Fraunhofer Institut
Experimentelles
Software Engineering

Proceedings of

Software Product Lines: Economics, Architectures, and Implications

Workshop #3 at 23rd International Conference on Software Engineering (ICSE2001), Toronto, Ontario, Canada, May 13

Editors:
Peter Knauber
Giancarlo Succi

IESE-Report No. 051.01/E
Version 1.0
May 2001

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

Table of Contents

Perspectives on Software Product Lines

Report on Second International Workshop on Software Product Lines: Economics, Architectures, and Implications
Workshop at 23rd 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
Center for Applied Software Engineering
Faculty of Computer Science
Free University of Bolzano - Bozen
I-39100 Bolzano-Bozen, Italy
(+39) 0471 - 315 640
Giancarlo.Succi@unibz.it

1 INTRODUCTION

Product line engineering is a recent concept and one of the hottest topics in software engineering aiming at synergy effects in software development. Diverse benefits like cost reduction, decreased time-to-market, and quality improvement can be expected from reuse of domain-specific software assets, several successful product line projects have been performed and documented [3]. Also non-technical benefits as result of network externalities, product branding, and sharing organizational costs have been observed.

Following the remarkable success of the "First International Workshop on Software Product Lines: Economics, Architectures, and Implications" held at ICSE 2000 in Limerick [1], this second workshop stresses more the non-technical, that is, business and organizational aspects of product line adoption and institutionalization. Another major topic of interest are product line tools, as tool support seems to become more and more critical for the success of product line approaches. Different tool concepts have been proposed and discussed during the workshop. Requirements for tools and respective solutions seem to become more concrete, maybe resulting from the fact that the technical concepts and solutions of product line approaches are better understood and can therefore be better supported with tools.

The strong emphasis on establishing contacts and giving experts and practitioners from academia and industry a platform for discussion has been continued during this second workshop.

Section 2 of this workshop summary describes the formal structure of the workshop. In Section 3, a short summary of the invited talk on issues and opportunities in product line research is given. Section 4 summarizes key points of the presentations of the workshop participants that were given based on their submitted papers that are fully available at [2].

In Section 5, the plenary discussion is described and its major lessons learned are summarized. Section 6 concludes this paper.

2 WORKSHOP STRUCTURE

During the starting session of the workshop in the morning, a detailed schedule for each of the sessions was defined. The decision was to limit the presentations of position papers to 15 minutes each in order to save more time for general discussion at the end. Also, it was decided to not have a dedicated panel at the end of the workshop and instead allow for plenary discussion.

Thus, the workshop was structured into the following parts:

Three discussion sessions allowed for short presentation of theoretical and practical issues of product lines. The first of these sessions was concerned with product line introduction and organizational aspects, as well as with extensions of current product line concepts. The second session addressed the handling of product line assets including their derivation for concrete products, change management, and tool support. The third session was concerned with modeling and evaluation of product line architectures and testing of generic assets. The most important topics of each session are briefly summarized in section 4.

In an invited talk, Paul Sorenson reported from two case studies exploring the potential of product lines and summarized some lessons learned as input for future research opportunities.

More details on this talk are given in section 3.

A final plenary discussion with the objective to define some product line research agenda concluded the workshop. According to this objective, the discussion was structured into the three parts "Where are we?", "Where ought we to be?", and "How do we get there?".

More information about the topics of this discussion is given in section 5.

3 THE INVITED TALK

Paul Sorenson gave an invited talk entitled "Issues and Opportunities in Software Product Line Research".

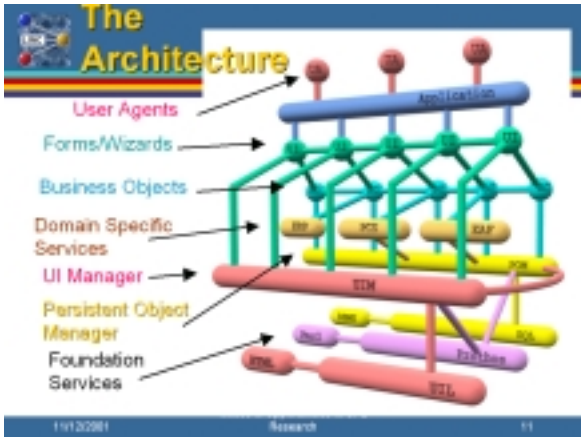


Figure 1: Product Line Experience (P. Sorenson)

He presented two case studies where the product line infrastructure was implemented in form of frameworks, issues that arose during their design and implementation, and lessons learned. In the first case study, an Engineering Application Framework was developed with the underlying goals to allow for easy evolution of the UI and workflow, to support deployment-related activities, and to anticipate refactoring of services. The resulting product line architecture is organized as set of domain-specific application frameworks composed of sub-frameworks where each sub-framework provides a small set of services (cf. Figure 1). The applications derived from this infrastructure are collections of services with varying degree of coupling. In the second case study, a framework was developed which can be used to build small client/server applications that run on the web (e.g., multi user games).

A couple of lessons learned can be derived from the case studies: Frameworks can be difficult to learn and understand, so a means of lowering the learning curve is needed. Tutorial sessions to show users what can be done with the framework, tools to investigate the framework and explore the use of hooks without having to understand the whole framework, and documentation including examples how to use the framework proved to be very useful. On the other hand, some research questions still need to be addressed. For example, it proved to be hard to decide if a framework is compatible with a particular application to be built or not. Another open issue is the evolution of frameworks: what constitutes a framework breakage and how can applications be protected from framework evolution?

4 THE SESSIONS

The topics addressed in the three presentation and discussion sessions can be clustered around four topic areas: product line introduction and organization, generic assets, architectural issues, and product line approaches. Due to timing constraints, it was not possible to allocate each of these topic areas to exactly one workshop session.

4.1 Topic Area 1: Product Line Introduction and Organization

Concerning this topic area, economic and organizational aspects of software product line introduction were addressed: when and where it is worth to introduce a product line approach in an organization, what are the expectations of the stakeholders concerned, how should convincing business cases look like, and can the risks of a product line transition be managed by following an incremental approach for product line introduction?

In his presentation, *Sholom Cohen* emphasized the need for a solid business justification, mainly based on a projection of costs for the current software development mode vs. costs of product line development. He argued to start with a data collection from legacy programs to provide a basis for comparison and then do a pilot study of product line development in order to collect data as basis for projected savings from product line development. As an example he presented a case study where three different reuse scenarios were analyzed: first, construction and then routine maintenance of an asset base for an initial set of domains, second, the later enhancement of that asset base, and finally, the later extension of the asset base into new domains. The potential benefits of the different approaches can then be illustrated with figures like Figure 2 which is taken from a case study.

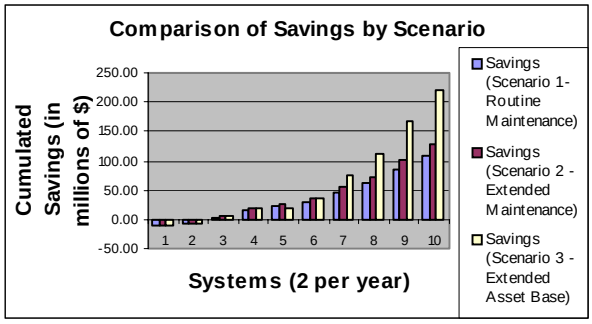


Figure 2: Comparison of Savings (S. Cohen)

Dirk Muthig argued to introduce product lines always in an incremental way. He presented some dimensions along which the incremental introduction can happen, two of which are software development life cycle stages and sub-domains.

Following the first dimension, stage after stage of the development life cycle would be transitioned towards product line mode, always encompassing both, domain and application engineering. Following the second dimension, one sub-domain after the other would be focal point of (full) domain engineering activities, thus getting revenues from the more promising sub-domains early while others are still handled product- (i.e., not domain-) specific.

In any case, following a non-incremental strategy implies changing products, processes, organizational structure, and the way people think at the same time, causing high upfront investment and uncontrollable risks. Instead, an incremental introduction enables to collect useful data to prove (and

quantify) success and benefits of the product line approach as well as to control risks and react early in case of problems.

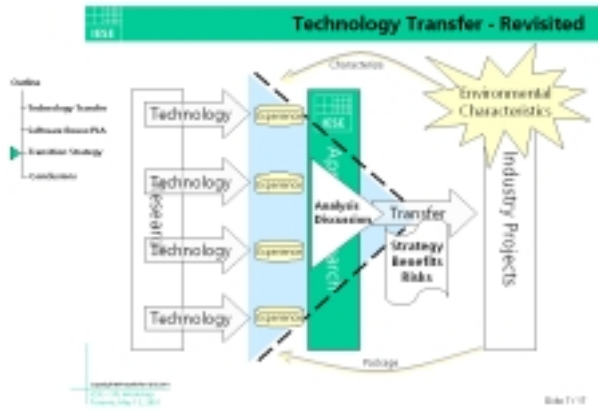


Figure 3: Dimensions for the introduction of PL (D. Muthig)

The paradigm of experimental software engineering in technology transfer based on data collected in the past (i.e., experience, cf. Figure 3) supports this incremental transfer.

4.2 Topic Area 2: Generic Assets

In his presentation, *Klaus Pohl* explained, how scenarios can be used to bridge the conceptual gap between solution space (represented by requirements) and solution space (as defined by the architecture), especially in product line development. Scenarios help to identify changes in requirements and differences between user-specific and product line functionality while at the same time reducing the complexity of product line description through the means of abstraction.

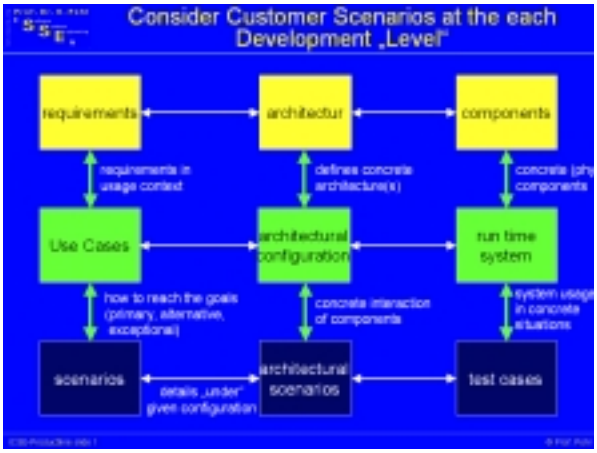


Figure 4: Customer Scenarios (K. Pohl)

Figure 4 illustrates the use of customer scenarios at each development level. The nine different artifacts used in this figure can be characterized at the meta level, including typed dependencies for each artifact type.

This way, traces among the artifacts can be modeled that help ensuring consistency when, for example, requirements are changing or, in the product line case, the generic artifacts are instantiated and adapted to user-specific needs.

John D. McGregor proposed a way to integrate test activities and their related assets in a product line context. Contrary to single system development organizations, product line organizations typically provide more continuity.

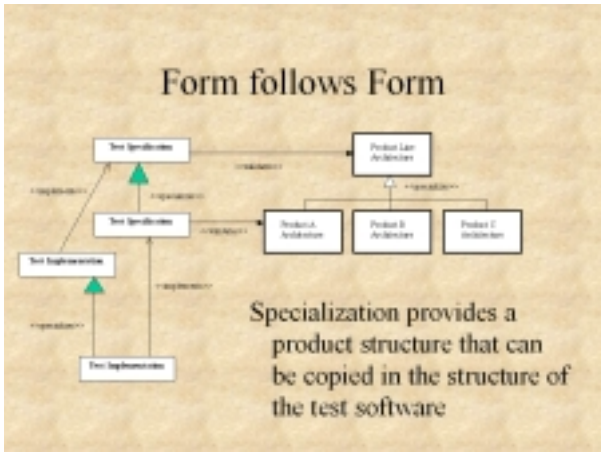


Figure 5: Integrating test activities in PL (J.D. McGregor)

This can be used to save testing effort while at the same time getting good estimates of the reliability of the products from the line. Figure 5 demonstrates how the test software should reflect the structure of the products derived from the product line. Since testing may require resources in the range from 50% up to 200% of the resources needed for the development itself, savings in the testing process due to process and/or test software optimizations can have significant impact on the overall performance of the development project.

In his presentation., *John MacGregor* argued in favor of a bottom-up product derivation process (cf. Figure 6) over a top-down parameterization process: Starting with the

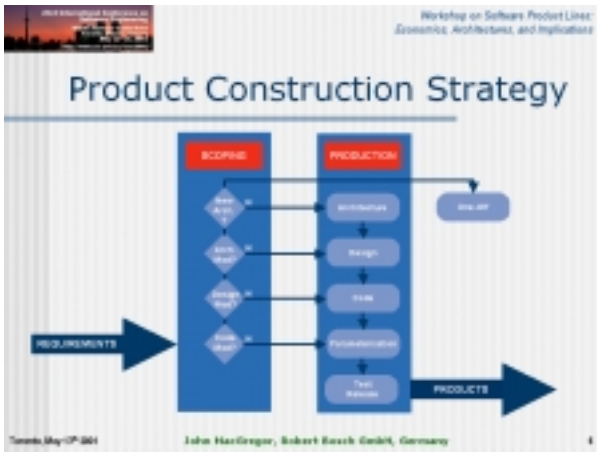


Figure 6: Integrating top-down and bottom up product line development (J. MacGregor)

requirements for a new product from the line one should first try to realize the new product using parameterization of code or to modify the code of the most similar existing product; the design would stay untouched. If that fails, a design adaptation has to be made, keeping the architecture intact. If that fails again, the architecture has to be modified in order

to adapt to the new requirements, in the worst case a new architecture has to be developed. In all cases, changes to higher abstraction levels have to be propagated down to the code.

An appropriate asset repository keeps the latest update of all assets touched, thus ensuring that all previously existing products from the line can be regenerated with equal or better quality without further intervention.

Giancarlo Succi presented Holmes, a framework supporting the different tasks for domain engineering while supporting multiple users, maintaining data and change consistency, and providing semantic support for relationship comprehension. Instead of re-implementing functionality that exists already in other, specialized tools like Rational Rose, Holmes allows for their easy integration.

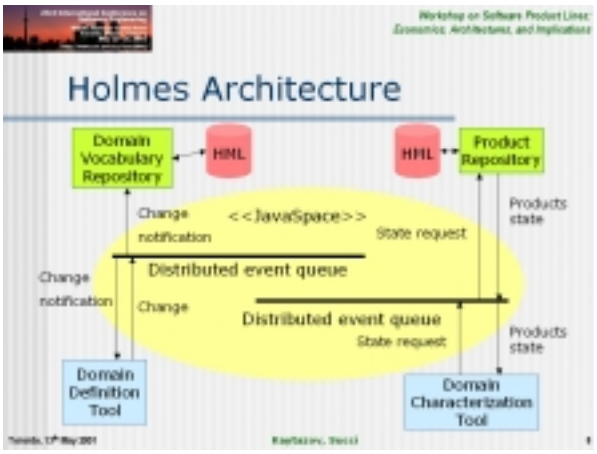


Figure 7: Architecture of Tool Support (G. Succi)

As shown in Figure 7, Holmes is based on JavaSpaces. Messages that can be passed from tool to tool are represented as objects together with their respective functionality. Queues are used to group related data, 3rd-party tools interested in one or more of these groups can then subscribe to the respective queues. Ongoing work is the realization of a critiquing tool written in Prolog that listens to (potentially) all queues and informs the user about potential flaws, independent of the tool creating the respective messages.

4.3 Topic Area 3: Architectural Issues

In his presentation, *Hassan Gomaa* demonstrated how to represent product line model variability in UML diagrams. He argued that UML is a well-known and established approach for modeling a software design, also for concurrent, real-time, and distributed applications. Analysis and design methods for UML exist. Thus, using UML for product lines will ease the transition from single systems to product lines. Use case diagrams are in a single system context used to define functional requirements for a system. For product line modeling, they can be used to model features and feature dependencies, including kernel, optional, and variant use cases.

Class diagrams can represent static/structural relationships, UML stereotypes are used to indicate kernel, variant, and optional classes. Statecharts and collaboration diagrams,

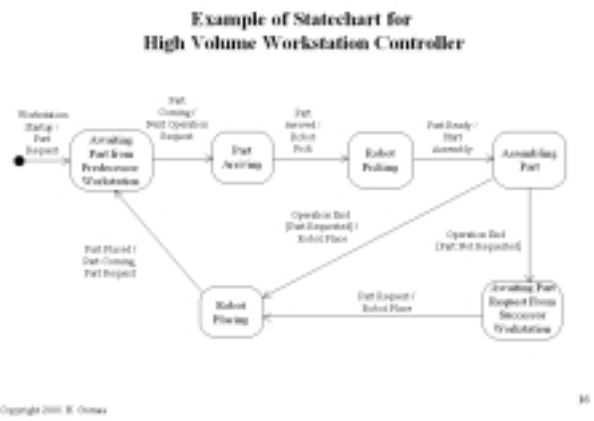


Figure 8: Using UML for PL (H. Gomaa)

again classified as either kernel, optional, or variant, are used to represent the dynamic behavior of the system (Figure 8 shows an example statechart). Future work should encompass the development of an UML-based software engineering environment for software product lines.

Andrea Savigni presented the concepts of Kaleidoscope, a reference architecture for command and control systems, and experience from two industrial case studies. The main property of Kaleidoscope is the separation of concerns, in particular, of computation, distribution, and strategy. The components are distribution-neutral, this way information alignment and information processing is kept strictly separate; separate entities, so-called projectors, take care of

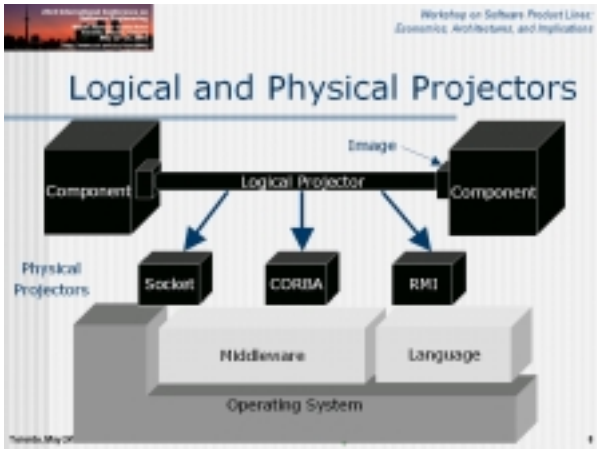


Figure 9: The Kaleidoscope Reference Architecture (A. Savigni)

the alignment (cf. Figure 9). The components are also strategy-neutral, that is, a separate entity takes care of triggering projectors for aligning and components for processing information.

The advantages of Kaleidoscope could clearly be demonstrated in two industrial projects. Experience from these projects shows that time-to-market can be reduced when configuration activities are assigned to domain experts (i.e., not to software engineers). Product line architectures,

as demonstrated with Kaleidoscope, are considered to be invaluable for large-scale reuse across applications and can be seen as the fundamental step in the transition from craftsmanship to actual engineering practice.

In his presentation, *David Rine* described requirements for and the design of a test framework product line for low earth orbit satellite constellations. This framework provides a common set of test threads that are required for the deployment, operations and maintenance, and anomaly detection and resolution phases. The test threads cover significant parts of satellite components that need to be defined and executed on the majority of satellite constellations with a minimum amount of data storage required.

The framework definition was done in four phases. In First, the focus area of research for the following activities was determined; this was necessary because it would have been impossible to achieve complete coverage of all possible constellations. In the second phase, the major nominal threads of testing necessary for each component on focus were identified. Then, a set of analysis criteria was determined and finally, each test thread was analyzed against these criteria. As a result, a common reusable set of test threads for multiple satellite constellations was developed under the strong resource constraints given by the domain.

4.4 Topic Area 4: Product Line Approaches

Juan Carlos Guzmán presented work and concepts from the embedded systems context. Usually, software and hardware are modeled using different notations and modeling techniques. For example, a variety of HDLs is used for protocol specification, special languages are used for software architecture description, and different programming languages are used for software coding. On top of that, lots of formal languages and mathematical notations exist. On the other hand, a high level of systematic reuse across soft- and hardware can only be achieved by a comprehensive design approach for complete systems.

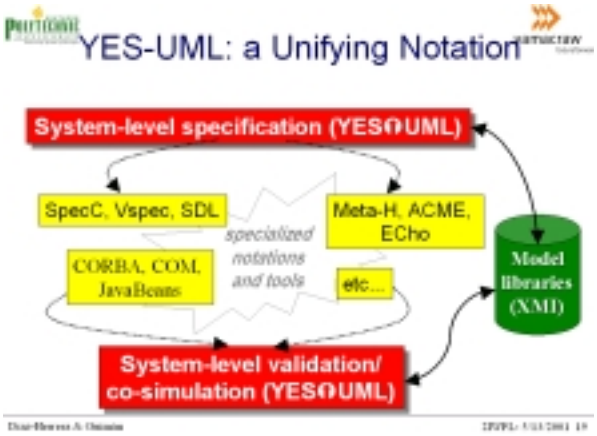


Figure 10: Structure of YES-UML (J.C. Guzmán)

For this purpose, he presented an UML extension called YES-UML that is developed and used in the Yamacraw project (see Figure 10). YES-UML shall contain concepts from description languages for hardware (like SpecC and SDL), software architecture (like Meta-H and ACME), and components (like CORBA and JavaBeans). All language concepts are represented based on XML. The start is made with SpecC, supporting tools are on their way. Using YES-UML as common representation for concepts and design of both, software and hardware, is expected to enable the use of existing software reuse approaches like product lines for complete system design.

In his talk, *Jeffrey Thompson* presented initial efforts in n-dimensional and hierarchical product lines.

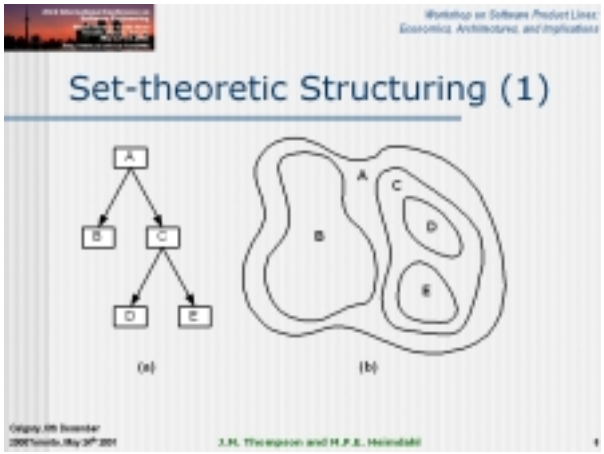


Figure 11: Formalizing n-dimensional and hierarchical product lines

N-dimensional product lines address the issue that many families are multidimensional in that they include several different, yet completely valid organizations that in turn may each have meaning in the context of the domain. In hierarchical product lines, sub-families can refine variabilities of the respective parent family or can add commonalities and variabilities unrelated to the parent. Such hierarchies can be represented like set structures (cf. Figure 11) and be handled using set-theoretic operations. Hierarchical product lines enable an easier extension of the family (the parent family can be much broader than otherwise), the possibility of refactoring in a way that sub-families have meanings in terms of the domain, and the possibility of an incremental product line introduction where one sub-family is implemented first, resulting in payoffs that can be used to finance further efforts in the implementation of other sub-families.

5 THE PLENARY DISCUSSION

A plenary discussion concluded the workshop. The discussion was organized in three major parts:

- Where we are
- Where we ought to be
- How to get there

The plenary discussion was heavily participated with contributions from several people, including -in order of appearance: Peter Knauber, Klaus Pohl, John D. McGregor, Andrea Savigni, James Paulson, Sholom Cohen, Dirk Muthig, Hassan Gomaa, Juan Carlos Guzmán, Mikko

Raatikainen, Paul Sorenson, Jan Bosch, Mike Bingle, Amr Kamel, John MacGregor, Jeff Thompson, Karl Reed. Moderator was Giancarlo Succi.

5.1 Where we are

Process modeling requires more investments in product lines, so managers have to be willing to accept initial investments that are likely to result in returns later.

Ideally, it should be possible to proceed incrementally, asking for small, incremental investments in exchange to a small, incremental transition to a product line approach.

However, not only incremental adoption of product lines is technically difficult, and does not make easier to convince managers, nowadays with very short term objectives to pay for something customers are not going to appreciate in the short term.

The scenario is not so new, as the same happened in the '60s and in the '70s with the moving to structured programming.

And the one thing we can be sure about and that we have learnt in the last 15 years of research on product lines and reuse, is that management support is essential.

The *good news* is that now software is viewed more as a product, and not any longer as just code. Companies are now realizing this, without really providing the essential support from the product, the product is not going to be of an adequate quality.

The *bad news* is that still among researchers there is no consensus whether product lines in software require more investment than in other disciplines, such as hardware or manufacturing.

A possibility to avoid stalling could be to start introducing systematically product lines in domains prone to product line approaches, such as those of component based or parameterisable.

More business case analyses are required, insulating the different costs occurring in product lines, outlining all the different benefits, including quality, time to market, lower know-how turnover, … Altogether, .the need for more accurate process models is the need for more costing and business advantage models.

It is interesting to analyse the connection between **product lines and frameworks**. So far, often people have made the claim that there is an intrinsic link between product lines and architectures. However, this is not so obvious. Moreover, trying to extend the framework into a product line may result in a breakage of the structure of the framework.

Testing for product lines should become more systematic. It should be performed right from the beginning of the process. The issue appears to be on how testcases can be derived. Formal documentation is required for an appropriate testcases. The level of documentation overhead depends on the desired quality. But developing test cases is itself a fairly lightweight work. Most of the automation required by practitioners is already present in existing tools.

Novelties in the area of product lines come from the ideas of **variation points** and **hotspots**, which have been popular since the early '90s.

An idea that is emerging is to use product lines as a tool for **safety critical** applications, or even to **certify** reference architectures for lines of safety critical applications.

5.2 Where we ought to be

Product lines are intended capture the evolution of software products and to last for a fairly long amount of time. Therefore, one of the hardest and most important aspects to consider is the **ability of product lines themselves to evolve** and change. Still, the only variability that they should have is that really needed to capture the required evolution, as a too general product line would end up being not manageable.

Also, it should be possible to determine when a given product line cannot be extended beyond a certain level, and that, for instance, a given functionality could not be added to an existing product line with a reasonable effort.

In simple terms, it should be possible to answer the question "How much more does this additional feature cost?"

Limited research has been posed on this aspect. Some work has been performed for the "Maintainability Assessment." It is an analysis focused on scenarios and maintenance profiles that enable to infer properties of the underlying system

Still, a better understanding is required to invest in, and produce product lines that are effectively usable.

It is also important to **perceive and appreciate the difference between evolution and variation**. Variation does not require a change of architecture. Evolution does.

A possible route could be the organization of the variation. Changes that are not taken into consideration in the variation points are really hard to manage. However, part of the problem is that it may not be known in advance what is variability and evolvability.

There is also a school of thought thinking that the problem is not evolution per se but the added complexity coming from evolution.

It is also important to improve the **use of compatibility** and incompatibility as means to develop open product lines. They can be profitably used both as means of developing open/close architectures, and as a tool to develop suitable economic models.

There are lots of other issues that require a deeper understanding. There is a gap between assets and products. It is still unclear how to bridge analysis assets and architectural assets. The longly-advocated paradigm shift from architecture based reuse to reuse based architectures has not occurred yet.

5.3 How to get there

It is hard to give advices in such an open area, where lots of promises have been made in the past year, most of which have not been kept.

It appears that **the creation of a database of codified experiences would be extremely useful**. Being able to systematize the existing experience into a single framework would create an infrastructure on which several open questions could be answered. This would be even more relevant if also industrial experience were to be used.

However, there are lots of issues related to the creation of such open database. Some relate to the quality of the data inserted in it. There is a non trivial problem of how to compare the data stored in it; meta-analysis could be a possible solution, but has not been used widely in software engineering. Moreover, suitable confidentiality agreements should be set with the companies from which the experience is to be extracted.

With such a database it should be possible to draw more definite answers to questions, such as the following:

- The use of domain isomorphism from requirement engineering and domain engineering to predict the likely variability in specifications
- The difference, if any, in the notion of changeability in time (evolvability) and in space (variability)
- The relevance of frameworks for product line architectures

6 CONCLUSION

Altogether, the workshop was 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 2002 in Buenos Aires the "Third 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 1st ICSE Workshop on Software Product Lines: Economics, Architectures, and Implications, Limerick, Ireland, 2000, Fraunhofer IESE Report No. 070.00/E, 2000, available at http://www.iese.fhg.de/pdf_files/iese-070_00.pdf

[2] Software Product Lines: Economics, Architectures, and Implications, Peter Knauber, Giancarlo Succi (Editors), Proceedings of 2nd ICSE Workshop on Software Product Lines: Economics, Architectures, and Implications, Toronto, Canada, 2001, Fraunhofer IESE Report No. 051.01/E, 2001, available at http://www.iese.fhg.de/pdf_files/iese-051_01.pdf

[3] P. Clements, L. Northrop: Software Product Lines. Practices and Patterns, Addison-Wesley, 2002

Predicting when Product Line Investment Pays

Sholom Cohen
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15217 USA
+1 412 268 5872
sgc@sei.cmu.edu

ABSTRACT
Many organizations require financial justification before proceeding on a product line approach. The product line approach may appear very attractive in an intuitive sense. There are obvious benefits in speedier time to market and higher quality. But, without the cost figures, the decision makers won't budget funds or personnel resources to carry out the up-front asset construction tasks. In addition, not all organizations are ready to commit up front to a full asset set, one that covers most if not all product line features. They favor a more incremental approach that tackles areas of highest and most readily available commonality first.

This position paper defines a few key factors to consider in taking an incremental approach to fielding a product line. It also sets out the data needed to make a compelling business case.

Keywords
Product line investment, business case, cost-benefit

1 INTRODUCTION

An organization makes a sizeable commitment of resources when it turns from stovepipe or silo approaches to a product line [5]. Occasionally, an organization is faced with a desperate situation – go product lines or fail to meet customer demands [1,2]. The organization must commit the resources because it simply cannot continue to produce systems one-at-a-time. More often, the organization has a number of options for upgrading from a stovepipe approach: flexible architecture, framework evolution, component strategy, product line, or others.

A product line approach is often viewed as the most risky of these alternatives due to:

- long lead time to develop assets
- insufficient number of systems that could potentially use the assets
- drain on critical resources

However, product line approaches also offer many advantages including financial, quality, and non-tangible benefits (e.g., developer satisfaction)[2]. Generally, the business case must be made financially, either in terms of reduced development costs or reduced time-to-market. Given the many options, how do you determine when product line investment pays?

2 THE BUSINESS CASE FOR PRODUCT LINES

Several authors [3, 4, 5, 6] have discussed economic issues relating to reuse. They generally tout the lower development costs and higher quality of software developed from assets. But in making the case, their economic considerations are generally based on the following assumption: we will develop assets for a domain or for a product line and reduce the costs of developing software systems that would otherwise be produced in stovepipe fashion.

Weiss' model shows that product line investment pays after 3 or 4 systems [6]. The organization recovers the costs of developing and maintaining the assets as the costs of product development come down. However, this model and others make assumptions about the custom-built portions of each system. A generator or comprehensive asset base is used to construct the systems in Weiss' model. The custom-built portion is small compared to that being generated or produced from component assets.

What if an organization wants to pursue an incremental approach to developing assets and fielding a product line? This approach can show the benefits to be gained with less up-front commitment and shorter lead-time. But what about costs of an increment approach? What data are required and how should they be presented? What are the benefits to be gained? How do you know when the product line investment pays under this strategy?

3 BUSINESS CASE FOR AN INCREMENTAL APPROACH

To establish when product line investment pays, the organization must compare the current, stovepipe approach to product development following the product line approach. The organization may also consider a number of alternative product line strategies:

Scenario 1. Build and maintain baseline assets to cover a subset of system features and rely on them as the basis for future development.

Scenario 2. Build and invest in refinements to baseline assets. Maintenance activities go beyond those of Scenario 1.

Scenario 3. Expand feature and asset coverage into new domains not currently addressed by existing assets.

Five key factors influence the cost projections for applying a product line approach. The factors that will support making the product line comparison includes the following:

1. Schedule for developing new systems

For example, there will be, on average, two new systems per year in the product line.

2. Historical costs for developing comparable systems

Table 1 illustrates the historical costs of developing systems in an organization using a stovepipe approach. The business case uses these data to determine when the product line investment pays. (Note: the dollar amounts used in the following tables are based on actual organization results for legacy costs, asset development costs, and savings from an initial product line development.)

Program Name	Lines of Code	Development Cost (in millions)	Maintenance Cost per year (in millions)	Number of Years in Maintenance	Cost with Maintenance (in millions)	Cumulative Costs with Maintenance
AA	122500	17	1.70	4.00	23.80	23.80
BB	300000	46	4.60	3.00	59.80	83.60
CC	286000	31	4.10	2.00	39.20	120.80
DD	640000	63	15.00	1.00	78.00	198.90
EE	500000	118	15.00	0.00	118.00	316.80

Table 1. Historical Costs of Systems

The table includes maintenance to show the true lifecycle costs for the system. Lifecycle costs also offer more realistic comparison with lifecycle costs under product line development. The figures do not include any projections for net present value or other "cost of money" figures. These could be included in the analysis for greater accuracy in determining the true cost of the product line investment.

3. Characteristics of each new system (size, complexity, level of commonality)

The cost projections assume that the characteristics of new systems are similar to those of past systems. This assumption will require normalizing the *degree of reuse* (DOR); that is, the percent of reuse that will apply to a new system. Given the baseline assets, the DOR is 25%, meaning that 25% of a new system may be derived from existing assets.

In addition, the projections will use a standard figure for the costs of reuse (COR) for applying those assets including:

- Developing requirements and design to point where assets may be used
- Determining assets to use and how to use them
- Asset testing and integration
- Reporting results of asset use back to asset developers

The COR may vary from 0% (where there is no effort to produce the end product) to 100% (where the cost to reuse assets equals the cost of developing the end product software from scratch). For the cost projections, reuse costs of 10%, 25%, and 40% will offer a range of possible values. These costs include training to familiarize developers with core assets including the architecture, components, and test facilities. The analysis in the paper includes only the 10% and 25% COR.

4. Ability to use existing assets; need for new asset categories

The cost projections also assume that where assets are available, new systems will make use of those assets. No one system can make use of all the assets. The projects will assume DOR based on earlier experience.

5. Cost comparisons

Costs for the stovepipe approach start with legacy data. These historical data form the basis for estimate, allowing projections for reasonable size and complexity of the new systems. Cumulative costs across these systems include initial development costs plus maintenance for the individual systems.

The three product line scenarios show the use and sustainment of baseline assets, enhanced sustainment, and building new assets. While there will be savings from cost avoidance for future systems under each, these savings must be reduced by development and sustainment costs of the assets. Given the following assumptions:

- two systems per year over five years, labeled System 1 through System 10
- using exis ting assets under Scenario 1
- DOR = 25%
- COR = 25%

the total cost avoidance will be almost $108 million. Cost avoidance shows the following relationship:

cost avoidance = cost without reuse – cost with reuse – reuse expenditures

See Table 2 for the results of the analysis across 10 systems.

System	Cost without Reuse	Cost with Reuse	Cumulative Reuse Expenditures	Cumulative Savings over 5 years
System 1	25.50	20.72	16.00	-11.22
System 2	50.15	40.75	16.80	-7.40
System 3	114.55	93.07	17.60	3.88
System 4	176.65	143.53	18.40	14.72
System 5	219.95	178.71	19.20	22.04
System 6	261.20	212.23	20.00	28.98
System 7	354.20	287.79	20.80	45.61
System 8	439.70	357.26	21.60	60.84
System 9	572.70	465.32	22.40	84.98
System 10	698.20	567.29	23.20	107.71

Table 2. Cost Savings under Scenario 1 (Five year period)

Scenario 2 invests in improving the usability of the existing assets. The goal of this scenario would be to reduce COR from 25% to 10%. Asset improvement would include the following:

- development tools or asset enhancement that reduce the costs of developing requirements and design to the point where assets may be used
- training in asset use
- improved asset test software
- streamlining feedback of asset users to asset maintainers

Although enhanced maintenance would double sustainment costs, the savings would increase by $18 million (in addition to the $108 million shown in the first option). These increased savings result from the lower cost of reuse. Table 3 shows the details.

The next analysis looks at extending the assets into new domains. In this analysis, there is a second investment of $16 million for new assets to increase DOR to 50%. The analysis includes the enhanced maintenance levels shown in Table 2. From the additional investment of $16 million in new assets, the net increase in savings is $94 million. Table 4 shows the details.

System	Cost without Reuse	Cost with Reuse	Cumulative Reuse Expenditures	Cumulative Savings over 5 years
System 1	25.50	19.76	16.00	-10.26
System 2	50.15	38.87	17.60	-6.32
System 3	114.55	88.78	19.20	6.57
System 4	176.65	136.90	20.80	18.95
System 5	219.95	170.46	22.40	27.09
System 6	261.20	202.43	24.00	34.77
System 7	354.20	274.51	25.60	54.10
System 8	439.70	340.77	27.20	71.73
System 9	572.70	443.84	28.80	100.06
System 10	698.20	541.11	30.40	126.70

Table 3. Cumulative Savings under Scenario 2

System	Cost without Reuse	Cost with Reuse	Cumulative Reuse Expenditures	Cumulative Savings over 5 years
System 1	25.50	19.76	16.00	-10.26
System 2	50.15	38.87	17.60	-6.32
System 3	114.55	88.78	19.20	6.57
System 4	176.65	136.90	20.80	18.95
System 5	219.95	160.72	38.40	20.83
System 6	261.20	183.41	41.60	36.19
System 7	354.20	234.56	44.80	74.84
System 8	439.70	281.58	48.00	110.12
System 9	572.70	354.73	51.20	166.77
System 10	698.20	423.76	54.40	220.04

Table 4. Cumulative Savings from New Asset Investment

4 COST COMPARISONS

The asset base must be sustained over time as assets are used in real products. Sustainment addresses defect reports and, as well, incorporates feedback from users. Costs occur in three main categories:

1. Routine maintenance including defect repair and limited perfective enhancements of existing assets based on user feedback
2. Development of new assets – extending the asset base with assets in existing domains (Scenarios 1 and Scenario 2) or in new domains (Scenario 3 only)
3. Improved packaging of assets to support new programs – improving the ability of users to make use of assets (Scenarios 2 and 3)

The activities that may be considered "routine" maintenance are covered by the $1.6 million per year figure, 10% of the development costs.

These projections show results over a five-year period with two systems per year. It's possible that more than two systems will be under development per year. The trend charts offer even more dramatic results, given the assumptions. Figure 1 plots the results from Scenario 1. With the limited DOR of 25% and an assumed COR of 25%, the gap between product line costs vs. stovepipe costs grows dramatically. Scenarios 2 and 3 illustrate the affect of decreasing COR and increasing DOR, respectively.

Under these scenarios the gap grows more quickly. Figure 2 illustrates the increased savings. These savings will come sooner if more systems are produced than under the nominal two systems per year.

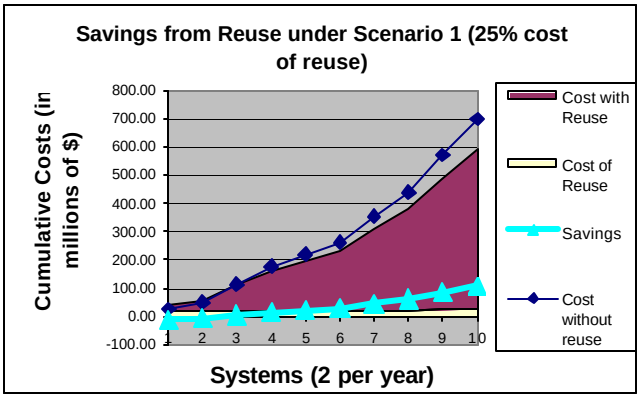


Figure 1. Savings under a Product Line Approach

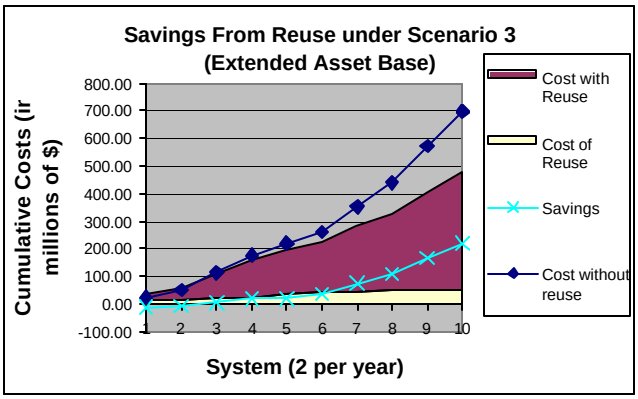


Figure 2. Savings under Scenario 3

Another comparison looks at the three scenarios and their respective cost savings. This shows that the third scenario generates much greater savings. However, there is a dip in savings when the expenses for domain expansion hit, between systems 4 and 7. The organization must be prepared to absorb this negative return until it recovers the cost of new asset development on future systems.

5 DEMONSTRATING THAT PRODUCT LINE INVESTMENT PAYS

Tables 2-4 show the cost benefit that the organization will realize from the product line investment. Plotting these results shows the trend towards even greater savings. While total system costs decrease by 15% for Scenario 1, they decrease by 30% under Scenario 3.

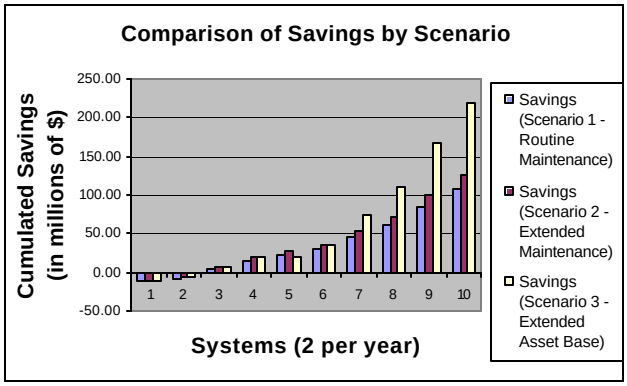


Figure 3. Comparison of Savings under each Scenario

These scenarios follow an incremental approach. The up-front investment costs are much lower than for a full asset base. The organization obtains a product line demonstration capability in short order, possibly two years or less. Because the assets are not comprehensive, product development users face less risk when they use the assets. The incremental approach tracks the anticipated results to show that the organization is or is not meeting goals. They may cut off the investment if projections do not materialize.

These analyses demonstrate the need for thorough tracking of cost information. While they are based on actual results, they include long-term projections that will be strengthened by future product line application.

ACKNOWLEDGEMENTS

This document has been adapted from analyses done with SEI customers. The author extends his thanks for their participation in generating the results.

REFERENCES

1. Brownsword, L.; Clements, P. A Case Study in Successful Product Line Development (CMU/SEI-96-TR-016). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1996.

2. Dager, J. C. "Cummins's Experience in Developing a Software Product Line." *Software Product Lines: Experience and Research Directions. Proceedings of the First Software Product Line Conference (SPLC1)* Denver, CO, August 2000, Kluwer, 23-46.

3. Lim, W.C. Managing Software Reuse. Upper Saddle River, NJ: Prentice-Hall, 1998.

4. Poulin, J. S. *Measuring Software Reuse*. Reading, MA: Addison-Wesley, 1997.

5. Reifer, D.J. *Practical Software Reuse*. New York, John Wiley & Sons, New York, 1997.

6. Weiss, D.; Lai, C.T.R. *Software Product-Line Engineering: A Family-Based Software Development Process*. Reading, Ma: Addison-Wesley, 1999, 45-49.

PRODUCT LINES IN THE CONTEXT OF EMBEDDED SYSTEMS

JORGE L. DÍAZ-HERRERA AND JUAN CARLOS GUZMÁN
Department of Computer Science, Southern Polytechnic State University
1100 South Marietta Parkway, Marietta, GA 30060-2896
Tel: +1 770-528-4281 FAX: 1 770-528-4281 E-mail: {jdiaz, jguzman}@spsu.edu

ABSTRACT

Successful product line engineering requires the achievement of design-for-commonality and control-of-variability, on the one hand, and "ease of adaptation" on the other. What are successful product line solutions in the context of embedded systems? How can they be designed as product lines, and at the same time allow the assimilation of (new) technology effectively, and offer the flexibility required for design-with-reuse? Many embedded systems involve the concurrent design of both special-purpose hardware (ASICs) and software, since functionality is partitioned accordingly. Interestingly enough, ***ASIC design for reuse****, what we call PLSIC for Product Line-Specific Integrated Circuit, is a new paradigm in hardware design. Interest in PLSIC is growing due to the capability offered by silicon technology where designers can integrate complete systems on a chip, and the difficulty in reducing time-to-market due to the sheer complexity with the large number of gates now available. In this paper we illustrate the problems and present our approach. PLSIC design suffers from many of the same problems already encountered with software components, their integration into a complete system, however, is less a problem than for software, manly due to the relatively stable hardware component integration technology and standardization efforts.*

1 INTRODUCTION

Plummeting hardware costs are leading to rapid growth in new application areas, and especially in the embedded systems arena. The embedded system market, and more specifically systems-on-a-chip or system-on-a-package (SoC/SoP), is expected to reach more than $26.5 billion in 2003 [1]. The projected shipment of processors in the year 2000 world-wide would have been more than 8,200,000,000 units, and the vast majority of them (98%) are found in embedded applications [2].

Silicon technology has allowed the building of complete systems on a chip with tens of millions of transistors (50 million by year's end). With this greater capability, the complexity of applications tends to increase; and the complexities of the dynamics of the real world with which they have to interact make embedded systems development a very complicated task indeed. The sheer number of such systems plus the complexities of embedded software, point to the need for finding better solutions faster.

Due to ever-shorter product life cycles, the competitiveness of development organizations depends increasingly on the ability to improve their design and development processes faster than their competitors. One possible way to reduce the productivity gap is to provide a *universal SoC platform*, agreed upon by all vendors; this is highly unlikely to happen. Another approach is to *automatically synthesize* SoC blocks from high-level specifications; this will have interoperability problems for particular applications and fabrication technology. Automation is regaining interest due to novel parsing and source processing technology [3, 4, 5]. Can this technology be used for hardware components?

A more promising approach, and the one discussed in this paper, is to move our focus from engineering single systems to *engineering families of systems* by identifying "reusable" solutions that support the future development of multiple systems and building SoC from "reusable IP[1]." An organization that focuses on a "range of similar" products can potentially take advantage of *economies of scope* [6], a benefit that comes from developing one asset used in multiple contexts, thus reducing development, maintenance, support and marketing costs, since products would share a common design, development effort, and actual components. Scope economies arise when the cost to develop multiple products together is less than the sum of the cost of developing the products individually. The overall goal is to produce quality products consistently and predictably by moving toward an asset-supported, component-based development approach.

The underlying assumption is that the benefits from the reuse of components will offset the potential extra cost for the increased organizational and design complexity. For example,

- Organization reduces cycle time and cost of new applications by eliminating redundancy. E.g., a 65% code reduction was achieved by redesigning five similar applications into a product line [7].

[1] IP = Intellectual Property, broadly speaking, any *packaged* knowledge that can be bought and reused (e.g., algorithms, software, designs, test benches, blocks or core)

? Building systems from a common component base *reduces risk* and improves quality by using *trusted, proven components*.
? An asset-based approach allows the management of legacy systems more efficiently, increasing the likelihood of *longer time-IN-market* [8].
? An organization based on product lines evolves a common marketing strategy, and strengthens core competency around strategic business interests and goals [9].

The benefits from reuse are clear, provided that assets can be reused several times. Assets designed for reuse are significantly more expensive, but significantly less so if subsequently reused. To benefit from the similarities that products may have, we must make product offerings on a common domain organized into product lines, and this transition must be planned carefully encompassing not only technical issues but also organizational and business aspects [10, 11].

In hardware, the (re)use of existing and pre-verified designs, known as design reuse [12, 13], is seeing as a most promising approach to "bridge the gap between available gate count and designer productivity." [14] How can IP or cores be designed as product lines, and at the same time allow the assimilation of (new) technology effectively, and offer the flexibility required for SoC design-with-reuse? What are successful product line solutions in the context of embedded systems?

In the rest of this paper, we address these and other questions referring to the technological challenges of building product lines in the context of embedded systems. Due to space constraints however, we do not discuss specific technology in detail. Rather, we present a roadmap for the various phases and kinds of artifacts produced. Section 2 discusses the problems of systematic reuse in this context. Section 3 presents current industry focus in terms of infrastructure and technology, and a brief introduction of our framework. We conclude with a discussion of the effort that lies ahead.

2 REUSE IN AN SOC CONTEXT

Reuse in general needs to be planned-for, since it requires a consolidation of understanding of systems in a domain in order to *exploit commonality* and *anticipate diversity*. The formal interaction between asset producing and asset utilization activities, whereby applications are assembled from prefabricated artifacts, is achieved if:

- analysis identifies explicitly variations to anticipate adaptations, and
- design for adaptability is engineered a priori to create assets for future software development. Assets are indeed ***investments*** for future work.

In the hardware arena, the issues and problems associated with reusability are fundamentally the same than those of reusing software components. There are important differences, however. Component interfaces are far simpler and more standard. There are several industry attempts to make the design-for-reuse process more visible and earlier in the development cycle with the development of reuse-driven design methodologies and tools. An industry group has been formed, the Virtual Socket Interface Alliance [15], to facilitate the adoption of design reuse, and to foster the development of industry standards.

In our context, that of embedded systems, an asset is ***reusable functionality***, also known as *intellectual property* or *IP*, that may be implemented in hardware or software. IP components are known in the hardware world as macros, core, or blocks. In this paper, we use these terms interchangeably. They refer to a stand-alone unit of a complete SoC design. System designers also make a distinction between *hard IP* and *soft IP*. The former is functionality fully designed, placed, and routed by the provider. The latter is delivered as synthesizable code usually as RTL (Register Transfer Logic) code.

Two fundamental component features that affect composition ability and reuse-payoff are *scope* and *granularity*. A component's scope can be domain-independent, domain-specific, or product-specific. The second component feature of granularity has two dimensions, namely fine-grained (small-scale) and coarse-grain (large-scale) granularity. The former is typically found in domain-independent components, whereas the latter are typical of application subsystems. Component functionality is less with the size of it, but reuse profit is directly proportional with size.

As it happens for conventional software reuse, IP reuse actually occurs at various levels of granularity and of generality. *Domain-independent IP*, such as processors, device controllers, standard interfaces, etc., clearly need to be extensively designed for reuse. *Domain-specific IP*, such as multi-media, jpeg encoders, data communications, digital signal processing, etc., are to be designed for reuse to fit several different products in a given market in a short time frame. The semantics of the component are domain-dependent, and hence have little or no use outside domain. Such domain-specific components make certain assumptions about how they will be used, reducing generality but increasing its usability. The traditional conflict between design-for-reuse and design-with-reuse [16] shows up here.

Product-specific components are reusable within a specific application (to generate various instances or

products). The semantics of the component are bound to a specific application type. The product line may dictate a generic architecture, and component roles will be developed to fit it. Typical product-specific components are entire architectures developed internally, or for technology nearing retirement.

The issues addressed during IP reuse are similar to those of conventional software reuse, as follows:

- solve a general problem
- design-for-reuse: create a ***reusable asset,*** i.e. one that is fully documented, has good code and robust scripts; is verified independently of any chip design, and with high-level of confidence
- design-with-use: create a ***usable asset***, i.e. one that is adaptable to multiple technologies, and that is usable in a variety of simulators

The technical problems are also fundamentally similar to those encountered in reusing conventional software as well:

- inappropriate design representation
- incomplete design information
- unavailable or obscured supporting scripts
- not appropriately archived design (configuration management)
- unavailable tools or lack of interoperability

2.1 IP DESIGN-FOR-REUSE/USE

The problems associated with design-for-reuse are basically three-fold: (1) Design-for-Commonality, (2) Analysis-of-Variation to anticipate adaptation, and (3) Control-of-Variability.

Design-for-Commonality forms the basis for standardizing assets to build products in a domain by encapsulating common features of related products, and by defining a common architecture for related products. Each component is a means of achieving one or more reusability features. They must have syntactically and semantically clear specification separate from implementation details and independence from environment. Today, each IP vendor uses its own specification and documentation style.

Like commonality, adaptability must be engineered a priori, and thus, *Analysis-of-Variation* must explicitly identify variations that anticipate adaptations. Optional parts are very important. Possible combinations of optional features must be supported by the product line architectural design and the flexibility incorporated in the component design, otherwise it may be impossible to reuse an asset "as-is" because commonality and specificity are mixed. This would make it necessary to modify the asset when reused in the new context, and this should obviously be voided whenever feasible. Configuration models help here.

Control-of-Variability provides flexibility in the assets to meet requirements for a variety of products without compromising commonality; it requires careful design to include appropriate levels of parameterization, generalization and specialization, and extension. Like commonality, adaptability must be engineered a priori, and thus, analysis must explicitly identify variations that anticipate adaptations.

There are well-known ways to control variability in software at the component level (e.g., class hierarchies, generators, generic parameters, configurations, etc). In hardware, the closest mechanism to adaptability is the use of Field Programmable Gate Arrays, but there is little experience in using this technique for PLSIC (Product Line Specific Integrated Circuit).

An important challenge here is to develop IP that is predictable, portable, and modifiable. For example, a soft IP is portable but not predictable, since it must be synthesized in a given setting before verification and use. On the other hand, a hard IP is predictable but not portable since it has been instantiated into one technology and it cannot be modified and ported to another technology.

In summary, the design-for-reuse of IP requires:

- IP as a Virtual Component (VC) with all the necessary information for use (functionality, adatability, test, etc.) in a standard high-level (higher than RTL), specification language
- Parameterized test-benches and scripts for simulation and verification of soft-IPs. The idea is to be able to verify correct functionality for all values of all parameters (very difficult problem).
- Scripts for re-configuring during reuse and for mapping to a design flow (e.g., paritioning)
- Tighter version control. It is interesting to note that because hardware wares out, hardware components are usually replaced not by a newer version, but an instance of the same. Care must be taken when replacing a component during maintenance with a compatible version.

2.2 IP DESIGN-WITH-REUSE

Systematic reuse only occurs when reusable assets are planned and created as a separate activity from application development. The achievement of design-for-commonality and control-of-variability requires the establishment of a reuse infrastructure, an overall framework that integrates the corresponding set of modeling, planning, and asset construction activities necessary for systematic reuse, and that, at the same time, allows the assimilation of technology effectively.

There are several issues that the IP integrator using reusable components will face. These include (1) design exploration support, (2) Standard IP design and acquisition process, (3) Specification of reusable IP ate the appropriate level of abstraction, and (4) System approach.

During the design exploration phase, designers need to know what design alternatives exist for the problem in hand, and, more importantly, which IP exist that can support these alternatives and what is the impact of (re)using one IP vs. another (assuming there could be several vendors).

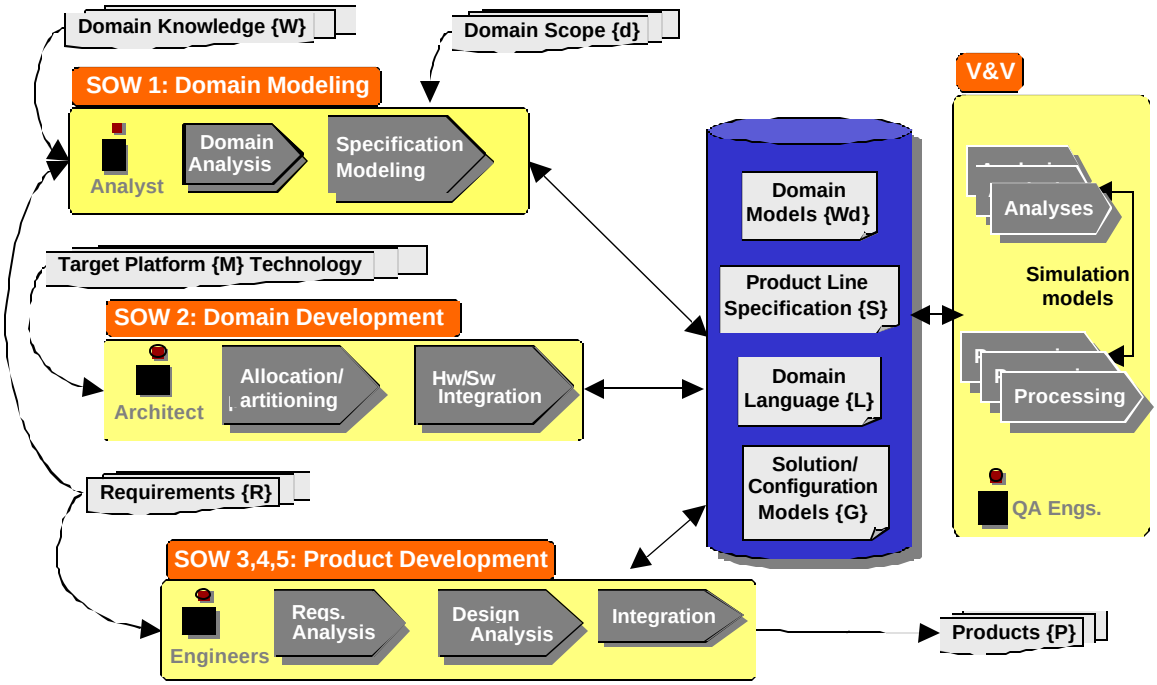


Figure 1: Yamacraw Embedded Systems Design Flow

A standard system design and IP acquisition process. There are three basic phases in the overall systematic reuse process: Domain Modeling, Domain development (i.e., Product Line Design and Implementation), and Product Development [17]. In Figure 1 below, we illustrate the interrelations between these three phases. We use the formalism introduced by Gunter et al [18] in their reference model for requirements engineering to label the arrows.

The primary information elements manipulated are in the domain knowledge {W}, which encompasses known facts about the domain environment or outside ***W***orld. The domain modeling activity is constrained by the chosen scope {d} thus sub-setting the domain knowledge to the specific set of products {Wd} based on strategic goals and organizational mission. The ***R***equirements {R} define needs from an end-user point of view. The specification {S} is produced as a precise description of {Wd} from where an optimal ***P***roduct {P} can be built. The target platform, or ***M***achine, {M} provides the specific computing environment(s) on which the delivered products and assets will execute.

The Product development process is multiplexed to correspond to the various product lines. Generative application development can be applied, whereby the application engineer states requirements in abstract terms, from where application generators produce the desired system or component. Figure 2 illustrates this concept of application-specific environments.

Code for IP detailed specification is not done, for the most part, with reuse in mind, nor is the design of test benches. Most system companies are using a the RTL language to write "macros" (another name given to a typical functional block). This is very inappropriate to achieve high level of IP reuse since it is difficult to specify variability (it is similar to specifying software components in assembly language).

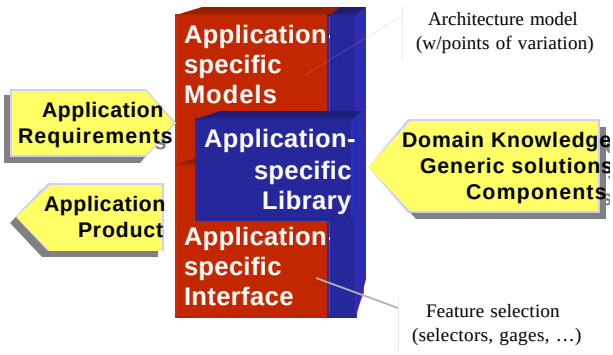


Figure 2: Application-specific environments

Furthermore, hardware must now be portable across different software platforms! (E.g., different operating systems and other system software). This suggests that

a complete system design approach is needed to achieve high level of systematic reuse.

3 IP AND SYSTEM SPECIFICATION

In general, we need modeling methods and tools ranging from system specification languages, architectural styles, frameworks, and architectural description languages, to component meta-models, patterns, and component description languages. All these models then serve as the basis for requirements and design analysis for a specific product, and they support creation of component libraries.

There are several levels of abstraction when specifying requirements for embedded systems; these go from the high-level system concept to the register transfer level. SoC's components for both hardware and software blocks are typically specified using different notations and modeling techniques. These notations include a wide variety of hardware description languages, continuous modeling languages, protocol specification languages, and dynamic synchronization languages, as well as software architecture description languages, and programming and module interconnection languages, together with a plethora of formal languages and mathematical notations.

A critical aspect is the ability to design "reusable" components from specification to silicon with the understanding of their need for adaptation when reused. For this, we must be able to capture the design information in a consistent and "standard" form, in such a way that engineers other than the original designers can integrate components. Components must not be designed for a single technology, since they must be "portable" and independent of specific simulators, capable of being independently verified in different environments (or chips). Finally, to aid in the validation and verification processes, specification languages must be either formal or executable (or both).

Previous attempts to address the multiple-notation and analyses problem have taken two avenues, a *co-simulation approach* based on a common backplane, and a *compositional approach* based on a single, "integrated" model. In the former the various analysis tools corresponding to the different notations are cognizant of the semantics of an underlying common backplane (see Figure 3). The basis for the second compositional approach, see Figure 4, is the translation of all different notations to a single "integrated" notation with its own analysis tool.

We propose a broad-spectrum integrating notation, YES-UML, to address the multiple-language, multiple-analyses problem of embedded systems design by combining levels of abstraction (vertical integration), and heterogeneous conceptual models (horizontal integration). The idea is to use a general modeling language, such as UML [19], and define mappings to concepts from existing notations supporting various modeling techniques. This mapping works two ways, namely at the front-end analysts deal directly with UML models, and at the back-end designers are able to use their conventional analyses tools which are "seamlessly" connected via UML implementation technology such as XMI [*20*]). Thus, YES-UML will be a series of extensions to UML to support existing modeling notations. This way, we reduce the semantic gap between system-level specifications and IP-level implementation decisions. It is worth mentioning that UML itself is an attempt to unification of several orthogonal elements such as data (classes), behavior

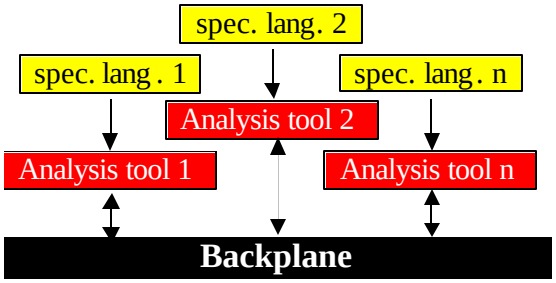


Figure 3: *Co-simulation Approach*

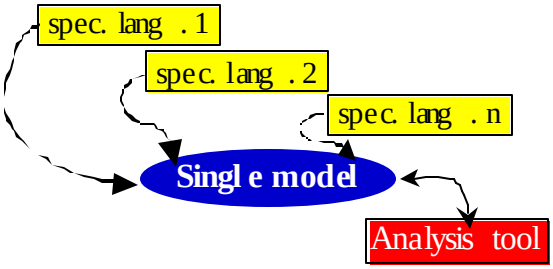


Figure 4: Compositional approach

(states), and execution flow (actions). See Figure 5.

4 CONCLUSIONS

In this paper we have argued establishing a causal relationship between models representing reusable assets, created by applying first principles, and the application of these models to actually develop the delivered application, by the routine application of solution models. We conjecture that the separation of models and product artifacts provides a suitable framework for integrating technology in the pursuit of systematic reuse.

An important aspect to secure institutionalization of systematic reuse is the attainment of "economies of scope." Organizations must invest in creating models of the common capabilities of related products, a process commonly known as domain engineering or *design-for-reuse*. Organizations use these models as IP assets supporting the creation of products that meet increasingly changing requirements, a process also

commonly known as product development or *design-with-reuse*.

A number of specialized notation, methods, and tools are being put forward for tackling the problem within the context of embedded systems. It is interesting to note that a number of UML extensions are being proposed to model real-time behavior. In addition, there is a move to create isomorphic (and semantic homomorphisms) mappings between system-level notations and UML [21].

Using these high-level notations together with architecture exploration tools will go a long way to achieve the elusive goal of IP reuse.

5. REFERENCES

[1] Chiang, S-y. *Foundries and the Dawn of an Open IP Era*. IEEE Computer, April 2001, pp 43-46.

[2] Tennenhouse, D. *Proactive Computing* Communications of the ACM, *Vol. 43, No. 5.*

[3] DeLine, R. *Avoiding Packaging Mismatch with Flexible packaging*. IEEE Trans. on Software Engineering, Vol. 27, No. 2, 2001:124-143.

[4] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns.* International Conference on Software Engineering, May, 1999.

[5] Kiczales, G., Lamping, J. Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J-M., and J. Irwin. *Aspect-Oriented Programming*. European Conference on Object-Oriented Programming, Finland. Springer-Verlag LNCS 1241. June 1997.

[6] Withey, J. *Investment Analysis of Software Assets for Product Lines*. CMU/SEI-96-TR-010. Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University. 1996.

[7] Bratthall, L. and P. Runeson. *Architecture Design Recovery of a Family of Embedded Software Systems*. In proceedings on TC2 First Working IFIP Conference on Software Architecture. P. Donohoe, Ed. Kluwer Academic Pub. 1999.

[8] Brownsword, and Paul Clements *A Case Study in Successful Product Line Management*. CMU/SEI-96-TR-016. (Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1996).

[9] Díaz-Herrera, J. L, Knauber, P. and G. Succi. *Issues and Models in Software Product Lines*. International Journal of Software Engineering and Knowledge Engineering, Vol. 10, no. 4.

[10] Cohen, S., Friedman, Martin, Solderitsch, and Webster. *Product Line Identification for ESC-Hanscom.* CMU/SEI-95-SR-024, Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University. 1995.

[11] Tully, J. *Design Reuse: Business Issues for Users. The silicon system, SoC viewpoint, Spring 2000, pp6-9. Design Reuse: Business issues for Users, March 2000, Gartner Research. www.gartner.com*

[12] Keating, M. and P. Bricaud Reuse Methodology Manual: for system-on-a-chip designs. Kluwer Academic Press, 1999.

[13] Thomas, T. *Tehnology fr IP Reuse and Portability.* IEEE Design & Test of Computers, Oct-Dec 1999, pp 7-13.

[14] Gajski, D.D.; Wu, A.C.-H.; Chaiyakul, V.; Mori, S.; Nukiyama, T.; Bricaud, P. *Essential Issues for IP Reuse.* IEEE Proceedings of the ASP-DAC 2000. pp 37-42.

[15] *Virtual Socket Interface Alliance*. http://www.vsi.org.

[16] M. Becker, and J. L. Díaz-Herrera. *Creating Domain-Specific Libraries: a Methodology and Design Guidelines*. IEEE International Conference in Software Reuse, Rio de Janeiro, Brazil, November 1-4, 1994.

[17] Díaz-Herrera, J. L. and V. Madissetti. *Embedded Systems Product Lines*. Software product lines, ICSE Workshop. Limerick, Ireland. June, 2000)

[18] Gunter, C. A., Gunter, E. L., Jackson, M. and P. Zave. *A Reference Model for Requirements and Specifications.* IEEE Software, May/June 2000 pp 37-43.

[19] OMG Unified Modeling Language Specification, Version 1.3 First Edition: March 2000 (http://www.omg.org/)

[20] OMG *XML* Metadata Interchange (XMI) Specification, Version 1.0, June2000 (http://www.omg.org/)

[21] Díaz-Herrera, J. L. *An Isomorphic Mapping for SpecC in UML*. OMER-2: Object-oriented

Modeling of Embedded RT-Systems, Herrsching am Ammersee, Germany, May 2001.

Modeling Software Product Lines with UML

Hassan Gomaa

Department of Information and Software Engineering
George Mason University
Fairfax, VA 22030-4444, USA

hgomaa@gmu.edu

1. Introduction

This position paper addresses how the Unified Modeling Language (UML) [Booch99, Rumbaugh99, Gomaa00a] may be used for modeling software product lines [DeBaud99] (also referred to as software product families and families of systems). The paper considers how the various views of the UML, the use case view, the static modeling view, and the dynamic modeling view, may be used for modeling software product lines. It also considers how the feature model, which has been used for modeling the common and variable requirements in families of systems, can be integrated with the UML. Other papers addressing this topic are [Atkinson00, Cohen98, Gomaa00b, Griss98, Jacobson97].

As an example of modeling software product lines, a factory automation product line is considered in this paper. The problem has been simplified to fit this paper. This product line encompasses three mutually exclusive kinds of factory systems, high volume low flexibility manufacturing systems, low volume highly flexible manufacturing systems, and factory monitoring systems.

2. Use Case Model for Families of Systems

The functional requirements of a system are defined in terms of use cases and actors [Jacobson97]. An actor is a user type. A use case describes the sequence of interactions between the actor and the system, considered as a black box.

For a single system, all use cases are required. When modeling a software product line, kernel use cases are those use cases required by all members of the family. Optional use cases are those use cases required by some but not all members of the family. Some use cases may be variant, that is different versions of the use case are required by different members of the family. The variant use cases are often mutually exclusive.

In the Factory Automation product line, the *View Workstation Status* use case is kernel, whereas the *Create/Update Process Plan* use case is optional. There are three mutually exclusive use cases to represent the three different kinds of factory systems, *Factory Monitoring, High Volume Manufacturing* and *Flexible Manufacturing.*

3. Feature Analysis

Feature analysis is an important aspect of domain analysis [Cohen98, Gomaa95, Griss98, Kang90]. In domain analysis, features are analyzed and categorized as kernel features (must be supported in all target systems), optional features (only required in some target systems), and prerequisite features (dependent upon other features). There may also be dependencies among features, such as mutually exclusive features. The emphasis in feature analysis is on the optional features, since the optional features differentiate one member of the family from the others. In modeling software product lines, features may be functional features (addressing software functional requirements), non-functional features (e.g., relating to security or performance), or parametric features (e.g., parameter whose value can be set differently in different members of the product line).

In the object-oriented analysis of single systems, use cases are used to determine the functional features of a system. They can also serve this purpose in families of systems. Griss [Griss98] has pointed out that the goal of the use case analysis is to get a good understanding of the functional requirements whereas the goal of feature analysis is to enable reuse. Use cases and features may be used to complement each other. In particular, use cases can be mapped to features based on their reuse properties.

Functional requirements that are required by all members of the family are packaged into a kernel feature. From a use case perspective, this means that the kernel use cases, which are required by all members of the family, constitute the kernel feature. Optional use cases, which are always used together, may also be packaged into an optional feature.

In the UML, use case relationships can also be specified. Thus, common functionality among several use cases can be split off into an abstract use case, which can then be included in other use cases. This dependency among use cases is analogous to feature dependency, where one feature may require another feature as a prerequisite. Careful usage of the use case dependencies can greatly help in feature analysis.

Thus in the factory automation domain, *High Volume Manufacturing* and *Flexible Manufacturing* are optional use cases. However, the common functionality among the two use cases is split off into an abstract use case called *Factory Production*. The abstract use case is used by the other two (now more concisely described) use cases (Fig. 1). Based on this, we can choose to have features corresponding to each of these use cases, where the *High Volume Manufacturing* and *Flexible Manufacturing* features both require the *Factory Production* feature. Furthermore, the *High Volume Manufacturing* and *Flexible Manufacturing* features are mutually exclusive.

Another form of use case relationship is the *extend* relationship, where one use case may extend another when certain conditions hold. An extension relationship can also be mapped to a feature dependency, where one optional feature may depend on the other; thus some members of the product line may have both features whereas others only have the base use case. In the example, the *Flexible Manufacturing with Storage* use case can be used to extend the *Flexible Manufacturing* use case (Fig. 1). Thus, introducing intermediate part storage in the factory is an extension of the case where no intermediate part storage is provided. As before, each use case is mapped to a feature where the *Flexible Manufacturing with Storage* feature requires the *Flexible Manufacturing* feature. Some factory systems will need both features while others will only need the *Flexible Manufacturing* feature.

4. Static Model for Families of Systems

A static model for an application domain has kernel classes, which are used by all members of the family, and optional classes that are used by some but not all members of the family of systems. Variants of a class, which are used by different members of the family of systems, can be modeled using a generalization / specialization hierarchy. UML stereotypes are used to allow new modeling elements, tailored to the modeler's problem, which are based on existing modeling elements [Booch99, Rumbaugh99]. Thus, the stereotypes <<kernel>>, <<optional>>, and <<variant>> are used to distinguish between kernel, optional, and variant classes.

An example of a generalization/specialization hierarchy is given in Fig. 2, in which the factory workstation controller class is specialized to support three variants, the *Flexible Workstation Controller*, the *High Volume Workstation Controller*, and the *Monitoring Workstation Controller*. The *High Volume Workstation Controller* is specialized to support a further three variants, the *Line Workstation Controller, Receiving Workstation Controller*, and *Shipping Workstation Controller* classes as shown in Fig. 2.

5. Collaboration Model for Families of Systems

As with single systems, the collaboration model is used to depict the objects that participate in each use case, and the sequence of messages passed between them [Booch99, Gomaa00a]. In families of systems,

the collaboration model is developed for each use case, kernel or optional. Once the use cases have been determined and categorized as kernel, optional, or variant, the collaboration diagrams can be developed. For each feature, the objects that are needed to support the feature are determined and depicted on a feature based collaboration diagram. With this UML based approach, the objects are determined from the use cases. It should be noted that on the feature based collaboration diagram, the message sequence numbering, which is shown on individual use case based collaborations, is usually not shown.
This view is very important as it is used to determine how the objects interact with each other to support a given use case. The interconnected objects in a collaboration diagram supporting one use case depend on (and hence communicate with) objects supporting a prerequisite use case. Thus the objects in the *High Volume Manufacturing* and *Flexible Manufacturing* features can depend on and communicate with objects supporting the prerequisite *Production Management* feature and the inherently prerequisite kernel feature. Objects in the

6. Example of UML Description for a Software Product Line

In the factory automation product line, the four use cases that deal with generating, storing, and retrieving factory workstation and alarm status are kernel. Hence they are grouped into a kernel feature. Each of these use cases is further refined to depict the objects that are required to participate in executing the use case and the sequence of messages passed between them. The objects that participate in these kernel use cases are allocated to the kernel feature and depicted on one feature based collaboration diagram. Thus Figure 3 shows the three objects for this feature, two servers, the *Workstation Status Server* and the *Alarm Handling Server*, which store workstation status and alarms respectively, and one client, the *Operator Interface*, which views workstation status and alarms.

The *Factory Monitoring* feature depends on the kernel feature. This feature supports one class, the *Monitoring Workstation Controller* (Fig. 2), of which there are multiple instances in a system. Combined together, the two features form a *Factory Monitoring* system (Fig. 3), which is one of the members of the software product line. This figure shows a collaboration diagram for the *Factory Monitoring* system, consisting of the objects from the two features. On the figure, the lower three objects are from the kernel feature and the *Monitoring Workstation Controller* is from the optional feature. The three kernel objects are also used in other factory systems including the *High Volume Manufacturing* system, another member of the software product line. In this system, the instances of the *Line Workstation Controller, Receiving Workstation Controller*, and *Shipping Workstation Controller* classes (Fig. 2), which support the *High Volume Manufacturing* feature, all send alarms to the *Alarm Handling Server* and status messages to the *Workstation Status Server*.

7. Conclusions

This paper has described how the Unified Modeling Language (UML) may be used for modeling software product lines. In particular, it has described how the use case view, the static modeling view, and the dynamic modeling view may be used for modeling software product lines. The paper has also described how the feature model, which is used for modeling the common and variable requirements in families of systems, can be integrated with the UML. Thus a member of the software product line is described in terms of features and its architecture is described in terms of feature based object collaborations. Although each member of the product line can be described by a feature based collaboration diagram, it is not possible to describe the architecture of the product line in this way. This is because the product line architecture includes provision for mutually exclusive features and hence feature based collaborations that cannot be combined in one system. The product line architecture is described instead using a static model and depicted on a class diagram.

8. References

[Atkinson00] C. Atkinson, J. Bayer, D. Muthig, "Component-Based Product Line Development: the KOBRA Approach", Fraunhofer Institute Report, 2000.

[Booch99] G. Booch, J. Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide", Addison Wesley, Reading MA, 1999.
[Cohen98] S. Cohen and L. Northrop, "Object-Oriented Technology and Domain Analysis", Proc. International Conference on Software Reuse, Victoria, June 1998.
[DeBaud99] J.M. DeBaud and K. Schmid, "A Systematic Approach to Derive the Scope of Software Product Lines", *Proc. IEEE Intl. Conf. Soft. Eng.*, LA, May, 1999.
[Gomaa95] Gomaa H, "Reusable Software Requirements and Architectures for Families of Systems", Journal of Systems and Software, May 1995.
[Gomaa00a] H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison Wesley, Reading MA, 2000.
[Gomaa00b] H. Gomaa, "Object Oriented Analysis and Modeling for Families of Systems with the UML", Proc. IEEE International Conference on Software Reuse, Vienna, Austria, June 2000.
[Griss98] M. Griss, J. Favaro, M. D'Alessandro, "Integrating Feature Modeling with the RSEB", Proc. International Conference on Software Reuse, Victoria, June 1998.
[Jacobson97] I. Jacobson, M. Griss, P. Jonsson, "Software Reuse - Architecture, Process and Organization for Business Success", Addison Wesley, 1997.
[Kang 90] Kang K. C. et. al., "Feature-Oriented Domain Analysis," Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.
[Rumbaugh99] J. Rumbaugh, G. Booch, I. Jacobson, "The Unified Modeling Language Reference Manual," Addison Wesley, Reading MA, 1999.

Fig. 1: Use Case Dependencies and Feature Dependencies

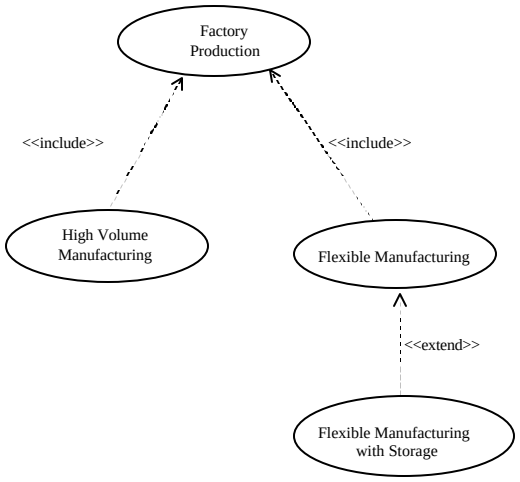


Fig. 2: Generalization / Specialization Hierarchy for Workstation Controller

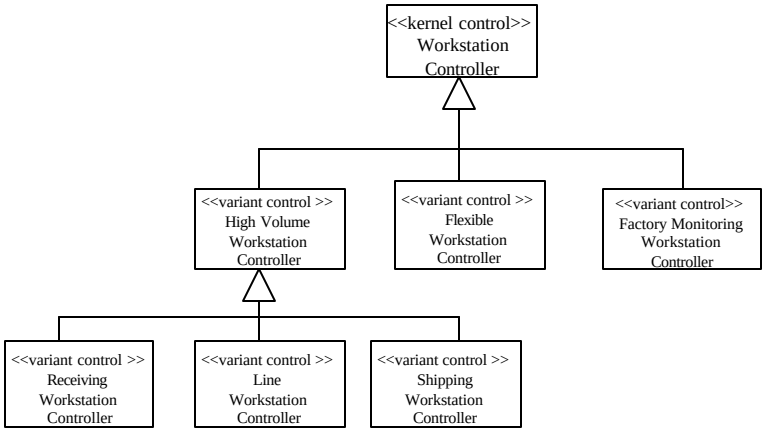
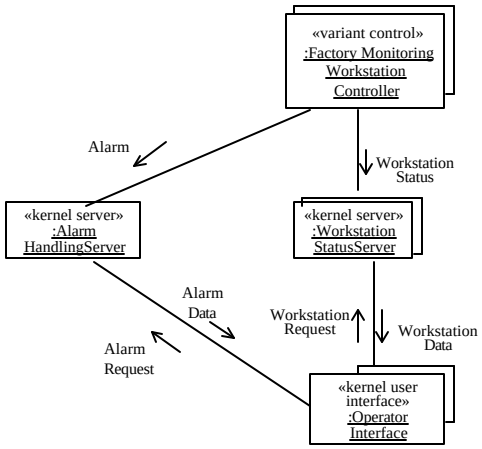


Figure 3 Collaboration Diagram for Factory Monitoring System



A Proposal for a Product Line Product Derivation Process

John MacGregor
Robert Bosch GmbH
john.macgregor@bosch.com

Abstract

The construction of products out of product line assets is often depicted as following a classic product development cycle starting with a complete requirements analysis and continuing with complete design, coding and testing phases.

This paper makes the assumption that in domain analysis, the domain problematic has been sufficiently characterized so as to render repetition of the analysis, design, and even the coding phases superfluous for the production of a large part of the products in a product line. The process resulting from the resolute application of this assumption follows practically the opposite course to the aforementioned classic cycle.

In the context of the development of embedded systems, where there can be hundreds, even thousands, of actual and potential product variants, the effort necessary to reach this degree of characterization is warranted. The proposed approach minimizes the time and effort spent in production through minimizing the search for appropriate assets and assessing their suitability.

1. Introduction

1.1. Background

The Product Line Approach divides the development of software products into two processes: domain engineering and application engineering.

In domain engineering, the domain or domains relevant to the product line are characterized and the commonality and variability in the affiliated products are abstracted to produce generic domain feature or requirements models, a platform architecture and various ancillary models. These models are used in turn to produce the core assets, which include, among other things, requirements templates, generic software components, test suites and documentation.

The intention is that these core assets be (re)used to accelerate and shorten the development or creation of all the products used in the lifecycle of a software product, including, naturally, the product itself. In order to accomplish this as efficiently as possible, another part of the domain engineering process is dedicated to tracing and documenting relationships between related assets so that the assets appropriate to the task of building a specific product can be quickly identified, understood and used.

Note that the product line models separate commonalities and variabilities. Variability with respect to a particular aspect does not mean that each product is unique with respect to that aspect or that the variability cannot be characterized in an algorithm. A third task of domain engineering, other than the construction of the assets and their traceability, is the grouping of variant whose variability is the same or similar. This grouping is then expressed as parameters with respect to the individual variation points in the feature models, architecture models, etc.

Ideally, the developer should be able to identify the points of modification / development for a new product by comparing the requirements for the product with the feature or requirements models. When the (new) product requirements correspond to the variability foreseen in the domain models, then the developer must only realize the parameters with the methods foreseen by the domain analysis. (See the “Parameterization” section which follows)

These core assets should ostensibly be stored in an asset repository, which provides a facility for storing and retrieving the appropriate assets.

After this point in the definition, product line theory becomes fuzzy as there are few descriptions of what specifically should happen, and up until now, no general agreement about how to proceed further. This being said,

it should also be recognized that there is a wide spectrum of software products that can be viewed as product lines and an even wider spectrum of organizations that develop them. It is extremely difficult to characterize, let alone generalize, the situation within which a software developing organization would find itself when developing a product line product.

It is nonetheless somewhat amazing that it is not recognized that considerations related to the product construction process also play a role in defining the structures required in the asset repository, and ergo, the steps necessary in domain engineering to provide these structures.

This paper stems from work in the ESAPS Project [ESAPS 1999] related to producing a requirements specification for an Asset Repository [Mergel et al. 1999]. In order to verify the specification, we needed to understand how the repository would be used, which led to the creation of a process description for application engineering.

Although the intent is neither to specify application engineering definitively nor to hinder alternative views of the subject, this paper provides perhaps an illustrative aid to understanding the concepts underlying the product construction process and how it relates to asset management.

1.2. Context

At the moment, the boundaries between the asset repository and its environment are ill-defined. The repository stores and restores objects for every other entity in the product line and that in different ways with respect to use and reuse. It also plays a significant role in all product line processes.

There is a complex relationship between the IDE's (Integrated Development Environments), the configuration management system and the asset repository however, which makes considerations related to storage and retrieval considerably more complex. For example, how are assets used to produce a concrete project? Are the assets left in the repository (viewed) or are they removed (i.e. brought into an independent IDE project)? Are they put back (i.e. registered in the asset repository as being part of the specific product for historical purposes) – or checked into the configuration management system to allow further modification access (and logically removed from the asset repository)? The assets are also part of the domain process, and as such could be modified or improved based on the experiences obtained in application development. That is, not only the product produced but also the individual assets would also be subject to change and configuration management.

2. Constructing Products

In concord with the purpose of this document, the focus of the descriptions is asset management. [MacGregor 2000] describes an overall system whose focus is probably quite atypical of product line systems. However, it does detail the interactions between application and domain processes as well as between the productive and management processes, and the description is in these aspects probably quite representative of product line processes. It therefore provides a process framework within which these scenarios can be understood.

When considering these scenarios it is helpful to keep the fact in mind that there is a spectrum of software product types that comprise product lines. Again, the intent of this section is not to provide a definitive description of all possible product lines, rather it represents a first step in understanding the problem area and hopefully covers a representative cross-section of the underlying spectrum of asset repository usage.

Manufacturers of embedded systems belong to a breed of companies whose products belong in a (relatively) stable market and in a stable, well-understood technical environment. Furthermore, embedded system manufacturers produce both a large number of instances of a product and a large number of variants of a product type. These large numbers bring both economies of scale and economies of scope which both allow and dictate a better understanding of product variance and justify the effort in parameterising general product software and in automating variant production to cover a large portion of product variants.

At the other end of the spectrum, there are companies who find themselves in a fast-changing market and/or technical environment where the dynamics are not necessarily well understood. They may neither produce a large number of instances of a product nor many product variants. Here, the effort to automate the development of individual product software instances may not be justified.

Both types of companies, and all companies coming between the two extremes have good reasons to build product lines and therefore asset repositories, however. The way in which the two types of companies deal with the asset repository could differ. This stems from the assumption that the companies that parameterize ("parameterizers") do not always modify assets to produce new products, while the others ("modifiers") are always forced to modify assets.

2.1. Schema

The following presents an idealized product line member development schema in order to establish a pattern to be used in the process description.

Parameterization

Where parameterization occurs, there is minimal interaction with the asset repository. Parameterization encompasses all actions taken to configure and assemble the assets to produce a viable product. These include, among other things, passing values to runnable modules (or setting environment, or "shell" variables), assigning values to pre-processor variables, instantiating abstract classes, setting compiler flags or otherwise influencing compile behavior.

The product's requirements were foreseen in the domain requirements model so that the modifications have no effect on the assets themselves. Essentially, all modifications are isolated and specific to the product being developed. Should the product require maintenance, it can be regenerated based on the parameter values. These values therefore represent the minimum, but also sufficient, amount of information that must be stored in the configuration management system for maintenance.

The construction task therefore becomes (merely, with respect to the asset repository) to identify how the requirements of the new product correspond to the parameters in the requirements model and to build the product.

Code Modification

In the case where code must be modified, the product requirements are also compared to the domain requirements model, but the parameter values foreseen are not appropriate. The question now becomes "What must be modified" under the sensible assumption that both the amount of code to be modified and the amount of investigation be held to a minimum.

The minimum amount of investigative effort would be to examine the design in light of the requirements to establish what parts of the code must be changed and how they must be changed. This means that the changes are local changes with impacts only at the component/subsystem level. This scenario ends when the modifications can be scoped through examination of the domain design.

For completeness sake, it should be added that the code would then be modified, and that possibly new parameters or new parameter values would be set. The domain requirements model and domain code assets would also be updated. All code modifications, new parameters, test modifications, and so on that are unique to this product must be stored in the configuration management system under the same philosophy as under parameterization, above.

Design Modification

This case follows inductively from the Code Modification case. Given that the requirements are outside of the scope of the domain design, then the architecture must be examined in order to scope the changes in the design. Assuming that the architecture must not be changed, that is, no fundamental changes to the infrastructure of the architecture, changes can be made in the component design and the new design documented. The rest follows as in the Code Modification case.

Architecture Modification

When the requirements of the new product dictate that the architecture be changed, it is high time to consider making a one-off product and separating it from the product family.

Assuming that the architectural modifications remain relevant to the product family, then the changes are made, documented and filtered down through the design, code and parameterization.

2.2. Discussion

Figure 2 (next page) presents the process underlying the schema graphically.

Note that the basic assumption underlying this model is that superfluous effort is avoided. Considering that the domain assets have already (hopefully) been engineered for reuse, it is a waste to use other assets (higher up in the hierarchy) when they are sufficient. Product construction approaches (references) other than the one presented here seem to go top-down. That is, a product architecture is derived first from the domain architecture and then the rest of the development process follows according to waterfall principles. The characterization presented here contrasts in that the process is a backtracking process.

This schema represents a significant departure from the classic "six pack" [SEI-STR 1997] [Foreman 96] [ESAPS 1999] in a number of

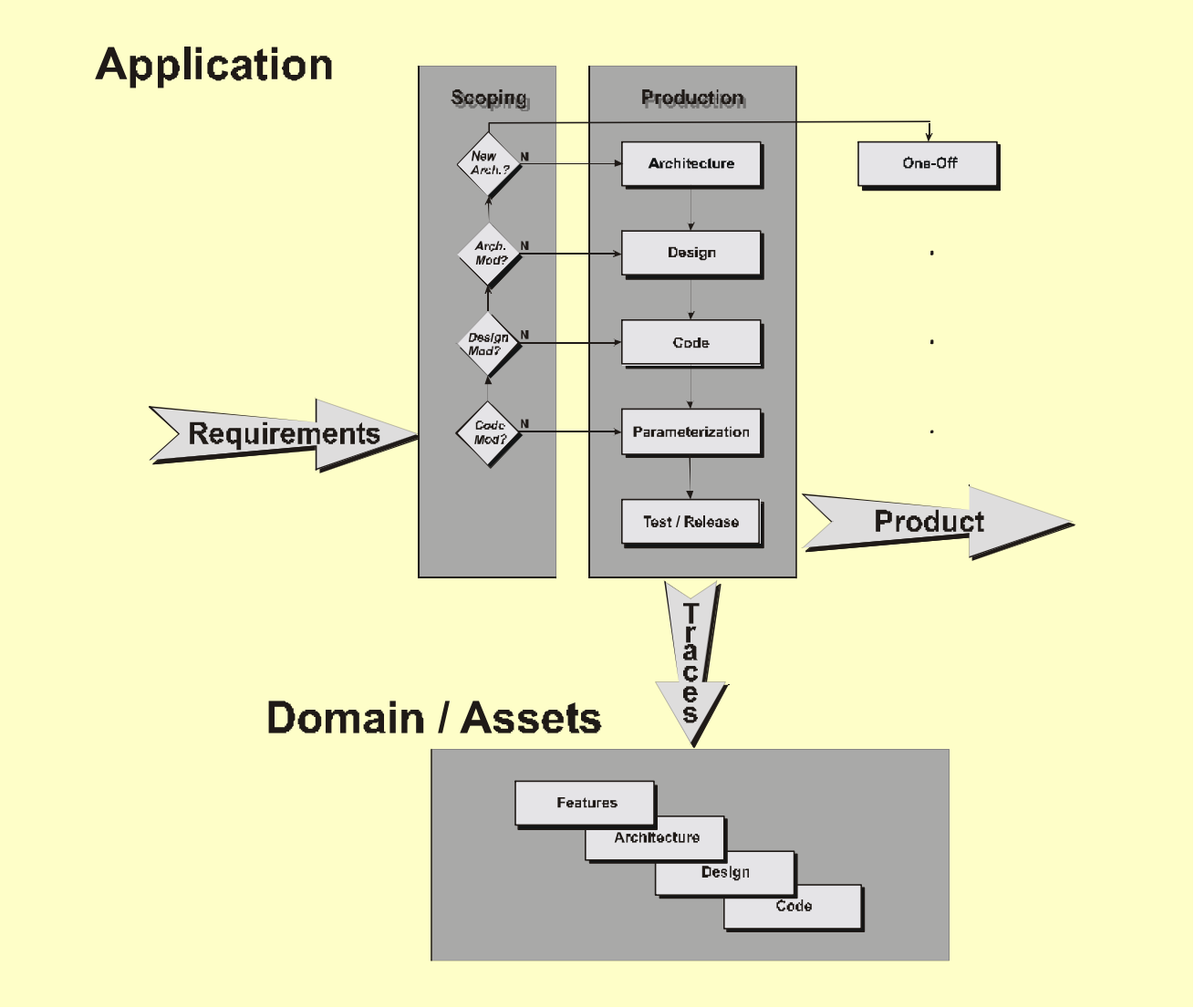


Figure 2: Proposed product derivation process

respects.

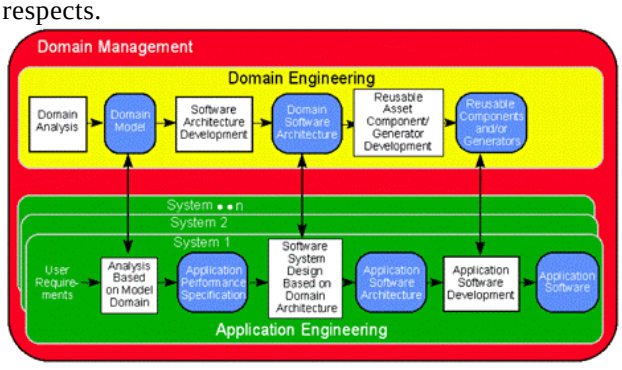


Figure 1: The "Sixpack" Model [SEI-STR 1997]

As first discussed in [MacGregor 2000], this traditional representation ignores time aspects. It is a tautology that a component must first be built before it can be used/reused. This means that, at the time that products are built from the asset repository, the analysis, design, code, test and release phases of the domain process are already finished, at least for a part of the complete product family scope. From this point on, either new components will be added to the repository to cover new functionality, or the existing components will be modified. In other words, when the application process is in production, the domain process is in maintenance.

This brings a couple of new aspects of the asset repository to light. Narrowly defined, the asset repository is the repository for the tailorable assets that represent the total variability spectrum within the product family. That is, the asset repository contains the domain assets. Assuming that there is a logical boundary

to the configuration management system, it makes no sense that the repository, *per se*, contain either older versions of the assets or archive information about products already produced. These are issues better left to the configuration management system.

In a certain sense, after the assets, the components of the product family, have been produced, the active domain engineering process disappears and its tangible assets are rolled into the asset repository. For the illustrative purposes of Figure 2, it therefore does not matter whether the bottom half of the diagram is labelled "Domain" or "Asset Repository".

Over and above that, all product family products, past, present or future, can arguably be derived from the domain assets. It can only be assumed that in the course of time the assets will improve with maintenance. When maintenance modifications have not explicitly precluded the reconstruction of historical products, it can only be hat equally good or better versions of historical products can be produced from the latest versions of the assets.

Consider also the following:

- Although the possibility of building a one-off product is only mentioned in the last section of the schema, the possibility should always be considered. There is a fundamental trade-off between increased generality to cover new requirements in subsequent family products and the increased complexity and maintenance effort required to incorporate the changes.

  The development of one-off products naturally has no effect on the asset repository. It foregoes the immediate benefits of the product line, however, in that development resources are redirected away from the product line and that the assets are at least partially not reused. Long-term, it represents a separate product version branch that must be maintained separately and in competition to the product line.
- This schema also makes the implicit assumption that all product development that results in modifications to the code, design or architecture and which do not result in a one-off product, will undertake modifications to the assets and then generate the product. That is, the assets will be updated immediately, and that the modifications do not occur as part of asset maintenance. It is also possible for expediency to produce the product and then incorporate the modifications in the assets at a later point. This duplicates effort unnecessarily and opens the process to unintended sources of variability or errors through two sets of modifications.
- It can be seen that the seemingly large difference between the modifiers and the parameterizers is actually not so big with respect to modification problems. In our (limited) experience with other companies, it seems that companies that do not produce products through parameterization still must perform an investigation to scope the work for a new product. It can be that they produce the product as a delta from the previous product. They have nonetheless analysed their domain with respect to commonalities and variability and would search along the same lines through the code, design and architecture to establish the minimum amount of modifications necessary.
- Although only the effects of changing or new (customer) product requirements were discussed and this aspect is true for both types of companies, there may be incentives to change due to product-technology evolution or organizational technology evolution; case tools, for example which might also affect product parameterization, code, design or architecture.

3. Conclusions

This paper has investigated the ramifications of the assumption that, after domain engineering has fully analyzed the product line domains, produced the core assets and the traceability relationships between them, the application engineering process should minimise the effort in identifying the appropriate components and assembling them together to produce a new product. In doing so, the aspects related to asset management have been considered.

It has reached the following conclusions:

- Although there may be differences among software developing organizations with respect to the economy of developing assets for reuse, the types of reuse they have is shared, insofar as the economies of their situation permit.
- The application derivation process itself (insofar as the current requirements were not foreseen in the domain analysis) consists of backtracking through the products of the

domain engineering process to find the minimum level of modification that meets the requirements at hand.

- Theoretically, there should be no difference between product construction and asset maintenance, so long as it makes sense to be able to generate product line members that have already been instantiated.
- If all product line products, actual or planned, can be built from the platform in the asset repository, the problem of asset repository versioning becomes less significant, when not irrelevant.
- At the time that products are being constructed, there is no domain engineering process, although a domain maintenance process could exist.

4. Outlook

This schema is predicated on the assumption that the product developer can easily assess the impacts of the requirements for a new product by examining the feature model whereas the prevailing methodologies tend to start with an examination of the domain commonalities and variabilities at the software artifact level.

We are currently investigating the application of artificial intelligence configuration approaches to allow tracing between the feature level and the artifact level.

5. References

[ESAPS 1999] "ITEA-ESAPS Full Project Proposal; European ESAPS Consortium", June 1999.

[Foreman 96] Foreman, John. "Product Line Based Software Development- Significant Results, Future Challenges." Software Technology Conference, Salt Lake City, UT, April 23, 1996

[MacGregor 2000] J. MacGregor: "A Product Line Process for the Production of Platform Software at Bosch"; Proceedings of Software Product Lines: Economics, Architectures, and Implications, P. Knauber and G. Succi (Eds.), workshop in conjunction with 22nd International Conference on Software Engineering (ICSE), Limerick, 2000, Fraunhofer IESE TR 070.00/E

[Mergel et al. 1999] M. Mergel, S. Thiel, S. Ferber: "Product Line Asset Classification and Dependency Specification"; Consortium-Wide ESAPS Deliverable BOSCH-WP3-T3.3-01, 30.6.2000.

[SEI 1999] P. Clements, L. Northrop: "A Framework for Software Product Line Practice Version 2.0"; Software Engineering Institute, Carnegie Mellon University, July 1999

[SEI-STR 1997] "SEI Software Technology Review**"** (Online); Software Engineering Institute, Carnegie Mellon University, 1997. URL: [http://www.sei.cmu.edu/str/descriptions/]

An Incremental Transition Strategy is Key to a Successful Introduction of Product Line Engineering

Dirk Muthig

Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
Dirk.Muthig@iese.fhg.de

ABSTRACT

The motivation for almost every technology transfer project is an organization's problem or goal. Software reuse is a discipline that is a promising candidate for improvement activities in case of many problems software organizations typically have. Nevertheless software reuse - and also product line engineering as one orientation towards reuse - is only one of many fields in which an organization can improve. Consequently, product line engineering must compete before any transfer project with alternative approaches. Unfortunately, most of the existing product line engineering methods ignore this competition so that data is missing that helps arguing for product line engineering in this competition.

In this paper we describe how strategies for transitioning towards product line engineering can be defined. We think that their clear definitions in conjunction with accompanying measurement programs are the basis for monitoring the transfer projects themselves, as well as sharing comparable data on product line transfer in the research community.

1 INTRODUCTION

The initial motivation for almost every technology transfer project is an organization's problem or goal (the distinction between problem and goal depends on whether you are an optimist or a pessimist). The technology to be transferred is supposed to help solving the problem and achieving the organization's goal. As usual, there are several, alternative ways to do so, that is, improvements in different fields or disciplines can eventually achieve identical goals. If there is a best way at all, it surely depends at least on diverse characteristics of an organization, its current situation, the concrete problem, and the stated goal.

Software reuse is a discipline that is a promising candidate for improvement activities in case of many problems software organizations typically have these days. Nevertheless software reuse is only one of many fields in which an organization can improve. Product line engineering again is only one of many possible orientations towards software reuse. Consequently, product line engineering must compete before any transfer project with alternative reuse approaches and also with approaches from other fields, for example, focusing on processes, tools, or planning. Unfortunately, most of the product line engineering methods ignore this competition by describing only the final, ideal state of a product line organization. They typically do not provide guidance on how an organization can or should transition into this final state in a systematic and controlled way.

Without this guidance, the introduction of product line engineering into an organization is more a risky adventure than a controlled improvement project. Additionally, only a few experience reports have been published on (successful) transfer projects towards the ideal product line engineering state. None of the available reports provides concrete and detailed data, if there is data at all, on the particular organization, context, investments, and achieved improvements. Already in 1993, Arrango called researcher for addressing serious empirical validation of methods and models, as well as the publication of project data on domain analysis projects and their results [1]. Since then the situation has not changed significantly, for instance, hardly any data can be found in the proceedings of last year's first international conference on software product lines [2].

In order to learn more about the impacting factors on product line transfer projects, more data and experience must be published and shared among research groups and software organizations. Therefore, measurement programs must accompany transfer projects to systematically collect the data necessary to check for the particular project whether the stated goal has been achieved, but also to make data collected in different projects comparable in the research community. The set-up of a measurement program requires detailed information on the organization, its practices, the planned changes, and the anticipated effects. The implementation of all changes at once would be a very radical change to the organization and its practices. The degree of novelty would be very high and potential innovations would typically be perceived as chaos. Therefore, the installation of the planned changes must be broken into smaller pieces. The pieces are then successively introduced into the organization and monitored by the measurement program. Each step can be validated and thus the overall transition

can be systematically controlled. This process is compatible to the generally accepted quality improvement paradigm (QIP) [3].

The plan and the documentation of the series of transfer steps are an incremental transition strategy – it is also customized when the specifics of a target organization are taken into account.

In this paper, we argue that an incremental and customized transition strategy is key for a successful introduction of product line engineering into an organization. On one hand, it reduces the risk and uncertainty for the target organization. On the other hand it enables researchers to learn more systematically from project to project.

In the next section, we return to the roots of product line engineering and motivate it as one orientation towards reuse. From this viewpoint, we then derive possible, incremental transition strategies for projects transferring product line engineering into an organization.

2 SOFTWARE REUSE

In this section, we return to the roots of product line engineering, which is one orientation towards software reuse. We start with the description of a basic reuse model and its problems to show how product line engineering tries to solve them.

2.1 Basic Reuse Model

The general model of a basic reuse is visualized by Figure 1. The output of an application engineering project is, of course, an application but additionally all assets the application consists of. These assets are stored in an asset base with the intent to reuse them during future application engineering projects. In these future projects, it is tried to develop parts of the application reusing assets existing in the asset base. As many organizations experienced, the pure basic approach does not achieve the expected improvements because it does not solve the following problems:

- Documentation: Each asset put into the asset base must be documented to enable its reuse. The problem is which information in which degree of detail is needed by the reuser.

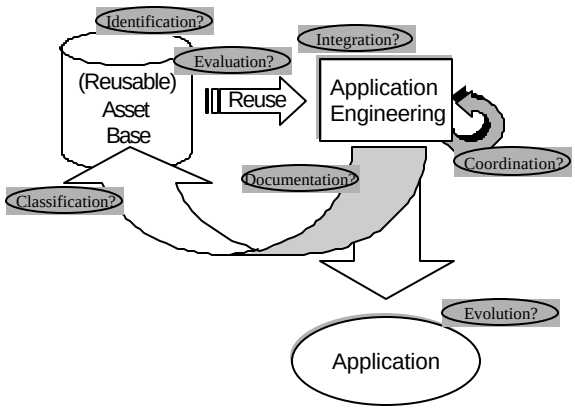


Figure 1: Basic Reuse Model

- Classification: Besides the information needed by the reuser, all assets and the documented information must be structured according to a common classification scheme. The classification scheme is the key for accessing the asset base, and must, therefore, be defined in a way that supports the reuse processes the best.
- Identification: As Tracz [4] said, there must be reusable software before you can reuse software. Consequently, during the application engineering process an engineer must know about assets, which are reusable in a given context. To be independent of the personal knowledge of the asset base, an engineer must be enabled to identify reuse candidates, for example, via the classification scheme. Unfortunately, classification schemes have proven to work only in a particular set of reuse contexts. Additionally, only a small, manageable set of good candidates should be identified. The reason is that reuse only when the reuser experiences directly that reuse saves time and effort.
- Evaluation: When a set of reuse candidates has been identified, the candidates must be evaluated with respect to adaptation and/or development effort. In order to make a confident decision concerning the selection of a reuse candidate or the development of the needed component without reuse, a systematic and repeatable evaluation approach is required.
- Coordination: When an asset has been selected to be reused, it will be adapted to match the context in which it will be reused. An organization must - to benefit from reuse as whole - coordinate concurrent application engineering projects to avoid identical adaptations within different projects. Therefore, beside organizational issues, the reuse contexts of projects must be comparable.
- Integration: Reused assets must be integrated with the application under development. Thereby, it is necessary to remove conflicting assumptions about the environment and architectural mismatches. Unfortunately, an asset's assumptions about its environment are often implicit and therefore neither documented nor visible.
- Evolution: When an asset has been reused while building several systems, what is the rule rather than the exception for successful reuse programs, the maintenance of the asset becomes more complex than for assets only used in a single application. Then, maintenance of several application must be coordinated to share as much of the maintenance effort as possible, and to improve the reusable asset base.

2.2 Domain Engineering

Domain engineering has its way to address all problems of the basic reuse model and thus to enable software reuse. Domain engineering splits the overall life cycle into

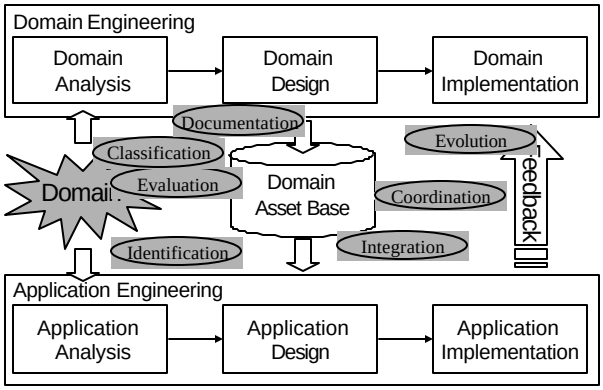


Figure 2: Domain Engineering Life Cycle

two main phases: domain engineering (i.e., development of assets reuseable in the domain) and application engineering (i.e., development of applications in the domain with reuse of domain assets). The life cycle of domain engineering is shown in figure 2.

Domain engineering uses the real-world application domain as the means of tackling the reuse problems described above.

- Documentation: The entities and their relationships of the domain are used to document the reusable assets. For example, a component supports a specific set of activities that are meaningful in the domain.
- Classification: Assets are classified according to the structure of the domain, which is identified by domain experts during domain analysis activities.
- Identification: The reuser knows the reuse context, which is usually defined by a problem in the domain that must be solved. Then and because the assets are classified according to domain abstractions, reusable assets can be identified by searching the asset base for solutions of the domain problem under consideration.
- Evaluation: Reuse candidates can be evaluated also using the domain abstractions. Differences are expressed in domain variabilities and the distance between the candidates' and the required domain context is a measure for estimating the reuse effort.
- Coordination: The split of the development life cycle enforces feedback of the application engineering projects to the domain engineering activities. This is also the channel where similar reuse or change requests can be coordinated. Again, the domain abstractions are the key for comparing them and planning their realizations.
- Integration: Domain engineering also defines a common reference architecture, the domain design, for systems in the domain. This simplifies the integration because the assets are built reusable for this architecture and implicit assumptions concerning the domain, which is the same for all reusable assets.
- Evolution: Analogously to the coordination of application engineering projects, domain engineering channels maintenance requests concerning reusable assets. Maintenance of reusable assets is mainly done as part of domain engineering and, therefore, maintenance effort is coordinated and minimized.

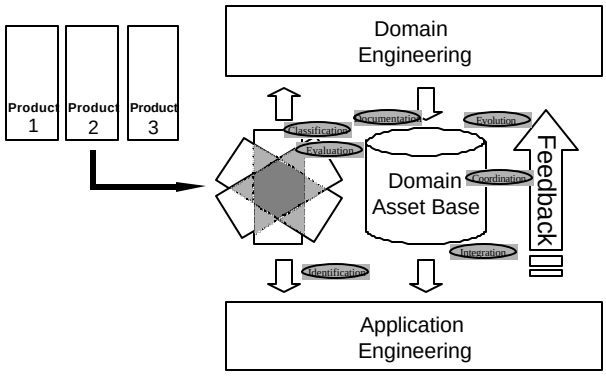


Figure 3: Reuse within product line engineering

2.3 Product Line Engineering

Practical experience with domain engineering has shown that the definition of the domain under consideration is problematic. When the domain has been chosen too small, the domain model and the reference architecture do not address important issues. In this case, significant changes will be necessary in future projects and thus the whole domain engineering effort is questioned. When the considered domain is too big, the effort invested in domain engineering is higher than really necessary, which will increase the already big investment. Product line engineering solves this scoping problem by analyzing a set of concrete (existing, planned, or future) products and defining the domain according to these products' characteristics. Everything required by one of these products is part of the domain - everything else is outside. The concept of defining the domain through a set of products is visualized by figure 3.

3 TRANSITION STRATEGY

In our context, a transition strategy describes the steps planned to be taken while moving an organization from single system development to product line engineering. The final goal of a product line transfer project is schematically shown in figure 3.

Before incremental aspects of transition strategies towards product line engineering are discussed, the necessary key conceptual change is emphasized. The only prerequisite for the product line life cycle is an existing abstract understanding of an organization's product line. To view the systems as a whole and define them through common and variable characteristics is indeed central to every activity in the product line life cycle. Domain analysis is requirements engineering but for a set of products, the construction of a reference architecture is architecting but for a set of products, and so on.

Unfortunately, a product line view requires a shift to the way many people in an organization think. Product

line engineering makes domain knowledge and the domain asset base to the central elements of the whole life cycle. Experience shows that the way people think cannot be changed very easily. Hence, each transition step towards product line engineering must also focus on establishing the product line way of thinking in an organization. This can be done best by clearly separating aspects of single system development from aspects of product line development. That is, before introducing product line aspects, a reduced variant of the product line process is introduced and used for developing and documenting a single system.

3.1 Customization

A product line method, in general, must be customized to the target organization. This includes the definition of the performed processes, as well as the used products. In both cases, changes should be minimized to reduce the degree of novelty within a transfer project. The established practices and the structure of an organization must be reflected in the processes. The key abstractions and entities of application domains can be captured best by different model sets and thus impact the selection of used products.

The customization of products and processes is typically related with the final, ideal state of a product line engineering method. For example in [5], the customization of our PuLSE method [6] is described. Hence, through customization the final goal of a product line transfer project for a particular organization is defined., which is the input for defining a transition strategy. Customization, therefore, defines what must be transferred but thus only indirectly impacts the transition strategy.

3.2 Increments

The increments for introducing product line engineering are in a three-dimensional space spanned by life cycle stages, orientation, and subdomains.

Life cycle stages, such as requirements engineering or implementation, are vertical columns in the product line cycle. Each increment must partially cover both, domain engineering and application engineering, because only in such cases an organization can visibly benefit from an increment.

Whether an increment is focused more on internal improvements (e.g., improving the coordination of developments during domain engineering) or on external improvements (.i.e., customer interaction during application engineering) is defined by its *orientation*. An example for an externally oriented requirements engineering stage is the construction of generic business process, which are discussed and instantiated during application engineering with the customer. An internal orientation is the integration of requirements on different systems to get an asset base that is a better starting point for application engineering projects. The orientation is different and consequently different roles are mainly involved in such an increment (e.g., consultants as peers to customers, or the leader of the core development team). Nevertheless, the opposite orientation must also be considered in both cases. The instantiated models from consultants are more standardized and thus also help to systematize the creation of new system variants. Consequently, the creation of new system variants must be part of the transfer project. In the other example, requirements on new system variants must always be mapped onto the internally integrated requirement model to benefit from it and to maintain it.

The introduction of domain engineering as a whole corresponds to a waterfall model. First, the complete domain asset base is constructed and then applications are built with reuse of the reusable assets. In software development of single systems an incremental life cycle model is seen by most organizations as the model that works in practice. An incremental life-cycle model for product line engineering in that sense, is a *subdomain* by subdomain transition. A subdomain corresponds to a part of the systems in the product line; it may be a system module, a component, or a feature group.

4 CONCLUSIONS

In this paper we describe how strategies for transitioning towards product line engineering can be defined. We think that their clear definitions in conjunction with accompanying measurement programs are the basis for sharing comparable data on product line transfer in the research community. Further, we believe that an incremental and customized transition strategy is key for a successful transfer project.

REFERENCES

[1] G. Arango. Domain Analysis Methods, in Software Reusability, W.Shaefer, R. Prieto-Diaz, and M. Matsumoto (eds.), Ellis Horwood, 1993

[2] P. Donohoe. Software Product Lines - Experience and Research Directions, Kluwer Academic Publishers, 2000

[3] V.R. Basili, G. Caldiera, and H.D. Rombach. Experience Factory. In John J. Marciniak, editor, Encyclopedia of Software Engineering, volume 1, pages 469-476. John Wiley & Sons, 1994.

[4] W. Tracz. Confessions of a Used Program Salesman. Institutionalizing Software Reuse. Addison-Wesley, 1995.

[5] K. Schmid and T. Widen. Customizing the PuLSE Product Line Approach to the Demands of an Organization Software, Process Technology, 7th European Workshop, EWSPT'2000, Lecture Notes in Computer Science 1780, Springer, Feb. 2000

[6] 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, in Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), May 1999

Scenario-Based Change Integration in Product Family Development

Klaus Pohl, Mathias Brandenburg, Alexander Gülich
University Essen, Software Systems Engineering,
Altendorfer Str. 97-101, 45117 Essen, Germany
email: pohl@informatik.uni-essen.de

1 Introduction

A product family defines a framework for developing customer specific applications in a particular domain. When defining a customer specific application the developers have to take the product family assets as well as application specific assets into account. In the case of a change, e.g. a requirements change based on customer specific needs, the developers must consider all relevant assets of both the customer specific application and the generic parts of the product family.

As already experienced in traditional software development errors made during change integration cause costly rework, especially if they are made at the requirements or architecture level. In addition, the integration of a change into a product family or a derived application is by far more difficult than the integration of changes into a single product. The main reason for this is that, in comparison with single product development, the interrelations among product family assets and between the product family assets and the derived costumer specific applications are much more complex and thus the number of assets being potentially affected by the change is much larger.

Thus, even a small change could have large effects on the product family and its customer specific applications. For example, a small change at the requirement level may effect several architectural and implementation components and thus require the re-execution or even the adaptation of several test cases. In addition, such a change may not only effect the generic product family asset, but also one or a whole set of customer specific application assets. Therefore even a small change may have a large impact in product family development.

For all those reasons, the developer has to be supported in identifying all assets potentially effected by the change integration and adapting them accordingly.

The change integration support sketched in this position paper facilitates the consistent integration of various types of changes, e.g. requirements changes, architectural changes, test case changes. For simplification we focus in this position paper on requirements changes. We that scenarios and use cases significantly facilitate change integration and product evolution in product family development, for two main reasons:

a) *Changes to requirements are quite likely to be identified first at the scenario or use case level*. Scenarios and use cases have proven in many application domains to facilitate the discussion and elicitation/specification of functional and non-functional system features;

b) *Scenarios and use cases help to bridge the conceptual gap between requirements and software architectures*. By defining and interrelating use case scenarios and architectural scenarios as well as scenarios with requirements and architectural scenarios with architectural (configuration) models requirements and architectural artifacts are related at a fine-grained, semantically rich level which is essential for supporting change integration.

In section 2 we briefly characterize change situations in product family development. In section 3 we describe the main roles scenarios can play in product family development. In section 3 we sketch how scenarios facilitate the interrelation of information at the requirements and software architecture level and thus how scenarios facilitate change integration and product evolution support.

2 Change Situations in Product Family Development

One can distinguish between three main change situations (types) in product family development:

(1) Change/evolution of the product family assets (upper part of Figure 1).

(2) Change/evolution of customer specific application assets (lower part of Figure 1).

(3) Development of a new customer specific application.

The main reason to distinguish between the evolution of a product family and a customer specific application asset is that if a product family assets is changed it is quite likely that several customer specific applications are effected. Changes of the type (1) thus tend to be more complex and effect more assets than changes of the type (2).

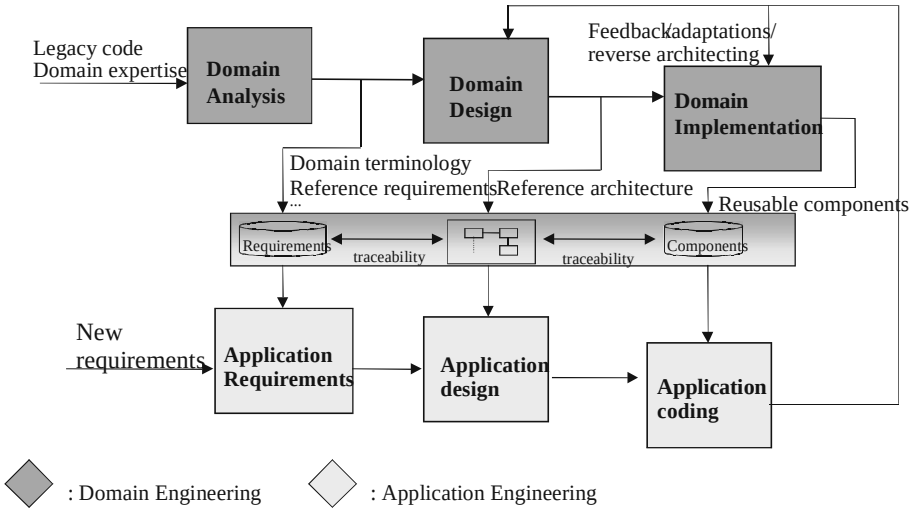


Figure 1: The generic ESAPS product-family engineering process

In addition, we view the development of a new customer-specific application as a special change situation. When developing a customer-specific application the customer specific requirements must be somehow implemented. The "implementation" of a customer requirement can lead to four major development situations (or combination of them):

a) *Reuse of product family assets*: If the customer requirement matches with one (or more) product family requirement(s), the corresponding product family assets can be reused to satisfy this particular customer requirement;

b) *Evolution of the product family*: A customer requirement which is of general concern and which cannot be satisfied through reuse should be implemented as generic product family asset and thus leads to an evolution of the product family;

c) *Developing application specific assets*: If a customer requirement cannot be satisfied through reuse and if this particular requirement is not of general concern, application specific assets have to developed;

d) *Evolution of a specific application*: If a new requirement for an existing customer-specific application has been elicited or if an existing requirement changes the customer-specific application has to be adapted.

Thus, the derivation of a customer-specific application from the generic product family assets can be viewed as a change process itself .

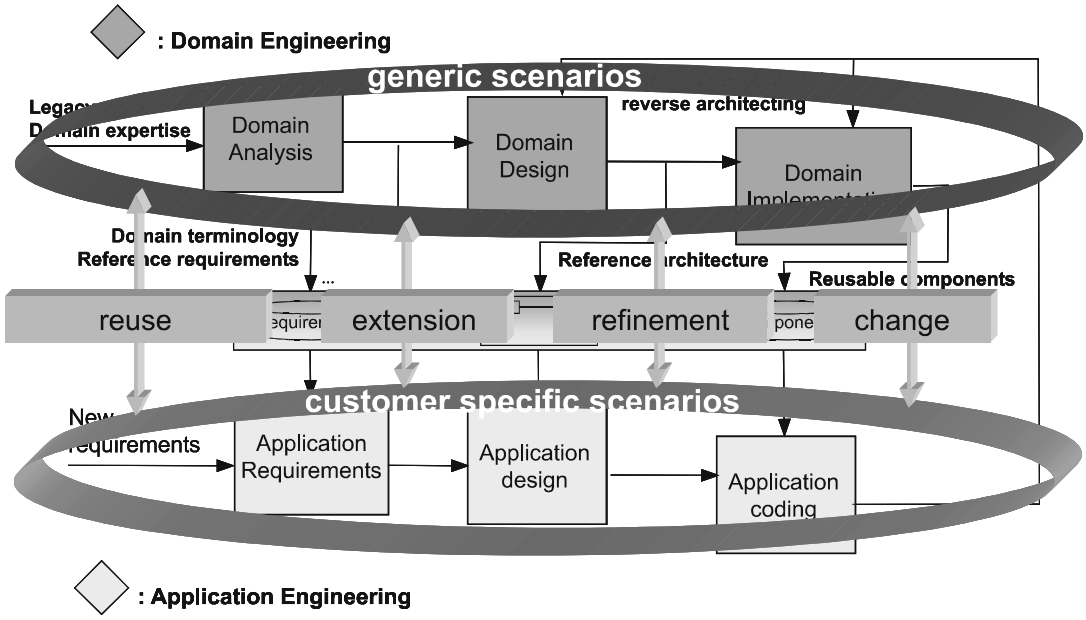


Figure 2: Scenarios in Product Family Context

3 Use of Scenarios in Product Family Development

Scenarios (e.g. use cases) provide an ideal means to select and/or extend the functionality offered by the product family according to customer specific needs. Based on the assessment of existing scenarios which are supported by the product family, the customer can identify the functionality that should be included in his specific application, i.e. the customer selects the use cases (or parts of them) which are required to satisfy his needs. Moreover, he can express specific requirements or wishes by modifying and extending existing scenarios or adding new ones. Customer specific changes are thus quite likely expressed in the first place at the use case or scenario level.

Consequently, we can distinguish two types of scenarios in product family development. ***Generic scenarios*** describe the overall functionality of the product family and thus define the functionality common to all products developed under the product family. ***Customer specific scenarios*** define the functionality required by a specific customer. Both types of scenarios represent requirements, illustrate how the architecture realizes the requirements and can be refined to test cases and can thus be used within every development phase.The generic scenarios are used to express the capabilities of the product family to the customer. They thus form an initial set of potential customer specific scenarios.

This is expressed by the reuse arrow in Figure 1. The customer can now adapt existing scenarios to his particular needs by either refining the description of the generic scenarios (depicted by the refinement-arrow in Figure 1) or by modifying the exiting scenarios, e.g. adding new steps, changing the step sequence, removing unrequired steps (depicted by the change-arrow in Figure 1). The customer can of course also extend the initial set of scenarios, i.e. define new scenarios which add a functionality not provided by the system so far (depicted by the extension-arrow in Figure 1). He can, obviously, also drop generic scenarios and thereby indicate that he does not need a certain functionality of the product family. Thereby, the customer expresses the application specific changes at the use case and scenario level. Scenarios in general and use cases in specific are thus a central means for identifying customer-specific requirements and for deciding about the functionality of a customer-specific application.

To support change integration in product families we therefore put scenarios in a central role. Scenarios are used to *identify and classify possible changes* (such as adding functionality to a scenario, refining a functionality or an actor, incorporating a new scenario, definition of alternative scenarios, consideration of application specific error handling procedures and the like) and to *analyze the effects of those changes* on the software architecture and even test cases. Moreover, scenarios are used to *structure product family development information* and to use this information for *supporting change integration*.

4 "Integrating" Requirements and Architectural Information using Scenarios

In all change situations identified in section 2, the developer must consider all effected product family and application specific assets. This is by far not trivial, especially if the satisfaction of the customer requirements leads to an evolution of the application or even the product family itself. Identifying all assets which might be effected by a change is a complex and error prone task. Important assets could easily be overseen and might thus be not correctly adapted.

Thus, the developers must be supported in identifying product family assets potentially effected by a change. It is commonly agreed that traceability is a prerequisite for identifying artefacts effected by a change and for supporting effective change integration. In the case of a requirement change, the software engineer must be able to identify the potentially effected parts of the software architecture. Vice versa, in the case of an architecture change she must be able to identify the effects the change has on the requirements and there fulfilment by the system.

The support which can be provided heavily depends on the trace structure and the comprehension of this structure by the developer. To provide a semantically rich trace structure which is easy to comprehend we adapt the well proven meta-model driven approach (cf. [Pohl, 1996]) which suggest to define orthogonal meta models for each artefact type (in our case the requirements and the architecture artefacts). Each of those meta model defines the concepts and the relations required to capture the relevant information about the artefacts.

An obvious problem of interrelating the requirements and architectural artefacts is that the conceptual distance between both artefacts is quite large. Moreover, it is quite common that a requirements artefact has an n:m relation to a architectural artefact and vice versa. In other words, a requirement artefact often influences the definition of more than one architectural artefacts (e.g. a component) and vice versa the definition of a component is often influenced by more than on requirement artefact. Those (as experienced in several approaches) interrelating just requirements and architectural artefacts does not go very far.

To achieve a semantically more rich integration of requirements and architecture artefacts we thus suggest to include use cases and scenarios for three main reasons:

- use case and scenarios describe concrete system interactions executed to achieve a certain goal. They thus put a set of requirement artifacts into a context. A set of use case can be used to define the intend use of the system instead of dealing with all possible use. Use cases thus reduce the complexity of all possible system-user interactions to a set of desired interactions;
- requirements changes are identified more frequently at the scenario and use case level. Scenarios provide an ideal means for communicating product family functionality to the customers and for defining customer specific requirements as extensions, modifications or refinements of existing scenarios or by adding new scenarios;
- scenarios enable us to interrelate the software requirements and software architecture components on a much more fine-grained level and thereby the large conceptual distance between the requirements and software architecture artifacts can be "bridged".

Consequently, we suggest to define six orthogonal meta models to capture and interrelate requirements and architectural information (see Figure 3).

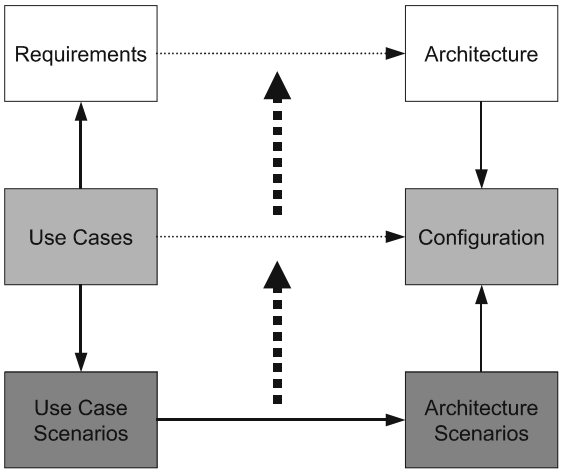


Figure 3: Interrelating Requirements and Architectural Assets

At the requirements level we suggest to define three type of meta models, namely:

- a ***requirements meta model*** which defines how requirements for the product family and the customer specific applications should be defined. Obviously, such a requirements meta model defines several requirements types (e.g. functional requirements, quality requirements) and for each requirements types several attributes (e.g. textual description, reference number, priority, orign, …). The requirements model we currently use is described in [Pohl, 1996]).
- A ***use case meta model*** which defines how use cases should be defined. The use case meta model essentially uses the attributes of the use case template typically being provided for specifying use cases as a basis and adds domain specific details. A use case defines at a high level the use of the system functionality in a given context, e.g. interaction performed with the system to achieve a certain business goal. To define use cases we currently use an extended version of the UML use case meta model (see [Brandenburg et al., 2001] for details).
- A ***use case scenario meta model*** which defines the concepts for specifying satisfying and exceptional scenarios. In general, a scenario is defined as sequence of actions to be executed to satisfy the goal of the associated use case. We currently use an extended version of the scenario meta model presented in [Pohl and Haumer, 1997].

Similarly, we suggest to use three types of meta models at the architectural level, namely:

- an ***architectural meta model*** which defines the components (well-defined software-building-block that possesses an application-oriented functionality which semantically belongs together), the connectors (a design-level element which represents communication and/or interaction-relation between internal components) and the link types which can be used to bind a component and a connector with complementary interfaces together.
- An ***architectural configuration meta model*** which defines a (set of) concrete software architectures. It thus reduces the possible combination of the components, the connectors and the links (interface types) defined in the architectural model.
- An ***architectural scenario meta model*** which defines concrete architectural interaction sequences under a given architectural configuration. A architectural interaction sequence consists of atomic architectural

interaction steps. A (or a set of) architectural interaction sequence is executed to achieve a certain goal, i.e. it is the "realization" of a use case scenario definition.

The six models allow us to capture requirements and architecture information in much more detail. Especially the scenarios at the requirements and the architecture side allow us to establish a tighter integration between the requirements and the software architecture. Briefly this is achieved by interrelating the (functional) requirements and the uses cases and by detailing the use cases in the associated use case scenarios. Those scenarios can easily be mapped to the architecture scenarios by mainly detailing the associated information and distinguishing between various system components. The architecture scenarios themselves can be viewed as "instances" of the architecture configuration. The architecture configurations describe how architecture elements are composed. Thus an architecture scenario defines the interaction between architecture components under a given architecture configuration. Finally, the architecture components can thus be easily mapped to the architecture elements.

To summarize, requirements and architecture components are now related via use case scenarios and architecture scenarios (cf. Figure 3). This interrelation can of course be used to establish and/or explain relations at the more abstract levels, i.e. relations between use cases and architecture configurations and between requirements and architecture elements (details about our approach can be found in the ESAPS deliverable [Brandenburg et al., 2001]).

The approach is currently being used in two industrial product line approaches, one in the medical and the other in the automotive domain. Besides the specialization of the six model types sketched above we also added adequate model at the "implementation level" which allow us to trace the effects of use case and scenario changes not only to the architectural level, but also to the test case and run time system level and vice versa.

5 Presentation at the Workshop

At the presentation at the workshop we will

a) motivate why you should use scenarios and use cases in addition to traditional software engineering models throughout (!) the development process (and not just during requirements elicitation and specification)

b) provide details on the requirements and architectural meta models, especially about their interrelations and how those interrelations can be used to support change integration

c) provide an adapted example of the application of the approach to a real industrial product family.

We think, that the presentation will provoke discussions about

a) change integration and product family evolution

b) role of requirements and scenarios in change integration

c) role of scenarios in product family development in general, e.g. elicitation of customer specific requirements, communication of product family features to customers, definition of typical usages of product families

6 References

[Brandenburg et al., 2001] Brandenburg, M.; Pohl, K.; Strembeck, M.: "Product and Project specific trace capture for product families", ESAPS-Report (D3.1a (E1.2b); and "Goal Driven reuse of experience; guide for trace usage" ESAPS-Report D3.2b (E1.2c), January 2001

[Dömges and Pohl, 1998] Dömges, R.; Pohl, K.: „Adapting Traceability Environments to Project-Specific Needs", In: Communications of the ACM, December 1998

[Jacobson et al., 1999] Jacobson, I.; Booch, G.; Rumbaugh, J.: "The Unified Software Development Process", Addison-Wesley, 1999

[Pohl and Haumer, 1997] Pohl, K. ; Haumer, P.: „Modelling Contextual Information about Scenarios", In: Proc. of REFSQ 97 Workshop, 1997

[Pohl, 1996] Pohl, K.: "Process-centered requirements engineering", John Wiley & Sons, 1996

[Pohl et al., 1997] Pohl, K.; Dömges, R.; Jarke, M.: „Towards Method-Driven Trace Capture", In: Proc. of Caise 97, 1997

[Pohl et al., 2000] Pohl, K.; Reuys, A.; Strembeck, M.: "Intentional and contextual process modelling language for defining method guidance", ESAPS-Report (E1.2a/D2.1e), June 2000

[Weidenhaupt et al., 1998] Weidenhaupt, K; Pohl, K.; Jarke, M.; Haumer, P.: "Scenario Usage in System Development: A Report on Current Practice", In: IEEE Software, March 1998

[Weinberg, 1997] Weinberg, G.M.: " Quality Software Management Volume 4: Anticipating Change", Dorset House Publishing, 1997

Test Framework Product Line for Low Earth Orbit Satellite Constellations

David Rine*
David Fortini
Nader Nada
Mahmoud Elish
Department of Computer Science, MSN 4A5
George Mason University
4400 University Drive
Fairfax, Virginia USA 22030-4444
Drine@cs.gmu.edu
703-993-1546

ABSTRACT
This Paper identifies the Test Framework Product Line, which will provide a common set of test threads for Low Earth Orbit satellite constellations. These test threads will provide significant satellite component coverage with minimum amount of data storage. Due to the data storage and visibility limitations of low earth satellite constellations, an automated and compressed means of testing needs to be implemented. The Test Framework Product Line will contain the common set of test threads required for the deployment, operations and maintenance, and anomaly detection and resolution phases. The following phases of research were used to address these issues: The first phase of research determined the applicable satellite orbits, phases of deployment, and satellite components, which will be the Focus Area of Research (FAR), for framework test components. The second phase of research identified the major nominal threads of testing necessary for each component in the FAR. This will provide the candidate Test Threads to be included within the determined Test Framework Product Line (TFPL). The third phase of research developed a set of analysis criteria, which addresses the Hypothesis and Research questions. The fourth phase analyzed each test thread in the TFPL against the analysis criteria, which was determined during the third phase of research. The TFPL will be refined to the test threads, which satisfy the analysis criteria. In conclusion there is a common set of test threads that can be reused for multiple Low Earth Communications payload satellite constellations. These test threads will also preserve precious on-board resources and execute in an efficient manner.

KEY WORDS: Satellite Software, Testing, Reuse, Product Lines, Test Framework.

1 INTRODUCTION

This research identifies the Test Framework Product Line, which will provide a common set with minimum amount of data storage. Due to the data storage and visibility limitations of low earth satellite constellations, an automated and compressed means of testing needs to be implemented. The Test Framework Product Line will contain the common set of test threads required for the deployment, operations and maintenance, and anomaly detection and resolution phases.

Wireless Satellite Communications Technology is in its infancy state and is rapidly growing in terms of research and production. In the past, satellites have been an expensive commodity and designed for specific uses with myopic earth coverage and visibility. As of recent there are many start-up efforts in this area which satellites are being produced in the manner of mass production similar to the automobile industry. These satellite constellations will provide complete coverage of the earth and provide connectivity for phones, faxes, and pagers in a reliable manner eventually eliminating the need for wire communications.

As this area grows, there is a need for efficiency in the development of hardware and software components and the testing of these components. Solar Power and Propellant Fuel are two precious commodities for a satellite that determine the life of the vehicle. One major limitation of a satellite vehicle is the amount of space and data storage on board. With today's technology, there has been significant progress in the reduction in the footprint of computers and the increase amount of data storage that requires minimum space for storage devices. However, these aspects are still limitations with onboard satellites due to the weight of these devices and the additional fuel needed to maneuver the vehicle in space.

Low earth orbiting satellites provide more visibility passes per day but have shorter visibility windows than that of a Geosynchronous or high orbit satellite. Due to the small window of visibility and onboard storage, an efficient and compressed approach to testing needs to be implemented when performing tests on orbit. This research paper identifies the common threads of testing [3] for each satellite component that requires the minimal amount of storage and visibility time for execution.

of test threads for Low Earth Orbit satellite constellations. These test threads [3] will provide significant satellite component coverage

2 REVIEW OF PRIOR RESEARCH

There is not an abundant amount of literature on Space Mission Operations and Design available to the public. Several mission applications in the past were classified and special clearances are needed to access the information. There was also a lack of coordination and documentation in this area due to its infancy and isolation of stove-piped operations. Two comprehensive textbooks [2, 5], which are available to the public, are referenced as background. Both of these textbooks are part of a collection of books in the Air Force Academy's Space Technology Series. This series intends to provide practical approaches to designing and operating space systems and capture experiences by key people in Space Programs of the past. The first textbook [5] is titled Cost-effective Space Mission Operations and covers processes, tools, data that can aid in performing space mission operations in a better and more cost-effective way. The second textbook [2] is titled Space Mission Analysis and Design that provides information that aids the user in defining a space mission to meet broad and poorly defined objectives.

3 ORGANIZATION

This paper is organized in the following way:

- Each four phases of research are addressed in order.
  - The first phase of research determined the applicable satellite orbits, phases of deployment, and satellite components, which will be the focus area of research (FAR), for framework test components.
  - The second phase of research identified the major nominal threads of testing [3] necessary for each component in the FAR. These test threads will be the initial entries of the Test Framework Product Line (TFPL) [1, 4, 6, 7].
  - The third phase of research developed a set of analysis criteria, which addresses the Hypothesis and Research questions.

- The fourth phase analyzed each test thread in the TFPL against the analysis criteria, which was determined during the third phase of research. The TFPL will be refined to the test threads, which satisfy the analysis criteria.

4 TECHNICAL PROBLEM STATEMENT PROBLEM STATEMENT

Due to the data storage and visibility limitations of low earth satellite constellations, an automated and compressed means of testing needs to be implemented. Tests need to be performed out of sight because the amount of time for line of sight vehicle contacts is limited and needed to perform real-time state of health operations for the vehicle. There is a limited amount of storage on board so out of site time tagged commands must be designed in an efficient manner to insure maximum test coverage with the minimum amount of commands. The completion of these tests in a short period of time is also important since this area of technology is very competitive and "time to market" operational capabilities is of the essence. In summary, test scenarios need to be develop to provide complete coverage, with minimum amount of data storage and visibility and completed in a timely manner.

This Test Framework Product Line would reduce cost and initial activation time of the constellation [1, 4, 6, 7].

Their input was instrumental when applying each one of the test threads to the analysis criteria that resulted in the ultimate TFPL.

5 RATIONALE: PROBLEM IMPORTANCE

Efficient on orbit test scenarios is important for satellite communications networks for several reasons. It will preserve precious data storage space on board of the vehicle, provide quicker turnaround of vehicle's use within the constellation, and identify potential end of life vehicle characteristics, which will allow for a suitable replacement in the network constellation in a proactive manner. A common set of test threads, which can be applied to multiple satellite constellations, will insure major thread coverage and save valuable time during the test and integration phase.

6 HYPOTHESES AND RESEARCH QUESTIONS

- What are the generic sets of test threads that are required for initial stages of satellite deployment? Can they be packaged into a common set of test threads that could be reused for other satellite networks?
- What is the common set of test threads required maintaining the health and status of a satellite network constellation? Can they be incorporated into a generic can set of threads that can be reused for future constellations?
- What is the generic set of test threads required for vehicle anomaly and resolution? Can they be packaged into a common set of test threads that could be reused for other satellite networks?
- Which of the test threads mentioned above require minimal amount of storage in order to efficiently use limited allocated resources on board?
- Which of the tests mentioned above be stored on board the vehicle and executed in the future automatically without the need of using precious and limited line of sight contacts?

7 POTENTIAL APPLICATIONS

The thesis of our research has focused on Low Earth Orbit Satellite Constellations with Communication Payloads. The Test Threads [3] identified are generic and may be tailored for different types of constellations and payloads. Newly deployed or existing satellite constellations can use this model as a starting point to refine existing test threads or develop new test threads that are generic yet applicable to the current domain. The goal is to identify a set of reusable test threads, which can serve as a basis for a Test Framework Product Line [1, 4, 6, 7], which can be easily modified to insure complete testing coverage is performed on the major components of a satellite.

8 RESEARCH APPROACH PHASES IN RESEARCH APPROACH

- The first phase of research was to determine the applicable satellite orbits, phases of deployment, and satellite components, which will be the focus of research for framework test components. This phase is necessary because of the amount of research necessary to provide complete coverage of all possible scenarios related to the variables mentioned of above is excessive and to insure an in depth analysis is performed within this research

paper. This selected area of research will be referenced as the FAR (Focused Area of Research).

- The second phase of research was to identify the major nominal threads of testing [3] necessary for each component in the FAR. This will provide the candidate Test Threads to be included within the determined Test Framework Product Line (TFPL) [1, 4, 6, 7].
- The third phase of research was to develop a set of analysis criteria, which addresses the Hypothesis and Research questions.
- The fourth phase was to analyze each test thread in the TFPL against the analysis criteria, which was determined during the third phase of research. The TFPL will be refined to the test threads, which satisfy the analysis criteria.

9 RESEARCH DESIGNS & METHODS FOR EACH PHASE: EMPIRICAL OR ANALYTICAL METHODS AND MODELS

The first phase of research was to narrow down the specific type of satellite constellation to be included in the focus area of research. The selected constellation was a Low Earth Orbit constellation with a civil communication payload. The rational for this selection is two folded. First, this type of constellation inhibits the types of problems mentioned in the initial problem statement. There is a short visibility time window and limitations on available data storage onboard. Second, the second author worked with several satellite engineers who are experts with years of experience on this type of constellation and payload. Their experience and participation was critical to validating our thesis.

The second phase of research was to select which satellite components were to be included in the focus area of research. There were three major criteria, which determined the selection process. First, the components had to be substantial and common within all satellite constellations. Second, the components had to exhibit significant test activities [3] during the Launch, Early Orbit Checkout, Operational and Maintenance, and Anomaly resolution phases of spacecraft operations. The third criteria was the components had to be significant elements pertaining to a civil communications payload where I had several subject experts available.

The third phase of research was to determine the selection criteria for which test threads were to be included in the Test Framework Product Line [1, 4, 6, 7]. Five major criteria items were identified: Level of Importance, Vehicle Uniqueness, Phase of Lifecycle, Size of Test Procedure, and Line of Site or Stored Command Execution. These five items were selected to address the hypotheses and research questions and are further defined in the next section, which describes the fourth phase of research.

10 EMPIRICAL METHODS: CASE STUDIES AND EXPERIMENTS

The fourth phase of research involved a technical questionnaire, which was distributed to several satellite experts in the area of components that were defined in the FAR. There were weights and values assigned to each of the five criteria items derived in the third phase of research.

The criteria item, Level of Importance, relates to the critically of a test thread if it was not performed and the functionally was due to fail. The following five values were assigned to this criteria:

1. *Not Important* : If this test thread was not executed it would not have a critical effect on the safety of the vehicle or payload.
2. *Important for other reasons* : The test thread is important but not relative to the vehicle's safety or payload.
3. *Payload Critical* : The test thread is important for the payload to operate successful but not critical to the vehicle safety.
4. *State of Health, Normal Operation and Maintenance* : The test thread is critical to the daily operations of the vehicle, which could lead to a vehicle safety issue.
5. *Vehicle Safety* : The test thread is critical to the life of the vehicle and if not performed properly it could lead to the vehicle to remain inoperable.

The second criteria Vehicle Uniqueness relates to the test thread being specifically associated with distinct vehicle features. There were five weights assigned to this criteria:

1. *Very Unique*
2. *Mostly Unique*
3. *Somewhat Unique*
4. *A Little Unique*
5. *Not Unique*

The third criteria, Phase of Lifecycle, had three values assigned and identified during which phase was the test thread executed:

1. *Launch and Ascent*
2. *State of Health (Operation & Maintenance)*
3. *Anomaly and Resolution*

The fourth criteria, Size of Test Procedure, identified the amount of storage required onboard the vehicle to execute the test thread and the associated telemetry values. The sizes were derived from a specific implementation of a Low Earth Orbit Constellation with a civil communications payload. The allocated resources reserved for test products and telemetry as well as real-time test experience was used to determine the size associated to each weight:

1. *Very Large* : Storage required to house test products that exceed two megabytes.
2. *Significantly Large* : Storage required is greater than one Meg but less than two Meg.
3. *Large* : Storage required is greater than five thousand bytes but less than one Meg.
4. *Medium* : Storage required is greater than one thousand bytes but less than five thousand bytes.
5. *Small* : Storage required is less than one thousand bytes.

The fifth and last criteria was whether the test thread required a Line of Sight vehicle contact or whether it could be stored onboard and automatically execute at a later time.

The two values associated with these criteria were:

1. L : for line of sight
2. S : for stored command

A questionnaire was developed and distributed to specific satellite component engineers. Each engineer mapped the test threads associated with the component of their expertise to each of the five criteria items described above. A master table was created and the average values of all completed surveys were included with our results. This was critical information that determined which test threads were selected for the TFPL.

11 RESEARCH RESULTS

The research results are a generic set of test threads that fulfill the criteria of the hypothesis and questions mentioned previously. This Test Framework Product Line would contain threads which were not satellite specific, require minimal time and storage vehicle resources, and critical for mission operations.

12 ANALYTICAL RESULTS

The first three phases of research required in depth analysis of satellite constellations, specific payloads and common components. The selection of a Low Earth Satellite constellation with a civil communications payload was predetermined and should remain static through out our research. This selection was strategically chosen due to its characteristics tightly coupled to those addressed in the hypotheses and questions. It was also selected due to the amount of experts immediately available and accessible to the second author at his current work assignment.

13 EMPIRICAL RESULTS

The empirical results identified a set of test threads [3] which are critical, require minimal vehicle resources, and not unique to a specific vehicle. We also identified a set of critical test threads that are critical but not satisfy the other two criteria of uniqueness and efficiency. These test threads will have to be executed due to their critically to the life of the vehicle. In contrast, we would expect a set of test threads that meet the two less important criteria but not be critical in nature to the vehicle's safety. These should not be included in the TFPL.

When organizing the selected test threads by phases, we would expect to see the majority of test threads to be associated with the phase of Operations and Maintenance. The other two phases of Launch and Ascent, and anomaly detection and resolution will be a smaller set due to their duration and number of events associated with them.

14 PLANS FOR INTERPRETING RESULTS

After all the questionnaires were completed and each entry was averaged on the master table of analysis, the following selection criteria were applied:

- All test threads, which have been identified critical for vehicle safety, will automatically be included in the TFPL.
- All test threads with a level of importance of Payload Critical, Vehicle Safety or State of Health and a vehicle uniqueness value of 3-5 will be included if:
  - it can be a stored command which executes in the future

- or it is a line of site command, which requires a size that is not very or significantly large.

15 INTERPRETING RESULTS

Each test thread in the master table was applied to the criteria explained in the *plans for interpreting the results* section. All test threads that meet the criteria are included in the TFPL and organized by subcomponent. All of the assigned values are preserved in the TFPL Table for further analysis of the results.

16 IMPLICATIONS FOR THEORY OR PRACTICE

The results of this research provided a good starting point for creating an initial set of test threads for a new satellite constellation or adjusting an existing set of test threads [3]. The results identified which test threads should be the area of focus and which test threads might be reused from an existing set of test threads for another satellite system [1, 4, 6, 7]. The test threads, which are not vehicle unique, should require minimal modification for reuse. It will also identify critical test threads, which need to have preallocated space and time reserved to insure vehicle safety.

17 LIMITATIONS OF RESULTS

The results of this research are depicted as test threads which are a high level entity. Further refinement of each test thread needs to be performed before the implementation phase. The satellite components selected were based on a common set for the majority of satellites. If the vehicle in question has a very unique complex payload, additional identification of payload specific test threads needs to be identified and applied to the criteria used for the TFPL.

The amount of assigned storage for test products and telemetry was based on an average low earth orbit constellation with a civil communications payload. The results will be different if the system in question has significantly more or less storage available. If this is the case, the size of storage associated with values assigned to *size of test procedure* criteria, needs to be adjusted and reapplied to the test threads.

The results of this investigation were based on a Low Earth Orbit satellite constellation, which emphasizes limited line of site opportunities for command execution. If the constellation contained geosynchronous satellites where constant visibility is not and issue, then more weight would be emphasized on the amount of storage versus whether the command could be stored onboard for future execution.

18 DOMAIN RESEARCH DETAILS

The first phase of research was to determine the applicable satellite orbits, phase of Operation, and satellite components, which will be the focus of research for framework test components. This phase is necessary because of the amount of research necessary to provide complete coverage of all possible scenarios related to the variables mentioned of above is too much material to cover. To insure an in depth analysis is performed within this research paper, a selected area of research will be determined and referenced as the FAR (Focused Area of Research).

There are several orbits used for Earth Referenced Missions; such as Geosynchronous, Sun synchronous, Molniya, Frozen Orbit, and Repeating Ground Track. Selecting a mission orbit is often highly complex and involves in-depth trade studies considering factors such as availability of launch vehicle, earth coverage, payload performance, communication links, and other technical constraints. The orbits mentioned above have unique characteristics and applications, which need to be examined when selecting a specific orbit.

Geosynchronous Orbits maintain a nearly fixed position above the equator and is usually used for communications and weather. Sun Synchronous Orbits rotate and maintain approximately constant orientation to the sun and are usually used for weather and earth resources. Molniya Orbit's apogee and perigee do not rotate and is usually used for high latitude communications. Frozen orbit minimizes changes in orbit parameters and is primarily used for applications, which require stable conditions. Repeating Ground Track Orbits contain subsatellite trace repeats and is used for applications, which require constant viewing angles.

A Geosynchronous Orbit may provide the best earth coverage but requires excessive propellant, instrument resolution, or power. The mission needs of a communication system may determine that the value of providing continuous coverage may outweigh the cost and performance loss due to the distance of Geosynchronous Orbit. A trade study between GEO

and LEO (Low Earth Orbit-below 1000km) may also be beneficial. GEO provides better Earth coverage and has a longer life; whereas, LEO have better instrument performance and requires less propellant to reach orbit. Our FAR will a frozen orbit LEO constellation that will be used for a communication network.

Spacecraft Operations consists of four major phases: Prelaunch, Launch, Early Orbit Checkout, and Normal Operations. Prelaunch usually starts two years before launch and consists of the following tasks: Development of flight plan, Development of training plan, Identify simulator requirements, Integrate and test support systems. Launch activities include supporting launch team and transfer spacecraft to initial orbit. Early Orbit Checkout usually starts two days after launch and may continue up to a six month period. Tasks during this period include validation of components, subsystems, subsystems and their interfaces, systems, instrument processing, and external interface protocols. Calibration of instruments, maneuver of spacecraft to mission orbit and detection/analyzing anomalies is also tasks performed during this period. Normal Operations usually begins thirty days after launch and continues for the life of the spacecraft, which is unique for each specific vehicle. Tasks during this phase include maintenance of ground software/database and flight software, resolution of anomalies, process and distribution of payload data, real-time operations, and disposal of non-operational spacecraft.

Test activities are concentrated during the prelaunch and early orbit checkout periods. Our FAR will address these phases of spacecraft operations.

All spacecraft possess a common set of subsystems:

- Attitude Determination and Control
- Communications
- Command and Data Handling
- Power
- Thermal
- Structures and Mechanisms
- Guidance and Navigation

There are several different types of payloads such as:

- Communications Both Civil And Military
- Scientific Applications Such As Space Shuttles/Platforms/Stations
- Telescopes
- Biological
- Solar
- Stellar
- Earth Looking Remote Sensing Applications Such As Fire Or Nuclear Burst Detection Reconnaissance
- Oceanography And Weather
- Special Strategic Defense Initiatives Including Laser/RF/Particle Beams
- Intelligence
- Jamming
- Space Surveillance and Tracking.

Our FAR will consist of the common subsystems mentioned above and a civil communications payload.

The Attitude Determination and Control Subsystem (ADCS) stabilizes and orients the vehicle. The vehicle uses sensors to determine its attitude and actuators to control it.

Major test threads within this subsystem include:

- Determine pointing of sensor
- Mapping the image to the target
- Establish relationship between the sensor optics or antenna pointing and the orientation in space
- Locate ground reference point on target image
- Compute attitude necessary to yield observed displacements of the reference location using an orbital model for specified vehicle
- Use two ground reference points to determine vehicles roll, pitch, and yaw
- Target areas with no reference points, i.e. Ocean, compute attitude for the orbit section that was over land and propagate attitude information forward to the point when it is over the ocean.
- Compute geolocation using spacecraft ephemeris information
- Send geolocation information via command and control link

Communications subsystem also referenced as the Telemetry, Tracking, and Control (TT&C) subsystem is the interface between the satellite and Earth or other satellites. Major test threads within this subsystem include:

- Carrier tracking (2-way coherent and noncoherent communication,1-way communication)
- Command reception and detection
  - ⇒ Acquire and track uplink carrier
  - ⇒ Demodulate carrier and subcarrier
  - ⇒ Derive bit timing and detect data bits
  - ⇒ Resolve data-phase ambiguity if it exists
  - ⇒ Forward command data, clock, and in-lock indicator to the subsystem for command and data handling
- Telemetry modulation and transmission

⇒ Receive telemetry data streams from the command and data handling subsystem or data storage subsystem
⇒ Modulate downlink subcarrier and carrier with mission or science telemetry
⇒ Transmit composite signal to Earth or relay satellite

- Ranging
  ⇒ Detect and retransmit ranging pseudorandom code or ranging tone signals
  ⇒ Retransmit either phase coherently or noncoherently
- Subsystem Operations
  ⇒ Receive Commands from the subsystem for command and data handling
  ⇒ Provide health and status telemetry to the C&DH subsystem
  ⇒ Perform antenna pointing for any antenna requiring beam steering
  ⇒ Perform mission sequence operations per stored software sequence
  ⇒ Autonomously select omni antenna when spacecraft attitude is lost
  ⇒ Autonomously detect faults and recover communications using stored software sequence

Command and Data Handling (C&DH) system performs two major functions. It receives, validates, decodes, and distributes commands to other spacecraft systems. It gathers, processes, and formats Spacecraft State of Health and Mission data for downlink.

Major test threads within this subsystem include:

- Receive command message
- Validate command message
  ⇒ Reception of synchronization code
  ⇒ Check command message length
  ⇒ Match spacecraft address
  ⇒ Match fixed unused message bits
  ⇒ Detect lack of errors using error check polynomial code
  ⇒ Update reject or accept message counters
- Decode command to determine output type and specific interface channel
  ⇒ Discrete commands are fixed amplitude and pulse duration
    ♦ High level discrete command used to drive latching relay coil or fire an ordnance device
    ♦ Low level discrete command open collector interfacing with digital logic
  ⇒ Serial command 3-signal interface used to indicate interface is active

Data handling combines telemetry from multiple sources and provides it for downlink

⇒ Receives analog telemetry data
  ♦ High-Level Analog : Telemetry Channel with information encoded as an analog voltage which does not need measurement excitation
  ♦ Low-Level Analog: Telemetry Channel with information encoded as a low analog voltage which amplification is needed
  ♦ Passive Analog: A Telemetry Channel with information encoded as a resistance

- Analog Telemetry is converted digital form
  ⇒ Bi-Level (Discrete) Input: Telemetry Channel representing two state information
  ⇒ Serial Telemetry (Digital) Interface: 3-signal interface used to transfer digital data from an external source to data handling equipment
- Format data into serial stream of continuous data for downlink
- Optionally provide telemetry data for to on-board computer

The Electrical Power System (EPS) provides stores, distributes, and controls spacecraft electrical power.

Major test threads within this subsystem include:

- Supply a continuous source of electrical power to spacecraft loads during the mission life
- Control and distribute electrical power to spacecraft
- Support power requirements for average and peak electrical load
- Provide converters for ac and regulated dc power buses, optional
- Provide command and telemetry capability for EPS health and status
- Protect spacecraft payload against failures within the EPS
- Suppress transient bus voltages and protect against bus faults
- Provide ability to fire ordinance, optional

The purpose of a thermal-control subsystem is to maintain all the elements of a spacecraft system within their temperature limits for all mission phases.

- A majority of testing of the thermal-control subsystem occurs during spacecraft design and development. There is minimal testing required

during launch and on-orbit phases; therefore this subsystem is out of our FAR and will not be included in our TFPL.

The Structures and Mechanism Subsystem attaches the spacecraft to the launch vehicle and provides the functionality for separation from the launch vehicle.

Major test threads within this subsystem which occur during Launch/Ascent and mission operation phases include:

- Steady State booster accelerations
- Vibro-acoustic noise during launch and transonic phase
- Propulsion system engine vibrations
- Transient loads during booster ignition and burn-out, stage separations, vehicle maneuvers, propellant slosh, and payload fairing separation
- Pyrotechnic shock from separation events, deployments
- Thermal Environments

The Navigation and Guidance subsystem consists of two parts. Navigation to determine the satellites position and orbital elements as a function of time. Guidance adjusts orbit to meet some predetermined conditions.

Major test threads within this subsystem include:

- Orbit Maintenance: maintaining orbital elements not timing of satellite location in orbit
- Station keeping: maintaining the satellite in a predefined box
- Orbit propagation: integrating the equations of motion to determine where at satellite will be at a given time
  - ⇒ Real-time orbit determination provides where a satellite is at the present time
  - ⇒ Definitive orbit determination is the best estimate of the satellite position and orbital elements at some earlier time.

The following is a list of each criterion and their associated values. Each test thread below will be applied to each criterion and assigned an associated value. The following is the master table, which contains an average value for all completed questionnaires. This is the full set of test threads, which will be refined further when applying the criteria associated with the Test Framework Product Line.

Level of Importance

1. Not Important
2. Important for other reasons
3. Payload Critical
4. State of Health, Normal Operation and Maintenance
5. Vehicle Safety

Vehicle Uniqueness

1. Very Unique
2. Mostly Unique
3. Somewhat Unique
4. A Little Unique
5. Not Unique

Phase of Lifecycle

1. Launch and Ascent
2. State of Health (Operation & Maintenance)
3. Anomaly and Resolution

Size of Test Thread

1. Very Large (Greater than 2 meg)
2. Significantly Large (Greater than 1 meg; Less than 2 meg)
3. Large (Greater than 5k bytes; Less than 1 meg)
4. Medium (Greater than 1k; Less than 5k)
5. Small (Less than 1k)

Execute During Line of Site or Stored CMD

L – Line of Site
S – Stored CMD

Table 1 Master Test Thread Table (See Appendix)

The following table (Table 2 – See Appendix) represents the Test Framework Product Line [1, 3, 5, 6]. The criterion, mentioned in the *Plans for interpreting results* section, has been applied to the previous Master Table. The TFPL only contains those test threads that have successfully satisfied the criteria mentioned above.

Table 2 Test Framework Table

19 SUMMARY

The results of this paper depict that there is a Test Framework Product Line [1, 4, 6, 7] which contains a common set of reusable test threads which are critical for mission operations, require minimal time and storage resources, and are not satellite unique. There was a subset of threads that were critical to the vehicle safety but not satisfying the other two criteria of time and space, and stored commands. In contrast, test threads that met the latter two criteria but were not critical to the vehicle safety did not get selected for the TFPL.

The candidate test threads for some of the specified satellite components were selected in their

entirety within the TFPL. These components appear to be more generic in terms of test threads than others. The components which all of the candidate test threads were selected were communications subsystem, ranging, subsystem operations, electrical, power subsystem, structures and mechanism subsystem, and navigation and guidance subsystem. The two components, which a subset of the candidate test threads, were selected were attitude determination and control subsystem and command and data handling subsystem. These components appeared to be unique to each satellite vehicle but still contained a subset of generic test threads, which were included in the TFPL.

This investigation identified a common set of test threads for a subset of satellite components, which need to be defined and executed on the majority of satellite constellations. The corresponding test procedures need to be defined for each of the common test threads, which are included in the TFPL. These test procedures need to be generic in nature and easily modified to be reused on other satellite constellations. Proper test coverage and reduction in time and effort will be realized when a template for these test procedures is defined. This is the first stage in developing a product line approach for this domain of important satellite test software [1, 4, 6, 7].

REFERENCES

[1] Baldo, J. Moore J., Rine, D., 'Software reuse standards,' Standards Review: ACM Perspectives on Standardization 5, 50-57. 1997.

[2] Boden, Daryl G, Larson, Wiley J., 'Cost-effective Space Mission Operations,'
McGraw-Hill, Inc., 1996.

[3] Dustin, E., Rashka, J., Paul, J., 'Automated Software Testing: Introduction, Management and Performance,' Addison-Wesley Publishers, 1999.

[4] Frakes, W., Fox, C., 'Sixteen questions about software reuse,' Communication of the ACM 38, 75-87 and 112. 1995

[5] Larson, Wiley J., Wertz, James R., 'Space Mission Analysis and Design,' Second edition. Microcosm, Inc. and Klunar Academic Publishers, 1992.

[6] Rine, D., Sonnemann, 'Investments in reusable software. A study of software reuse investment success factors,' The Journal of Systems and Software 41, 17-32. 1998.

[7] Rine, D., Nada, N., 'Three empirical studies of a software reuse reference model,' Software – Practice and Experience Journal 30, 685-722. 2000.

APPENDIX OF TABLES

Table 1 Master Test Thread Table

Component Test Threads	Level of Importance	Vehicle Uniqueness	Phase of Lifecycle	Size of Test Thread	Execute LOS or Stored
Attitude Determination and Control Subsystem					
Determine pointing of sensor	**2**	**5**	**2**	**4**	**L**
Mapping the image to the target	**1**	**5**	**2**	**2**	**L,S**
Establish relationship between the sensor optics or antenna pointing and the orientation in space	**4**	**4**	**2**	**3**	**S**
Locate ground reference point on target image	**1**	**5**	**2,3**	**2**	**L,S**
Compute attitude necessary to yield observed displacements of the reference location using an orbital model for specified vehicle	**1**	**5**	**1,2,3**	**3**	**L,S**
Use two ground reference points to determine vehicles roll, pitch, and yaw	**2**	**2**	**2**	**3**	**S**
Target areas with no reference points, i.e. Ocean, compute attitude for the orbit section that was over land and propagate attitude information forward to the point when it is over the ocean.	**1**	**5**	**2**	**2**	**L,S**
Compute geolocation using spacecraft ephemeris information	**5**	**1**	**2**	**4**	**S**
Send geolocation information via command and control link	**5**	**2**	**2**	**4**	**S**
Communications subsystem					
Carrier tracking (2-way coherent and noncoherent communication,1-way communication)	**5**	**3**	**1,2,3**	**3**	**L**
Command reception and detection	**5**	**3**	**1,2,3**	**5**	**L**
Acquire and track uplink carrier	**5**	**4**	**1,2,3**	**5**	**L**
Demodulate carrier and subcarrier	**5**	**3**	**1,2,3**	**5**	**L**

Component Test Threads	Level of Importance	Vehicle Uniqueness	Phase of Lifecycle	Size of Test Thread	Execute LOS or Stored
Derive bit timing and detect data bits	**5**	**4**	**1,2,3**	**5**	**L**
Resolve data-phase ambiguity if it exists	**5**	**4**	**1,2,3**	**3**	**L**
Forward command data, clock, and in-lock indicator to the subsystem for command and data handling	**5**	**3**	**1,2,3**	**4**	**L**
Telemetry modulation and transmission	**4**	**4**	**1,2,3**	**5**	**L**
Receive telemetry data streams from the command and data handling subsystem or data storage subsystem	**4**	**3**	**1,2,3**	**5**	**L**
Modulate downlink subcarrier and carrier with mission or science telemetry	**4**	**3**	**1,2,3**	**5**	**L**
Transmit composite signal to Earth or relay satellite	**4**	**4**	**1,2,3**	**5**	**L**
Ranging					
Detect and retransmit ranging pseudorandom code or ranging tone signals	**4**	**4**	**1,2,3**	**4**	**L**
Retransmit either phase coherently or noncoherently	**4**	**3**	**1,2,3**	**4**	**L**
Subsystem Operations					
Receive Commands from the subsystem for command and data handling	**5**	**3**	**1,2,3**	**3**	**L,S**
Provide health and status telemetry to the C&DH subsystem	**4**	**3**	**1,2,3**	**3**	**L,S**
Perform antenna pointing for any antenna requiring beam steering	**4**	**4**	**1,2,3**	**3**	**L,S**
Perform mission sequence operations per stored software sequence	**5**	**4**	**1,2,3**	**3**	**L,S**
Autonomously select omni antenna when spacecraft attitude is lost	**5**	**4**	**1,2,3**	**3**	**L,S**
Autonomously detect faults and recover communications using stored software sequence	**5**	**4**	**1,2,3**	**3**	**L,S**

Component Test Threads	Level of Importance	Vehicle Uniqueness	Phase of Lifecycle	Size of Test Thread	Execute LOS or Stored
Command and Data Handling Subsystem					
Receive command message	**5**	**2**	**1,2,3**	**4**	**L,S**
Validate command message	**5**	**2**	**1,2,3**	**4**	**L,S**
Reception of synchronization code	**5**	**2**	**1,2,3**	**3**	**L,S**
Check command message length	**4**	**3**	**1,2,3**	**3**	**L,S**
Match spacecraft address	**5**	**3**	**1,2,3**	**3**	**L,S**
Match fixed unused message bits	**4**	**3**	**1,2,3**	**4**	**L,S**
Detect lack of errors using error check polynomial code	**5**	**2**	**1,2,3**	**4**	**L,S**
Update reject or accept message counters	**4**	**4**	**1,2,3**	**5**	**L,S**
Decode command to determine output type and specific interface channel	**5**	**3**	**1,2,3**	**3**	**L,S**
Discrete commands are fixed amplitude and pulse duration	**5**	**2**	**1,2,3**	**3**	**L,S**
High level discrete command used to drive latching relay coil or fire an ordnance device	**5**	**2**	**1,2,3**	**3**	**L,S**
Low level discrete command open collector interfacing with digital logic	**4**	**2**	**1,2,3**	**3**	**L,S**
Serial command 3-signal interface used to indicate interface is active	**4**	**3**	**1,2,3**	**3**	**L,S**
Data handling combines telemetry from multiple sources and provides it for downlink	**3**	**3**	**1,2,3**	**3**	**L,S**
Receives analog telemetry data	**4**	**2**	**1,2,3**	**4**	**L,S**
High-Level Analog : Telemetry Channel with information encoded as an analog voltage which does not need measurement excitation	**4**	**2**	**1,2,3**	**4**	**L,S**
Low-Level Analog: Telemetry Channel with information encoded as a low analog voltage which amplification is needed	**4**	**1**	**1,2,3**	**4**	**L,S**
Passive Analog: A Telemetry Channel with information encoded as a resistance Analog Telemetry is converted digital form	**4**	**1**	**1,2,3**	**4**	**L,S**

Component Test Threads	Level of Importance	Vehicle Uniqueness	Phase of Lifecycle	Size of Test Thread	Execute LOS or Stored
Bi-Level (Discrete) Input: Telemetry Channel representing two state information	**4**	**1**	**1,2,3**	**4**	**L,S**
Serial Telemetry (Digital) Interface: 3-signal interface used to transfer digital data from an external source to data handling equipment	**4**	**1**	**1,2,3**	**4**	**L,S**
Format data into serial stream of continuous data for downlink	**4**	**1**	**1,2,3**	**4**	**L,S**
Optionally provide telemetry data for to on-board computer	**4**	**2**	**1,2,3**	**4**	**L,S**
Electrical Power Subsystem					
Supply a continuous source of electrical power to spacecraft loads during the mission life	**5**	**3**	**2**	**3**	**S**
Control and distribute electrical power to spacecraft	**5**	**3**	**2**	**3**	**S**
Support power requirements for average and peak electrical load	**5**	**3**	**2**	**3**	**S**
Provide converters for ac and regulated dc power buses, optional	**4**	**4**	**2**	**N/A**	**N/A**
Provide command and telemetry capability for EPS health and status	**5**	**5**	**2**	**N/A**	**N/A**
Protect spacecraft payload against failures within the EPS	**5**	**5**	**2**	**3**	**S**
Suppress transient bus voltages and protect against bus faults	**4**	**3**	**2**	**N/A**	**N/A**
Provide ability to fire ordinance, optional	**5**	**4**	**1**	**3**	**S**
Structures and Mechanism Subsystem					
Steady State booster accelerations	**5**	**1**	**1**	**N/A**	**N/A**
Vibro-acoustic noise during launch and transonic phase	**5**	**1**	**1**	**N/A**	**N/A**
Propulsion system engine vibrations	**5**	**1**	**1**	**N/A**	**N/A**
Transient loads during booster ignition and burn-out, stage	**5**	**1**	**1**	**N/A**	**N/A**

Component Test Threads	Level of Importance	Vehicle Uniqueness	Phase of Lifecycle	Size of Test Thread	Execute LOS or Stored
separations, vehicle maneuvers, propellant slosh, and payload fairing separation					
Pyrotechnic shock from separation events, deployments	**5**	**1**	**1**	**N/A**	**S**
Thermal Environments	**5**	**1**	**1**	**N/A**	**S**
Navigation & Guidance Subsystem					
Orbit Maintenance: maintaining orbital elements not timing of satellite location in orbit Stationkeeping: maintaining the satellite in a predefined box	**4**	**3**	**2**	**3**	**L**
Orbit propagation: integrating the equations of motion to determine where at satellite will be at a given time	**4**	**3**	**2**	**3**	**S**
Real-time orbit determination provides where a satellite is at the present time	**4**	**3**	**2**	**3**	**L**
Definitive orbit determination is the best estimate of the satellite position and orbital elements at some earlier time	**4**	**3**	**2**	**3**	**L**

The following table (Table 2) represents the Test Framework Product Line [1, 3, 5, 6]. The criterion, mentioned in the *Plans for interpreting results* section, has been applied to the previous Master Table. The TFPL only contains those test threads that have successfully satisfied the criteria mentioned above.
Table 2 Test Framework Table

Component Test Threads	Level of Importance	Vehicle Uniqueness	Phase of Lifecycle	Size of Test Thread	Execute LOS or Stored
Attitude Determination and Control Subsystem					
Establish relationship between the sensor optics or antenna pointing and the orientation in space	**4**	**4**	**2**	**3**	**S**
Compute geolocation using spacecraft ephemeris information	**5**	**1**	**2**	**4**	**S**
Send geolocation information via command and control link	**5**	**2**	**2**	**4**	**S**
Communications subsystem					
Carrier tracking (2-way coherent and noncoherent communication,1-way communication)	**5**	**3**	**1,2,3**	**3**	**L**
Command reception and detection	**5**	**3**	**1,2,3**	**5**	**L**
Acquire and track uplink carrier	**5**	**4**	**1,2,3**	**5**	**L**
Demodulate carrier and subcarrier	**5**	**3**	**1,2,3**	**5**	**L**
Derive bit timing and detect data bits	**5**	**4**	**1,2,3**	**5**	**L**
Resolve data-phase ambiguity if it exists	**5**	**4**	**1,2,3**	**3**	**L**
Forward command data, clock, and in-lock indicator to the subsystem for command and data handling	**5**	**3**	**1,2,3**	**4**	**L**
Telemetry modulation and transmission	**4**	**4**	**1,2,3**	**5**	**L**
Receive telemetry data streams from the command and data handling subsystem or data storage subsystem	**4**	**3**	**1,2,3**	**5**	**L**
Modulate downlink subcarrier and carrier with mission or science telemetry	**4**	**3**	**1,2,3**	**5**	**L**
Transmit composite signal to Earth or relay satellite	**4**	**4**	**1,2,3**	**5**	**L**
Ranging					
Detect and retransmit ranging pseudorandom code or ranging tone signals	**4**	**4**	**1,2,3**	**4**	**L**

Component Test Threads	Level of Importance	Vehicle Uniqueness	Phase of Lifecycle	Size of Test Thread	Execute LOS or Stored
Retransmit either phase coherently or noncoherently	**4**	**3**	**1,2,3**	**4**	**L**
Subsystem Operations					
Receive Commands from the subsystem for command and data handling	**5**	**3**	**1,2,3**	**3**	**L,S**
Provide health and status telemetry to the C&DH subsystem	**4**	**3**	**1,2,3**	**3**	**L,S**
Perform antenna pointing for any antenna requiring beam steering	**4**	**4**	**1,2,3**	**3**	**L,S**
Perform mission sequence operations per stored software sequence	**5**	**4**	**1,2,3**	**3**	**L,S**
Autonomously select omni antenna when spacecraft attitude is lost	**5**	**4**	**1,2,3**	**3**	**L,S**
Autonomously detect faults and recover communications using stored software sequence	**5**	**4**	**1,2,3**	**3**	**L,S**

Component Test Threads	Level of Importance	Vehicle Uniqueness	Phase of Lifecycle	Size of Test Thread	Execute LOS or Stored
Command and Data Handling Subsystem					
Receive command message	**5**	**2**	**1,2,3**	**4**	**L,S**
Validate command message	**5**	**2**	**1,2,3**	**4**	**L,S**
Reception of synchronization code	**5**	**2**	**1,2,3**	**3**	**L,S**
Check command message length	**4**	**3**	**1,2,3**	**3**	**L,S**
Match spacecraft address	**5**	**3**	**1,2,3**	**3**	**L,S**
Match fixed unused message bits	**4**	**3**	**1,2,3**	**4**	**L,S**
Detect lack of errors using error check polynomial code	**5**	**2**	**1,2,3**	**4**	**L,S**
Update reject or accept message counters	**4**	**4**	**1,2,3**	**5**	**L,S**
Decode command to determine output type and specific interface channel	**5**	**3**	**1,2,3**	**3**	**L,S**
Discrete commands are fixed amplitude and pulse duration	**5**	**2**	**1,2,3**	**3**	**L,S**
High level discrete command used to drive latching relay coil or fire an ordnance device	**5**	**2**	**1,2,3**	**3**	**L,S**
Serial command 3-signal interface used to indicate interface is active	**4**	**3**	**1,2,3**	**3**	**L,S**
Data handling combines telemetry from multiple sources and provides it for downlink	**3**	**3**	**1,2,3**	**3**	**L,S**
Electrical Power Subsystem					
Supply a continuous source of electrical power to spacecraft loads during the mission life	**5**	**3**	**2**	**3**	**S**
Control and distribute electrical power to spacecraft	**5**	**3**	**2**	**3**	**S**
Support power requirements for average and peak electrical load	**5**	**3**	**2**	**3**	**S**
Provide converters for ac and regulated dc power buses, optional	**4**	**4**	**2**	**N/A**	**N/A**
Provide command and telemetry capability for EPS health and status	**5**	**5**	**2**	**N/A**	**N/A**
Protect spacecraft payload against failures within the EPS	**5**	**5**	**2**	**3**	**S**
Suppress transient bus voltages and protect against bus faults	**4**	**3**	**2**	**N/A**	**N/A**

Component Test Threads	Level of Importance	Vehicle Uniqueness	Phase of Lifecycle	Size of Test Thread	Execute LOS or Stored
Provide ability to fire ordinance, optional	**5**	**4**	**1**	**3**	**S**
Structures and Mechanism Subsystem					
Steady State booster accelerations	**5**	**1**	**1**	**N/A**	**N/A**
Vibro-acoustic noise during launch and transonic phase	**5**	**1**	**1**	**N/A**	**N/A**
Propulsion system engine vibrations	**5**	**1**	**1**	**N/A**	**N/A**
Transient loads during booster ignition and burn-out, stage separations, vehicle maneuvers, propellant slosh, and payload fairing separation	**5**	**1**	**1**	**N/A**	**N/A**
Pyrotechnic shock from separation events, deployments	**5**	**1**	**1**	**N/A**	**S**
Thermal Environments	**5**	**1**	**1**	**N/A**	**S**
Navigation & Guidance Subsystem					
Orbit Maintenance: maintaining orbital elements not timing of satellite location in orbit Stationkeeping: maintaining the satellite in a predefined box	**4**	**3**	**2**	**3**	**L**
Orbit propagation: integrating the equations of motion to determine where at satellite will be at a given time	**4**	**3**	**2**	**3**	**S**
Real-time orbit determination provides where a satellite is at the present time	**4**	**3**	**2**	**3**	**L**
Definitive orbit determination is the best estimate of the satellite position and orbital elements at some earlier time	**4**	**3**	**2**	**3**	**L**

Evaluating the Kaleidoscope Product-Line Architecture for Monitoring and Control Systems

Andrea Savigni
Department of Computer Science
University College London
Gower Street
London WC1E 6BT United Kingdom
+44 (0)20 7679 3699
a.savigni@cs.ucl.ac.uk

ABSTRACT
This paper presents an evaluation of the Kaleidoscope product-line architecture, that has been successfully used over the past few years for building monitoring and control systems in various application domains. The architecture is based on the separation of communication, computation, and strategy in order to foster modularity and reuse. Two case studies are briefly presented.

Keywords
Software Engineering, Software Architecture, Monitoring and Control Systems, UML, Reflection, Real-Time Systems

1 RATIONALE FOR AND OVERALL AIMS OF THE WORK

Building monitoring and control systems is a very challenging activity, as these systems present software engineers with a number of complicated issues, such as distribution, timeliness, scalability, complexity of configuration.

While the literature does contain some examples of reference architectures for this class of systems, such examples are usually not general enough, in that they tend to be confined to specific application domains or make strong assumptions about the environment. This is for example the case of the well-known NASREM architecture [2, 3], that is specifically oriented towards robot telecontrol, and assumes global memory, which is not always possible to implement. Others, such as [5] are oriented towards embedded systems. Others (e.g., [10]) are oriented towards simulation rather than actual control.

All in all, the literature does not contain any standard, established, specific product-line architectures for this class of systems. Thus, these systems are usually built on an ad hoc basis, which severely hinders reusability and modularity.

The overall aim of our work was to establish a general framework for monitoring and control systems. This framework is composed by:

- Kaleidoscope [15, 16], a product-line architecture that constitutes the object of this paper;
- a design methodology to assist engineers in building monitoring and control systems based on that architecture;
- an object-oriented framework containing a set of ready-to-use building blocks that engineers can exploit when building new Kaleidoscope-based systems.

This paper will only cover the first point (i.e., Kaleidoscope); the other two will not be taken into consideration due to space limitations.

Kaleidoscope was developed using UML [14]. Thus, it will be described in the sequel using standard UML terminology. Sections 2 through 4 briefly set forth the fundamental ideas the architecture is based upon. Some specially important principles are particularly emphasised as such. Section 5 compares the approach with related work. Section 6 presents an evaluation of the approach. Finally, Sect. 7 draws come conclusions and sketches the future developments.

2 IMAGES AND LAYERS

The Kaleidoscope architecture is based upon the following principle:

Principle 1 *The fundamental activity of a monitoring and control system is to align different* images *of the same application-domain entity, represented at different abstraction layers.*

A fundamental distinction is made between *conceptual images* and *concrete images*. A conceptual image is a class that models entities of the application domain (e.g., a CO_2 sensor). Attributes and associations of a conceptual image model all the information related to the domain entity at all

abstraction levels. For example, the conceptual image of a CO_2 sensor might include `currentValue`, `history`, `average`, and so on. An object in the application domain (e.g., `sensor24`) is modeled by an instance of a conceptual image. Conceptual images also include the appropriate methods for computing derived values from more elementary ones.

A concrete image is the realisation of a conceptual image at a specific abstraction layer. As such, it typically contains a subset of the attributes found in its "parent" conceptual image. A concrete image exports an appropriate subset of the methods defined by its corresponding conceptual image. Components [18] are defined as collections of instances of concrete images.

A "standard" set of images has been identified, which can be customised at will by the designer. This includes an acquisition image (basically a driver of the acquisition device), a processing image (which is the main locus of computational activity, including for example statistical processing or decision support), a presentation image (whose goal is to export information out of the system in the form of graphical representation, geographic transmission, etc.), a persistent image (that connects to an external DBMS to permanently store data), an activation image (basically a driver of the activation device), and a simulation image (that feeds the system with simulated data in case of unavailability of the "real" data or for what-if analyses). From this picture, some sort of symmetry clearly emerges between monitoring (the *upstream* information flow) and control (the *downstream* information flow). This does not happen by chance, but is the expression of a general principle:

Principle 2 *Monitoring and control are to be considered as conceptually equivalent. They are simply two complementary aspects of information alignment.*

Particular care has been taken in order to make system configuration feasible by a domain expert rather than a software engineer. To that aim, a rather sophisticated, reflective [12] mechanism has been devised, based on *facets* [15] i.e., small objects that implement image attributes, and that can be added to images at configuration time rather than at compile time. In this activity, several concepts borrowed from reflection play a major role. A Java-based (prototype) tool was implemented to render this activity as easy as possible.

3 SEPARATING DISTRIBUTION FROM COMPUTATION

A monitoring and control system is most likely a distributed one. Thus, one crucial problem is how to deal with distribution issues in a systematic and general way. This leads to the following

Principle 3 *In a well-designed monitoring and control system, information alignment and information processing should be kept strictly separate.*

In other words, information processing should always be performed on local concrete images. The above principle is but one instance of the general principle of separation of concerns. In this case, it means that distribution issues and "semantic" issues (i.e., computation that is directly related to the application domain) should not be mixed.

Entities concerned with image alignment are called *projectors*. A projector connects two concrete images, generally residing on different nodes, and takes care of all (and only) the alignment issues. In particular, projectors completely encapsulate low-level communication details, such as underlying middleware (if any), network protocols, and the like. Projectors, like images, are passive entities. They export methods that allow the strategic components (see Sect. 4) to perform information alignment.

In order to further foster reuse, Kaleidoscope draws on the concept of Architectural Reflection [7, 6, 8, 20], introducing a meta-level representation for topology and strategy (see also Sect. 4). The system maintains a meta-representation of its own architecture, hosted by an entity (the *topologist*) which is separate from images and projectors. These, in turn, have no knowledge of the overall system architecture. In reflective parlance, the topologist *reifies* the system's architecture into some internal representation, which is maintained at run-time. The use of reflection has the goal of achieving *transparency*, in that some aspects of the system architecture can be changed at run-time (for instance by adding or removing a projector) without the components noticing.

4 STRATEGIC ISSUES

The next question to be answered is: Who, and when, commands image alignment? This leads to the following:

Principle 4 *Neither images nor projectors should embed any activation strategy. All strategic issues should be dealt with by separate entities.*

This principle has the goal of maximizing code reuse. Since images and projectors are passive entities, they can be reused under many different strategies. A separate entity, called *strategist* [17], takes care of activating projectors (to align information) and images (to process information). All communication among images happens through commands issued by a strategist on a projector; this, in turn, extracts information from its source image and transfers it to its target image. The strategist reifies the system's behaviour into an internal representation, any change of which reflects back into the system in the form of changes in the latter's behaviour.

To summarise, Kaleidoscope defines three major aspects of a system, namely computation, distribution, and activation, each of which is orthogonal, and should be kept separate, with respect to the others.

Finally note that, although the strategist is usually referred

to as one entity, it can have any implementation, from a simple, centralised component to a complex distributed system made up of timers, reactive components, time-driven engines, and so on. The fundamental principle is that it must be kept separate from images and projectors. Kaleidoscope's strategic aspects are dealt with in more detail in [17].

5 RELATED WORK

The principles that lie at the base of Kaleidoscope have obviously been influenced by several contributions, starting from the very idea of programming-in-the-large [9].

The idea of separating computation from distribution and execution strategy is not new. The field of Coordination Models and Languages contains several examples of such systems, for example Linda [1], JavaSpaces [19], and Manifold [4]. Kaleidoscope builds on many of the concepts underlying these systems (in particular Manifold's strict separation of computation from communication) to define a *concrete* product-line architecture for monitoring and control systems.

Our approach is also close to the field of Object-Oriented frameworks [13], of which it might be considered an example. However, one important difference with respect to that discipline is the fact that Kaleidoscope keeps a (logically) centralised description of the system architecture at runtime, which is generally not the case in frameworks. As anticipated in Sect. 1, an object-oriented framework has been developed to be use as a support for building systems based on the Kaleidoscope product-line architecture.

Obviously, the whole structure of Kaleidoscope was heavily influenced by reflection, as already anticipated above. This is true in particular of meta-level representations, covered in Sect. 3.

Finally, the whole communication scheme, in which information alignment is triggered by the communication system, rather than by components, has several similarities with Kopetz's Time-Triggered Protocol [11].

6 EVALUATION OF THE APPROACH

Since Kaleidoscope claims to be a product-line architecture for building actual software systems, it would be useless if it remained at the conceptual stage. For the same reason, it is the authors' opinion that evaluating this approach can only be done "inductively" i.e., on the basis of the success of actual software systems based on the architecture. Apart from several minor prototypes, two major projects have been developed based on it. The first project consists of a very large and complex integrated traffic monitoring and control centre for a major European city. Design has been successfully completed and an early implementation is being carried on.

The second project involves the design of an indoor environmental monitoring system employing innovative sensors and wireless communication. Several major international industrial and academic partners participate in the project, in such diverse areas as climate control, telecommunications, solid state physics. The project is by now practically finished.

These two projects, in such diverse areas, gave us the opportunity to test the overall approach and the Kaleidoscope product-line architecture in particular. This practical evaluation gave sound and very promising results; more precisely:

- the use of different abstraction levels greatly helped manage the complexity of the application domain. This was especially true of the traffic control application, in which the sheer number of domain entities to be taken into consideration was scary;
- the separation of computation and distribution was crucial, in that it allowed us to consider one issue at a time when designing systems. In particular, in the case of the environmental monitoring system we had to provide three incremental prototypes, employing respectively a centralised machine, a standard TCP/IP LAN, and wireless transmission. By the way, this is a general and reasonable requirement on the part of project manager, as it allows to spread risks across successive releases. The separation of computation and communication made it possible to reuse the very same software components in all three situations;
- keeping strategy separate also greatly helped in the environmental monitoring system. In fact, two different sensor working modes were explicitly required: low consumption, employing batteries, and high consumption, in which sensors were simply using mains. This problem turned out to be surprisingly hard, as it involved a complete change in the activation strategies and also in the communication paradigms. Having strategy embedded in components and/or connectors would have meant providing two different sets of components and connectors (i.e., practically two different systems) according to the specific functioning mode. Our solution, on the contrary, allowed us to change modality on the fly, thus yielding an actual non-stopping system (which is frankly more than required and expected);
- the configuration activities could largely be assigned to domain experts, which greatly reduced the time-to-market.

These positive experiences, and the fact that they were achieved in such different application domains, lead us to think that a sound product-line architecture is invaluable towards achieving large-scale reuse across applications. Based on our practical, industrial experience the same level of reuse could not be gained through bespoke systems. In a sense, a sound product-line architecture is a fundamental

step in the transition from craftsmanship to actual engineering practice, in that it helps to code and establish a vast amount of knowledge and experience that would otherwise remain confined in the designers' own professional experience.

7 CONCLUSIONS AND FUTURE DEVELOPMENTS

This paper presented the Kaleidoscope product-line architecture for monitoring and control systems, and evaluated its soundness in the light of two real-life, industrial projects. The success of this particular architecture encourages to think of product-line architectures as means to capture and formalise a vast amount of domain knowledge and expertise in order to reuse it across different projects.

As anticipated in Sect. 1, current research is focused on a design methodology that serves as a guide for designing monitoring and control systems using Kaleidoscope. The methodology focuses on automating the transition from analysis (that certainly requires an expert human being) to design, so that the designer is relieved of much of the burden usually associated with detailed design activities. To that aim, it makes an extensive use of advanced UML features such as stereotypes to tag conceptual images in order for a tool to be able to derive design diagrams from analysis ones in a (partially) automated way.

ACKNOWLEDGEMENTS

Since its inception, the Kaleidoscope product-line architecture was developed in close cooperation with Prof. Francesco Tisato. Without his support and guidance it would have never come into being. I also wish to thank the team who cooperated with me in the "second project" (see Sect. 6), and particularly Daniela Micucci.

REFERENCES

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *Computer*, 8(19):26–34, august 1986.

[2] J. Albus, R. Lumia, J. Fiala, and A. Wavering. NASREM – The NASA/NBS Standard Reference Model for Telerobot Control System Architecture. In *Proceedings of the 20th International Symposium on Industrial Robots*, Tokyo, Japan, October 4–6 1989.

[3] J. Albus, H. McCain, and R. Lumia. NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM). Technical Report NISTTN 1235, 1989 Ed, National Institute of Standards and Technology, Gaithersburg, MD, April 1989.

[4] F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and Its Implementation. *Concurrency: Practice and Experience*, 50(1):23–70, Feb 1993.

[5] P. Binns, M. Englehart, M. Jackson, and S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation and Control. *IEEE Transactions on Software Engineering*, 20, 1994.

[6] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. A Fresh Look at Programming-in-the-Large. In *Proceedings of COMPSAC '98*, Vienna, Austria, Aug 13-15 1998.

[7] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification. In *Proceedings of Euromicro 1998*, Florence, Italy, March 8-11 1998.

[8] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Rule-Based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level. In *Proceedings of Automated Software Engineering – ASE'99*, Cocoa Beach, Florida, USA, Oct 12-15 1999.

[9] F. DeRemer and H. H. Kron. Programming-in-the-large versus Programming-in-the-small. *Transactions on Software Engineering*, SE-2:80–86, June 1976.

[10] L. G. Huang. Simulating Traffic Control Systems with the Flip-Tick Architecture. In *Proceedings of the IEEE 1999 International Conference on Systems, Man, and Cybernetic*, Tokyo, Japan, October 12-15 1999.

[11] H. Kopetz and G. Grnsteidl. TTP – A Protocol for Fault-Tolerant Real-Time Systems. *Computer*, pages 14–23, January 1994.

[12] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, volume 22 of Sigplan Notices, pages 147–156. ACM, Oct. 1987.

[13] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.

[14] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, December 1998.

[15] A. Savigni and F. Tisato. Kaleidoscope. A Reference Architecture for Monitoring and Control Systems. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, USA, February 22-24 1999.

[16] A. Savigni and F. Tisato. Designing Traffic Control Systems. A Software Engineering Perspective. In *Proceedings of Rome Jubilee 2000 Conference (Workshop on the International Foundation for Production Research (IFPR) on Management of Industrial Logistic Systems – 8th Meeting of the Euro Working Group Transportation - EWGT)*, Roma, Italy, September 11–14 2000.

[17] A. Savigni and F. Tisato. Real-Time Programming-in-the-Large. The Case of Monitoring and Control Systems. In *Proceedings of ISORC 2000*, Newport Beach, CA, USA, March 15–17 2000. IEEE.

[18] M. Shaw and D. Garlan. *Software Achitecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[19] Sun Microsystems. The JavaSpaces Technology. http://java.sun.com/products/javaspaces/.

[20] F. Tisato, A. Savigni, W. Cazzola, and A. Sosio. Architectural Reflection. Realising Software Architectures via Reflective Activities. In *Proceedings of the 2nd Engineering Distributed Objects Workshop (EDO 2000)*, Davis, California, USA, November 2–3 2000.

People Issues in Developing 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

In this paper we discuss the importance of people and their behavior in the introduction and continuous operation of product lines. We give some hypotheses on how these people issues can be formed in a way favorable to product line development. In particular, we focus on the relation between the *mind set of people*, *communication patterns*, and the *organizational structure*. The hypotheses we provide are derived from our own industrial experience as well as from industry reports. However, this work is in an early stage and mainly aims at contributing to an ongoing discussion of technology transfer in a product line context.

Keywords

Organizational Structure, Communication Patterns, Technology Transfer

1 INTRODUCTION

In this paper we aim at a contribution towards answering the question: *how can we successfully support organizations in their shift towards product line development*?

In current research on product line development, the focus is often set on technical issues: *what is the best way to do scoping, how can we best capture results of domain analysis, how can we support the description of variability in reference architectures, etc.* [12, 13]. While these are very important questions to ask (and to answer), they leave out what we regard as a major aspect of technology transfer in the context of product line development: the importance of people.

Thus, in this paper, we want to focus exclusively on people issues. From our work in transferring software product line approaches into industrial practice we often saw how important people issues are [6, 9]. This is especially true, if we focus on transforming a single system company into a product line organization.

Driven from the motivation of technology transfer, we focus on the question of what does it mean to get the people issues right and what are its implications on technology transfer. While we try to give initial answers to this question, our main goal is the clarification of the problem. In Section 2 we describe in more detail the issues on which we want to focus here. In Section 3, we will discuss what we learned from our experiences as well as from other reported experiences on technology transfer in product line development. In Section 4, we will summarize our main hypotheses as success factors for product line technology transfer. Finally, in Section 5 we will conclude.

2 SCOPE OF THE ANALYSIS

People issues are already a rather broad are. However in our discussion here, we actually make it even larger, as we will also include communication aspects, management issues, and organizational concepts. Basically, every experience report (e.g., [3, 4, 5, 9]) as well as every discussion on technology transfer issues (e.g., [1, 9]) addresses these issues as well. What is missing from our point of view is a description of how these aspects interact into providing a supportive environment to product line development and comparing this with experience from the field. One of the goals of this paper is to provide an initial model of this interaction.

Our *hypothesis*, which was derived from our own experience in technology transfer, is:

> **The dominating success factor for product line development is the mind set of people.**

With this we mean, that only if we are successful in making aspects like reuse or the integrity of the product line an important goal for the individual developer, we will be successful in making product line development a reality in the corresponding organization. Luckily, this does not mean that we are at the mercy of what is on the mind of people in the development organization, but we assume that this mind set can be actively influenced. Actually, we believe now that it has to be changed on the way from a single system organization to a product line organization in order to make technology transfer successful.

Here, we will focus on two main aspects: *communication patterns* – what is communicated to whom – and *organizational structure* – which stable role assignments are given to which person – and what dependencies exist among the people. From our point of view dependencies exist among these aspects as is illustrated in Figure 1. However, this picture is only meant to be illustrative, not a complete description of all relationships. For example, we do not make the assumption that the mind set is *only* influenced by the communication patterns. We will actually discuss further influences in this paper. The picture is meant to highlight the three main areas (mind set, communication patterns, and organization) which we will discuss in this paper and their relation.

In this paper, we mean with *mind set* those aspects like

beliefs, attitudes, priorities and values, that are driving forces in determining the behavior of the developers. We will focus on the specific priorities and believes people have in mind, which in turn determines what they focus upon in their work and where they do invest their time in.

The assumption in Figure 1 is that the mind-set is influenced by soft factors, in particular, the *communication* people are involved in.[1] Typically, in a work-place environment these communications are structured in a certain way. Certain types of information are typically communicated with certain people depending on their needs and interests. If these relations are somewhat stable over time and not people-specific we term them *communication patterns*. A typical communication pattern in a non-product line organization (which is not derived from the organizational structure) is that developers mourn to each other that they have to sacrifice the integrity of the architecture for making a deadline.

The communication patterns are partially determined by the overall work organization, as they influence what information needs to be communicated to which person. The *organizational structure* determines who is concerned with which part (functionality-wise) and which aspect (life-cycle perspective) of the product line. Again, this influence is not a complete one. Rather, the organizational structure provides constraints on the overall communication patterns. Thus, the organizational structure has also in an indirect manner a strong impact on the mind set of people.

The fact that the organization structure, the communication patterns, and the resulting products a company develops are strongly interrelated has already been emphasized by the classical quote by Conway [15]:

> **Organizations which design systems are constrained to produce systems which are copies of the communication structures of these organizations**

Here, we advocate that this relationship can be extended to product line engineering: our core problem is to find adequate organization forms and communication structures that mirror the needs of product lines.

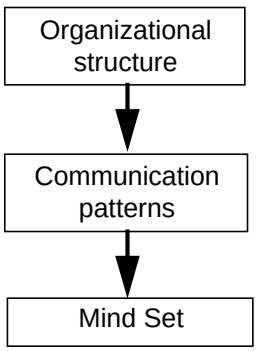


Figure 1. Schema of influences of soft factors

1. It is also influenced by previous experiences, the raising of the people, etc. Aspects that are well studied in psychology, but beyond the scope of this paper.

3 DEVELOPING A PRODUCT LINE PERSPECTIVE

In the previous section we discussed how mind set, communication, and organization interrelate. In this section, we discuss in more detail how they interact in a product line context and how they should be formed in order to support the introduction of product line development.

3.1 Mind Set

As we discussed above, we regard the intrinsic motivation of people – or as we called it above: the mind set of people – a key aspect for the successful development of a product line. In particular, we believe it to be necessary that people show three main attitudes:

1. They aim at avoiding rework, i.e., they work pro-actively towards making it easier to integrate future products. In order to enable this we need to make the product line visible to the people and even more to make it a personal experience to people that they can reduce the amount of effort they need by having this attitude.[2]

This attitude is intrinsically linked to the following two aspects:

2. People should focus on the product line instead of the product, i.e., the predominant question for them should be: what does the change I am about to make mean for the whole set of products our organization is developing?

3. The people should value the product line higher than the individual product. An obvious sign for this is the willingness to sacrifice a product deadline over the conceptual integrity of the product line. This is of course something that needs to be supported and communicated by management, as we will discuss below.

All these attitudes taken together form a common principle:

> **Developers need to feel responsible for the well-being of the product line.**

3.2 Communication

However, the mind set can usually not be influenced directly. Thus, we need to influence it – to the extent possible – through communication. This holds both for a technology transfer situation, where people need to be acquainted with these attitudes in a step-wise manner, as well as during the continued development of a product line, where these attitudes need to be supported and kept up.

Thus, these attitudes need to be continuously communicated to the product line development personal. The following aspects are particularly important to communicate:

- People need to feel involved and need to be convinced.
- The product line needs to be made visible to people.

The following means can help to communicate this. (Here, we focus on their communication aspect):

2. Note, that we emphasize rework instead of reuse here for two reasons: first we regard reuse as a means to an end, namely saving effort; second this was the attitude that we experienced in successful product line organizations.

1. Trainings
2. Champions
3. Management messages
4. Involving practice

Each of these communication forms has its unique set of advantages and disadvantages. Out of these different forms *trainings* are probably the most straightforward. In the context of technology transfer they are typically used when people are trained for new technical activities that are particularly relevant for product line development. Typical of this are trainings on domain analysis or architecture concepts. This can and obviously should be used to reinforce the need for product line development and to understand any problem people have with this shift in attitude, thus making it a two-way communication. The advantage of this communication form is that the time of people is explicitly reserved for this activity allowing them to focus on it. Also in such a forum any difficulties and problems can be addressed more directly than in every-day practice. The disadvantage of trainings is that this is typically a one-time activity, which is on a comparably abstract level for the developers, as it predates their practical involvement. It will only show results, if the communicated information and attitudes are found practical in daily experience (cf. *involving practice*).

Different interpretations of the term "*champion*" are used in literature. Here, we mean a person with strong social and communication skills, who is in continuous contact with the development personal and is in favor of product line development ideas and communicates this strongly and convincingly. The champion may have, but need not have a management role (cf. *management messages*). The champion helps to keep up commitment for product line practices and helps other people in seeing the product line. There are no disadvantages of champions as far as we can see. However, there is not always an appropriate person available in the environment. In the worst case it can also be that a person who would have the necessary skills to fit this role is a strong opponent of product line practice. This is about the worst thing that can happen to such a program.

As very important for changing practices in an organization we found the *messages management* sends. With *messages* we do not mean so much the general speech management is giving but more the indirect signals it is sending. For example, suppose a manager strongly claims support to product line concepts. However, he is willing to sacrifice the product line integrity by making architectural shortcuts in order to meet a product deadline. This message will come across and will have a strong effect on the people, as they will also feel evaluated on the possibility of meeting the deadline instead of contributing to the product line. Similar messages are:

- *sufficient funding* – does management provide sufficient funding (both money and personnel) to support product line development.
- *a strong product line vision* – e.g., management is willing to forgo a customer contract that does not fit the product line vision.

From our experience, as well as from other experience reported in literature [1, 4, 5] we believe that management commitment is a key factor in product line success. This is also related to the organizational structure, which we will discuss in Section 3.3.

While all aspects mentioned above are contributing to a product line attitude in an organization, we believe the aspect of *involving practice* to be key. With the term *involving practice* we mean: the way of working the individual person experiences in everyday practice. We believe from our experience that only if this changes – and people can relate benefits to the change – they will accept and support the change to product line practice.

Practice strongly depends on management decisions and in particular the organization of work (cf. Section 3.3). Other important aspects are the personal involvement in product line aspects and the personal benefit people achieve through product line thinking.

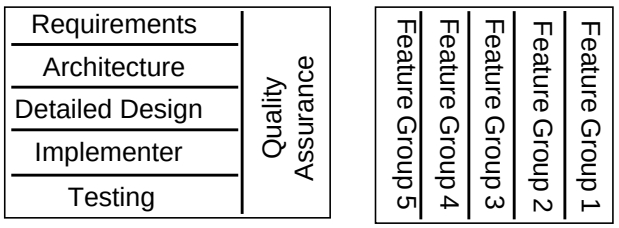
In order to make the product line visible we need to achieve a common understanding among people involved in the development of the different products. This is often a difficult task as people may have different vocabularies. Some researchers regard the development of a common vocabulary as a core task of domain analysis (e.g., [4, 7]). If a common terminology has been established, the exchange of people between product development teams is another way of making the product line visible throughout the organization. However, achieving product line visibility is strongly related to the aspect of organization, thus we will discuss it in more depth in the following section.

3.3 Organization

Here, we do not restrict ourselves to the "official" organization, but we look at any form of somewhat stable role-relationships. This is a broader view as is taken in other work on this topic (e.g., [11]). The organization of work plays a predominant role in structuring the practice of people and what they need to communicate about. In particular, this has a strong impact on whether they will be able to see the product line. The tasks people have to work on, the problems they have to tackle, etc. are strongly influencing the perspective of people (i.e., their mind set), as they will determine whether considering product line issues is also the best solution as an individual.

Here, we want to distinguish *formal* and *informal organization*. The formal organization determines who is officially in charge of what aspect of work, who has management responsibility for which part of the organization, how are the management hierarchies structured, etc. A good overview of different formal organizational models that are used in the context of product line development is given in [2]. On the other hand we have the informal organization, relating to temporary working groups, as well as semi-official bodies. These are typically not found on the organization's structure chart, but they have a strong impact on the communication patterns in the organization (an example of this is the architectural steering board in the Owen approach [3]).

Two fundamentally different possibilities for constructing a software development organization exist: phase-wise and



a) Phase Organization b) Feature Organization

Figure 2. Organizational Prototypes

feature-based organization, which map roughly to the process- and product-oriented organizations as they are termed in [14]. If we look at the formal organization within a product unit, we often see a phase-based organization. However, from our experience, as well as from existing success stories on product line development, we came to the conclusion that in terms of an organizational structuring the predominant structure should be based on features, not development phases. Note, that these are extreme cases. Real organizations usually lie in-between. For example, the per-feature organization is typically restricted to detailed design, implementation, and unit testing (where a single person can be responsible for more than one of these phases).

In a featured-oriented organization certain development groups are responsible for certain aspects of the software in terms of functionality. This organization mode is particularly meaningful in the case that the same group is responsible for this functionality across different products. An example of this is the Owen-organizational model (cf. [3]) where the same person is always responsible for the same code-module. We also saw this working in practice, where across a range of different products the same person was always responsible for the same functionality.

From our point of view the feature-based organization holds important promises for product line development. As the same people are charged with developing similar functionality for different products, this leads to two important effects:

- It becomes easier to ensure the integrity of the architecture as the same person responsible for a part of it also makes changes to it.
- People experience a *personal* benefit in their work by focusing on reusability aspects.

As the organizational modes described above are rather extreme, for large development projects a combined mode of operation is typically the most appropriate.

For example, a certain group might be responsible for the overall requirements and the customer relationship, detailed requirements, design and implementation are developed by a different group, structured according to features (main functionality areas). Integration of the software as well as end-user testing (as opposed to module level testing) is routinely made by a different group which may actually overlap with the first group. Overall architectural integrity of the product line may be ensured by designated architects or an architectural steering board composed of members of the second group.[1] Similar approaches are exhibited by the Owen approach [3] which implies having an architectural steering board composed of members of the different product teams. Another approach is the organization used at CelsiusTech [5] which has only requirements personnel in the application engineering units, while development is completely done in the domain engineering unit.

The organization forms exhibited by CelsiusTech [5] and Owen [3] are actually two ends of a spectrum as is shown in Figure 3, which is based on a figure from [3]. In extreme cases, dedicated application engineering personnel can vanish completely [5]. This maps to the domain engineering unit model as described in [2]. However, if we look at the two cases more closely, we see that while the large difference holds for the formal organization, in terms of the informal organization the two cases are actually rather close together, as in terms of the informal organization both companies mostly adher to an organizational model based on features.

From the case studies we analyzed [3, 4, 5, 7], as well as from our own industrial experience [6, 8, 9], we came up with the following organizational rule:

> **The larger the reuse potential in a product line and the larger the exploitation of this potential, the stronger should feature-orientation dominate the organizational structure.**

As a corollary, we can also say that *the domain engineering*

1. This highlights also a different aspects: it is a requirement that an organizational body – be it a manager or a board of people feels responsible for the product line. We are aware of cases, where assigning a manager as the head of a group of product units brought already a shift towards product line reuse.

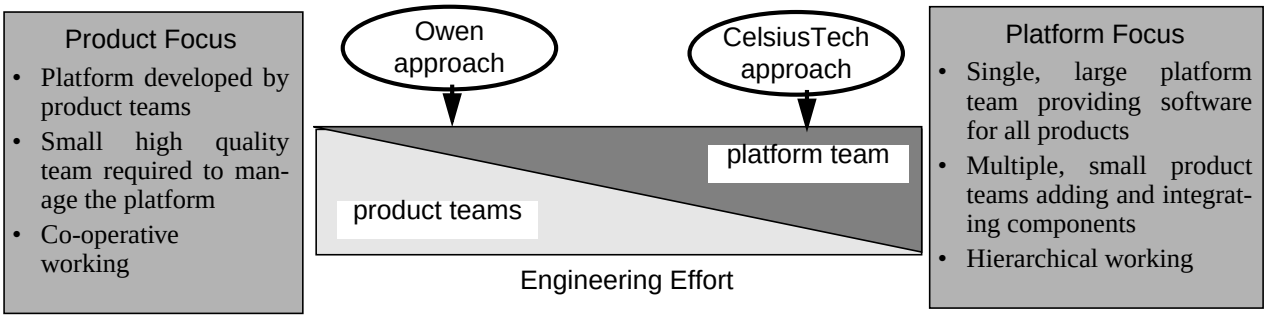


Figure 3. Product focused vs. platform focused development organizations [3]

unit should be structured in a feature-oriented way.

Besides being able to experience personally the need for reusability, the organization structure should also enable the developers to see the product line at large.

First of all, this implies that a large range of people can contribute to what is regarded as the product line both now and in the future. This implies that the product roadmap is not only defined by management and marketing. In the context of Cummings [4], this was emphasized as a major success factor. At best, the involvement of people is direct, or at least a stakeholder of each group should be involved. Whatever approach is used, it is of key importance to involve as many stakeholders as possible [7]. If the groups are rather large, then again we can make use of a feature-based decomposition. The different people in a group of developers are made responsible for different functionality areas and then contribute accordingly to the different across-product groups. Thus, for each product division stakeholders are integrated into the definition of each part of the product line description and everybody (or at least most people) are integrated in this process.

This broad involvement is particularly important so that across an organization a common language can be discovered. But stakeholder involvement has reportedly an important effect in the context of transition to a product line as well:

- It helps to reduce the fear of change people have.
- It makes them feel responsible for the results.
- It helps to avoid the *not-invented-here* syndrome.

4 SUCCESS FACTORS FOR TECHNOLOGY TRANSFER

In this section, we want to summarize the main principles that can be extracted from the discussion given above. From our point of view these should be key concerns when introducing product line development in an organization:

- In the case of small organizations (less than 10 people): bring them together.
- For larger organizations select subsets for the different activities (e.g., developing domain analysis)
  - assign people based on topics
  - select representatives (for very large organizations) to do the modeling
- In case of multiple divisions, exchange personal
- Have the same people responsible for the same functionality (organization follows software architecture), this implies a restricted use of a phase-wise work-breakdown.
- Get somebody in charge of the product line
  - this may be a managerial role *or*
  - a steering board with members from the different product units, which has at least management backing
- A clear product line vision (what can be future products) should be established, and management should be aware that it itself is evaluated based on its adherence to this vision.

In terms of organizational principles we were surprised to discover that the organizations do not necessarily adhere to the established principles of domain vs. application engineering units [2].

For example in the Owen case study [3] a domain engineering unit hardly exists. Architectural integrity is only assured by a steering board. On the other hand in the CelsiusTech case study [5] an application engineering unit hardly exists. Application engineering is only responsible for capturing the customer requirements. All development is performed in the domain engineering unit. Thus, product development can be performed successfully rather independently of the formal organization. However, as we discussed above in all the cases we regard as successful, the informal organization was structured according to the aforementioned principles.

While the success factors we presented here were derived in a more analytical manner, they are in good agreement with success factors derived from large-scale field studies [10].

5 CONCLUSIONS

In this paper, we discussed the aspect of people issues in transitioning to and continuously operating a successful product line project. In particular, we focused on the mind set of people and its relation to communication patterns and organization. We based our analysis of these issues on our individual experience in this area and existing case study reports.

From our discussion we derived some success factors we regard as key aspects in introducing product line development. We expect to discuss the importance of these issues in practice with the participants of the workshop.

REFERENCES

[1] Thomas Wappler. *Remember the Basics*. In: Software Product Lines—Experience and Research Directions, Proceedings of the First Software Product Lines Conference (SPLC1). Ed. Patrick Donohoe, pp.73–84, Kluwer Academic Publishers, 2000.

[2] Jan Bosch. *Organizing for Software Product Lines*, In: Third International Workshop on Software Architectures for Product Families (IWSAPF-3), pp.126–143, 2000.

[3] Peter Toft, Derek Coleman, and Joni Ohta. *A Cooperative Model for Cross-Divisional Product Development for a Software Product Line*. In: Software Product Lines — Experience and Research Directions, Proceedings of the First Software Product Lines Conference (SPLC1). Ed. Patrick Donohoe, pp. 111–132, Kluwer Academic Publishers, 2000.

[4] James C. Dager. *Cummin's Experience in Developing a Software Product Line Architecture for Real-Time Embedded Diesel Engine Controls*. In: Software Product Lines — Experience and Research Directions, Proceedings of the First Software Product Lines Conference (SPLC1). Ed. Patrick Donohoe, pp. 23–46, Kluwer Academic Publishers, 2000.

[5] *CelsiusTech — A Case Study in Product Line Development.* In: Software Architecture in Practice. Len Bass, Paul Clements, and Rick Kazman. Chapter 16 with

Lisa Brownsword, Addison Wesley, 1998.

[6] Peter Knauber, Dirk Muthig, Klaus Schmid, and Tanya Widen. *Applying Product Line Concepts in Small- and Medium-Sized Companies.* IEEE Software, Vol. 17, No. 5, pp. 88–95, 2000.

[7] Pierre America and Jan van Wijgerden. *Requirements Modeling for Families of Complex Systems.* Third International Workshop on Software Architecture for Product Families (IWSAPF'3), March 15-17, 2000, Las Palmas de Gran Canaria, Spain. Springer LNCS 1951.

[8] Paul Clements, Cristina Gacek, Peter Knauber, and Klaus Schmid. Successful software product line development in a small organization. In Paul Clements and Linda Northrop, editors, *Software Product Lines: Practices and Patterns*, chapter 11. Addison Wesley Longman, 2001.

[9] Klaus Schmid, Ulrike Becker-Kornstaedt, Peter Knauber, and Florian Bernauer. *Introducing a Software Modeling Concept in a Medium-Sized Company.* Proceedings of the 22th International Conference on Software Engineering (ICSE'00). pp. 558–567, 2000.

[10] David C. Rine and Robert M. Sonnemann. *Investments in reusable software: A study of software reuse investment success factors*. The Journal of Systems and Software. Vol. 41, No. 1, pp. 17–32, 1998.

[11] Wayne C. Lim. *Managing Software Reuse.* Addison-Wesley-Longman, 1998.

[12] Peter Knauber and Giancarlo Succi, editors. *Proceedings of the First Workshop on Software Product Lines: Economics, Architectures, and Implications, Limerick*, number 070.00/E in IESE Technical Report, 2000.

[13] Patrick Donohoe, Editor. *Software Product Lines: Experience and Research Directions* — Proceedings of the First Software Product Line Conference (SPLC1). Kluwer Academic Publishers, 2000.

[14] Werner Mellis. *Process and Product Orientation in Software Development and their Effect on Software Quality Management.* M.J. Wieczorek and D.B. Meyerhoff (eds.): Software Quality — State of the Art in Management, Testing, and Tools. Springer, 2000.

[15] Conway, M. E. *How Do Committees Invent*. Datamation, Vol. 14, No. 10 (October 1968): 28-31.

Tracing Features with Decision Models

Zoë Stephenson, John McDermid
Rolls-Royce University Technology Centre,
Department of Computer Science,
The University of York,
Heslington, York YO10 5DD, UK
(+44) 1904 432749
{zoe.stephenson,john.mcdermid}@cs.york.ac.uk

1 INTRODUCTION

Attention has been paid in recent years to the possibility of using conventional product-line techniques to produce software. This approach makes use of the similarity between different products in that product line, describing that set of products as a family[8] with common and variable features[2]. Features are included or excluded from a particular family member according to the particular goals, or requirements, placed on that family member. Those features are then composed together to create the product.

Descriptions of the legal combinations of features for a particular set of products are well-understood[3], showing the way in which features are selected, the content of the features, and describing feature dependencies either alongside or within the description of legal feature combinations. Since the work in this paper is intended for families of safety-critical systems, the available techniques for describing and implementing features are limited, especially those involving run-time flexibility and automated code generation.

As a further goal of this work, the reduction of the cost associated with making a change is sought. Given a product line of sufficient breadth, a majority of the changes to a product will be covered by a different product in that product line. However, for a safety-critical product family, the range of potential feature combinations is much larger than the number of products that are being manufactured. The products themselves are composed of standard components, rather than being generated through another tool or making use of dynamic object-oriented techniques such as reflection[1]. This implies that the work needed to change a product from one member of the family to another must occur by altering a set of components within a complete software product. Clearly, the impact of performing this task must therefore be minimised.

In this paper, we describe a strand of work that has developed a representation for features that helps to ensure that dependencies other than family selection are fully understood. These dependencies are then used, both to provide some of the information for feature selection descriptions, and to structure the product to minimise the impact of the most common changes.

2 FAMILY DESCRIPTION

Descriptions of families of software have been in use throughout software engineering. Parnas introduced a concept of program families[8], whereby advantage is gained by considering the common aspects of a set of programs, and then showing how they differ. This theme of similarity has continued throughout software engineering history. Language design has always encouraged 'internal' reuse, where a function that is to be used in many parts of the same program is stored once in its text, and referred to from many different points. The ability to share similarities between different programs starts with operating systems and system libraries, and progresses to utility libraries and toolkits, and further into component-based software and off-the-shelf component libraries. Successful applications of these concepts to create a large base of reusable software have mostly met with poor results; exceptions arise when the software fits into a well-defined and well-understood domain, although problems can then be found in accessing, understanding and using the software[10]. Object-orientation has had its own part to play in providing a common structural paradigm within which these issues can be addressed.

More recent developments in the use of product-line approaches for the construction of families of software have resulted in a variety of techniques, such as FODA[4], FAST[2] and MRAM[7]. Of these techniques, both FODA and FAST describe feature selection using a graph notation. FODA feature diagrams represent all features as nodes, and features are descendents of those features whose selection is a prerequisite. The arcs from each node are decorated to show their relationship (such as a mutual exclusion constraint, or a select-at-least-one constraint) and nodes themselves may be optional. The MRAM graph approach represents features and feature selections as nodes, with a feature selection node being selectable if its parent is selected, and performing a selection among its child nodes. This selection covers a sin-

gle optional node, a set of nodes from which at least one must be chosen, and a set of nodes from which exactly one must be chosen. The FAST approach specifies that a feature should identify the other features on which it depends; the realisation of this specification is embedded in an application engineering environment produced by the family analysis process.

Each of these description mechanisms serves two purposes. Firstly, they identify the features to be found in the family, and the context upon which that feature depends. Secondly, they give some structure to the selection of a legal set of features for inclusion into a family member. The supporting information for this selection process comes, in part, from the product line customers, and in part from the dependencies between features. Hence, it is important to ensure confidence in the validity of those dependencies. The remainder of this paper will focus on the description of features, rather than the description of legal combinations of features.

3 FEATURES AND DECISIONS

The features of a family of products are introduced into that family to satisfy particular goals placed on its members. For example, market research and focus groups for mobile telephones may indicate that the ability to send a text message to multiple simultaneous recipients is a marketable product feature. Similarly, an aircraft manufacturer may want an engine that can detect and record potential fan damage incidents, such as a bird strike.

In meeting such a goal, a software engineer makes a design step, deciding on a particular feature, or set of features, to meet that goal. In some cases, such a decision is simple, in others it may involve a complex trade-off based on many sources of information, and with wide-ranging consequences. To represent this form of feature introduction, as a design step made in the presence of contextual information, a knowledge-based representation approach is appropriate. These approaches record and maintain a body of knowledge about a particular design, such as the trade-offs and choices being made, and the context within which they are made. They contain automated assistance for the exploration of a design space, typically with multiple designers involved.

Decision approaches such as REMAP[11] and Redux[9] make use of an underlying constraint management system to maintain the information base. Redux uses a large, fixed contextual model within which assignments are made to describe potential solutions. REMAP solutions use design objects from existing design methods, and constraints among those objects are also maintained. Solutions are also linked to the requirements and decisions that drive their creation. Information can be added to show the assumptions made in arguing for and against particular solution strategies.

These decision-based approaches to design capture are intended for use in large collaborative projects in which many deliberations must be carried out before reaching an acceptable solution. They manipulate information that is not directly applicable to the description of dependencies in safety-critical systems families. Instead, only the introduction of the feature and the context of that introduction need to be recorded. In Redux, that information is represented by:

Goal Representations of aspects of the desired state of the product.

Assignment An addition to the description of the product.

For the REMAP approach, contextual information contains:

Requirement Representations of aspects of the desired state of the product.

Design Object The results of a particular decision.

Assumption A statement that provides additional support for a particular solution strategy.

The approach developed in this work makes use of these forms of contextual information to represent features.

4 DECISION-BASED REPRESENTATION

In representing a feature design step, there are two important categories of information that must be made explicit:

- The reason for introducing the feature — the goals or requirements that it meets, or the (inadequate) models that it augments;
- The context that describes the feature — the assumptions and design solutions that provide a reference within which the feature is introduced.

Between these two threads of reason and context must lie the point at which the software engineer has actually performed the design step. This suggests that there should be a modelling element in which the software engineer records what has taken place to introduce the feature. This is the decision strategy element. Connected to that element are the goals and existing solutions that caused the feature to be introduced, and the assumptions and solutions that form its context. The result of applying the action represented in the decision is given as resulting solutions and derived goals.

To be able to use this approach for safety-critical software, it must be possible to construct an argument that explains how that software meets safety regulations. The knowledge-based representation schemes represent some of this information as well as decision information, but it is interleaved with decisions to provide deliberation among decisions. An approach that is able to provide documentation of argumentation without explicit deliberation is the Goal Structuring Notation[5]. This approach has been used successfully in the construction

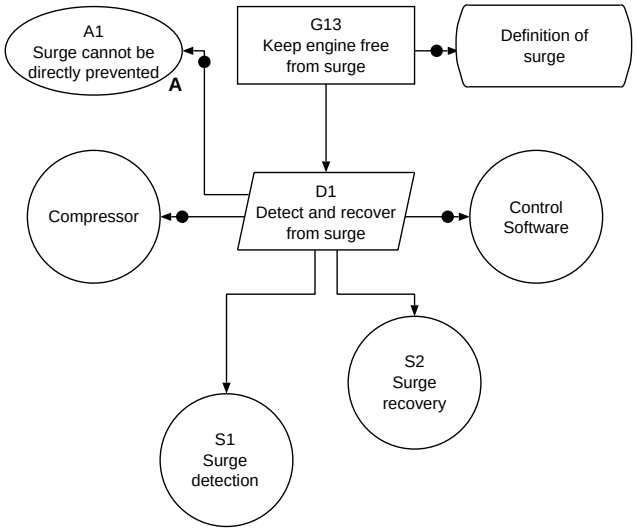


Figure 1: Decision Trace Notation

of safety arguments, and so provides an ideal companion to the work outlined in this paper. This companionship is made explicit in two ways:

- The generalised "context" element from GSN is adopted as a way of providing definitions that explain a software engineer's understanding of a particular element;
- The graphical arrangement and notation of GSN is adopted to ease understanding and tool support.

Since the new approach is primarily concerned with tracing the reason and context for a decision, it is called the Decision Trace Notation. An example of the diagrammatic form of the notation is given in Figure 1. Here, a goal element with a piece of contextual information drives a decision to introduce a detect-and-recover feature. The feature depends on descriptions of the compressor and the existing control software, and assumes that surge cannot be directly prevented. It results in software components to detect surge and perform recovery. In the diagram, the "reason" path, from goal to solution, is shown with directed arcs. Context relationships are shown using directed arcs decorated with filled circles.

5 PRIORITISATION AND CHANGE

Features are composed into a complete product by ordering them according to their dependencies, and building a software architecture to cater for their variation. However, it is possible for a large number of features to have the same dependencies, and would therefore seem to be introduced together. However, the change cost reduction strategy outlined in the introduction calls for the most common changes to have the least impact. This can be achieved by delaying their introduction as long as possible, thus introducing features according to how often they change.

To determine this measure of change, a simple heuristic is used: a feature which varies more often than another is considered to represent a more common change. This information is not directly available from a family selection representation, as that only indicates the choices that are available; a feature that is selected in 90% of the family members, but not in the rest, changes less than one that is selected in 50% of the members but not in others. While a more complex ordering rule would be possible, this simple heuristic is being used to evaluate the feasibility of the technique.

It is possible that the use of a change management ordering technique will conflict with the change context dependencies of the features. In this situation, the relationship between the change context, the reason for the feature, and the result of the feature must be assessed. In situations where the feature refines aspects of other features, the reason dependencies must be obeyed. However, in situations where the feature results in a policy for the use of other features, the change ordering will be more important.

6 CASE STUDY

An analysis was made of the features of two different engine functional areas, the core control algorithms, and the thrust reverser control system. The analysis approach used was tailored from other domain analysis approaches[6, 10, 4] to fit DTN elements. The stages of the process were:

Preparation Identify scope, sources, level of detail and common knowledge.

Collection Extract issues and variations from documentation and interview experts.

Tracing Record features using DTN elements.

Trace Validation Validate the feature set against documents and experts.

Variation Record feature variations and prioritise feature introduction.

Variation Validation Validate the family model against existing products.

In the control laws analysis, after three iterations, 34 features were identified. Of these, 24 were common to all control systems, and 10 varied to some degree. Four of the features were described as policies that influenced the incorporation of other features. The features were prioritised according to their dependencies and the change heuristics. These features were then introduced into a software architecture according to that priority ordering. Additional contextual information supplied architectural style, component organisation and system organisation policies. A validation exercise

was conducted with domain experts, comparing the features and the architecture to their knowledge of engine projects. Both were able to represent those projects at the given level of detail.

For the thrust reverser analysis, 14 features were identified. Of these, all were contingent on the inclusion of the reverse thrust feature itself, and of these, 6 were common for that feature, and 7 varied. These features were prioritised, and introduced into a software architecture. The architecture was then implemented for use with an engine system simulator, and a change was made to one of its variable features. The resulting rework was found to be proportional to the size of the feature, in both impact analysis and in the amount of new code.

7 DISCUSSION AND FURTHER WORK

The features dependency relationships that have been found by this approach have been indicative of the nature of the systems that were analysed. Further work in assessing features would find more complex dependencies between features. The use of Decision Trace Notation in describing the resolution of undesirable feature interactions should be investigated.

The approach is predicated on the assumption that the common changes that occur within a member of a product family will result in another member of that family. The use of the analysis and representation techniques for changes that move a product across that scope boundary, or for changes to that boundary itself, must also be addressed.

The change prioritisation heuristic, and its interaction with the change and reason dependencies, must be investigated further. An extension to the notation is being investigated in which specific relationships between the reason, context and result of a decision are represented explicitly. These relationships will form the basis of a more rigorous prioritisation approach.

Many of the features that have been discovered through the case studies have been similar in structure, and the decisions have largely been instances of domain-specific strategies. It seems appropriate to investigate the use of a domain-specific pattern catalogue of decision strategies for feature representation.

ACKNOWLEDGEMENTS

The work presented in this paper has been conducted in cooperation with Rolls-Royce plc., and part of the work was conducted through the EPSRC-funded CONVERSE (GR/L42872) project. CONVERSE was a part of the Systems Engineering for Business Process Change (SEBPC) managed research programme. We would like to express our thanks to the EPSRC for funding this research.

Rolls-Royce plc. has supported a number of University Technology Centres (UTCs), with which they conduct research into key aspects of engine development and design. A UTC in Systems and Software Engineering was founded in York in 1993. We would like to acknowledge the support of those staff from Rolls-Royce Controls Systems who participated in the studies.

REFERENCES

[1] F. Buschmann, R. Meunier, H. Rohnert, et al. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[2] J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6):37–45, Nov. 1998.

[3] K. Czarnecki and U. W. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[4] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, CMU/SEI Technical Reports, Nov. 1990.

[5] T. P. Kelly and J. A. McDermid. Safety Case Construction and Reuse Using Patterns. In *Proceedings of the 16th International Conference on Computer Safety, Reliability and Security*, pages 55–99, 1997.

[6] W. Lam and J. A. McDermid. A Summary of Domain Analysis Experience by Way of Heuristics. *ACM Software Engineering Notes*, 22(3):54–64, May 1997.

[7] M. Mannion, B. Keepence, H. Kaindl, et al. Reusing Single System Requirements from Application Family Requirements. In *Proceedings of the 21st International Conference of Software Engineering*, pages 453–463, May 1999.

[8] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2(1):1–9, Mar. 1976.

[9] C. Petrie. Constrained Decision Revision. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 393–400, July 1992.

[10] R. Prieto-Díaz. Domain Analysis: An Introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54, Apr. 1990.

[11] B. Ramesh and V. Dhar. Supporting Systems Development by Capturing Deliberations During Requirements Engineering. *IEEE Transactions on Software Engineering*, 18(6):498–510, June 1992.

Ideas on How Product-Line Engineering Can be Extended*

Jeffrey M. Thompson and Mats P.E. Heimdahl
Department of Computer Science and Engineering
University of Minnesota
4-192 EE/CS; 200 Union Street S.E.
Minneapolis, MN 55455 USA
+1 (612) 625-1381
{thompson,heimdahl}@cs.umn.edu

Abstract

Product-line engineering can result in cost savings and increases in productivity. In addition, in safety-critical systems, the approach has the potential for reuse of analysis and testing results which can lead to a safer system. Nevertheless, there are times when it seems like a product family approach should *work when, in fact, there are difficulties in properly defining the boundaries of the product family.*

In this paper, we present a position on n-dimensional *and* hierarchical *product families, which we have recently introduced. This paper focuses on our initial thoughts on how making n-dimensional and hierarchical families has the potential to affect the product-line development process as well as how using this approach might enable more organizations to use product-line approaches.*

1 Introduction

Although one of the main barriers to the use of product family techniques is one of process and organizational acceptance, technical issues have not been completely solved for product-line engineering. The techniques available work best for cohesive product families, where the variabilities do not have complex interdependencies. When this is not the case, it can be difficult to apply the product family approach even though there might be significant commonalities between the members of the family. As an alternative, we propose to view the families themselves in a multi-dimensional and hierarchical fashion. This helps us to deal with existing problems, for example, near commonalities, and helps to extend the approach to domains which, traditionally, would be difficult for product-line engineering.

*This work as been partially supported by NSF grants CCR-9624324 and CCR-9615088, and by NASA grants NAG-1-2242 and NCC-01-001.

We introduced the notion of *n-dimensional* and *hierarchical* product lines in [9]. In this paper, we present a position on how this techniques may affect the software product-line development process and how it may assist in the adoption of software product-line engineering within organizations.

2 Product-line engineering

The notion of a product family was introduced by David Parnas in [8]. The FAST (Family-oriented Abstraction, Specification and Translation) process advocates the creation of a specialized language for the domain for each family [10]. A similar process is mentioned by Campbell *et al.* in [3] and by Lam [6]; the differences between these works are primarily in the sort of artifacts produced by the domain engineering.

Current techniques for product-line engineering work well (1) the systems in the family share significant commonalities, and (2) The variabilities which define each family member have a straightforward decision model, i.e., it does not require many complicated rules to describe how the variability values are assigned to produce each family member. The first point describes the essential feature of product families that Parnas noticed in his work. However, the second point originates in the practical experience of many researchers who have labored to construct software product-lines. In general, the simpler the relationships among the variabilities, the easier it is to construct the product family.

n-Dimensional product families: Attempts have been made to organize the product family requirements in a hierarchical fashion [7, 8, 5, 6]. Lutz noted in her attempt to organize the variabilities into a tree that "there were several possible trees, with often no compelling reason to select one possible tree over another" [7].

Brownsword and Clements present a shipboard command and control systems family which contained 3000-5000 parameters of variation for each ship [2].

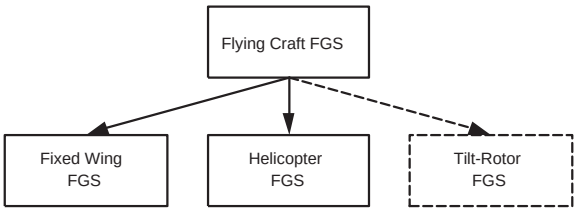


Figure 1. FGS product family covering flying craft

They state that "the multitude of configuration parameters raises an issue which may well warrant serious attention." In addition, they present three different views of the architectural layering of the base system in the case study which "do not conflict with each other; rather they provide complementary explanations of the same ideas."

Both these examples, as well as our own experience, illustrate the fact that often a product family is multi-dimensional; therefore, a hierarchical decomposition is not sufficient to capture the structure of the domain. We call such domains *n-dimensional product families.*

Hierarchical product families: Suppose that a company wished to construct a flight guidance system (FGS) for both fixed-wing aircraft and helicopters[1]. The FGS is responsible for issuing commands that keep the aircraft level, cause it to climb or descend, and so forth. Furthermore, the FGS must interact with other airborne systems. Many of the tasks that the system has to perform might be common across these two radically different aircraft. Nevertheless, the actual control of the aircraft is very different. Therefore, developing a single set of commonalities and variabilities which span this entire domain is difficult.

Some would argue that the FGS example is simply too diverse to be considered a product line. However, it is clear that these systems share much in common, which is the most important criterion for being a family. Thus, we propose the concept of a *hierarchical product family*.

Most previous attempts at product family structuring have focused on hierarchically grouping the *variabilities* while the *commonalities* remain the same for all family members [7, 6]. Notable exceptions are Parnas [8] and Brownsword and Clements who noted in their case study at CelciusTech [2] that sometimes product-lines exist within the main product line.

In our approach, *additional commonalities* which are *unrelated* to the parent product family can be added in the sub-families. The hierarchical decomposition of the FGS family is shown in Figure 1. Thus, the helicopter sub-family can have significantly different requirements than for fixed-wing aircraft, yet share many things in common as well.

[1]We would like to thank Steven P. Miller of Rockwell-Collins Inc. for this example

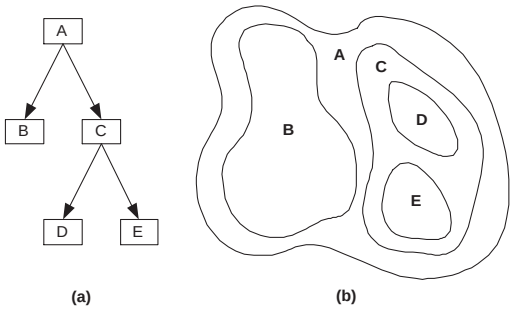


Figure 2. Hierarchical decomposition and subset structure

By structuring the requirements in this way, we focus on the structure of the domain itself. Furthermore, should the company wish to start building FGS systems for an entirely new set of aircraft, for example, tilt-rotor aircraft, this could be done while reusing many aspects of the FGS systems already implemented (Figure 1).

This will eventually effect the architecture and structure of the systems. For example, the product of the domain engineering for the parent family, Flying Craft FGS, might be a set of reusable components, whereas the product of domain engineering for the children might be a reference architecture or generation facility. The architectures for the fixed-wing aircraft and the helicopters could differ significantly and use the components from the parent family in different ways.

3 Structuring technique

One way to view a product family is as a set, where the boundaries of the set are determined by the commonalities, and the individual members of the set are distinguished by the values of their variabilities. Furthermore, the family may be undefined at some points within the boundaries due to, for example, illegal or non-functional combinations of variability values. This section shows how to use set theory to reason about the structure of product families (a more extended version can be found in [9]).

The most basic structure that can be represented with the set theoretic approach is the subset (Figure 2). This corresponds to a hierarchical decomposition of the family. The general requirements for any family **E** which is a subset of another family **C** is as follows:

- **E** must include all of the commonalities and variabilities in **C**; however, **E** may restrict the range or options available in the variabilities.
- **E** can add additional commonalities and variabilities which are not present in **C** as long as the additional they do not conflict with the commonalities or variabilities in **C**.

The first criterion is straightforward and necessary for the subset **E** to be completely contained within **C**.

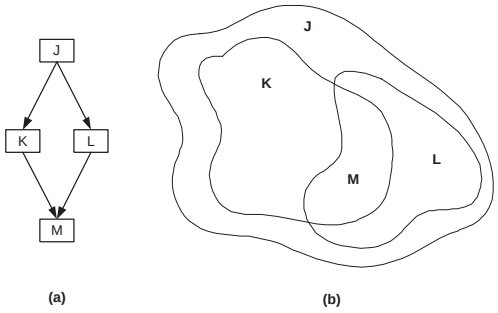


Figure 3. Set intersection and non-hierarchical structure

The second criterion defines the fact that **E** may wish to refine or restrict the values of the variabilities of **C**. It is possible for this refinement to result in an additional commonality. Additional commonalities can also be added which are unrelated to the parent family. For example, it is likely that the family of helicopters will need different commonalities than the family of fixed-wing aircraft. Finally, it is possible to add additional variabilities.

Another structure that can be represented using a set-theoretic approach is that of set intersection. The ability to represent a set intersection distinguishes this approach from the purely hierarchical structures which have been applied by others. This is shown in Figure 3. The constraints on any family **M** which is a subset of families **K** and **L** are as follows:

- **M** must include all the commonalities and variabilities of both **K** and **L**; however, it may restrict those variabilities as above for subsets.
- **M** may introduce additional commonalities and variabilities which are not present in **K** or **L**.

These structures can be used to reason about families which are both *n-dimensional* and *hierarchical*.

4 A brief evaluation

The structuring technique presented results in the creation of more families within the domain than with a traditional approach. However, these sub-families are more cohesive and simpler than would be the case if we created just one top level-family. We believe that this provides several benefits. First, the top-level family can now be much broader than was previously possible. Second, the overall family can be expanded and contracted by adding and subtracting sub-families. Finally, these techniques will allow a family to be more easily refactored as the definition of the family evolves over time.

The ability to draw a larger product family was an essential requirement for the structuring technique. This grows out of our own experiences with mobile robotics [4], where we had difficulty in applying the product family approach. This difficulty stems from the fact that the mobile robotics domain is both *n-dimensional* and *hierarchical*.

The mobile robotics domain breaks down along two clear dimensions: the hardware platform and the desired behavior. Each hardware platform conforms to a basic specification: it can move forward and backward, turn left and right, sense whether or not an object is in front of it. The hardware platform may also be equipped with a variety of sensors and actuators that give it additional capabilities; and, the various sensors differ greatly in the speed and accuracy with which they provide information. Thus, on the hardware side, there are many different configurations that must be modeled.

On the behavior side, we can imagine that a basic behavior might be a random exploration where the primary goal of the robot is collision avoidance and recovery. More complex behaviors can be added, for example, wall following, going through doors, and finding particular objects. Furthermore, those behaviors may be composed and combined to form a composite behavior. We might envision a behavior which includes the door navigation, a wall following behavior, and a high-level planner. The high-level planning behavior needs to communicate with the random exploration, door navigation, and wall following to direct the robot towards high-level goals. However, if the robot collides with an obstacle, then the lower level behavior will take over and recover from the collision. Thus structure of the behavioral dimension is much different from the hardware dimension and resembles Brooks' subsumptive architecture [1].

Certainly, a domain such as mobile robotics which absolutely requires n-dimensional and hierarchical product families will necessarily be more complex than a domain that does not require these techniques. Nevertheless, any domain can benefit from reuse of the artifacts at the top of the family hierarchy and a more traditional cost-benefit will exist towards the leaves of the family (along each particular dimension).

Another benefit of the technique is the ability to expand and contract the family as necessary. This ability is essential because it allows a more incremental development of product-lines than is facilitated by current approaches. Furthermore, it facilitates *family refactoring*; that is, the family can be redefined more easily as the product line evolves over time. Thus, this structuring technique has much potential to increase the usefulness of the product family approach.

One of the barriers to traditional product family approaches is that the whole organization must change to accommodate product-line oriented development. Many resources are required to develop the domain engineering support for the entire product line while at the same time continuing to produce products for existing customers. Our approach allows an organization to start out with a high-level product family and reuse just a few key pieces between the major product areas. As the payoff from this reuse makes more organizations resources available, the organization can then afford to make the family more rich (by refactoring and/or adding sub-families) and thus achieving more payoff from the effort.

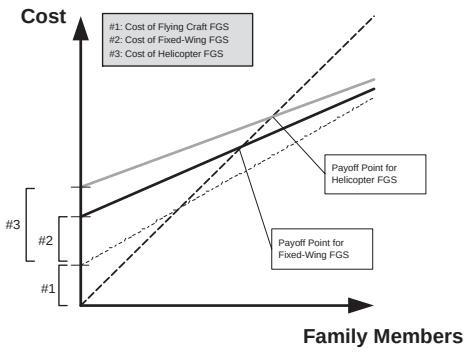


Figure 4. Cost-benefit of the FGS Family

Of course, these benefits do not come for free. The broader and more flexible view of product families allowed by our techniques will result in families which are more complex than traditional families. In addition, because of this broader view, it may be more difficult to determine what constitutes a viable family under our approach. Almost anything is related in some fashion or other and it may be difficult for organizations to decide when to define an encompassing family for a particular group of subfamilies. Nevertheless, we feel that these techniques hold promise and may serve to advance the frontiers of product-line engineering.

The cost-benefit analysis of our product-line engineering approach is more difficult because one must not only consider the cost of developing domain engineering support of the particular sub-family in which the member resides, but also all sub-families above that one in the product family hierarchy. For example, the cost-benefits for the FGS family is shown in Figure 4. The payoff for the fixed-wing FGS is shown by the thick black line and the payoff for the helicopter FGS family is shown by the thick gray line. As the figure demonstrates, the payoff point for the two subfamilies is different, because the cost of implementing each over and above the functionality provided by the flying craft FGS family is different. As the structure of the family becomes more complex, for example, through the creation of a deeper hierarchies and/or the use of multiple dimensions with constraints between them, this relationship will become more complex.

5 Conclusions and future work

In this paper, we have reflected on some current issues with product-line engineering and presented some ideas about our initial attempts towards extending the product family approach to better address these issues as well as support what we call *n-dimensional* and *hierarchical* product families.

When we examined problems that we and others have had in developing product families, we concluded that the difficulties in modeling near-commonalities and variability dependencies stemmed more from the structure of the domain itself and the lack of an ability to adequately capture that structure. Further, the simple structuring technique that we proposed in [9] is based on set theory, which provides a simple and elegant solution. This allows the analyst to capture the structure of the domain and not be biased by implementation or design concerns. This approach allows thinking about *n-dimensional* and *hierarchical* product families and encourages many different views of the system.

In the future, we intend to continue developing this approach to product-line engineering. We are already working on several industrial-sized case examples with good initial results. We look forward to reporting the outcome of that work, as well as presenting a more detailed view of this approach. Furthermore, in the future we intend to provide a more detailed description of the formal foundations of this approach and how it could be leveraged if the family members were specified in a formal specification language.

Acknowledgements

The authors wish to thank Steven P. Miller at the Rockwell-Collins Advanced Technology Center for his thoughtful comments and encouragement with regards to this work as well as his insights on how the structuring techniques presented here might be applied to the systems manufactured by Rockwell International.

References

[1] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, March 1986.

[2] Lisa Brownsword and Paul Clements. A case study in successful product line development. Technical Report CMU/SEI-96-TR-016, Software Engineering Institute, Carnegie-Mellon University, October 1996.

[3] G. Campbell, J O'Connor, C. Mansour, and J. Turner-Harris. Reuse in command and control systems. *IEEE Software*, 11(5):70–79, September 1994.

[4] Debra M. Erickson. Structuring formal requirements specifications for reuse: A mobile robotics case study. Masters Project, University of Minnesota, April 2000.

[5] Juha Kuusela and Juha Savolainen. Requirements engineering for product families. In *Proceedings of the Twenty-Second International Conference on Software Engineering (ICSE'00)*, pages 60–68, June 2000.

[6] W. Lam. Creating reusable architectures: Initial experience report. *ACM SIGSOFT Software Engineering Notes*, 22(4):39–43, 1997.

[7] Robyn R. Lutz. Extenting the product family approach to support safe reuse. *Journal of Systems and Software*, 53:207–217, 2000.

[8] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, March 1976.

[9] Jeffrey M. Thompson and Mats P.E. Heimdahl. Extending the product family approach to support n-dimensional and hierarchical product lines. In *The Fifth IEEE International Symposium on Requirements Engineering*, August 2001.

[10] David M. Weiss and Chi Tau Robert Lai. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

Structuring Test Assets in a Product Line Effort

John D. McGregor
Clemson University
johnmc@cs.clemson.edu

Introduction

The test assets for a product line should live within the same architecture and satisfy the same architectural constraints as the product line application. This paper discusses the structure and design of test information in a product line development organization. The basic premise is that the effort to create and maintain the required test information is minimized by being compatible with other structures related to the product line. Since the resources devoted to testing can often approach those devoted to development, this optimization can have significant impact on productivity and profitability. This work is based on a number of projects in several different companies and domains. Our experience began with projects that produced product families and more recently with product line approaches.

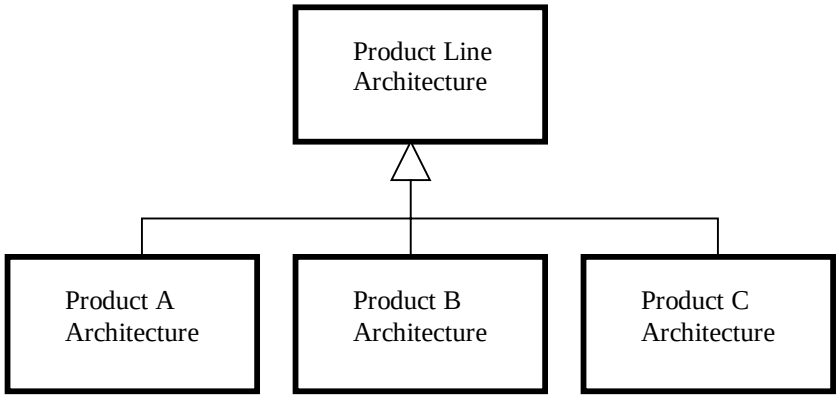


Figure 1

There are two significant structures in a product line that affect the structure of the test assets. There is a general product line architecture from which the architecture for each of the products is derived. This relationship is shown in Figure 1. The relationship is similar to the specialization relationship in object-oriented design.

The second structure is the set of relationships between the levels in the producing organization. Figure 2 shows that strategic management groups set the standards for quality attributes that are necessary to meet business objectives. The software engineers translate these levels into effort estimates. Tactical management translates these estimates into resource allocations. Some portion of these estimates and resources pertain to the testing process.

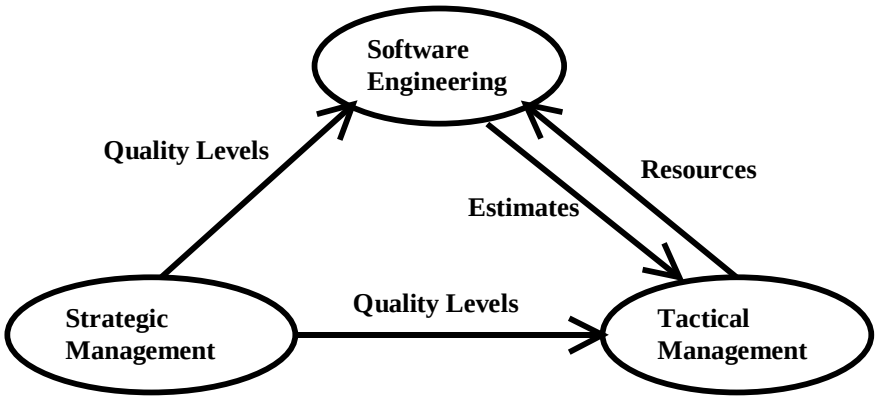


Figure 2

Testing Assets

The test assets needed to support a product line include:

- Test plans
- Test cases
- Test results and reports
- Infrastructure and tools
- Personnel

First we will provide a very brief description of each of these project assets and then we will consider the role of each type of asset in a product line.

Test plan - A test plan provides project personnel with a description of the types of testing that will be applied at a specific point in the development life cycle. The plan gives criteria for determining how much of each type of testing will be applied to the artifact under test. Template test plans can be created and used at the product line level and then specialized by the individual product teams.

Test case - A test case provides a stimulus/response pair for the artifact being tested. The stimulus is the set of all inputs that must be provided to establish a specific state in the system. The expected response is a description of the state the system should have reached after digesting all provided inputs plus all expected outputs. Many of the test cases developed by the Component Engineering Group can be propagated to the Product teams where they can be used to test the integration of the components into a product.

Test results and reports - These artifacts are assets with a limited useful lifetime. The test reports are used to direct the repair process. The reports may also be propagated to the other level (either upward or downward). They are propagated down to the product teams to identify potential problems with existing components. They are propagated upward from the product teams to provide the Component Engineering team with contexts for the failures that occur.

Infrastructure and tools - Testers use a variety of software tools to automate the process of providing the inputs to the system. These may be simple capture-and-playback systems running against a graphical user interface or they may be more sophisticated fault injection programs that control a piece of hardware. Scripts for the tools can be developed by the Component Engineering team and reused by the Product teams. The scripts are specialized by the Product teams using features available in the scripting languages.

Personnel - The people associated with the test process include those who write the test cases, those who apply them and those who create tools for use in the test process. The specialized testing knowledge of test personnel can

be made available by structuring the test organization along the same hierarchical structure as the product line organization.

Structures and Relationships

One of the basic principles of software development is that the architecture of a software system reflects the structure of the organization that built it. Successful organizations respond to this principle by organizing their people in the way that fits how they wish to structure the software. We take this a step further by structuring supporting materials such as test assets and documentation in parallel to the structure of the software products being constructed [2]. As a basic organizational scheme, the software engineering organization follows the basic structure illustrated in the left side of Figure 3.

Team Organization

Many product line projects identify two basic types of groups in the development organization [1]:

- component engineering/product line group
- product development groups

The product line group has a broader scope and a more general view of the products than do the product teams. They produce the product line architecture and many, if not all, of the components that populate the architecture. The product teams assemble these components into finished products.

Persons may have assignments that include responsibilities at both levels or the teams at the two levels may be largely disjoint. The component developers may also be assigned to component assembly teams. Personnel with testing responsibility often apply their specialized knowledge across the two levels.

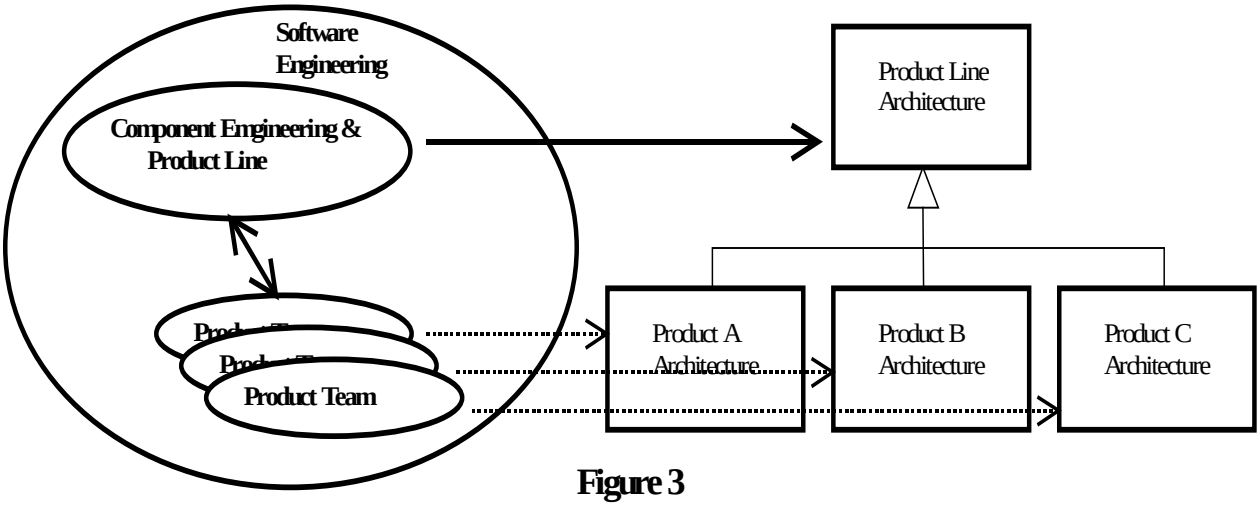


Figure 3

Test Plans and Architecture

Test plans are based on the component interfaces and the interactions that correspond to the connectors between the interfaces. As Figure 1 illustrates, there is a specialization relationship between the product line architecture and the individual product architectures. This relationship constrains how a test plan defined for an interface in the product line architecture can be reused in a specific product project.

Integration and system test plans are first produced at the product line level. The completeness of the test plan is determined by the completeness of the product line specification. Where the architecture is "parameterized", the test plan is as well. At the product level, the test plan is specialized by adding information about specific combinations of components needed to satisfy the product's requirements. Where techniques such as design patterns or ensembles

are used to describe the parameters, partial test plans are created at the product line level and assembled at the product level to produce comprehensive test plans.

Test Case and Software Organization

There are two relationships involved in the construction of test cases, as illustrated in Figure 4. First, the specified test case must be traceable to the feature that it is intended to test. Second, each test case specification corresponds to at least one test case implementation. That implementation may simply be the inputs and expected result that will be fed to a tool or it may be a script or software object that will actually execute the test.

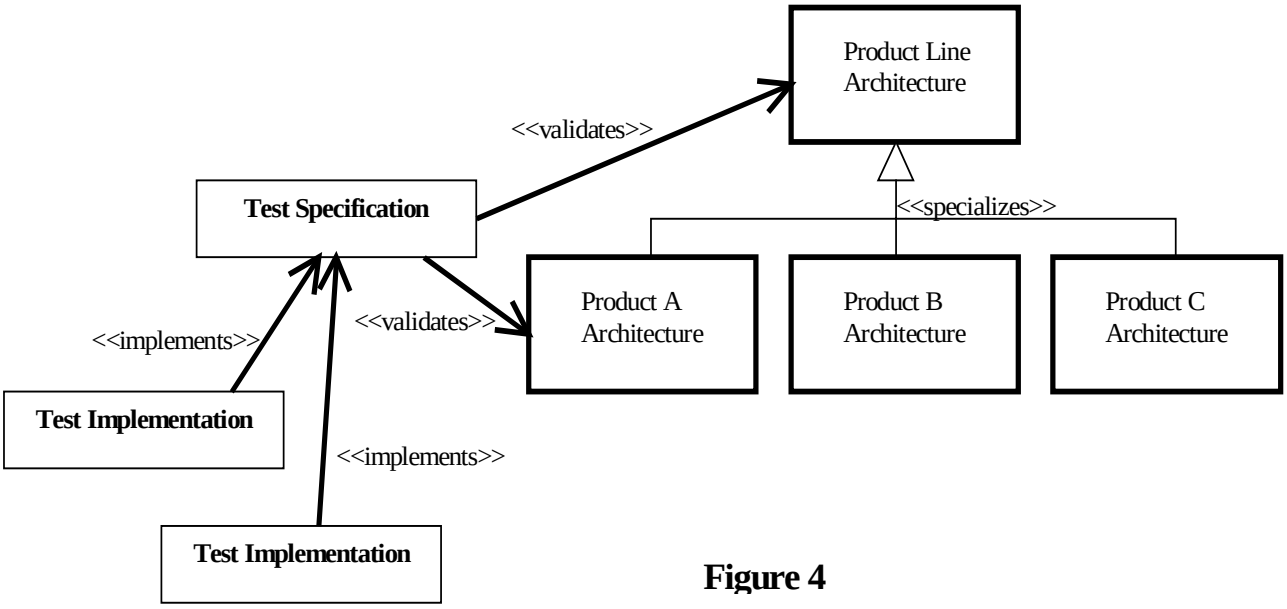


Figure 4

Test Reports and Results

The outputs from the testing process are reuseful assets. The reports are reused from one iteration to the next to track progress during regression testing. The test results are routed from product teams back to the component engineering group to provide direction for debugging and repair. This information is also directed to other product teams. This allows other teams to focus on testing in other areas of functionality.

Personnel

The "testing perspective" [3] is the viewpoint taken by a software engineer assigned any test responsibility. This perspective can be taught and some level of mastery is necessary for most engineers since every developer should be responsible for the initial testing of his/her own output. The number of engineers that are excellent at this perspective is very limited. As a result, those who are good at testing are usually shared among multiple projects. In a product line, testers can be used across multiple product development efforts.

Summary

The test assets required for a project can be developed more economically if the product line architecture is used as the organizing structure. Lines of communication and feedback that are in parallel to the architecture provide an organization in which change is facilitated. Our challenge continues to be to identify techniques that support this organizational scheme.

References

1. Paul Clements and Linda M. Northrup**,** *A Framework for Software Product Line Practice - Version 3.0,* http://www.sei.cmu.edu/plp/framework.html Software Engineering Institute, 2000.
2. John D. McGregor, "PACT- Parallel Architecture for Component Testing", *Journal of Object-Oriented Programming*, May 1997.
3. John D. McGregor, "The Testing Perspective", *Journal for Software Testing Professionals*, December 2000.

The Implementation of Holmes: a Tool to Support Domain Analysis and Engineering

Iliyan Kaytazov, Giancarlo Succi, Witold Pedrycz
Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB

Abstract

One of the serious obstacles in implementing DA&E is its complexity and the fair number of activities over which it spans. Consequently, a tool that alleviates the adoption of a DA&E methodology is required. Such a tool should be designed and implemented with a few considerations in mind. Given the diversity of DA&E activities, a possibility for smooth integration of third-party tools is a must. Furthermore, the design of such a tool should be well suited for frequent changes. The proposed implementation of such a tool claims to cover these prerequisites by heavily exploiting emerging object-oriented technologies and devising novel approaches to existing problems by using object-oriented practices.

1. Introduction

There are a number of existing methodologies for carrying out a DA&E. The commonality among them is that they are essentially based on coordinating a sequence of steps that start from the analysis of the market and end with the design of an actual software system [1]. This suggests that domain analysis is a large and complex process consisting of sundry and interdependent activities. The difficulty connected with the adoption of Domain Analysis imply that a tool that will alleviate the overhead of its adoption and will decrease the incidence of errors is needed.

There are a number of requirements that such a tool should meet in order to be useful. One of the basic and most important features is to store and organize information originating from different fields. That organization should allow performing a few operations against the stored data: run queries to ascertain whether a particular data item is already present or not, or check the status of a data item (outdated vs. up to date); implement a mechanism to propagate a change to a data item made in one document to all other documents that contain the same data item, if that is desired; support data and change consistency in the presence of multiple users.

[11] suggests that the purpose of a domain analysis support tool "should be to offer an integrated environment for collecting and retrieving the domain model and architectures". However, for some domain analysis activities, such as defining a domain vocabulary or building a domain model, there are already existing tools, which offer the required functionality. Incorporating these tools through interfaces will result in shorter development time for the support tool. Moreover, since these tools are specialized they will have superior features than a newly developed one. Another advantage of incorporating a third-party tool is the existence of users who are already familiar with it and thus utilizing the network externalities.

That brings the issue of integrating third-party tools. That area has already been studied by [5]. They identified two approaches for tool integration: tool-to-framework and tool-to-tool integration. Tool-to-framework integration suggests that the tools do not interface directly but instead use a framework to exchange data or operation calls. Conversely, when tool-to-tool integration is used a separate interface/adaptor should be created for every possible couple of cooperating tools.

[15] recognizes four different aspects of integration: user interface, data, control and process integration. Considering the fact that there are a number of different knowledge contributors and the necessity of integrating third-party tools a requirement for a common "look-and-feel" would be both unnecessary and restrictive. Control and process aspects of integration will highly aggravate the task of integrating tools without providing a substantial gain in functionality and since will not be considered.

As pointed out previously by [11], a support tool should be mainly responsible "for collecting and retrieving the domain model and architectures". For that purpose the most essential aspect of integration is data. Data integration can be achieved by direct data transfers among tools or by accessing and manipulating the data of a

common repository. Managing only that aspect of integration proves to be difficult. This is caused by the fact that "domain information is stored in a great variety of data sources, using different data models, access mechanisms, and platforms." [2].

All that considered an implementation of a support tool using a loose tool-to-framework integration and a shared data repository looks to be the most appropriate choice. It will eliminate the need of building numerous tool-to-tool adapters and the development of a complex method for accessing data distributed among the different tools.

Given the diversity of activities and knowledge involved in DA&E suggests that some sort of semantic support for the DA&E practitioner would be valuable. Such semantic support should try to suggest or find relationships within the information provided throughout the process of DA&E.

The previous analysis suggests that a domain analysis tool should support the following activities:

- Queries on existing data: filtering, sorting, and dependency relationships
- Link consistency management: ensures that link traces make sense
- Data and change consistency
- Support for multiple users
- Semantic assistance: warns user of potential problems
- Data integration with COTS tools.

2. Implementation of Holmes

2.1. Architecture

Holmes uses a blackboard architecture built using a Linda tuple space [8], as implemented by Sun's JavaSpaces [5]. A tuple space is a form of virtual, shared, associative memory that generatively provides a repository for tuples. One of its characteristic features is that an entity, which is put in the tuple space continues to reside there as long as some other process doesn't take it from the space. The existence of an entity in the tuple sapce is completely decoupled from the existence of the process that created that entity and put it in the space. That way a tuple space can be used as a persistent storage that holds data to be exchanged among processes which execution should not be constrained to overlap in time. Another advantage of using an implementation of a tuple space is that it provides for an easy way to synchronize a number of processes, which try to manipulate a set of objects. In that case a process is given a random object among those that meat the process' criteria. After the object is processed it can be put back in the tuple space or disposed of depending on the need.

In addition to the time decoupling a tuple space provides also space decoupling. In the JavaSpaces implementation this is realized by using Jini distributed technology. That way every process which wants to access a JavaSpace uses the same uniform way of connecting to it independent of the fact whether the JavaSpace is executed on the same machine or somewhere over a network.

All that considered the use of JavaSpaces gives an opportunity to implement a loose tool-to-framework integration that proved to be a better choice concerning the diversity of the tools incorporated. Furthermore, the access method to any entity stored in JavaSpaces eliminates most of the difficulties usually associated with multiple users accessing a common resource. Certainly, such a method will not be acceptable for any time-critical application but time is not an issue in our case.

The different tools incorporated in Holmes communicate using an algorithm similar to message based communication [Figure 1].In the proposed implementation there are a few improvements over standard message passing. As mentioned above the use of JavaSpaces waives the difficulty of initializing properly the communication channels between tools, after they are started. JavaSpaces stores messages as long as it is specified including infinity. Other improvements are that the messages that are exchanged among the tools are objects not data; a message queue implemented herein is not susceptible to the number of tools that "subscribe" to the data posted on that queue.

The advantage of using objects, compared to simple data, as messages is that an object encapsulates certain functionality. Thus, the tools that communicate through objects are decoupled from the format of the data stored in the object, which actually represents a message. This flexibility proves to be essential when dealing with diverse third-party tools, which use different output formats. Implementing messages as objects, in fact turns them into functional units that perform the communication protocol. That way changes made to the communication protocol leave the tools intact. The opportunity of using object-oriented design patterns, such as Adaptor, Visitor, Strategy [6], leave space for broad future evolution of the behavior offered by object-based messages.

Even though, there are no clearly defined communication channels for exchanging messages among tools, the objects, which represent these messages are organized into conceptual queues. The idea behind these queues is that they group messages belonging to semantically related data. Considering this organization, a tool subscribes to any queue that represents a flow of data the tool is interested in. A tool can act as both information provider or information subscriber. The way to do that is to instantiate an object of type "writer" or

"reader" respectively. All readers of a certain queue monitor the JavaSpace for specific types of messages. The order in which the readers subscribed to a queue is unrelated to the order in which they will actually read a message. In fact that order is unpredictable and varies from message to message. Nevertheless, the sequence in which messages are posted to the queue is preserved when they are read.

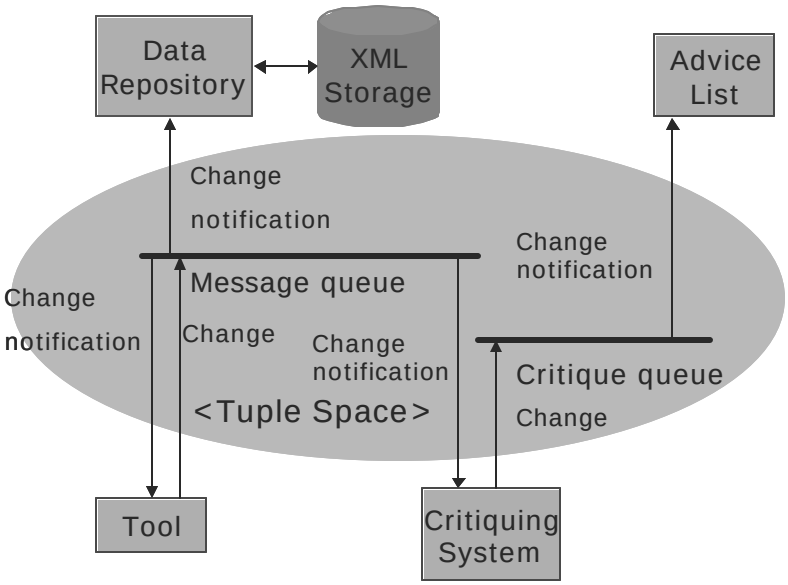


Figure 1: Architecture

(if a message M1 is posted to a given queue before message M2, than M1 will be read by a given reader before M2)

Tools that act as information subscribers are not aware and are not dependent on the type of tools that act as information providers, as long as they post messages that conform to the agreed interface.

As described so far, the framework, used to integrate all the tools provides for loose-coupling among the different tools in terms of time, type, and space. One disadvantage of the current framework is the possible performance loss compared to a having a framework that supports individual message channels among the tools.

2.2. Holmes' Tools/Persistent Storage

Holmes stores persistently its data through a number of tools, which are called repositories. Every repository stores semantically related information just like queues group messages belonging to semantically related information. When the Holmes tool is started all other tools that are interested in particular type of information post a request for that information. If the repository responsible for that information is stared and has activated its reader than it will notice the request, as all other tools that monitor the queue, and will reply posting the required data.

Similarly, when a tool makes a change to a certain piece of data than it posts the change on the queue. This gives a chance to all other tools, including the repository tool, to update their copies as well. When the Holmes application is terminated only the data contained in the repository tools is saved to persistent storage.

The above-given way of dynamically storing data ensures that the repositories itself could hardly become a bottleneck in the system. The data or parts of it is distributed and duplicated among the tools that manipulate it. As shown the use of JavaSpaces and objects as messages provides for parallel processing and polymorphic behavior on behalf of the tools.

The choice of how Holmes data is statically stored is also related to the tool integration requirement. Holmes data is stored using the Holmes Markup Language (HML), which is essentially XML [16] 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 rely on the growing popularity of XML as a data format, especially for Product Data Management systems [9] and UML CASE tools [12].

2.3. Critiquing System

As pointed out in the introduction some sort of semantic assistance would be highly valuable in DA&E support tool. The concept of design critics suggests a suitable approach for such assistance. Design critics are intelligent mechanisms that analyze a design and provide feedback to assist the designer in improving it [13]. According to [4], design critics should be embedded in the environment and actively but non-disruptively alert designers of potential problems suggesting potential solutions. [14] describes the critiquing approach as different from traditional software analysis approaches in that its focus is on the designer's cognitive needs. Traditional approaches attempt to prove correctness of a completed or almost completed system. Critiquing approaches, on the other hand, pessimistically detect potential problems in partially specified systems.

The critiquing system is implemented in Holmes similarly to the other tools. It subscribes for particular type of information by activating a reader to the associated queue. The critiquing system can be configured to monitor the data flow only on specific queues, which are of interest or to monitor the data flow on all queues. The critiquing system bases its responses on the data objects that it intercepts on a given queue without needing any prior knowledge about the tool that placed the data objects. The advantage of this approach is that the critiquing system is completely decoupled from the tools that post data to the JavaSpaces. This decoupling includes both the type of the tools and their number.

The critiquing system is based on the Prolog language. The knowledge about a specific field is represented in terms of Prolog clauses. Prolog was chosen as the scripting language of choice since Prolog clauses are very well suited to the description of relationships. Prolog scripts can also be used to make queries about dependency links on a particular data item.

Critics can be triggered when a particular content type is posted to the JavaSpace; the semantic meaning of the data is analyzed through the existing Prolog database, which represents the current knowledge about the domain. If any critique clauses are satisfied then a message, which describes the possible problem, is posted back to the JavaSpaces on a separate queue. Another tool which is called "Advice List" monitors that queue and displays the information contained from a given critique in a user friendly way.

The generation of critiques is based on user-modifiable rules written in Prolog. Since Prolog offers a possibility for dynamical assertion or retraction of clauses, the quality of the advice provided can be incrementally enhanced by experimenting with samples.

3. Conclusion and Future Work

The proposed implementation of a DA&E tool is based on a framework developed using Sun's implementation of a tuple space, called JavaSpaces. This framework provides for loose coupling among the integrated tools, and the framework. Since JavaSpaces are based on the object-oriented paradigm, any entity that is stored in the JavaSpaces should be an object. This condition, initially regarded as a constraint, enforced the development of a flexible, object-oriented mechanism for communication among the tools, as opposed to standard message passing.

Future work will be targeted at reducing the size of the current class hierarchy by adopting object-oriented design patterns mainly based on composition and delegation. Another component of future work will be focused on the development of the Critiquing System by adding Prolog clauses, which will identify possible pitfalls in a proposed design and suggest probable improvements. These suggestions will be based on existing reusable object-oriented design pattern.

Finally, a third component of future development will be the improvement of the visualization of domain information. The use of hyperbolic browser promises to give a better representation of existing data dependencies and relationships.

4. References

[1] Arango, G. (1994) "Domain analysis methods", Software Reusability, pp. 17-49, Ellis Horwood

[2] 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

[3] Domges, R. and K. Pohl (1998) "Adapting Traceability Environments to Project-Specific Needs", Communications of the ACM, 41(12)

[4] 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

[5] Freeman, Hupfer, and Arnold “JavaSpaces Principles, Patterns, and Practice” Addison-Wesley, 1999

[6] Gamma, Helm, Johnson, Vlissides “Design Patterns Elements of Reusable Object-Oriented Software”, Addison-Wesley Professional Computing Series, 1995

[7] Gautier, B., C. Loftus, E. Sheratt, and L. Thomas (1995) “Tool Integration: Experiences and Directions”, Proceedings of the 1995 International Conference on Software Engineering

[8] Gelernter, D. (1985) “Generative communication in Linda”, ACM Transactions on Programming Languages and Systems, 7(1)

[9] 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

[10] Griss, M., J. Favaro, and M. d’Alessandro (1998) “Integrating Feature Modeling with the RSEB”, Proceedings of the Fifth International Conference on Software Reuse

[11] 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

[12] 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

[13] Robbins, J. (1998) “Design Critiquing Systems”, Technical Report UCI-98-41, University of California, Irvine

[14] Robbins, J. and D. Redmiles (1998) “Software Architecture Critics in the Argo Design Environment”, Knowledge-Based Systems, 11(1)

[15] Wasserman, A. (1989) “Tool Integration in Software Engineering Environments”, Lecture Notes in Computer Science 467 – Proceedings of Software Engineering Environments, International Workshop on Environments

[16] W3C (1998) “Extensible Markup Language (XML) 1.0”, W3C Recommendation, REC-xml-19980210, URL: http://www.w3.org/TR/REC-xml

Document Information

Title:	Proceedings of Software Product Lines: Economics, Architectures, and Implications Workshop #3 at 23rd International Conference on Software Engineering (ICSE2001), Toronto, Ontario, Canada, May 13
Date:	May 2001
Report:	IESE-051.01/E
Status:	Final
Distribution:	Public