@isse.gmu.edu

G.A. Farrukh
The MITRE Corporation
1820 Dolley Madison Blvd
McLean, VA 22102-3481
farrukh@mitre.org

1 INTRODUCTION

In this paper, a software product line architecture is defined as an architecture for a family of systems that have some features in common and others that differentiate them. The architecture is described using an Architecture Description Language (ADL) [Shaw96] in terms of components and their interconnections. This paper describes two approaches for composing reusable software product line architectures from feature based domain specific architecture patterns. An *architecture pattern* is defined to be a set of interconnected components specified in an ADL in terms of their interfaces and interconnections. The first approach uses reusable black box architecture patterns, in which the entire pattern is reused without change, although components in a pattern may have configuration parameters passed to them at instantiation time. The second approach uses white box extensible architecture patterns, allowing the architecture to evolve after it has been deployed. The approaches are described and compared.

2 SOFTWARE PRODUCT LINE ARCHITECTURAL DESIGN

A *domain model* is an object-oriented analysis model for a family of systems. The domain modeling method [Gomaa95] allows the explicit modeling of the similarities and variations in a family of systems. A *feature* is an end-user requirement [Kang90] that is supported in the *software product line architecture.* In a family of systems, features may be *kernel,* i.e., required by all members of the family, or *optional,* i.e., required by only some members of the family. Optional features may also be mutually exclusive. A feature may require another feature as a prerequisite. Features are determined during domain modeling. Scenarios are also used to determine the objects required to support each feature [Gomaa95, Gomaa96].

During product line architectural design, a reusable architecture is developed for the product line family. Each object type in the domain model is mapped to a component type and each feature is mapped to a domain specific architecture pattern, showing the composition of interconnected components required to satisfy the feature.

In this paper, components types and architecture patterns are specified using an ADL, Darwin, which is part of the Regis configuration environment for parallel and distributed programming developed at Imperial College, London [Magee94]. The Regis environment uses the Darwin ADL for the external specification of each component type, while the internals of component types are programmed in the object-oriented language C++.

A Darwin component specification describes the external specification of the component type. For every simple kernel, optional, and variant object type in the domain model, an equivalent Darwin component type is developed. Every Darwin component type is defined in terms of the interfaces it provides and requires from other Darwin component types. An example of a Darwin component type is:

```
component Line_Assembly_Workstation_Controller (char* wkst_name)
{
   provide Part_Requested <port Part_Request_Type>;
   provide Part_Coming <port Part_Type>;

   require Workstation_Data <port Part_Type>;
   require Part_Sent <port Part_Type>;
   require Part_Request <port Part_Request_Type>;
   require Operation_Request <entry OpReq_Type, OpRes_Type>;
   require Alarms <port Alarm_Type>;
}
```

Our approach for developing reusable architectures uses domain specific architecture patterns. The problem that an architecture pattern solves is described by a feature. The solution of the architecture pattern is a set of interconnected components, describing the components and their interconnections. The context in which the pattern solution works is described in terms of the feature / feature dependencies, which are the constraints to be applied when combining features and hence architecture patterns to compose the architecture of a target system (one of the members of the family). Thus one feature may be a prerequisite for another or one feature may be mutually exclusive with another.

To relate the domain features to the reusable architecture, a scenario driven approach [Gomaa97] is used. An architecture pattern shows the collaboration among the components required to support a given feature, as well as communication with collaborating components in prerequisite architecture patterns.

For every feature in the domain model, an architecture pattern is specified using the Darwin ADL in the reusable architecture. There is one kernel architecture pattern, which is required by every member of the product line, and consists of non-variant components and their interconnections. For each optional feature, an architecture pattern is

developed, in which the optional and variant components needed to support it are defined as well as the interconnection between these components. In addition, interconnections are defined between the components in the architecture pattern and any kernel components used by these components. Interconnections are also defined to any optional or variant components defined in prerequisite architecture patterns.

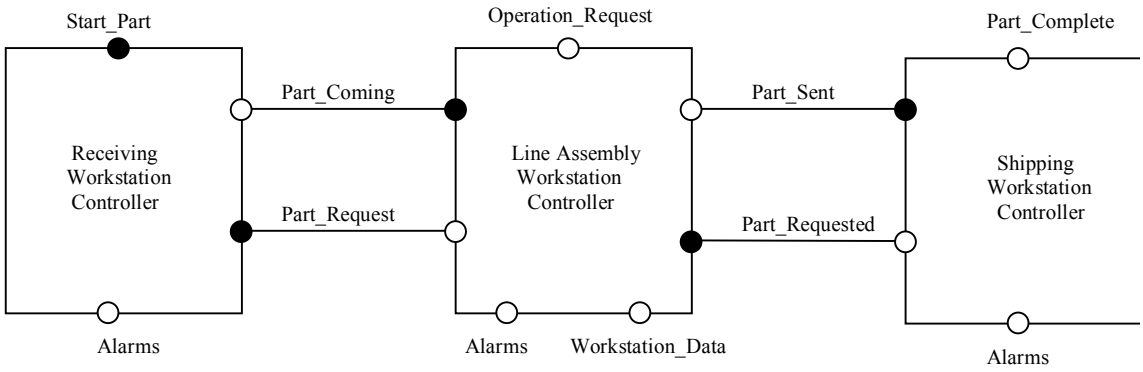


Figure 1: Software Architecture Diagram for High Volume Architectural Pattern

An example of a Darwin ADL description of an architecture pattern is given in Fig. 1 from a factory automation product line. The High Volume architecture pattern has three component types: Receiving Workstation Controller, Line Assembly Workstation Controller (which has a configuration parameter, workstation name) and Shipping Workstation Controller. The instantiation statements are used to create one or more instances of a component type:

inst
Receiving_Wkst: Receiving_Workstation_Controller;
Shipping_Wkst: Shipping_Workstation_Controller;
Line_Wkst: Line_Assembly_Workstation_Controller(wkst_name);

The pattern also defines the interconnections between components contained in the pattern. Message communication between two components involves connecting the *require* interface of the sending component to the *provide* interface of the receiving component using the Darwin bind statement. For example, the *Line Assembly Workstation Controller* sends the *Part Sent* message to the *Shipping Workstation Controller*:

bind Line_Wkst.Part_Sent -- Shipping_Wkst.Part_Sent;

In this simple example, the High Volume Pattern also depends on two other architecture patterns: the kernel pattern, which contains the kernel components, and the Factory Production pattern, which contains two other components. Any dependencies on components in these required patterns must be explicitly defined in the dependent pattern.

3 TARGET SYSTEM ARCHITECTURE COMPOSITION

Target system architectures, which are members of the product line family, are composed from domain specific architecture patterns, where the constraints for interconnecting architecture patterns are given by the feature/feature dependencies. Thus the relationship among product line features, which is defined during domain analysis, is preserved as constraints among architecture patterns. For a High Volume Manufacturing system, the High Volume, Factory Production, and kernel architecture patterns are needed, as shown in Fig. 2. The High Volume target system architecture consists of the composition of these three patterns involving the instantiation and interconnection among the components of the patterns. As both High Volume and Flexible Manufacturing systems, which are both members of the software product line, require the kernel and Factory Production architecture patterns, components in these architecture patterns are reused in different target systems. Furthermore, the High Volume and Flexible Manufacturing architecture patterns are constrained to be mutually exclusive by the feature/feature dependencies.

4 EVOLUTION OF SOFTWARE ARCHITECTURES

This section describes an evolutionary software architecture consisting of extensible white box architecture patterns. The application domain is that of federations of client/server systems [Gomaa99].

A Federation Interface Manager (FIM) is a software subsystem that mediates each member's interface to the federation. The goal is to allow each information system's clients and server to be integrated into the federation with the minimum amount of software modification. There are logically two kinds of FIMs, client FIMs and server FIMs. As new members of the federation can be added after the federation is operational, it is also necessary to decouple clients from servers through the use of object brokerage services. A high level view of the federation is shown in Fig. 3.

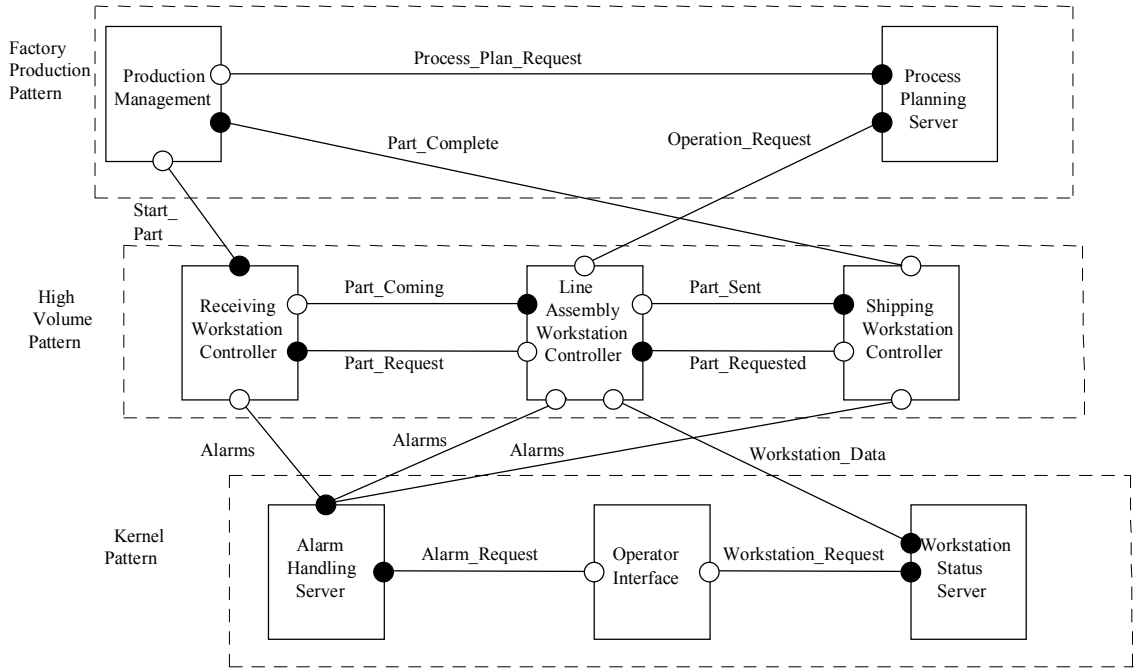


Figure 2: Software Architecture Diagram for the High Volume Target System

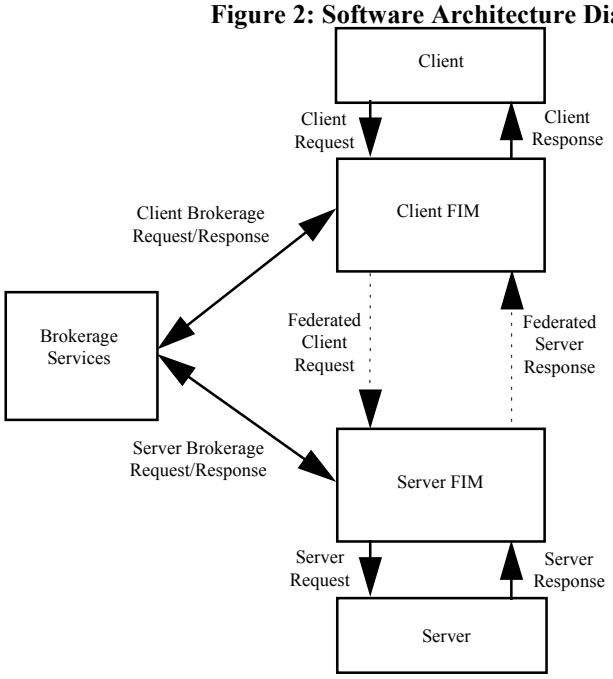


Figure 3: Federation of Client/Server Systems

The goal is to have FIMs that are reusable by each member of the federation. However, each member of the federation is likely to need a different variation of the FIM, since each FIM will need to reflect the characteristics of the individual client or server that it is interfacing to, as well as the characteristics of the federation, which are common. Thus a FIM has some aspects that are common to all members of the federation, and other aspects that are specific to each individual member of the federation. In addition, federation services can be split into those that are domain independent, e.g., brokerage services, and those that are domain specific, e.g., banking services in an electronic commerce domain. The characteristics of this problem make it very amenable to domain modeling.

The FIM architecture operates at three different levels of reuse: federation, domain, and application. At the **Federation Level,** the architecture is most abstract and can be used for any application domain. The overall structure of client FIMs,

server FIMS, and registration server, are defined. The architecture is defined in terms of the structure and interfaces of the components, and is specified in the Darwin ADL. It defines the overall control structure for the federation. The implementation of components providing federation level services is also domain independent.

The object brokerage services and the federation registration message are domain independent because registration services are the same regardless of the application domain. The header part of the federation transaction is also domain independent.

At the **Domain Level**, domain specific functionality is defined. In particular, domain level transactions are defined, e.g., for electronic commerce, which define the type of service provided for the domain, transaction requests containing service requests and parameters, and transaction responses, are defined.

At the **Application Level**, functionality for individual applications (clients and servers) is added. For example, for an ATM client, its transaction types and translation mechanisms are defined. At this level, clients and servers can actually instantiate their respective FIMs and become members of a federation.

The FIM architecture along with the C++ code represents a framework [Johnson97]. In the approach described in this paper, the framework consists of three architecture patterns (defined in the Darwin ADL) and components implemented in C++. The patterns are micro-architectures for the Client FIM, Server FIM, and brokerage services. The brokerage services form a black-box pattern, which is reused without adaptation. There is one instance of this pattern for a given federation. The Client FIM and Server FIM are white-box patterns, which need to be extended before they can be used. There are several variant implementations of these patterns in a given federation.

5. CONCLUSIONS

This paper has described two approaches for composing reusable software product line architectures from feature based domain specific architecture patterns. The first approach uses reusable black box architecture patterns. A feature describes the problem that an architecture pattern solves. The solution given by the architecture pattern is a set of interconnected components, with a description of the components, their interconnections and their pattern of communication. The context in which the pattern solution works is described in terms of the feature / feature dependencies, which are the constraints to be applied when combining architecture patterns to compose the architecture of a target system. When an architecture pattern is reused, its components are usually interconnected to different components each time. With black-box patterns, the entire pattern is reused without change, although components in a pattern may have configuration parameters passed to them at instantiation time. The second approach uses white box extensible architecture patterns, allowing the architecture to evolve after it has been deployed. Using extensible architecture patterns gives much greater flexibility, although considerably more knowledge of the architecture, and how to adapt it is required.

To support our research into software product lines, we have developed several tools [Gomaa96, Gomaa97]. Currently, we are developing an object-oriented UML based analysis and design method for software product line architectures, which is based on use cases [Gomaa98] from which architecture patterns are developed.

6. ACKNOWLEDGEMENTS

This research was supported in part by the NASA Goddard Space Flight Center, the Virginia Center of Innovative Technology, and DARPA. The authors gratefully acknowledge several valuable discussions with J. Kramer and J. Magee. The views and conclusions expressed are those of authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of NASA, DARPA, or The MITRE Corporation.

7. REFERENCES

[Gamma95] Gamma E., Helm, R., Johnson R., and Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[Gomaa95] Gomaa H., "Reusable Software Requirements and Architectures for Families of Systems", *Journal of Systems and Software*, April 1995.

[Gomaa96] Gomaa H., et. al. "A Knowledge-Based Software Engineering Environment for Reusable Software Requirements and Architectures," *J. Automated Software Engineering*, Vol. 3, Nos. 3/4, August 1996.

[Gomaa97] Gomaa H. and Farrukh, G.A., "Automated Configuration of Distributed Applications from Reusable Software Architectures", Proceedings IEEE International Conference on Automated Software Engineering, Lake Tahoe, November 1997.

[Gomaa98] H. Gomaa, "Use Cases for Distributed Real-Time Software Architectures", *Journal of Parallel and Distributed Computing Practices*, Vol. 1, No. 2, 1998.

[Gomaa99] H. Gomaa and G. A. Farrukh, "A Reusable Architecture for Federated Client/Server Systems", Proc. ACM Symposium on Software Reusability, Los Angeles, May 1999.

[Johnson97] Johnson, R.E., "Frameworks = (Components+Patterns)", *CACM*, Vol. 40, No. 10, Oct. 1997, pp. 39-42.

[Kang90] K. C. Kang et. al., "Feature-Oriented Domain Analysis," Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[Magee94] Magee, J., Dulay, N., and Kramer, J., "Regis: A Constructive Development Environment for Parallel and Distributed Programs", *J. Distributed Systems Engineering*, 1994, pp. 304-312.

[Shaw96] Shaw, M. & Garlan, D, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

ICSE/IWPL1 submission paper

A *bumon* Methodology for Product Line Conceptual Modeling

Masao J.Matsumoto, Masahiko Kamata and Kaoru Umezawa
University of Tsukuba,Tokyo
Graduate School of Systems Management
3-29-1 Otsuka, Bunkyo, Tokyo 1120012 Japan
mjm@gssm.otsuka.tsukuba.ac.jp

Abstract

This paper presents a workable methodology for product line modeling. Though some advantages of product line as compared to single shot have been recognized, actually workable methodology has not been presented well covering and supporting those essential points as core conceptual, structural and operational aspects that are incorporated into the architecture retaining throughout the lifecycle. Conceptual modeling is the first crucial step for structuring the architecture, so that a new methodology for it has been experimented in the internet-based enterprise modeling (*bumon*) project.

Keywords: Extended Objectives-Tree,

1.Introduction

Disciplines behind such success product line cases as AT&T ESA, IBM OS/360, Microsoft Windows are risky to follow as guideline. Because, all them might not won their victories by their excellent technology, but mainly by enormous capital investments and without any sound technology. It is hard to find any sound technological advancement in developing Windows [AND96]. It must be carefully checked to see whether there have existed the sound technology and architectural model at the beginning of the Product Line developments. In most cases, the model gradually came up along with, in most likely at the end of, the Product Line development life cycle as one of the resultants. In this paper, some acronyms are used for making the descriptions neat such that PL for *Product Line* and SSP for *Single Shot Product.*

This paper will discuss which methodology suites to PL development, especially for PL modeling assuming PL model-based development is a good approach. A couple of questions are raised on PL modeling and development and their methodologies, for example, does SSP development methodology not fit to PL development? In addition, what is particularity of PL development methodology? Why model-based approach is appropriate to PL development? What looks like PL modeling methodology, any special features?

The authors have conducted internet-based enterprise, interprise in short, modeling research project called bumon at University of Tsukuba, Tokyo campus since 1996 covering rather wide range of themes as formal and empirical approaches for interprise modeling, business process simulation, PL innovation. Some come-ups from bumon project are disclosed in this paper with focusing the PL modeling methodology named *Extended Objectives-Tree* that was experimented in the project.

2.Product Line Requirements

As opposed to *Single Shot Product*, Product Line seems have some unique features. In order to clarify what PL is, Table 1 shows the varaiability (difference) and similarity (commonality) between SSP and PL. The table lists up not only the SSP and PL natures but also characteristics of methodologies used for their modeling. SSP development methodology does not fit to PL development, since the methodology does focus on the SSP itself and needs not pay attention to product line family concerns. On the contrary, PL development methodology has to place more

emphasis on the family concerns, while SSP methodology needs not.

Table 1. Variability and Similarity observed in SSP and PL

Variability:	*SSP*	***PL***
Attitude	- Reactive to specific users' Requirements	-Proactive to arbitrary user groups forming market segments
Offer	-Stand alone product and not formed as a product family	-Family of several member products called poduct line
Evolution	-On demand-basis, que se la se la style	-Intention-basis, target specific market segments as developers want to get
Life cycle	-Single tired, SSP life cycle only	-Dual tired, PL life cycle as well as each member life cycles
Modeling	-Scoped only by SSP itself	-Scoped by PL family conscious
Similarity:	Both are Software Product, Solution Engineering used as well as Domain Analysis	

The variability in Table 1 shows requiring a different modeling for PL, since PL requirements differ from SSP's. Before step into PL modeling discussion, let's have a quick look at PL ideal factors that make PL more sound.

Necessary Conditions of sound PL include that the PL must have evolution ability, so that useful product member could be provided for use one after another along with the PL life cycle. Technical interpretation of this would be that the PL structure should be of evolution, extendable, and expandable through the life cycle. The PL development is required to meet high adaptability product with better quality with less cost with shorter time to market.

Sufficient Conditions of sound PL include that the PL is able to give greater benefits and profits both for users and developers with smoothly getting PL development and evolution consensus among the stakeholders. Today's economic situations are featured with rapid and ever requirement changing, stiff competition, high-graded product requirement. These features force PL development be done more effectively and efficiently.

3.Product Line Life Cycle

Usually it is very rare that such sound PL as meets the conditions described in the previous sections can be developed as a single shot product, but normally could been developed thorough a series of evolutionary developments in which the developers are trying to meet the market needs .

PL is usually developed through two different phases, namely, the initial development and the successive enhancement and evolution phases. PL is born in initial phase and may or may not grow up to a sound PL through series of development cycles.

Initial phase

The most important aspect of initial phase of PL project life cycle is whether the PL starts with its life cycle based on sound PL model. If not based on sound PL model, the PL will not grow up

smoothly, but may be have to change the basic architecture from a PL family member to another (or in case of within a PL family member, from a member version to another), since the upholding basic architecture falls into a ditch difficult to support the successive PL family members' requirements. Changing the basic architecture in the middle of PL development means discontinuity of the PL life cycle. This kind of change will be also happen in the middle of a family member development, namely, discontinuity of the family member life cycle.

Evolution phase

Once a PL is launched to user sites in marketplace, how the PL will keep growing up? In the PL evolution phase, at least two factors will be observed as key criteria which decide extent of the PL evolution, namely, the availability of feed back opinions on the PL enhancement from user sites and the evolution ability of the PL itself.

Through utilization of the PL by the users in marketplace, they may feed back their opinions on the PL to the developers. Those opinions are usually not known to the PL developers from the very beginning in the initial phase, but become gradually revealed along with actual users' use of it. The users' opinion never come up all at once in the initial phase, but be gathered one after another throughout the life cycle. The market demands, instead of each users' opinions, are also feed back to the PL enhancement development. The feed back information from user community or the market is thought valuable as the sources by which developers can make decision on the directions of PL enhancement. In case of no feed back information given, the developers have to make the decision on the PL enhancement and evolution by themselves.

For fulfilling those feed back opinions came along later, it is important to make judgement on whether the PL could be improved or enhanced. The conditions for making those improvements and enhancements possible, will be that the PL will not result in a hazardous situation in the course of its continuous enhancements. In other words, the PL model is able to keep supporting the PL enhancements along the PL evolution. What does the PL evolution ability means? The ability implies that not only the PL itself could be evolved,(this is so-called "Line evolution") but also each active family members, that are existing ones, could be evolved. Be sure not all of the family members are evolved well. Some members are not evolved for some reasons like the market needs dismissing. On the contrary, brand new member happen to be born in order for meeting new requirements. These are so called "Member evolution". The relationship between line and member evolutions seems aggregation. Aggregating all relevant members' evolution forms the PL evolution. An important issue is whether the developers are able to build such a model that can fulfill the needs for those member and line evolutions. In the later sections of this paper, the model itself is discussed in details.

4. PL Modeling

Why PL Modeling (Why not domain modeling)?

Domain modeling is usually carried out gathering systems development cases in the underlining application domain, making analysis of the cases, extracting commonality and varaiability features from the cases, and building a domain model that may be utilized for systems development in the domain. Domain modeling in general does not intend build any PL model. As discussed earlier in Chapters 2 and 3, a better way of PL development and more importantly enhancement for the evolution indeed relies on availability of appropriate PL model which member products development is based on.

Prerequisites for the modeling

There are several prerequisites that PL model must meet including as:

1. Clearly separate core functionality from trivial functions that PL supports,
2. Incrementally PL development must be possible with core functionality first,
3. Precisely elicit the PL requirements including potentials,
4. Appropriately make the PL adaptable to ever changing requirements,
5. Largely cover market share with long life cycle evolution ability,

What to model

PL architecture is a special case of domain model that is specializing at PL. The PL architecture serves as core portion for all member products, so that the architecture represents common structure, functionality, non-functionality as well as their varaiability which are made reference by PL members. A Priority is given to conceptual model in PL architecture, since the model is a key to construct a sound PL. The remaining aspects of the architecture should be supported by traditional technology.

How to model

Case-base method may be appropriately uses for PL conceptual modeling. However, it is known that sufficient cases are not always available for analysis. A methodology for making model incubation (Quick Evolution) must be useful, since PL requires as much extensible conceptual model. Extended Objectives-Tree is discussed with some others in later chapters.

5. PL Modeling Methodologies

It is widely believed that those methodologies invented as domain modeling, software engineering, or systems engineering could be used for PL modeling purpose. However, that is a wrong belief, since PL is neither equivalent to domain (PL is a sort of systems that is primarily oriented towards a set of product family-members) nor to single software or system. Some methodologies in software engineering and systems engineering might be utilized for PL modeling purpose, but they must not primarily be oriented towards PL modeling. What PL must differ from usual systems are those among that PL typically comprises a family of product members each of which has any of such family relationship to others as parent-children, brothers and sisters, uncle, aunt, cosines, nephews. The meanings of family relationship must be obvious, but how to produce each family member is not so obvious.

Basically, there are two different ways of developing a product line and its family member, i.e., develop it “with PL model” or “without PL model”. A model-based PL development is one way of developing any member of product line family and based on a model. That way of PL development must be better than non model-based development, since model shows a basic architecture by which each PL member must be built. A model is also useful as a guideline that shows a way of developing the PL. The model-based PL development obviously assumes that certain model for the PL exists. If no model does exist, one has to develop a model. A question is how one can develop such model that is useful for developing a PL member. Are there any methodologies which support PL model development? This and next chapters will focus in such methodologies that seem useful for PL modeling, especially for PL conceptual modeling. Theoretically, the conceptual modeling needs at least three different kinds of supports, let’s say,
Issues identification, Objectives structuring, and Commonality analysis. Issues identification comes first of all and is needed for identifying the issues surrounding the PL. Objectives structuring comes next and is needed for making PL objectives clear to define hopefully hierarchical structure of PL. Commonality analysis may be used to make PL relevant characteristics clear from those particular ones unique to each PL family member.

If the authors select some methodologies for PL conceptual modeling, they may be includes the Issue-based, Objectives-tree, Commonality analysis, and optionally Factor analysis methodologies. Those methodologies are strictly speaking, not invented for PL modeling but in general rather for arbitrary systems. The reason why touching on them in this paper is that the methodologies would be worth to experimental use to see whether they are useful for PL conceptual modeling.

5.1 Identifying Issues

It is said that if the issues about the problem, in this case the problem is PL, has been clearly defined, then more than half way of project might be progressed and the remaining parts must be accomplished without much difficulties. Paradoxically, if no issue is clearly made, then the project must be in uncertain and no one guarantees its sound progress. Issue identification is very important in development project and PL project is no exceptional on this.

The Issue-based may be useful for identifying issues surrounding the underlining PL. The Soft Systems Method, SSM in short, has been developed by Checkland et al [CHE90] and useful for identifying problem structure, especially with clear descriptions of each view to the problem based on their world view, so-called weltanschauunk. The SSM is particularly useful for ill-structured problem rather than well-structured. One must get sound views to the problem structure of the PL by the SSM at initial stage of the PL modeling.

As the other methods of issue-based objectives analysis, the authors can point The Interpretive Structural Modeling, ISM in short, and The DEecision Making Trail and Evaluation Laboratory, DEMATEL in short, both have been developed by Battel Columbus Laboratory, Swiss. The ISM is primarily-oriented towards large scale project analysis, in its issues identification and determining development objectives. ISM helps modeler to systematic seize issue chain identified in the problem and draw the multi-layered arc graph representing the issue structure. The DEMATEL method is primarily-oriented towards complex problem analysis like world model, ranking strength of issue relationships in a form of matrix representation [AKA92][KAM97]. As a traditional issue identification method, the KT method was proposed by Kepner et al to help analysts make clear analysis with focusing especially in such four aspects of problems as situation, problem, optimum solution decision, potential risk [KEP81].

The IBIS is another methodology that may be useful for identifying problem, issue, and arguments [CON88]. The IBIS may be helpful for one to find out any rational behind the issues showing all the relationships among three separate views surrounding the problem, namely, design issue, design conditions, design position and affirmative and/or negative arguments.
A ambitious research has been tackled by Software Reuse Special Interest Group headed by Komiya Seiichi trying to compromise those KT and IBIS to improved method[KOM93]. It is worth to note that the extensible improvement was made on development rational management through the research.

There are a lot of methodologies invented elsewhere. It is not this paper’s purpose to list up them exhaustively, but just points out they are fallen into two categories of ways of thinking, divergence and convergence, see, Table 2. Divergence and Convergence ways of problem analysis.

Table 2 Convergence vs. Divergence Study Method [KAM94]

Stages	Divergence		Convergence	
	Data used	Method	Method Type	Method
Set up Prob Seize	Problem	Interview Discuss	Evaluate	Several Evaluate Methods
As-is	Fact data	Brain method Play writing	Spacial	Relating
Issue	Issue data	Defects Play writing Brain	Cause-Effect	Fishborne Cause Analysis
Objectives Solution	Goal	Checklist	Evaluate	Several Evaluate
Structure	Idea data	Brain Wishes Listup Attribute Listup	Spacial	Relationship KJ method Cross method
Actual Plan		Gordon Shape Analysis Synetics NM method	Cause-Effect	Story method
Procedure	Work data	Brain Brain method	Temporal	Card-Part method Gant Chart
Total Eval	Reactive data	Questionare	Evaluate	Several Evaluate Method

6.2 Commonality Analysis

The Commonality analysis is a method that had been developed for domain analysis [DIA87]. Ruben Prieto-Diaz explored domain analysis method focusing in its process and proposed a way of representing domain taxonomy catching a hierarchy of objects, functions at the structure nodes and different kinds of relationships at the edges. Commonality analysis is a method that allows one to analyze any commonality, similarity, and variability that might exist in and among members of a product family. Weiss tried to apply the commonality analysis method to model of PL [WEI95].

The research theme of applying the Objectives-Tree analysis methodology to PL modeling, or if not the case, just software product developments, has been studied by several researchers including Parnas, Mittermeir, Umezawa, see, next sections for details.

6.3 Traditional Objectives-Tree Methodology

As discussed earlier in Section 3, PL objectives must address technical essential issues as well as mankind survival issues in environment and economics, since objectives are defined as representations of the PL strategy and the ways of putting it into practice, setting out goals to achieve and evaluating the performance. Objectives are traditionally used to focus on only certain specific product itself and typically not for a product line, since still rather week conscious to product line.

What does PL objectives differ from single product objectives? PL objectives must have perspectives of the PL's competitiveness over its lifecycle in the targeted marketplace, so that the objectives must usually cover longer time frame in time axis and more wider market scope in target market axis than objectives for a single stand-alone product.

Objectives-means method had first been proposed by Parnas [PAR79] and afterwards revisited by Mittermeire with an intention of applying it to software family requirement definition [MIT90], and the method provides a way to make a hierarchy of objectives with exploring each levels of

objectives' why-to-do as well as how-to-do. The exploration is to verify the hierarchy of the objectives. The procedures for defining objectives hierarchy seem rather ambiguous, but essential points include first, focusing at a major issue at which organization faces, second, thinking of how the issue can be accomplished, in turn, reading this how to do an issue for the lower level of objectives hierarchy. Repeating these steps of the procedures until four or five levels of hierarchy would be clearly defined. If the modelers are not confidence in the soundness of the hierarchy, they must verify the hierarchy each level by level to check relationships between a level of issue and its why should do and its how to do.

7. The bumon Objectives-Tree

The traditional method addresses only the hierarchy of issues. Though the hierarchy represents major aspect of objectives, there are still some other aspects of the objectives. One can optionally address functionality of PL depending on which aspect should be thought major factor in the PL project. Thus, PL objectives-tree can be defined.

In PL objectives-tree, it is important to define two different evolution aspects, horizontal and vertical. In horizontal evolution, one may deal with PL's characteristics including issues, functional and non-functional requirements, and structure. Non-functional requirements will cover performance and quality, ease to use human machine interface. In vertical evolution, one may define deliverable products showing parent-children relationships of the PL upon launching one new family member after another until the end of the total PL lifecycle. The Extended Objectives-Tree was first invented and experimented in bumon project prior to the deployment of the methodology into several practical PL developments.

1) Identify issues

Several issue identification methods are discussed in the previous section. In bumon project, SSM is mainly experimented, because problem nature as observed in PL is similar to a kind of social problems that fit to SSM.

2) Extract common characteristics

In top down approach, the modelers must break the PL objectives-tree down to the tree of each product member. Alternatively in bottom up approach, modelers may apply the same procedures to each product member, and then get product member objectives-tree. Modelers aggregate all objectives-trees, and could form PL objectives-tree.

Once the hierarchy of objectives is defined, try to extract common characteristics over the product line. This extraction is done through careful checking the objectives tree structure. Important thing is that those characteristics appear in each product members and should be dealt with major bone in the PL.

3) Make PL model

In previous section, common characteristics in PL are addressed. The term characteristics is off course a generic and could be interpreted into many others, for example, functional, non-functional, structural, behavioral. All these characteristics must be taken into consideration in making PL model. Taking the procedure mentioned in the previous section, one can define common functionality for the underlining PL. Likewise, one can define common structural and others, thus one can define the commonality of the PL with respect to non-trivial factors.

Using commonality analysis method, one can identify variability that may be exists in the PL. The authors have found some lessons learned through the experimental uses of the commonality analysis method undertaken in the bumon project at University of Tsukuba over these three years.

Commonality is rather easy identifiable, while variability difficult. The reasons why it is difficult to identify the variability are that commonality findings are close ended, while variability findings are rather open-ended. Variability must be identified in many senses, so that it is difficult for modelers to exhaustively identified every non-trivial aspects of a variety, let's say, difficult to find out every parameter sources representing for variability. See [MJM99] for details the lessons learned.

Variability leads to form a generic representation of PL model. Commonality leads to form a basic structure which each product line members must commonly incorporate. Basically, PL model can be constructed by the basic structure in generic representation.

8. Experiments

Experimental research has been done in bumon project at The University of Tsukuba in order to do verification of usability of the Extended Objectives-Tree Methodology in PL conceptual modeling. Human Resource Management (HRM) and some other domains are selectively experimented. Umezawa undertook his role to improve the relevant methods and apply them in HRM PL development, not only research base but also actual field practices [UME00].
For HRM, PL conceptual model has been made as:

1. Explore several product member in HRM with respect to functional and non-functional of the PL,
2. Check and extract core functionality from the PL Objectives-Tree as functions commonly appear throughout PL members. This is rather tough task, since it is somehow difficult to extract common function.
3. Construct conceptual model for the PL using those common functional.

.

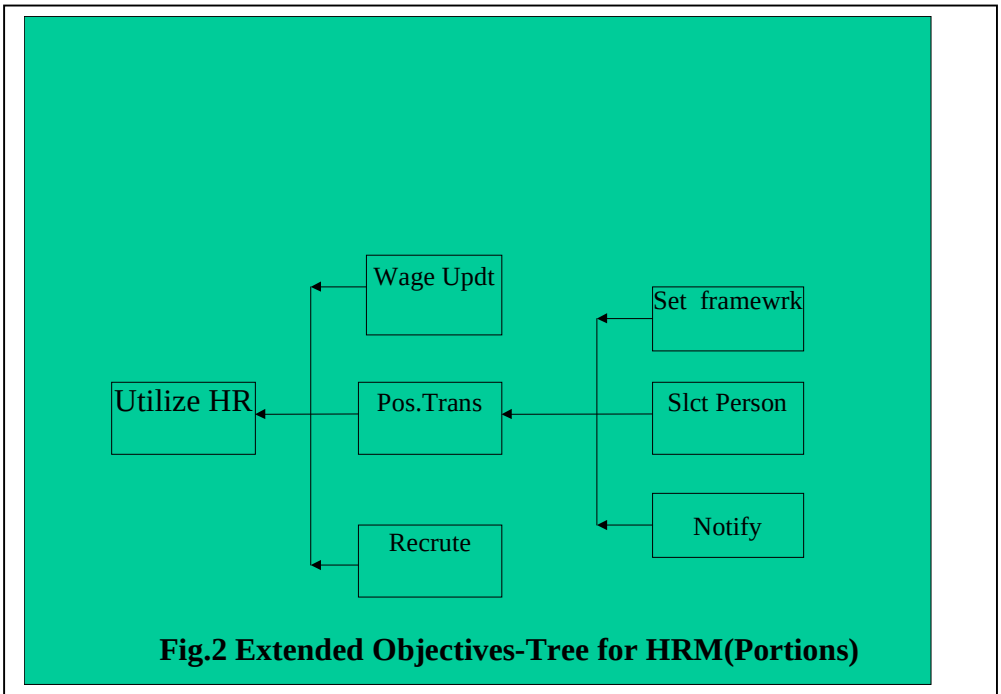


Fig.2 Extended Objectives-Tree for HRM(Portions)

8. Concluding Remarks

It is important to recognize what PL differs from stand-alone SSP type product. PL should keep

holding consistent model throughout its evolution lifecycle, otherwise both developers and users must take disadvantages, as discussed earlier chapters. PL modeling methodology is a key to success in PL. Specifically, in the modeling, such two analysis as issue identification and commonality and varaiability extraction are essentially needed to do. The bumon Objectives-Tree is helpful to form PL characteristic hierarchy on which based, conceptual model is built. What has been verification through bumon experiment and actual practices in PL development in field that the model-based PL is constructed and evolved well satisfying the requirements addressed in earlier chapters.

References

[AND96] Anderson, C.: An Operating System Development:Windows 3, Proc. ICSE,pp.101-Industrial Experiences Section 2D, 1996

[KEP81] Kepner, C.H. and Tregoe, B.B.: “The New Rational Manager”, Princeton Research Press, 1981

[AKA92] Akagi, S.: “Systems Engineering”, Kyoritsu-shuppan, 1992

[KAM97] Kamata,M.: “Research on Systematic Clarification Methodology for Problem Domain”, Master Thesis Paper, University of Tsukuba, Tokyo, 1997

[CHE90] Ceckland, Peter and Scholes, Jim: “Soft Systems Methodology in Action”, John Wiley & Sons, Ltd., 1990

[CHE81] Ceckland, Peter : “Systems Thinking, Systems Practice”, John Wiley & Sons, Ltd., 1990

[CON88] Conklin,J. and Begeman,M.L.: gIBIS: hypertext Tool for Exploratory Policy Discussion, Proc. CSCW’88, ACM, pp.140-152, 1988

[DIA87] Prieto-Diaz, R.: Domain Analysis for reusability, Proc.COMPSAC’87,pp.23-29, 1987

[WEI99] Weiss, D.M. and Lai, C.T.R.: ”Software Product –Line Engineering, A Family-based Software Development Process, Addison-Wesley, 1999

[WEI94] Weiss, D.M. : Commonality Reviews, AT&T Laboratories Technical Memorandum, BL0112650-940321-09, 1994

[WEI99] Weiss, D.M. Software Synthesis: The FAST Process, MultiUse Express, 1994

[KOM93] Komiya,S.: Reuse of Software Design Know-hows, 4th SIG C group, SPC Research Report, JUSE, in Japanese, pp.124-146,1993

[PAR79] Parnas, D.L.: Designing Software for ease of extention and contraction, IEEE Trans. on Software Engineering, SE-5, No. 2, 1979

[MIT90] Mittermeire, R.L. et al: An Integrated Approach to Requirements Analysis, Van Nostrand Reinhold, 1990

[UME00] Umezawa,K.: “A Definition Method for Product Line Architecture”, Master Thesis Paper, University of Tsukuba, 2000

[MJM99] Matsumoto,M.J.: Lessons learned in Commonality Analysis, R-Memo,University of Tsukuba,1999

ESAPS – Engineering Software Architectures, Processes and Platforms for System Families

Frank van der Linden
Philips Medical Systems B.V.
Veenpluis 4-6
5684 PC Best
the Netherlands
+31 40 276 4577
frank.van.der.linden@philips.com

1 INTRODUCTION

Across Europe 21 companies and research institutions work since July 1999 in a project "Engineering Software Architectures, Processes and Platforms for System Families" – ESAPS, project 99005 in the Eureka Σ! 2023 Programme. Based upon earlier and smaller scale experiments in ARES [1] and PRAISE [2], ESAPS aims to improve the state of practice in European industry with respect to the engineering of architectures, processes and platforms for system families in order to achieve significant higher levels of reuse and improved system quality

The ESAPS project is conceived as a 4-year project that has been divided into two phases of two years each. The first phase will concentrate on *the development of the approach and laboratory scale validation* of the individual technologies and technology integration framework. The second phase will focus on the *integration* of the individually validated technologies and *automation of the approach and industrial scale validation in various domains.*

The idea of a program family is not new and dates back to the seminal papers of David Parnas and Edsger Dijkstra [6, 4]. Nowadays system-families are becoming strategic business assets. For designing products we have to take into account an increasing number of user groups (so-called stakeholders) with diverging requirements. So we have to handle requirements and architecture in such a way that varying products can be derived using a common pattern – i.e. a family concept. The structuring of systems into system-families allows sharing of development effort within the system-family and as such counters the impact of growing system complexity. This makes it possible to sustain the rate of product innovation, while keeping guaranteed levels of overall system performance and quality. The fundamental concept of a system-family is based on domain-specific product architecture determining a layered set of platforms. A software engineering process focussed on pervasive reuse supports the family.

The ESAPS project aims to provide an enhanced system-family approach and enhanced domain-specific platforms for the application domains of the partners. ESAPS is designed to enable a major paradigm shift in the existing processes, methods, platforms and tools and comprises amongst others the change from engineering *single systems* to the engineering of multiple systems or *system-families.*

The basic idea is to have a large commercial diversity with a relative small technical diversity. It supports the development and evolution of product variants for different customer groups, e.g. for different countries. It reduces significantly the time to market for individual system variants. It enables a delta development approach.

2 ESAPS APPROACH

A system-family is defined as a group of systems sharing a common, managed set of features that satisfy core needs of a scoped domain. The idea behind a system-family approach is to build a new system or application from a common set of assets (domain model, reference architecture, components, platform) defined from earlier developed systems belonging to the same line. A software asset is a description of a partial solution. It might be a component, known requirements or design elements that an engineer uses to build or modify a software product.

ESAPS analysis tasks

One of the topics of ESAPS deals with analysis, separated in the analysis of the problem-domain and the solution-domain. The architecture should take into account all kinds of customer preferences but also preferences of the other

stakeholders. In particular we have an activity dealing with aspect analysis, incorporating the quality requirements (non-functional requirements) a product family has to meet. The book [3] distinguishes between runtime discernible, not run time discernible, inherent and business aspects.

Specific analysis and design techniques have to be developed and integrated to deal with the different qualities. Solutions may lie in (run-time or development-time) infrastructure support, in the design of specific interfaces, in the design of specific services or a combination of all. For example, concerns like configuration, testing, logging, initialization may be served by a combination of an infrastructure component and specific interfaces. These interfaces have to be designed for each of the components of the system family.

As can already be deducted from the different quality classes, the different qualities serve the needs of distinct stakeholders. Each of these qualities and interfaces are often very important from the view of a particular stakeholder.

As the development and marketing organization are important stakeholders, economic analysis is part of the aspect analysis investigations. Moreover, we have started special task dealing with economic analysis of the use of system family engineering.

It is our conviction that the system family approach only pays when there is a large amount of variability foreseen in the family. Only then the additional investments of setting up a product-line organization may benefit. However, Setting up the organization well may lead to possibilities of easy outsourcing, giving chances to middle size companies delivering specific and generic components that can be used within the family.

ESAPS design tasks
The ESAPS design tasks have a focus on how to share a common architecture within a system family. For instance, product line processes are considered. Jacobson et. al. [5] determines three process categories which should be combined to obtain a complete process for system family development. The first of which deals specific with all assets whose scope encompasses single products.

1. **application family engineering** covering the development of assets usable for the complete family
2. **component system engineering** covering the developments of single (platform) components to be used within system-family members.
3. **application system engineering** covering the necessary developments to construct the family members using the components developed in 2.

The reference architecture is an important system-family artifact, as it comprises the main architectural information of the complete family. It is an intermediate between the problem space, encapsulated in the domain model, and the solution space. Designing system families requires a way of architecting the commonality and variability in order to exploit them during the tailoring phase.

The system family architecture, or reference architecture, defines the components (which may be mandatory, optional, or alternative), component interrelationships, constraints, and guidelines for use and evolution in building systems in the system family. Consequently, the reference architecture must support common capabilities identified in the specification, the commonality, and the potential variability within the system family. The reference architecture is then used to create an instance of a particular architecture for a new variant. The system family scoping, defined in the analysis phase is essential for the development of a reference architecture since it defines the bounds for systems that will constitute the system family as well as the goals to be achieved and targeted by system family development.

ESAPS derivation tasks
Within this task the two aspects of system-family evolution are present: First, the family evolves because new members are added, secondly the separate system artifacts evolve separately into improved versions. An important tool for dealing with evolution will be derived from advanced requirements modeling and traceability techniques.

Deriving a new member of the family, defined by a set of new requirements, means to:

- identify those requirements which are common to all product variants and those which belong to specific variants
- select and adjust those parts of the system family architecture which can be reused in the new variant,
- to document and reuse the architectural decisions at the variation points which have to be taken to complete the variant architecture, and,
- identify and possibly adjust the components along with an adequate configuration that can be used in the new product.

Establishing and maintaining traceability between development artifacts is essential for an effective and error-free definition of a product specific reference architecture. More precisely

- the new requirements must be mapped onto the reference requirements defined in the domain assets
- the requirements must be interrelated with the corresponding parts of the reference architecture
- the architecture parts must be related with the system components

- the interrelations should be justified by recording important design decisions.

Capturing, documenting and maintaining the traceability information is labor and cost intensive. Traceability should thus be adjusted to product family specific needs and as far as possible be automated.

Moreover we need to be able to determine impacts of change and how changes propagate. Change to a product concerns modification related to defects or new requirements. It is then necessary to analyze the affected assets and determine whether changes have impact on reference assets. It is also necessary to keep track of the proportion of reference assets that are faulty. In general, this enables us to predict the properties of variants or new reference assets before actually building them.

Change may influence several system artifacts at different levels of abstractions such as components, architectural models, and requirements. Changing one artifact may require the adaptation of dependent artifacts, again at different abstraction levels. This dependency relationship refers to interactive change impact propagation. Change impact propagation is illustrated by references to potential changes required.

REFERENCES

1. ARES Web Sites, Available at Vienna <http://hpv17.infosys.tuwien.ac.at/ARES/> and Madrid <http://sirio.dit.upm.es/~ares/>
2. PRAISE Web Site. Available at <http://www.esi.es/Projects/Reuse/PRAISE/>.
3. Len Bass, Paul Clements, Rick Kazman, Software Architecture in Practice, 1998, Addison Wesley: ISBN 0-201-19930-0
4. E.W. Dijkstra, Notes on structured programming, O,J. Dahl, E.W. Dijkstra, C.A.R. Hoare, edss., Academic Press, London 1972.
5. Ivar Jacobson, et al; Software Reuse: Architecture Process and Organization for Business Success, 1997, ACM Press New York: ISBN 0-201-92476-5
6. D. L. Parnas On the Design and Development of Program Families, Transactions on Software Engineering, SE-2:1-9, March 1976.

Document Information

Title:	Software Product Lines: Economics, Architectures, and Implications Workshop #15 at 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland, June 10th 2000
Date:	June 2000
Report:	IESE-076.00/E
Status:	Final
Distribution:	Public