A survey on software product-line evolution

Mika Pussinen
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, 33101 Tampere
e-mail: mtp@cs.tut.fi

December 19, 2002

Contents

1 INTRODUCTION 2

2 ORGANIZATIONS AND PROCESSES FOR PRODUCT LINES 3

2.1 ESTABLISHING PROCESSES AND PLANS FOR PRODUCT LINES 4
2.2 ORGANIZATIONAL MODELS AND ASSET RESPONSIBILITY APPROACHES 6
2.3 AN EXAMPLE OF DOMAIN ENGINEERING CENTRIC APPROACH 9
2.4 AN EXAMPLE OF PRODUCT-FOCUSED APPROACH 12
2.5 OTHER ORGANIZATIONAL GUIDELINES AND ALTERNATIVES 15

3 RECORDING OF ASSUMPTIONS AND DESIGN DECISIONS 18

3.1 ON THE IMPORTANCE OF ASSUMPTIONS AND DESIGN DECISIONS ON EVOLUTION 18
3.2 SOLUTIONS FOR RECORDING OF DESIGN DECISIONS 19

4 METRICS FOR SOFTWARE EVOLUTION 24

4.1 ESTABLISHING AND TRACKING METRICS FOR PRODUCT LINES 24
4.2 PROPOSED METRICS FOR EVOLUTION 24

5 CONFIGURATION MANAGEMENT APPROACHES 30

5.1 APPROACH FOR COMPONENT BASED PRODUCT POPULATIONS 30
5.2 KOBRA APPROACH 31

6 HOW TO VISUALIZE AND TRACE EVOLUTION 34

6.1 EVOLUTION GRAPHS TO TRACE VERSION HISTORIES 34
6.2 TOPOLOGY DIAGRAMS TO TRACE COMPONENT RESPONSIBILITIES AND REUSE-LEVELS 35
6.3 DEPENDENCY TRACKING BETWEEN PRODUCT-LINE ARTIFACTS 36

7 OTHER GUIDELINES AND LESSONS FOUND FROM LITERATURE 41

8 SUMMARY AND CONCLUSIONS 46

REFERENCES 48

1 Introduction

In traditional software engineering, evolution of software occurs only in the maintenance phase in software system life cycle. A system is developed according to the initial customer requirements, and has normally limited possibilities to incorporate new requirements. As the system cannot be adapted to newly emerging customer needs, it is replaced with a subsequent version. In this scheme, software lifecycle phases and responsibilities are rather straightforward ones. There is a group taking care of development of a new system and another group is maintaining the old systems and giving customer support. Moreover, there are normally just few system generations varying from each other that have to be taken care of.

In the product-line approach, delivered software systems are organized around commonalities that are shared by a family of products. The commonalities are implemented as software core assets. A common architecture is designed to support a certain amount of variability in the product level. Those functionalities neither supported by nor conflicting with the common architecture can be implemented in the product level. If functionalities created in product level are noticed to be usable also in other products, they can be generalized. Then they become new core assets. Thus, a product line is a continuously evolving organism. In order to manage this kind of evolution, organization and organizational responsibilities have to be defined to support evolution. In the product-line approach, released products do not necessarily share a single system version they are built on. Instead, they can be built on various versions of core assets and glued together with product specific code. This makes configuration management and tracking of different versions and variations more challenging than in traditional single software system development.

The idea of this report is to examine product-line structures, roles, and other aspects that should be taken into account when preparing for an evolving product line. Which are the aspects that form rigidity and which will help to evolve a product line? Which kind precautions should be taken at the organizational level and in software development effort to prevent future problems? How one can measure evolution and evolvability of software systems? How to trace the evolution of product line artifacts?

This report proceeds as follows. Chapter 2 gives some general advices considering product-line processes and plans, but mainly discusses about organizational alternatives for product lines. Chapter 3 discusses the important role that the recording of assumptions and design decisions has on smooth product line evolution, and gives a couple of solutions for the recording of design decisions. Chapter 4 presents metrics for evolution found from the literature. Chapter 5 presents two configuration management approaches for product lines that have certain principles defined considering evolution aspect. Chapter 6 presents three different proposals for visual format that can be used to trace evolution of product-line artifacts. Chapter 7 lists other advices and observations found from the literature that have effect on product-line evolution.

2 Organizations and processes for product lines

In the product line literature, two major concerns can be detected. How to organizationally separate core asset development (domain engineering) and product development (application engineering) effort? How the responsibilities on different assets are shared and divided? An integral part of the roles and responsibilities are the processes defined for the deployment of assets. An extensive set of organizational alternatives for product lines is represented in [Bos00]. Solutions offered are mainly domain engineering unit centric i.e. platform-focused and product-line literature does not much offer solutions that cannot be fitted into the basic principles offered by [Bos00]. The set of models represented in [Bos00] is also referred in [ClNo01]. Otherwise [ClNo01] gives only the overall goal to organizational structure: "Organizational structure chosen for product line production must assign decision making responsibilities to make sure that evolution and core assets are managed to bring long-term benefit to the entire organization". The quote emphasizes the fact that definition of responsibilities is more important than the actual organizational model used. While Section 2.2 represents the basic set of alternatives, Sections 2.3 and 2.4 present two different approaches with attached roles and responsibilities.

In this chapter, organizational alternatives proposed for software organization are skipped and only alternatives used in product-lines are discussed, apart from few traditional reuse oriented approaches that are also referred in the product-line literature. Good representation and evaluation for organizational alternatives for object oriented project organizations (Section 2.3. Project organization and roles) can be found from [Kiv00] which is available via Internet. Another publicly available source about different software development team models is lecture notes for [Sch02] and its presentation considering teams (Chapter 4 Teams). [Sch02] explains democratic, chief programmer and synchronize-and-stabilize approaches, and gives a sketchy view on extreme programming teams. What is notably different in product-line organization from other organizations is the product-line vision: the investment made now on reusable quality core assets will pay back in future. For the smooth evolution, it is essential to define responsibilities between development of core assets and products, which can be seen in the product-line organization and in its roles. The essential difference from traditional software reuse is that all artifacts can be core asset, not just software components although they often are of main importance. Another noticeable feature for product-line organizations is the existence product-line champion or product-line sponsor (or dedicated sponsor group). The role of the champion is to maintain the overall organizational knowledge of the product-line approach and keep the vision clear. No short goal decisions should be made that would disturb the long term goals of the product-line.

2.1 Establishing processes and plans for product lines

In product-line thinking, there is no single software process that is dealing just with software engineering activity. Instead, there are multiple processes in multiple levels: from business case analysis to testing and from core asset level to coordinating level. In this section are given a few general advices considering establishment of processes and plans for product lines. Many times the organizations resemble the product line itself, but the organizational roles with related practices and processes are those, which will help survive with evolving product line.

In product line thinking all the artifacts are considered as possibly reusable assets that can be used to smooth future efforts. According to [ClNo01] reusable assets may include in addition to normal software components, for example

- training specific to product line,
- business case for the use of a product line for a set of products,
- set of identified risks for building products for product line.

Clements and Northrop emphasize that each core asset should have a process associated with it [ClNo01]. The process specifies how asset will be used in the development of actual products. An example process might state following guidelines when a new functionality is included into the product line:

- use the product line requirements as the baseline requirements,
- specify the variation requirement for any allowed variation point,
- add any requirements outside the specified product line requirements,
- validate that the variations and extensions can be supported by the architecture.

This kind of a process may also specify the automated tool support to accomplish these tasks, and provide automated procedures such as inform that document is ready for review or send a change request for core asset developers. Processes should also tell how the core asset base will be updated as the product line evolves and processes should be attached between core asset developers and asset users. Defined processes set bounds for each person's roles and responsibilities, and aim for efficient and successful collaboration. Electronic documentation about processes should provide only views according to roles of users and hide the unnecessary information for the reader's role. [ClNo01]

These individual processes form a production plan. Production plan can range from a detailed model to a more informal guidebook. Items covered in a typical plan are

- goal that should be achieved by the execution of plan,
- strategy used to achieve the goal,
- intermediate states that will be achieved during execution of plan,
- assignable and discrete activities to achieve intermediate states and the goal,

- the resources that the planned activities are allowed to consume.

Production plan should be incorporated with metrics defined to measure organizational improvement as a result of product line (or other process improvement). [ClNo01]

Before processes can be effectively used one has to define roles of individuals and communication interfaces, and offer enough knowledge for everyone involved in the form of training. During the execution of processes one has to follow-up defined measurements in intermediate states of execution. Process improvement can be achieved by using well-defined and efficient processes in future projects. Existing processes can be evolved by comparing them against different process models. When process is about to change impacts must be evaluated before execution of change in actual process. Additional aid can be gained by selecting tools that help the process.

Clements and Northrop list also characteristics (sometimes competing ones) of good plans [ClNo01]. A good plan

- is focused on the correct things in order to accomplish goals,
- is unambiguous (what is to be done and why, how it will be done and when and by whom and what resources are required),
- is brief in order to be easy to use and maintain,
- is sufficient in detail to be able to accomplish goal in spite of brevity,
- separates supporting information and volatile information from body of the plan,
- is internally and externally consistent (no contradictive sentences and dependencies must be identified and traceable),
- is usable combination of previously mentioned characteristics.

Practices mentioned above are mainly intended for technical management at project level. Clements and Northrop also list responsibilities for upper level of technical and organizational management [ClNo01]. Upper management should

- establish funding model and invest resources in the development and sustainment of the core assets,
- define adoption plan that describes the desired state of the organizations and strategy for achieving state,
- speed up cultural changes towards product-line approach and orchestrate technical activities in and iterations between the essential activities of core asset development and product development,
- ensure that these activities and the communication paths of the product line effort are documented in an operational concept,
- manage organization's external interfaces.

2.2 Organizational models and asset responsibility approaches

Bosch presents four organization models [Bos00]. The main principle in moving from an organization model to another is the size of the organization. The rule of thumb figures are offered for a number of personnel applicable with each of the models. Presented models are (recommended personnel size in parentheses):

- Development department (up to around 30 software-related staff members).
    - Products and product line assets are developed under a one unit and same staff participates in both core asset and product development projects.
- Business unit (up to 100 software -engineers in general case).
    - Each business unit concentrates on developing a certain product or product group and units share a common asset base. Common projects might be initiated to revise the asset base or to create new assets. The maintenance of a certain set of assets is given to a certain business unit based on fact how much each unit is using the set and how probable is that the unit is the one that will further develop the assets.
- Domain engineering unit (over 100 software engineers).
    - Develops and maintains reusable assets that are used by product engineering units (business units).
- Hierarchical domain engineering units (hundreds of software engineers/over 30 engineers in one domain engineering unit).
    - In this model, domain engineering units are specialized on assets for certain product lines although there is a domain engineering unit that maintains common platform for all products.

The main reason for moving from one model to another seems to be the avoidance of a situation when n-to-n communication will cause too much overhead. The purpose of re-organization is to break n-to-n communication net into n-to-one communication and force communication flow through certain communication points. Communication overhead is not the only reason for selecting appropriate organization model, e.g. different physical locations and cultures affect on decisions. Also advantages and disadvantages of different models are discussed in [Bos00]. Despite the fixed set of models offered here, it is good to remind oneself that the practice of product line approach is more a matter of attitude towards reuse-based software development than an organization model. Successful practice of product-line approach can be reached in any kind of organization but organization model may courage or discourage reuse.

More important than the actual structure of an organization are the responsibilities and processes around the evolution of core assets. Related issues are

- Who is responsible for the maintenance of a certain set of assets?
- How and when assets are released to the users of an asset?
- How the feedback considering enhancement of assets is handled?
- What to do in the case of responsibility conflicts?

[Bos00] represents four asset responsibility models [Bos00].

- **Unconstrained model**. Any business unit can extend functionality of any shared component but each unit is also responsible to verify that changes do not conflict with existing functionality and quality attributes are not affected.

- **Asset responsibles**. A revised asset is reviewed by an asset responsible who makes the decision if enhancements to an asset are such that they follow the best interests of the whole organization and the generality level is sufficient to accept the software changes as a subsequent version of the asset. If changes are too product specific, those are regarded as product variation that is to be maintained as a part of product code, not as a part of the shared asset base.

- **Mixed responsibility**. Maintenance responsibility of assets is shared between business units. A certain set of the assets is assigned to a unit that does the most extensive and advanced use of the asset. Division is made to diminish communication overhead because the most proposals for enhancements are originated from the same unit. Other units have no rights to modify the asset, and they have to make an engineering change proposal for a change or for an extension of the asset.

Table 1 contains an evaluation of the represented responsibility models based on the generality and the evolution of maintained assets.

Evaluated aspect	Unconstrained model	Asset responsibles	Mixed responsibility
Generality of assets	Especially the software assets may be driven in too product specific direction which degrades assets.	Overall generality is verified by a responsible but the responsible can be overridden by higher management in order to satisfy e.g. time-to-market requirements.	Generality is checked by a responsible unit which enforces generality for proposals that are from the other units. Approach offers a temptation to optimize assets for the own purposes of the unit.
Evolution	Evolution is not coordinated and the results of changes are only evaluated by the originator of evolution (=easily skipped).	The verification of the impacts of changes is coordinated but not the evolution itself. Some overhead exists because a verifier differs from the originator of evolution.	Evolution is coordinated by a certain organizational unit and changes are made and verified by the same unit. Approach causes delays in evolution in case proposals are made from the outside the responsible unit.

Table 1 Evaluation of responsibility models

Some conclusions drawn from the above presentation are listed in the following:

- Unconstrained model ruins the whole product-line approach.
- An asset responsible should have enough authority and time reserved for the task to be able to maintain the integrity of the asset he or she is responsible for.
- Sharing of responsibilities between assets needs some serious thinking in order to minimize delays caused by bureaucracy.
- Units have to agree upon the response times for the request outside of the own unit (either schedule or discard) in order to keep other units happy. If other units cannot get a service good enough, they will create their own solutions. Then product-line approach is ruined once again.

Clements and Northrop suggest that before the adoption of a new candidate to a core asset base happens, at least two component representatives must sponsor the new candidate which ensures usability at least in two products [ClNo01]. Owen model represented in [TCO00], adds the idea of changing responsibilities (see, Section 2.4.). If an asset is experiencing a considerable change, the responsibility is moved to a project/unit that benefits most from the change. In addition to responsibility sharing, also the core asset update schedules have to be agreed between core asset developers and product units. Different product units have different response requirements from core asset developers. Some need early beta versions with lots of new features, and some prefer versions that are as stable as possible with fewer features.

2.3 An example of domain engineering centric approach

The Family-Based Software Development Process (FAST) represented in [WeLa99] is an example of domain engineering centric approach. Development is strictly divided between domain engineering and application engineering phases. Also the organization and roles are divided accordingly. FAST is appropriate in the situation where variation in the end product can be captured by a central architecture and variations can be parameterized out of the central structure by help of an application description language, and only minor modifications are made after the generation. One of the responsibilities of the domain engineering unit is to get needs from domain experts, and develop an application description language and tools around it. According to the philosophy, no significant development is made in application engineering units. In a case where a major improvement is needed, it is given as feedback from application engineers to domain engineering unit. The next version of the language and tools probably support the new feature, if it is seen usable for multiple products, and it can be generalized or parameterized for the needs of the different products of the family. Although the division is made between domain and application engineering units, it is only the division inside a domain. This domain can be a sub-domain of a larger domain, and the whole product-line organization can consist of multiple domains and their sub-domains; the knowledge how to integrate the artifacts of these domains to form a concrete product forms also a domain of its own [WeLa99]. In the basic division between the application and domain engineering unit the asset responsibility is a rather straightforward subject. In this model core asset responsibility is always static unless new sub-domain divisions are being made. Domain engineering units of each domain have the responsibility and decision power over their core assets, and other units are giving feedback and change proposals.

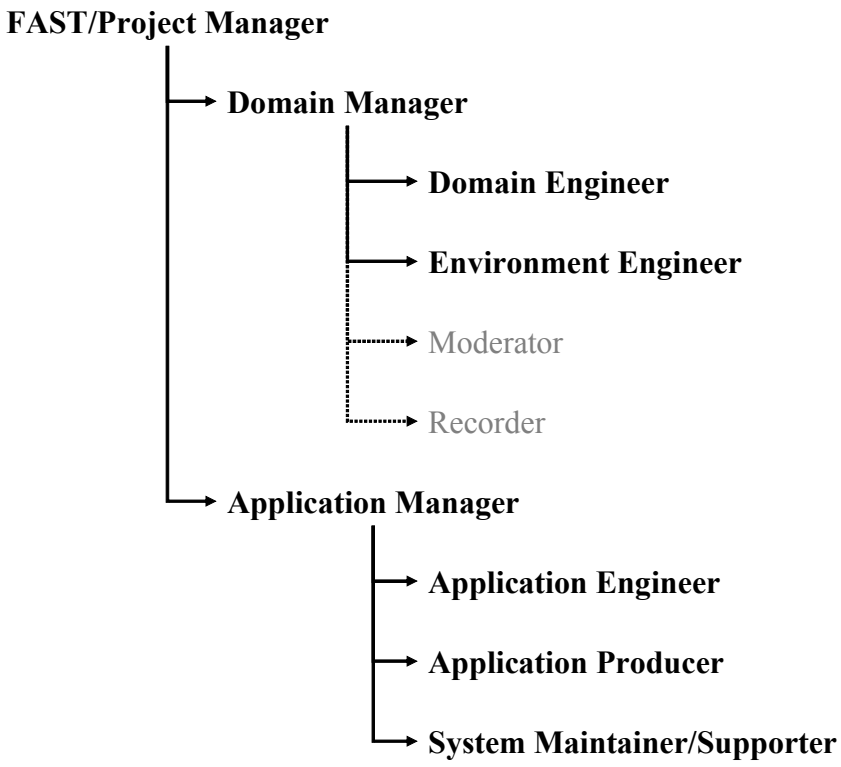

Figure 1 FAST role hierarchy [WeLa99]

The roles in the FAST organization (Figure 1) also reflect the division between domain and application engineering units. The hierarchy in the picture represents traditional supervise and report-to relationships. All the other roles are organizational posts except Moderator and Recorder which are acknowledged with duties but are not organizational positions in an organization. Instead, some of the domain engineers are probably given the duties of these roles (though it is not explicitly said in [WeLa99]).

FAST/Project manager manages the engineering and the evolution of the domain and the engineering of the applications in the domain. FAST/Project manager acts as domain level product-line champion/sponsor in FAST approach; [WeLa99] does not take into account coordination between domains that form the product-line as whole. FAST/Project manager is responsible for a process for creating application software system as members of a family and customer satisfaction. The other manager that responsible and interested in customer satisfaction is Application Manager which manages the production of applications. Domain Manager manages the engineering of the domain but otherwise have very similar responsibilities compared to Application Manager. Both managers monitor progress, set priorities, and allocate resources for needed tasks. It is not said explicitly in [WeLa99] but probably Application Manager is in frequent contact with Domain Manager and forward feedback from application development to domain engineering. Moreover, domain and application managers

negotiate together about priorities. All managers help in reviewing of artifacts if needed and share analysis responsibilities. These common analysis responsibilities are implementation assurance analysis, defect analysis, risk analysis, progress analysis, and resource analysis. [WeLa99]

In a case of change request, Domain Manager decides whether a proposed change is a domain change (possibly new family members) or just an implementation change needed in application engineering environment to be able to accommodate to a wanted change. Domain manager evaluates if the implementation or domain change is worth doing and either accepts or rejects it. Domain Engineer carries out activities needed to produce the domain model by using current software development methods, and by participating in the types of activities needed to create and maintain families. With the help of Environment Engineer, he or she creates a standard application engineering process and designs the application engineering environment. The Environment Engineer is responsible for carrying out activities needed to implement the domain model as application engineering environment. Environment Engineer creates an implementation library consisting of code, document, and test templates. As a part of the development of the application engineering environment he or she writes user's guide, training material, and reference manual for the environment. Moderator and Recorder both have the same responsibilities and target but from the different point of view. Their target is to establish a list about domain commonalities and variabilities, initialize domain term dictionary, and handle parameter sets used to produce needed variabilities. Moderator moderates the commonality analysis process by guiding and directing all group discussions, and Recorder records the result of the commonality analysis process. [WeLa99]

Application Engineer and Application Producer are responsible for creating new products by generating new family members in the limits of capabilities of the application engineering environment. Application Engineer determines and validates customer requirements. Application Producer is responsible for integrating system produced by application engineers(s) with any software that is not produced using the application engineering environment and he or she assists engineer to model the application. System Maintainer/Supporter is responsible for delivering the application to the customer and maintaining it after it is delivered. This role combines the skills and activities of Application Engineer and Producer. [WeLa99]

All the responsibilities in the FAST approach are well defined from the evolution point of view. One downside of the approach is the natural rigidity of the structure. Delays between a feature request and new application engineering environment supporting the requested feature may be substantial. Another downside is the requirement that domain variabilities must be able to capture into a parameterizable format, which may require a lot of effort. The generative approach is the most suitable one for mass markets. Common investment is paid back, if one is able to produce quality software products with minor variations in huge quantities, and the time window of deployment is long enough to bring back the original investment.

2.4 An example of product-focused approach

A community of development teams called "The Owen Firmware Cooperative" has been used in two product divisions of Hewlett-Packard in USA to produce firmware for printers and all-in-one products (the name of cooperative has changed because of confidentiality) [TCO00]. The product teams of the community are organized into a software cooperative in a very similar fashion compared to traditional cooperative organizations. A traditional cooperative organization consists of members with same economic interests. They may share manufacturing equipment, marketing and supplying channels in order to gain joint economic benefit. In a case of a software cooperative, members share managerial support, software assets and development practices in order to get common benefits. Individual organizational units are able to develop products in less calendar time and cheaper than they could do in isolated efforts. Cooperative approach still preserves product-oriented focus and allows doing of division specific decisions when appropriate. The rest of the section is based on the ideas presented in [TCO00].

At the time of the article the approach was expected to extend to other divisions and locations in near future. The model has reached its goals what comes to time-to-market, productivity and reuse figures. The model is represented as a product-focused approach instead of a platform-focused one to organize development of core assets. At the beginning of the paper a short comparison is made against a platform-focused view. The article claims that the following success criteria for platform-focused development are hard to maintain over long periods of time:

- good communication between platform and product teams,
- anticipation of future requirements for reuse platform.
- good knowledge of platform and efficient reuse of it in products.
- management support for platform that has no direct revenue stream.

The architecture of Owen-based products is represented by help of topology diagrams (see Section 6.2 and Figure 6). At the start of a new project, the topology is constructed by the project manager, the project lead, the division architect, and the asset manager. Each component is looked at and an attempt is made to find the closest match in the repository of components.

Owen components are large-grained components offering a logical block of functionality with well-defined interface. To support loose coupling between components Owen uses runtime binding for connecting components to interfaces. This is enabled by System Manager –component. Loose component coupling is regarded as a vital factor in facilitating the distributed and empowered development paradigm used by Owen.

Some Owen principles and guidelines are listed in the following:

- The services structure of the system is a hierarchy of components in client-server relationships in order to avoid circular references and control loops.
- Code itself is valued as an important asset, and is kept in healthy condition e.g. with the help of asset leads.
- Owen uses c-sets in software configuration management. C-sets are non-interacting logical changes that make changes to multiple files. The idea is to isolate changes as independent entities that can be included or excluded from a component according to product specific needs. It allows a product team to make isolated changes to the components that are only used by a certain product. The idea of c-set can be compared with the concept of task that is used in task-based configuration management environments.

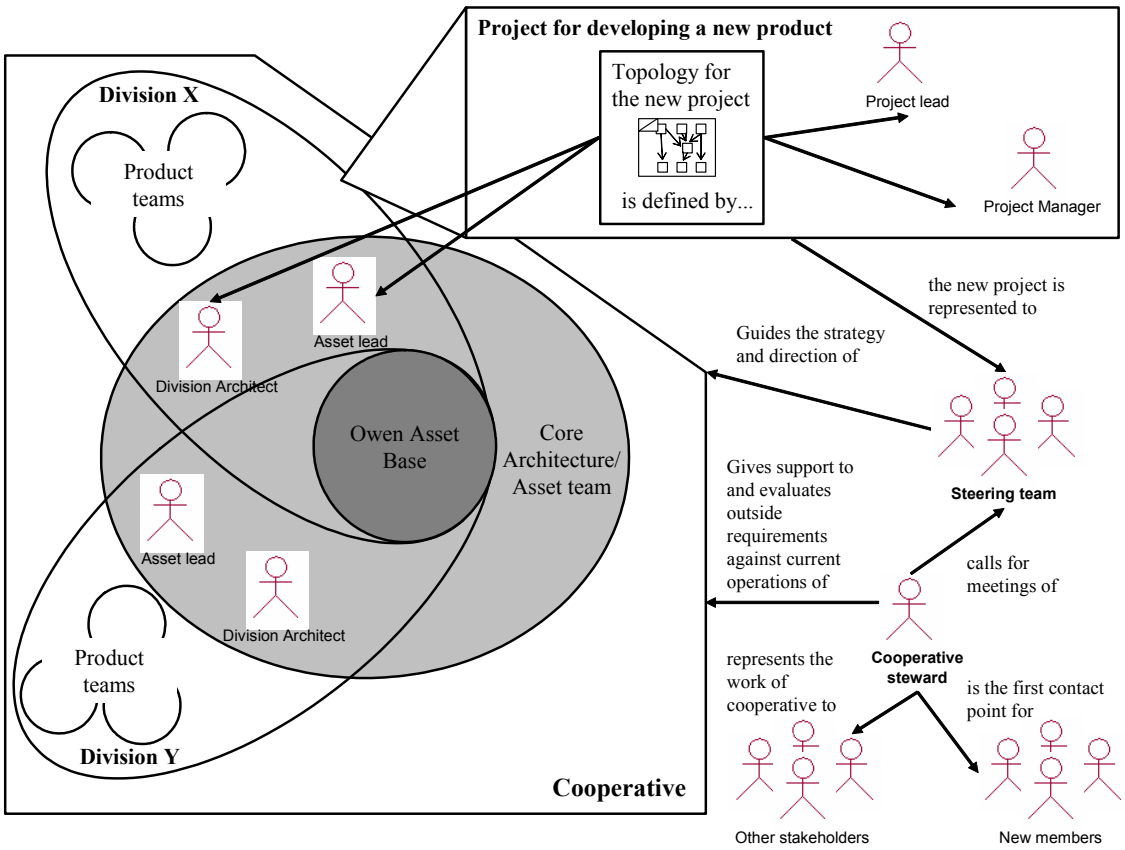

Figure 2 Owen's organizational model

In addition to project and product specific roles, Owen model (Figure 2) has the following groups and roles: steering team, cooperative steward, asset lead and architect role.

The purpose of steering team is to guide the strategy and direction of the cooperative, help to define and communicate goals of cooperative, and to establish and institutionalize its culture and values (higher level operation). The team makes operational decisions, resolves conflicts and brings new project teams to cooperative (lower level operation). All projects are represented on the team. Possible participants are project managers, technical leads, architects and asset leads. The team reinforces the shared goals of the cooperative, encourages desired behaviors and is able to make decisions in response to a changing environment.

These responsibilities of a steering team refer to the responsibilities of product-line champion or sponsor role mentioned in introductive paragraph of the beginning of Chapter 2. A person or a group must take care of those responsibilities regardless of the role name used in order to product-line approach be successful. In cooperative approach the sponsorship is divided between steering team and cooperative steward.

Cooperative steward is responsible for the smooth, sustainable operation of the cooperative. He/she convenes steering group meetings, acts as first point of contact for potential new community members and represents the work of the cooperative to management and other outside the community. He/she provides support to members of the community by empowering them to make decisions and get work done, by offering encouragement, and by helping community members feel valued. He/she must sense when architecture needs to be renewed or when current operation models need to be revised in order to match current business situation of the wider organization (strategic level).

Asset lead role is responsible for making the code base and related assets available to the project teams in the division and for making local developments and modifications available to other divisions in the cooperative. Asset leads work together to ensure the ongoing "health" and economic value of the asset base e.g. by assisting local engineering teams to comply with guidelines for architecture, documentation, and so on. The role may be combined with one of the architect positions, but it is important that there is a responsible person in each division.

Each division has an architect that is responsible for adapting and extending the architecture and framework to meet the requirements of new products. Architects are the primary technical decision making body and they work together to develop consensus on how the architecture should evolve and represents needs of their own divisions in the evolution of architecture. Architect is a contact point for other divisions when they need to make changes to components owned by the division of the architect. A division may have more than one architect but at least one is needed to ensure that product plans of the division are being included in the requirements on the architecture as it evolves. The role is not necessarily a full-time role.

Projects are responsible for project code but the owner of a component should provide some level of support and consultancy on the component. Ownership and ownership transition is decided according to the following rules:

- If a component is changed by 5% or less, it remains on the responsibility of the originating project or division.
- If a component needs to be changed by 25% or more, then the project needing the change becomes the new component owner.
- Ownership is negotiated between parties in situations between 5 and 25%.

In addition to above principles and practices, the article emphasizes the importance of social relationships. Possibilities for people to meet face-to-face should be organized in a periodic basis and whenever needed. It is an established practice that team members meet face to face at the outset of the work if engineers from different businesses in the cooperative will work closely together on a component, or rely heavily on each other's components.

The article lists also a set of general principles (not a reuse project, keep it simple, make it easy, value the community and get value from the community, value the code, ensure project team autonomy and empowerment, maximize utility, people matter) with observations that give actual meaning to the principles. A list of the aforementioned practices is also listed with comments on different perspectives (technical, business, social). At the end of the paper Owen model is also shortly compared with open-source projects.

2.5 Other organizational guidelines and alternatives

Software evolution can be seen as the result of multi-loop and multi-level feedback processes [Leh96, Leh00]. One should master this feedback nature of processes. Desire for functional extension can be seen as positive feedback that leads to pressure for growth and a need for continuing adaptation to external changes [Leh00]. Too rapid introduction of new features may lead to instability. On the responsibility of management is to observe the rate of change in response to information received about progress, system quality etc.; negative feedback, in the form of directives and controls to should be presented in order to limit the side effects of changes and drive evolution in the desired direction [Leh00]. Recognition of the feedback nature has lead to the following organizational guidelines [Leh00]:

- Determine organizational structures and domains within which the technical software development process including information, work flow and management control, and monitoring of changes. Both forward and feedback information should be controlled.
- In particular seek to identify the many informal communication links that are not a part of the formal management structure but play a continuing role in driving and directing the system evolution trajectory, and seek to establish their impact.

- Model the global structure using, for example, system dynamics approaches, calibrate and apply sensitivity analysis to determine the influence and relative importance of the paths and controls.
- In developing the models, include all personnel and activities that feedback information. And include also instructions that may be used to direct, control or change the process.
- In assessing process effectiveness, use these models to guide to identify interactions, improve planning and control strategies, evaluate alternatives and focus process changes on those activities likely to prove the most beneficial in terms of the organizational goals.
- In planning and managing further work, use the models as simulators to help determine the implications of the influences that are implied by the analysis.
- Devote some portion of evolution resources to complexity reduction of all sorts, restructuring, the removal of "dead" system elements and so on. Though primarily antiregressive, without immediate revenue benefit, this will help ensure future changeability, potential for future reliable and cost effective evolution. Hence, in the long run, the investment is profitable.

In addition to organizational alternatives represented in [Bos00] some conference proceeding papers refer to nowadays a little bit old-fashioned models targeting for software reuse. In [Rös98] the organization started as an ad hoc model but later evolved into the expert services model. Future target was to reach the level of a product center model. Both models aim for the organization supporting reuse. Fichman represents and evaluates both models as well as bunch of other ones [Fic01]. All the models are reuse centered models that can be applied at least into core asset development activity in product lines. Product centered and expert services model are described shortly in the following as explained in [Fic01]. The team producer model is also rated high in [Fic01], and therefore it is also included.

Product centered model is based on reuse center that employs reuse engineers and other reuse specialists (e.g., librarians, toolsmiths, methodologists). Center is responsible for identifying and evolving a formalized reuse process model and for developing and maintaining a robust reuse infrastructure. The center is also responsible for ensuring that application developers receive adequate training about reuse in general, and about the sanctioned reuse process in particular. Primary responsibility in day-to-day operations is to identify, acquire certify, classify and store reusable components and develop tools to support the storage and retrieval of components. Reusable components are acquired externally, from applications teams, or occasionally developed by the product center staff. [Fic01]

The expert services model is also created around a reuse center. There is though a crucial difference when compared to the previous model. Under the expert services model, reuse engineers from the reuse center are actually "loaned out" to applications development teams to assist them with the practice of reuse. Reuse engineers can help application engineers by pointing out potentially reusable components and by assisting in adaptation of components. While being "loaned" in application engineering, they are able to identify

best candidates from applications side for new reusable components. When application engineering project ends, the domain knowledge gathered can be brought back to reuse center. This model does not require a sophisticated reuse infrastructure or a large scale reuse training program as does the product center model. [Fic01]

In team producer model each team has its own supervisor, and resides at the same organizational level as two or more application teams. The reuse team and the application teams report through the same higher level manager. Under this model, as much or as little reuse is practiced as the higher level manager sees appropriate. [Fic01]

Highest rates Finchman gives to the expert services and team producer model. Both result in an efficient style of reuse because they rely on reuse specialists to do the most of the work of reuse. Both models shift the most of the costs and risk of reuse consumption and production outside application projects. Both models have high indirect costs, and require major structural changes to a development organization. Expert services model has high costs in form of the reuse center itself but does not require a sophisticated reuse infrastructure or a large scale reuse training program as does the product center model. Probable non-willingness to have outside members in application projects has been mentioned as another disadvantage to the expert services model. The disadvantage of the team producer approach is that it requires circumstances where multiple teams have high potential for reusability and both are reporting through the same manager. [Fic01]

Because product centered model can be thought as bunch of wise guys trying to give aid to others, it is not very probable to succeed without very good communication mechanisms between application engineering and reuse center. In expert services model reuse engineers are forced to use their own reuse components in real project circumstances. That guarantees true feedback and willingness to develop reusable components that are even easier and appropriate for future reuse.

As an interesting observation, Fichman lists possibilities wherein the lifecycle process efforts to make components reusable should occur. Following five options for reuse production were offered [Fic01]:

- pre-project domain analysis,
- in-project domain analysis,
- in-project generalization,
- post-project generalization,
- next-project generalization.

In product-line perspective, evolution of core assets after initial development happens either as post-project or next-project generalization.

3 Recording of assumptions and design decisions

3.1 On the importance of assumptions and design decisions on evolution

Software design process is based on multiple assumptions about the surrounding world in order to get software to fit into some mould that is formed based on these assumptions and constraints. Constraints affecting software development process are, for example, available funding, hardware resources, staff resources, time-to-market pressures, and requirements for software product. These assumptions and constraints are instantiated in the form of design decisions that pave the way to the design process. As time goes by, some constraints will vanish, and a part of assumptions will become invalid. That implies an inevitable change that has to happen in software. In that phase, it is invaluable to have knowledge about original assumptions, constraints and design decisions. Impacts of changes are impossible to estimate without knowing both the set of original assumptions and constraints and the current situation. The importance of the matter is extensively noted among the research community and industry. Among other references, importance is noted in [SWK01] that is an intermediate outcome of Swebok-project. The project attempts to characterize software engineering body of knowledge and provide topical access into it. Information in [SWK01] is gathered from recognized publications. A chapter about software maintenance states the following: "Software designed for maintainability facilitates impact analysis". Based on the previously presented logic, the impact analysis can be only facilitated if the design decisions and rationales behind them are recorded.

Lehman suggests that application and domain boundaries should be identified at the start of software development process [Leh98]. The implementation of software system is based on the assumptions and constraints that are derived from the identified system boundaries. These assumptions and constraints should be reviewed regularly, and there should be a pre-defined processes to recognize, capture and record assumptions, and assumptions should be reviewed when system bounds change (e.g. because application or domain changes). Also the likelihood of future change for each assumption should be recorded. These likelihoods will make the review process easier because one can concentrate on the assumptions that are most probable to change. In the product line point of view, this review process can be compared to a product line architecture assessment in which change scenarios are used to validate architecture against changing assumptions.

Lehman has examined software evolution issues since 80's, written many publications on issue, and invented the laws of software evolution. The basic idea of these laws is that when a software system grows big enough, the evolution will be constrained by its system dynamics. Lately he has revisited these laws and stated that software evolution is defined by multi-loop and multi-level feedback processes and these should be recognized in order to manage evolution successfully [Leh96]. These feedback processes are generated from different stakeholders and hence from different product line practices. His research has been extended during years 1996 to 2001 in the form of FEAST project (http://www.doc.ic.ac.uk/~mml/ feast/). A part of the work is to examine the evolution of

existing systems and, for example, find formulas to calculate system growth after a certain number of releases in large software systems. The other part of the work is to recognize feedback loops. Many publications raise more questions than give answers but in [Leh00] there are many guidelines considering recording and reviewing of assumptions and about metrics that can be used to track system growth (Chapter 4).

Another early bird in software evolution has been Parnas who already in 1994 has stated that every design should be reviewed and approved by someone whose responsibilities are for the long-term future of the product e.g. by maintenance personnel long before there is code [Par94]. Karhinen and Kuusela claim that recording alleviates analyzing of impact of change request and the implementation of change without degrading architecture by offering design decision and rationales (assumptions, constraints) behind them [KaKu98]. A noteworthy additional point to mention is the importance of active design decisions that are made but never recorded in a traceable way. The fact that design decisions are implicit in a program can cause at worst the rewrite of considerable sections. For example, Svahnberg and Bosch tell about a real world case where the attempt to update a read-only file system into a read-write system was actually implemented as rewrite from scratch because the earlier active decision was to write just a read-only file system, and nothing more (no support for planned evolution) [SvBo99]. All the places obeying the active decision could not be tracked, and it was more economical to rewrite the whole software.

3.2 Solutions for recording of design decisions

One way to record design decision is to use an architectural description language (ADL). Architectural description languages are not handled in here because they are often domain specific and there is a lot of literature dedicated to examine them, i.e. area is a large enough for its own research.

An alternative way to record design decisions is a design decision tree (DDT). The implementation of change requests normally trusts on local information of program, and system wide impacts are not taken into account. Rationales behind the earlier decisions should be known in order to estimate the impacts of a change and needed resources. Design decision trees are targeted to help this problem. [KaKu98]

DDT is a directed acyclic graph consisting of nodes that have three fields [JRL00]:

- actual design decision,
- requirements and constraints that must be fulfilled,
- implications and consequences of the decision.

Decisions can be expressed e.g. as design patterns. Each node shows reasoning behind one essential design decision, alternatives considered and consequences and the later decisions affected by this decision. Decisions proceed from a more general design decision to a more specific one. Earlier decisions are more expensive to reverse than the late ones. Therefore, it is a useful heuristic to address requirements based on the tightness

of the constraints they impose on the design; decisions that introduce fewer constraints may be made first because they are less likely to be reversed [JRL00]. This advice also helps one to avoid making of premature design decisions. "A design choice is premature if a better design choice is ignored because of the design information in the requirements document and the premature design decision could have been taken at later time" [ESAPS01]. The design decision tree notation also promotes comparison of different tactics before committing to a certain set of design decisions. The DDT notation is especially appropriate for quality requirements and requirements that describe overall properties of the system and are relevant during the architecture design phase. Only the architecturally significant decisions are useful to represent in the DDT format, not all the requirements.

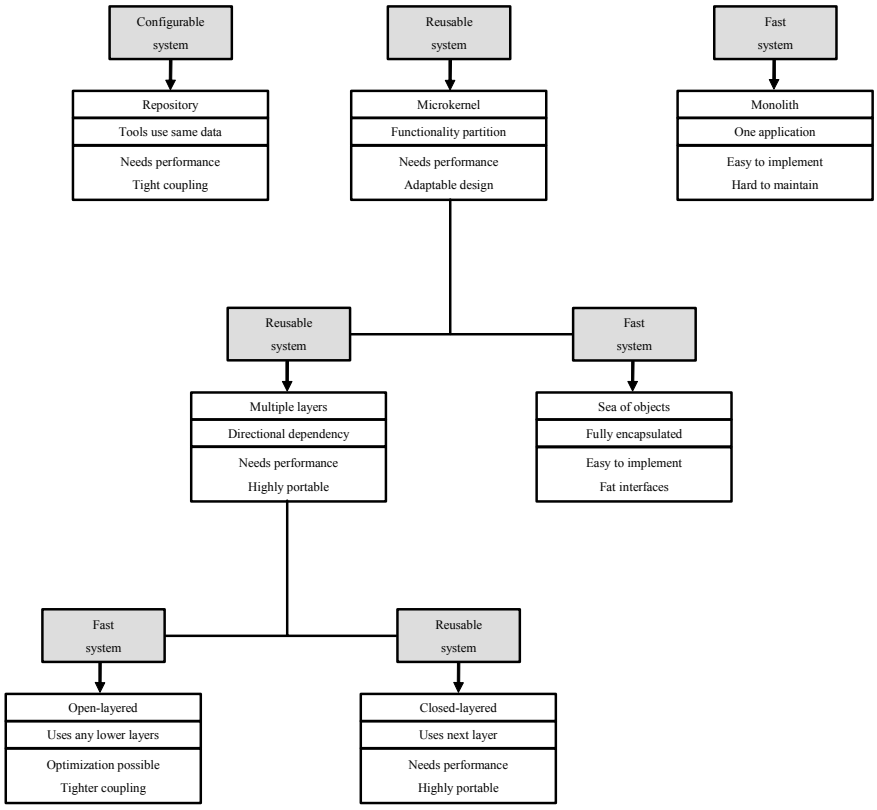

Figure 3 An example of design decision tree [Sav00]

In the DDT notation (Figure 3) each node represents a taken design decision. It is guarded by an entry criterion that is placed over the arc leading to the node. The entry criterion is the most general constraint that the decision must fulfill in order for it to be taken, e.g. fast performance or robust operation. Branching for alternative designs and their differing qualities happens in the context of preceding design decisions and problem requirements. Independent decisions may be ordered based on the tightness of constraints and variance is handled as an additional requirement. Based on the DDT forest we are able to show first decision that should be reversed by the change and estimate costs (and

resources needed) based on the place in tree. DDT provides a way to argue about different tactics and accumulate knowledge about the evolution of the system. [KaKu98]

Kuusela and Savolainen introduce a method for requirements capturing analysis [KuSa00]. The target for the analysis is to offer support for design decisions and offer mechanism to discuss about alternatives and to make sure that a new product of existing product family conforms to the family architecture. Requirements are structured into a definition hierarchy in which design objectives are defined by other design objectives or design decisions. The hierarchy is a logical AND tree, that is, the child requirements are used to define the meaning of the parent requirements [KuSa00]. The visualization of the definition hierarchy is represented on the left in Figure 4. The main idea of the tree is to split abstract requirements, e.g. overall quality attributes, into concrete requirements that can be implemented and therefore fulfilled. Abstract requirements are satisfied via the successful implementation and test of concrete requirements.

Savolainen enhances the DDT notation is towards new aspects by using definition hierarchies and impact values [ESAPS01, p.58]. The idea behind the combining of definition hierarchies and DDTs is to make sure that investment made in the requirement analysis phase is really harvested during the architecting and design phase. The approach binds the goals and product category division defined at the requirement phase to a design decisions. In order to illustrate which design decision is affected by which goals or sub goals, the idea of impact values is represented (Table 2).

Impact value	Classification	Rationale
0	No impact	This design decision does not affect the related requirement. (Normally these relations are not shown in the visualizations of the traces, however sometimes this may be stated to reduce misunderstandings.)
1	Low impact	This design decision has only limited impact in this dimension. The change in this decision may affect the satisfaction of the connected requirement.
2	Medium impact	This design decision has an impact in this dimension. The change in this decision probably affects the satisfaction of the connected requirement.
3	High impact	This decision has a decisive impact on the satisfaction of the related requirement. If the decision is changed the impact on the change must be verified.
4	Dominant decision	This decision has been made solely in order to satisfy the related requirement. Any change in either will have major effect in the related specifications.

Table 2 Impact value classification [ESAPS01]

The combined approach (Figure 4) does not only offer media to just compare different tactics. The format can be also used to represent variations between different product instances that are emphasizing different aspects of architecturally significant requirements. The combination gives insights about the requirements affected by a design decision. If decisions are made for a certain product that conflict with the abstract requirements for a product, one must either revise a decision or loose the requirements for the product. Also if the requirements change, one can identify places in the DDT affected by a change. When places are identified, one can better estimate the impact of changing or reversing some requirement.

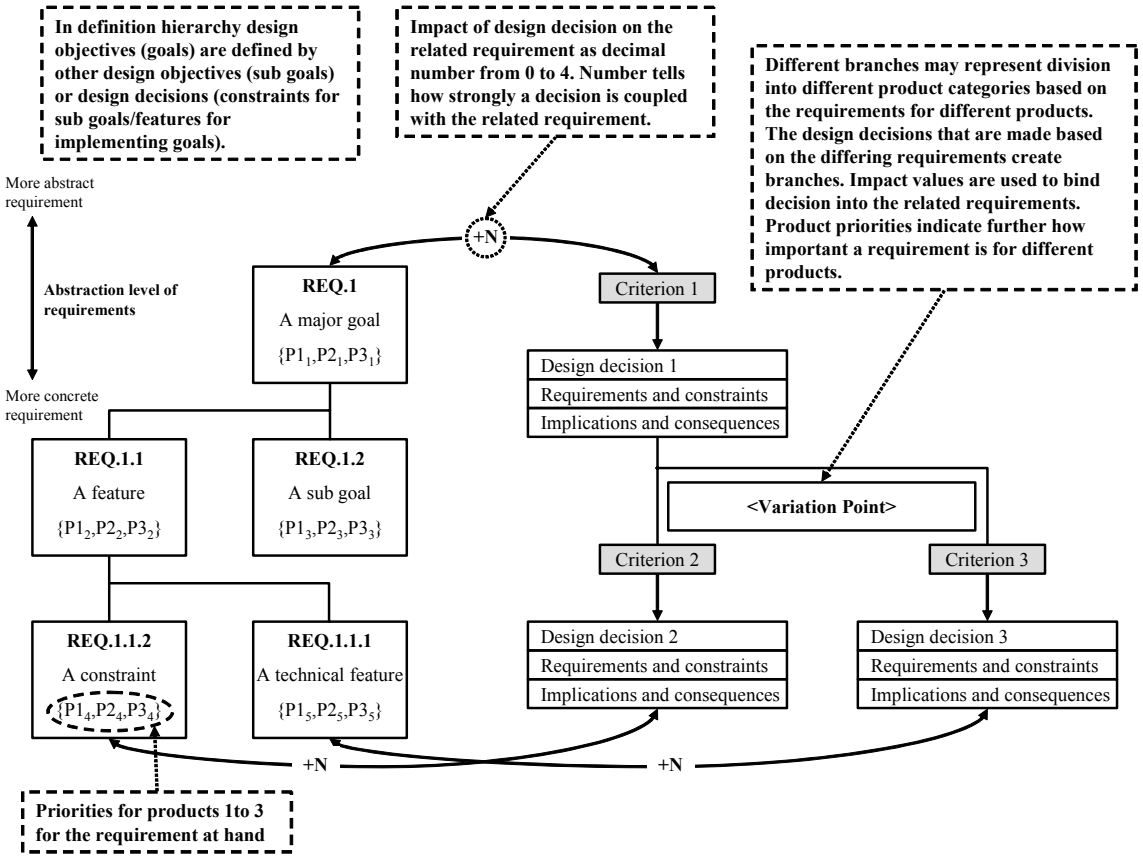

Figure 4 Describing product variations and requirement impacts with a DDT

The work-assessment for the addition of a new requirement could be made by attaching work amount estimates to the design nodes (e.g. man month value). Work amount estimates can be used to prioritize implementation of requirements. Some sub-requirement with low indication on overall system attributes but with a rather high work estimate can be deferred to a later development phase. These work amount estimates can be also reused later when either the maintenance or re-design is done on the system. There exist a requirement engineering tool based on the definition hierarchy method, but the tool is not publicly available.

Rösel represents a structure for documenting a design decision of a framework concept [Rös98]. This approach is more suitable for normal text based documentation of design decisions for framework concepts. In the following a generic structure for documentation of framework concepts is represented:

- name of concept,
- intent: purpose, problems to tackled,
- interface: provided,
- design,
    - Which classes/objects play role?
    - responsibilities and collaboration of the classes/objects
    - Which behavior is made available through this framework as a standard procedure?
- related design patterns/concepts,
- related classes/methods,
- sundry (optional),
    - This is a place to capture open points and documenting potential for improvements.
- discussion (optional),
    - In here is given a detailed description about aspects and design decisions
- reorganization (optional).
    - How to reorganize for better reusability?
    - design aspects: adaptability, flexibility, black-box vs. white-box framework
    - implementation aspects: readability, consistency, simplicity

4 Metrics for software evolution

4.1 Establishing and tracking metrics for product lines

Clements and Northrop present three essential activities that should be managed, as well as describes interaction between different practices to enable successful product line effort [ClNo01]. These three essential activities are core asset development, product development, and management. Defined metrics should be meaningful for each of these practices. Clements and Northrop do not define exact metrics but presents actions needed to establish and track metrics. Exploitation of metrics requires an initiation phase and an actual performance phase. During the initiation phase one has to

- designate goals to be tracked,
- define metrics that will be used to track progress toward those goals,
- identify the data that must be collected in order to derive those metrics,
- characterize the expected results and issues that may be discovered, based on any foreseen risks,
- specify how the data will be collected, when the data will be collected and by whom it will be collected.

At the performance phase one has to

- collect specified data,
- analyze and translate the collected data into metrics and compare them against the expectations that were characterized during the initiation phase,
- determine the actions that are needed to remedy any discovered issues,
- confirm whether those actions were appropriate for addressing those issues.

4.2 Proposed metrics for evolution

Although Clements and Northrop do not define metrics to be followed, it lists new interesting metrics for a core asset product manager. Information is received mainly from product development where the collection effort is guided by a product manager. Interesting metrics from product development are [ClNo01]:

- Usability of core assets: Which product assets and how often are used in product efforts?
- Quality of assets: How many bugs are found in core assets by the product developers?
- Infrastructure improvement: How much product efforts expend in finding, tailoring, and integrating assets?
- Opportunities for future asset or infrastructure: Where product efforts spend time otherwise?

Lehman presents a multitude of metrics to be measured. Measurement baseline consists of following items [Leh00]:

- process rates such as growth and faults
- changes over the entire system
    - units changed, units added, units removed and so on may be counted per release or per unit time.

Results over real time and over release sequence number must be compared and appropriate conclusions drawn. The second group of measures consists of numbers of people working with the system in various capacities. Measures to be followed are person days in categories such as specification, design, implementation, testing, integration and customer support, and costs related to these activities. Third group of measures relates to quality factor: pre-release and user reported faults, user take-up rates, installation time and effort, support effort etc.

The most nurtured idea in [Leh00] is the *safe rate of change*. According to [Leh00] the safe rate of change per release is constrained by the process dynamics. As the number, magnitude and orthogonality to system architecture of changes in a release increases, complexity and fault rate grow more than linearly. Successive releases focusing on fault fixing, performance enhancement, and structural clean up will be necessary to maintain system viability. Following measures are listed for giving indication of limits to safe change rates:

- models over a sequence of releases, for example, patterns of incremental growth,
- numbers of changes per release,
- numbers of elements changed (i.e. handled) per release or over a given time period.

Lehman gives rules that help to keep in pace of growth [Leh00]. In planning a new release or the content of a sequence of releases, the first step is to determine which of three possible scenarios exists. Let m be the mean of the incremental growth, mi the desired release content of the system in going from release i to release $i+1$, and s the standard deviation of the incremental growth both over a series of some five or so releases or time intervals. The scenarios may, for example, be differentiated by an indicator $m+2$. A release plan is identified as

- **safe**, when mi is less than or equal to m.
    - If the condition is fulfilled, growth at the desired rate may proceed safely.

- **risky**, when mi is greater than m but less than $m+2s$.
    - The release could succeed in terms of achieved functional scope but serious delivery delays, quality or other problems could arise. If pursued, it would be wise to plan for a follow-on clean-up release. Even if not planned, a zero growth release may to be required.
    - In addition one may implement a ‘preparation release’ that precleans the system, if limiting growth to around m is difficult or not possible.

Alternatively, allow for a longer release period to prepare to handle problems at integration, a higher that normal fault report rate, some user discontent.

- **unsafe**, when *mi* is close to or greater than *m+2s*.
    - It is likely to cause major problems and instability over one or more subsequent releases. At best, it is likely to require to be followed by a major clean up release which concentrates on fault fixing, documentation updating and anti-regressive work such as restructuring, the elimination of dead code etc.
    - Another alternative is to spread the work over two or more releases. Deliver the new functionality to users over two or more releases with mechanisms in place to return to older version, if necessary. Reinforce a support group in order to precede the release with one or more clean up releases or prepare for a fast follow on release to rectify problems that are likely to appear.
    - In either of the last two instances provision must be made for additional user support. Prepare to cope with and control a period of system instability, provide for a possible need for more than average customer support and accept that a major recovery release may be required.

Clements and Northrop present the idea of risky change level in a different context [CleNo01, p. 128]. It is suggested that a version control system should be able to store predefined levels of change and detect when considerable amount of changes have been done. Breaking the predefined levels activates the regression testing activity.

In [CJH01], measures are examined to define the evolvability of a software system. In the paper evolvability is defined as the capability of a software product to be evolved to continue to serve its customer in a cost effective way. The paper divides maintainability further into following sub-characteristics:

- **Analysability**. The capability of the software product to enable the identification of the part(s) to be modified.
- **Changeability**. The capability of the software product to enable a specified modification to be implemented.
- **Stability**. The capability of the software product to avoid unexpected effects from modifications of the software.
- **Testability**. The capability of the software product to enable modified software to be validated.
- **Compliance**. The capability of the software product to adhere to standards or conventions relating to maintainability.

The metrics defined by the ISO/IEC 9126 standard ([ISO91]) are examined in [CJH01]. As a part of the research external metrics for changeability presented in Table 3 were derived. "External product metrics cover properties visible to the users of a product; internal product metrics cover properties visible only to the development team" [Mey98]. The first two metrics in the table require special effort more than the final one. The first metric can be used in addition to proposed uses to evaluate the effort needed to mine legacy code in order to produce new product-line assets. The second one can be used to

evaluate usability of the core asset in different situations where it has been used in. The last formula is the easiest one to deploy. In case of defect correction, the number of changes made and the effort spent are normally collected via company's defect reporting system. A defect report should also list modified files and their versions to calculate the size of the changed software.

Metric name	Purpose of the Metrics	Measurement, formula and data element computations	Comments
Software change control capability	Can the modifier easily identify revised version? Can the modifier easily change the software to resolve problems?	X=A/B A = No. of change log data actually recorded B = No. of change log data planned to be recorded enough to trace software changes.	More relevant to analyzability than changeability.
Parameterized modifiability	Can the modifier easily change parameters to change software and resolve problems?	X=A/B A = No. of occasions when modifier changed software by changing parameters B = Total no. of changes made.	In the standard, this metric is designed as: X = 1 -A/B A= no. of occasions when modifier fails to change the software by using parameter. B= no. of occasions when modifier attempts to change software by using parameter. (B is subjective and hard to collect.)
Modification complexity	Can the modifier easily change the software to resolve problem?	T=Sum(A/B)/N A = Effort spent on a change B = Changed software size N = No. of changes	Size can either be measured in modules or LOC.

Table 3 External metrics for changeability [CJH01]

The authors of the paper also collected a large set of metrics from [ISO91] and evaluated them by two experienced software engineers. Metrics were evaluated based on the importance of a metric, on the ease of collecting the data for the metric from the historical data, and on the ease of future collection of the metric. The following final set of metrics was selected either according to the importance of the measure or according to both the importance and easiness of collection. The selection process can be, of course, criticized but Table 4 represents at least one kind of selection of metrics to measure evolution and evolvability according to different sub-characteristics (factors). Metrics

were gathered as two different sets, one that were rated to be easy to collect from historical data and those that were easy to collect in future if needed. From the both set were selected only those metrics that exceeded predefined limit for overall importance of the metric. In Table 4 those metrics that were excluded from historical metrics set but were included due to importance and easiness to collect in the future are marked with asterisk (*). Metrics are so generic that they can be applied in any type of software development. For example, a target of application for metric number 5 in product-line development could be the follow-up of the efficiency of feedback processes between core asset and product development.

No	Factor	Metric name	Measurement, formula and data element computations
1	Analysability (External)	Cause analysis efficiency	X=E/N E=Effort (staff-hour) spent on finding the cause of failure or parts to be modified. N=No. of change requests for which the parts to be modified have been identified.
2	Analysability (External)	Cause Analysis capability	X=1-A/B A=No. of changes for which the parts to be modified are still not found. B=Total no. of requested modifications.
3	Analysability (External)	Outstanding analysis effort (*)	X=A / B A=Estimated effort required to identify the parts to be changed. B=No. of unresolved change requests.
4	Changeability (External)	Modification complexity	T=Sum(A/B)/N A = Effort spent on a change B = Changed software size N = No. of changes
5	Changeability (External)	Change cycle efficiency (*)	T = Sum(Tu) /N Tu = Trc-Tsn Trc = Time at which user receives a revised version Tsn= Time at which user finishes sending request N = No. of times user receives a revised version from modifier.
6	Changeability (External)	Change implementation effort	X= E / N E = Effort to change the software N = No. of changes made
7	Changeability (External)	Outstanding change effort (*)	X = E /N E = Estimated effort to implement unresolved changes. N = No. of changes which have been successfully analyzed but not implemented yet.
8	Changeability (External)	Average change size	X=A/N A = Size of changes that have been made N = No. of changes made
9	Stability (External)	Change success ratio	X=Na/Ta Y=((Na/Ta)/(Nb/Tb)) Na=No. of occasions in which a user encounters failures during operation after software was changed. Nb=No. of occasions in which a user encounters failures during operation before software was changed. Ta=operation time during specified observation period after software was changed Tb=operation time during specified observation period before software was changed

10	Testability (External	Re-test efficiency	X=E / N E=Effort spent on testing to ensure that the software has been successfully modified. N=Total no. of modifications
11	Testability (External	Test re-design efficiency	X=E / N E=Effort spent in designing new test strategy and test cases N=Number of modifications
12	Analysability (Internal)	Change recordability (*)	X=A/B A=Number of changes to functions/modules with change comments confirmed in review B=Total number of functions/modules changed
13	Stability (Internal)	Change impact	X=1-A/B A=Number of detected adverse impacts after modifications B=Number of modifications made

Table 4 Metrics for evolvability [CJH01]

5 Configuration Management approaches

This chapter presents two configuration management approaches meant for product lines. They state some basic principles and rules which are used to cope with the evolving product line.

5.1 Approach for component based product populations

In [Omm01] the approach to following configuration management issues is represented: version management, temporary variation, permanent variation, build support and distributed development. Version management and temporary variation rely on capabilities of configuration management system but each package development team uses its own system.

A package team issues formal releases of a package with the version identification tag consisting of a major number, a minor number, and a 'patch letter'. Each this kind of release must be internally consistent i.e. all components should compile and build, and run without errors. Each package should be tested sufficiently for the products relying on the package. Customers download the release from the intranet and insert it in their local configuration management system. They do not see all of the detailed versions of the files in the package and only maintain the formal release history of the package. The golden rule is that each release of a package must be backward compatible with the previous release of the package. If the rule is applied, only the last release of each package is relevant, which simplifies version management. However, this is not always possible and a silver rule is applied. The silver rule states that all existing and relevant client software of the package should build also with the new release of the package.

The approach recognizes two kinds of temporary variation: "in the small" and "in the large". The first case happens when one developer wants to fix a bug while another developer adds a feature. The bug fix is separated into a temporary branch. Both tasks can be done in isolated working environments until changes in the temporary branch are integrated into a main branch. The latter case of temporary variation is used just before the release of a product utilizing the package. A new branch that is targeted just for product release is created. No new features are introduced into this branch, only bug fixes are done. All the new features are added into the main branch. The temporary branch is closed after product is released and work is continued in the main branch. Naturally, the bug fixes have to be propagated also into the main branch but the paper does no mention anything about it.

For permanent variation the approach relies on the component technology called Koala (presented in [Har01, p. 30]). The choice makes variation explicit in architectural level, instead of hiding it in the configuration management system. It also allows a late choice between compile time diversity and run-time diversity. Third argument is the possibility to exercise diversity outside the context of configuration management system, which is used to separate build support from configuration management.

Because the build support is separated from configuration management, the choice of configuration management system is left open, and only the recommendations are given for cost-efficiency. The tools used for build are not relevant in here but another argument behind the selection is to enable development also when one is not able to connect to a configuration management system. The final principle represented in the paper is a "small company approach" for the development of packages and for the distributed development. Packages can be delivered (cf. buying third-party components) between units without a demand for the use of a common configuration management system.

5.2 KobrA approach

KobrA is a method for component-based software represented in [ABB+02]. Information of this section is taken from the referenced book as a whole. In the method, evolution management (maintenance) is divided into three activities: configuration management, change management, and maintenance planning. Configuration management is a static management of different artifact incarnations that is responsible of synchronizations of different versions of a framework and products (the book uses word applications instead). Change management consists of methods and techniques involved in handling changes: categorization and evaluation of change requests and the assignment of appropriate formal change operators to change requests. Maintenance planning is responsible for constructing infrastructure for the change and configuration management activities. It customizes process according to organization and project factors, allocates resources for maintenance activities and defines how they should be applied.

KobrA divides coupling between framework and application into high and low coupling. High coupling means easier integration. Environmental facts, e.g. common skills and common asset base and nearly located framework and application teams, support high coupling. Negations, i.e. geographical distance, different skill sets, and different tools, promotes low coupling. KobrA approach assumes low coupling between the framework and its applications which mean that more complex and sophisticated techniques are needed in integration of modifications. The approach is based on a change-oriented model, i.e. versions are regarded as result of changes applied to some artifact in the product line. Maintenance tries to separate concerns as well as does the design approach. In the design approach, specification (abstract behavior of component), realization (environmental context realization of component, realization of component's specification), and implementation aspects are separated.

In the approach, every configuration item is self-contained and they are versioned independently of other configuration items. This is enabled by explicitly viewing dependencies as first class assets in their own right, which allows configuration items to maintain their own local configuration management and minimizes the number of artifact types needed in the configuration mechanism. In the normal approach a group of configuration items forms a configuration. The book is claims that by recognizing artifacts at all levels of granularity as configuration items (e.g. frameworks, applications, KobrA components etc.) and by viewing the dependencies between them as objects, the explicit concept of configurations is not needed. However, later in the book the word

configuration is used anyway, which contradicts the definition. One can notice also other contradictions along the way but still the approach is worth of examination.

The version state of configuration items is captured in the form of a version vector composed of four orthogonal values:

- **Revision**. A number that distinguishes different semantic incarnations of the artifact over time.
- **Variant**. A string that distinguishes co-existing incarnations of an artifact instantiated for the needs of particular product line family members.
- **Edition**. A number that distinguishes different cosmetic incarnations of the artifact over time.
- **Quality state**. A boolean value that indicates whether the artifact is in a satisfactory quality state according to the consistency rules.

The approach is not applicable without tool support. For example, consistency rule checking should be done automatically by maintenance system (on responsibility of maintenance planning activity). A physical versioning system that follows the approach should be implemented based on the KobrA metamodel.

The following items are connected to a change object: change request, change cause (corrective, adaptive, perfective or preventive), change justification (rationale, cost-benefit evaluation), and change description (details). Change object has links to artifacts affected by a change in order to record the impacts and identify and select the affected configuration dependencies and evolution graphs (see Figure 5 from Section 6.1). A change in some artifact sets it, and all the artifacts dependent on it, to the unstable state (Quality state is unchecked, i.e. boolean value is false). In a stable framework each KobrA component, with specifications and realizations (i.e. valid realizations of valid specifications) of a framework, are in stable state. A change can be a semantic change (the meaning of specification changes), when a new revision is created, or a cosmetic change (specification is described in different way but the meaning stays the same), when a new edition is created. After the modification all dependant components have to be checked against consistency rules. After the consistency check the edited component and the dependant components can be set to stable state. After revision all the dependant components have to be revised too, given a new revision number and run through a consistency check. Revisions or editions do not change the entities of an existing framework, instead new assets are created and the old version of framework is always present in project repository. Changes can be visualized by help of the evolution propagation graphs (see Figure 5).

Change requests belong to two main categories. They are either application specific requests originating from a customer or from application engineers, or framework change requests from domain experts or from framework engineers. The type of the change cause can be any of the cause types listed previously along with change object. A change can be implemented in application level if it is customer-specific and there is no need to support the changed feature in future releases. If the change relates also to other applications and

there is a need to support the changed feature in future, it is wiser to do change at framework level. If the change is implemented by changing the framework, the customer may also get changed and additional features that were not a part of the original product version. When and how changes are implemented is always a strategic decision and there are no fixed rules. Anyhow, the approach lists the following basic integration schemes:

- **Framework change integration**. An asset is changed within a framework, and the change is propagated to other assets within the framework to assure consistency.
- **Application change integration**. An asset is changed within an application and the change is propagated to other assets within the application to assure consistency.
- **Feedforward change reintegration**. Changes within a framework are propagated to the existing applications generated from the framework.
- **Feedback change reintegration**. Changes within an application are propagated to the framework from which it was generated.

Integration happens always within a framework or an application, and reintegration happens between a framework and an application.

6 How to visualize and trace evolution

Product-line engineering requires more than tracking of software artifacts and their version histories. One must be able to trace evolution of non-software artifacts, different variants of software artifacts and the owners of the core assets. Configuration management tools can be used to do some work, but still there should be visual formats to help the trace effort and offer medium for discussion.

6.1 Evolution graphs to trace version histories

Evolution graphs can be used to capture version histories [ABB+02]. An evolution graph, represented in Figure 5, is normally tree-shaped. Merges are only needed when there are temporary branches. A graph is created for each framework and for each application derived from the framework, and also for physical components and deliverables. Each node of the graph denotes a version of the corresponding artifact. A branch represents an evolution step and assigns increments to the version vector of an artifact (see Section 5.2 for the definition of version vector). Changes between two versions are described via the triple $\Delta = (S^+, S^-, T)$, where S^+ stands for a set of new assets, S^- for a set of retired assets and T for a set of base transformation relations. Transformation relations contain change rules used to transform a certain artifact into a new version. Triples are marked down to the evolution graph but nothing is said about how the change information presented by the triples is stored and shown to user when required. The idea is though that one should be able to trace every new, changed or retired asset between subsequent versions.

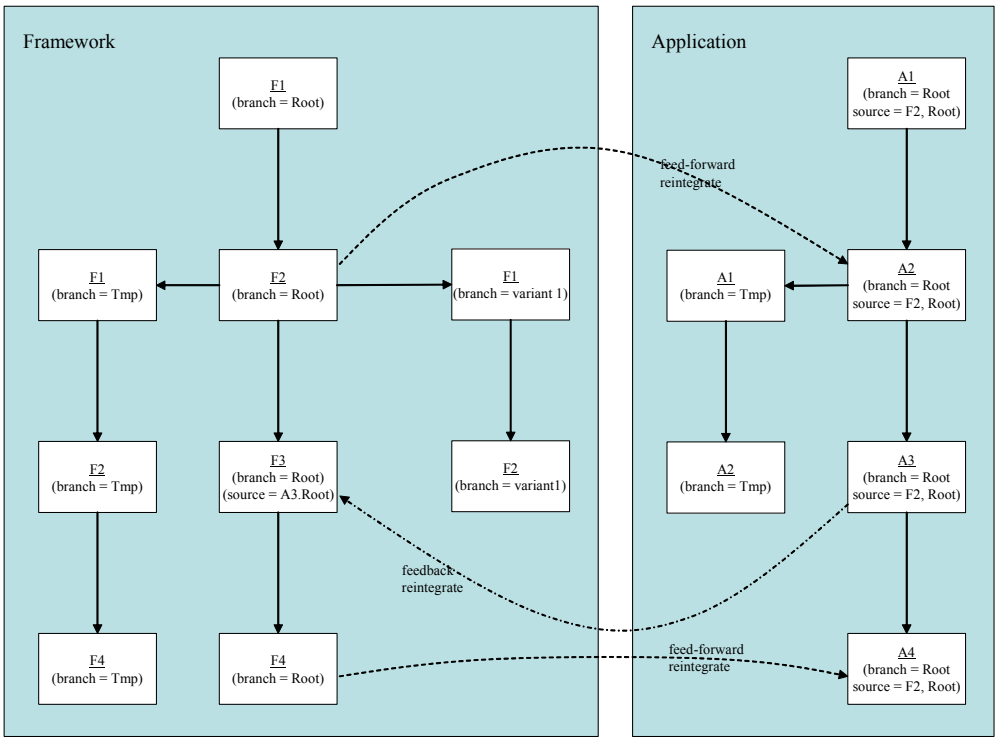

Figure 5 Evolution propagation graph [ABB+02]

6.2 Topology diagrams to trace component responsibilities and reuse-levels

The architecture of Owen-based products (see Section 2.3) is represented by help of topology diagrams that act as a "common language" for the architecture. Each diagram shows the components, their names and the interconnections (Figure 6). Each component is annotated with the name of the engineer responsible for its development. Color-coding is used to show

- which components are leveraged directly from existing Owen assets,
- which components are partially leveraged,
- which components are completely new,
- which engineers are working on which components.

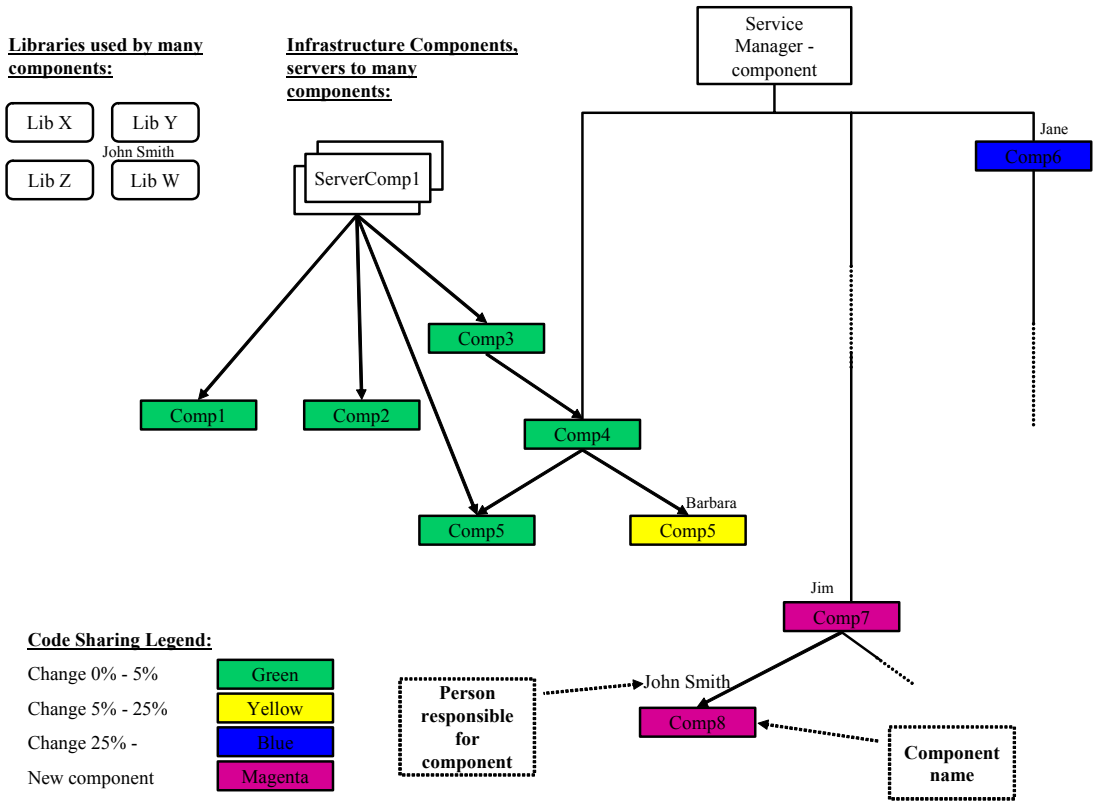

Figure 6 Topology diagram (adapted from [TCO00])

Topology diagrams offer a sense of how much Owen is offering "free", which components need to be modified, and how large portion of development is completely new. The new development is hoped to be focused on the areas that add value over and above previous products (which is the target). The diagrams act also as a quick reference

guide to find out who is doing what, and who is likely to be affected by a particular change (i.e. offers a tool for change impact analysis).

6.3 Dependency tracking between product-line artifacts

Schach and Tomer present two ways to visualize evolution of different software development artifacts, evolution trees and propagation graphs [ScTo00]. They can be used to trace dependencies between artifacts. The paper discusses trends in software engineering that emphasizes greater need for traceability. These trends are incremental development and product-line approach. In other words, software engineering is moving further away from the traditional design-and-implement-first-then-maintain approach. Incremental development is a solution targeted against changing requirements during software engineering project. Artifacts produced in the earlier phases in a project are refined as requirements are refined, i.e. maintenance is done already during the project. In the product-line approach maintenance happens in another form as products are no more developed from scratch, instead product development reuses existing artifacts. Existing software artifacts do not necessarily fit into a new product as they are, and artifacts go through adaptation before they are used in the target product. After adaptation the reused software artifacts need to be tested to confirm that they still fulfill their original purpose. To help the process one must be able to trace and reuse also the other artifacts associated with the software artifact: specification, design, test cases, test software, etc. As software product is adapted, related artifacts should be modified accordingly and adapted test software can be used to verify the correctness of adaptation.

The evolution tree (Figure 7) has two-dimensions: development and maintenance axis. If development is performed from scratch, the starting point is an initial (empty) artifact denoted by ϕ. Letters R_i, S_i, D_i, C_i denote versions of the requirements, the specification, the design, and the code, respectively. The product-line approach adds to the picture a third dimension, reuse. In this three-dimensional view, individual products form evolution tree planes and the evolution of core assets is captured on its own plane.

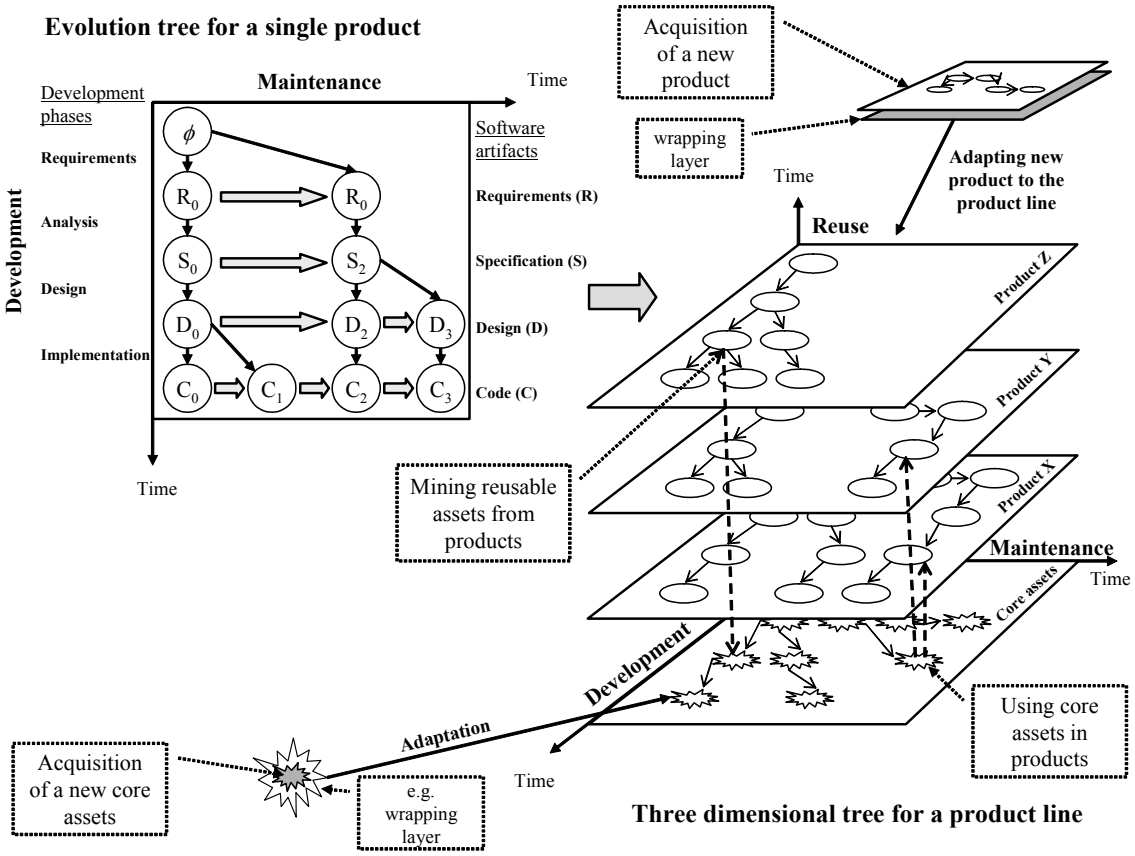

Figure 7 Evolution tree (adapted from [ScTo00])

The evolution tree idea represented in Figure 7 does not offer much traceability support as such, but gives an insight about the different level of maintenance activities that are part of the product-line approach. The figure represents a situation in which acquired software components or products cannot be reused as they are (black-box reuse) but are adapted instead (glass-box reuse). Schach and Tomer list the typical forms of glass-box reuse happening when existing products are mined for new core assets, or existing core assets are used in new products. In the following, a core asset base is seen by all product development units in an organization and a product base is just seen, and used by, an individual unit.

There are two approaches for using core assets in a product [ScTo00]:

- A core asset is copied into the product base and necessary product-specific adaptation is made.
- A core asset is modified and stored back to the core asset base before copying it to the product base (the modified and generalized asset is available also to other products needing the same functionality).

Correspondingly, there are two approaches to transfer a product component to a core asset base [ScTo00]:

- Product components are copied and then modified before storing them to the core asset base. Examples of possible modifications are generalization of the functionality or wrapping of legacy code behind the interface that is usable by other products.
- The product components modified in product base are copied to the core asset base without further modifications. This is the least likely situation according to the paper. An example of this kind of situation could be a core asset clone that has gone through some non-product specific improvements in the product base and the improvements are needed by other products.

The fundamental idea behind the propagation graph is the fact that a single requirement splits to multiple specification artifacts which may further generate new requirements at design level, and finally, a single design requirement will affect multiple implementation artifacts. During a maintenance task the impacts of the change has to be evaluated and the efficiency and the correctness of the solution is strongly dependent on the ability to locate the artifacts that are affected by the maintenance task. An example of propagation graph is represented in Figure 8.

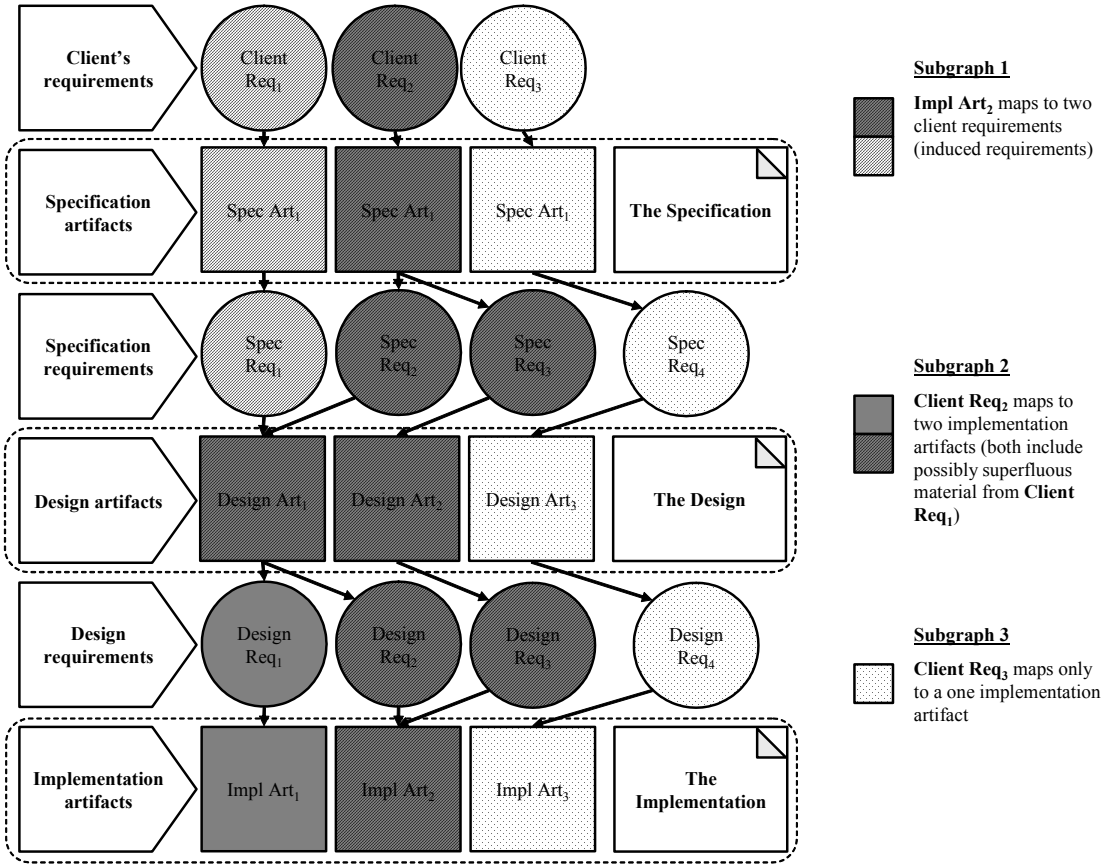

Figure 8 Propagation graph (adapted from [ScTo00])

The other aspect in the propagation graph is the way it is used to evaluate the fitness of an artifact for reuse. Schach and Tomer state the following: "An artifact can be reused only if the requirements that it implements are also requirements of the new product" [ScTo00]. In other words, the extracted subgraph of the propagation graph formed around an artifact should not conflict with the requirements of the product trying to reuse the artifact.

Three different examples of subgraphs are represented in Figure 8. The subgraph 1 implies an effort to reuse the implementation artifact 2. The subgraph 1 maps to the requirements 1 and 2. If both requirements are also the requirements of the target product, there are no problems. Large scale re-use can be achieved best if it is possible to use an implementation artifact that share same requirements between different products. If another requirement is an additional requirement, it is called an *induced requirement* according to the paper. In the case of a functional requirement, an implementation artifact may implement a feature that is never used in the target product. The induced requirements might be the reason why so many accidental reuse initiatives have failed and, because of the maintenance implications, it is almost always wrong to reuse artifacts that result in induced requirements that are not also requirements of the target product [ScTo00]. If either requirement 1 or requirement 2 in the subgraph 1 conflicts with the requirements of the target product, the implementation artifact cannot be used in the product. For example, if one of the requirements is conflicting with quality requirements of the product.

In the subgraph 2 of Figure 8 reuse is examined from the requirements point of view. In order to include the requirement 2 into a target product, one has to use the implementation artifacts 1 and 2. As traversing from the implementation artifacts back to the requirements (subgraph 1), one can see that also the requirement 1 is bound with the requirement 2 as already discussed in the previous paragraph. If the requirement 1 is neither the requirement of the target product nor the conflicting requirement, the requirement 1 is still producing superfluous material to the target product via the implementation artifacts. Too much of superfluous material may cause overhead but the worse problem is maintenance. Maintenance of a product gets hard if it is not known which part of the code is a part of the product and which part is superfluous.

The subgraph 3 of Figure 8 represents the ideal and non-problematic situation for reuse when a requirement maps exactly to one implementation artifact. That can be achieved in planned evolution but it is not very probable to happen in real life. This kind of a mapping is possible for functional requirements, but quality requirements almost always scatter into the multiple artifacts. A widely used way of implementing functional requirements is to group related functionalities into a same dynamic link library. In that way implementation artifacts almost always implement multiple requirements. On the other hand, this kind of grouping may promote large-scale reuse if organized well.

It is clear that propagation graphs cannot be maintained and used without tool support. The paper claims that there is a CASE tool under development to support evolution tree and propagation graph models. Nothing more specific about the matter was available via

the World Wide Web. Only another pointer to the CASE tool could be found from [SJW+02] that has been published this year.

7 Other guidelines and lessons found from literature

In many articles Lehman emphasizes the importance of recording and evaluation of recorded assumption as already mentioned in Section 3.1. In [Leh00] is discussed some points considering tracking, recording and reviewing of assumptions. In the requirements capturing and architecture definition phase it is natural to discuss about assumptions and constraints, but usually some decisions related to solution is deferred to a later development phase. Whatever is not included in the specification and is left to the judgment of the assignee or whoever he/she wishes to consult and any such decision must be confirmed as not being inconsistent with the specification and recorded [Leh00]. If these individual decisions are not recorded, they result as hidden assumptions in a software product restricting the evolution of the product. The recording of the assumptions should be the goal of every process in the every phase of development effort. In order to enable the recording of assumptions one has to ensure right conditions for capturing, recording, and reviewing the assumptions. The right conditions include training, defined documentation formats, change management procedures and availability of needed informational resources.

As an example of organizational actions Lehman gives the following [Leh00]:

- Train personnel to seek to capture and record assumptions, whether explicit or implicit, at all stages of the process in standard form and in a structure that will facilitate their being reviewed.
- Separate validation and implementation teams to improve questioning and control of assumptions.
- Evolution team members should have access to all appropriate domain specialists.

The following advice is given on documentation [Leh00]:

- Any decision not being inconsistent with the specification must either be documented in an exclusion document, or formally approved and added to the specification or to other appropriate user documentation. Users need to be informed of the assumptions that have effect on the use of software system. To treat this as a replacement for recording in the system documentation cannot be considered as best practice.

The following advices are given about reviewing of assumptions [Leh00]:

- Estimate and document the likelihood of change in the various areas of the application domains and their spread through the system to simplify subsequent detection of assumptions that may have become invalid as a result of changes.
- Assess the impact if the assumption was to be invalid. For example, what changes would need to be made to correct for the invalid assumption?
- Validate all assumptions by individuals with the necessary objective and contextual knowledge.
- Verify the validity of assumptions with users and/or other stakeholders.

- Review relevant portions of the assumption set at all stages of the evolution process to avoid design or implementation actions that invalidate even one of them.
- Review the assumptions at intervals to facilitate detection of any assumptions that have become invalid. Review should be guided by the likelihood or expectation of change or reviews should be triggered by events.

Rösel lists the following lessons as the most important ones in developing and utilizing product line framework [Rös98]:

- No framework can be a success without application success.
    - The amount of realized and delivered applications measures success.
    - The first two or three applications give the essential requirements.
    - Keep an open mind for future needs and flexibility but retain in reality.
    - There is no future until applications are 'out there' within a window of opportunity.
- There can be no framework without a customizable architecture.
    - It is important to find key concepts which are general and 'simple' enough to survive many years of framework usage.
- Object-oriented principles are more important than language choices.
- There is no framework acceptance without documentation.
- There is no framework without organizational changes.
    - One should be able to provide a win-win situation between framework suppliers and users.

As an example of sustaining key concept Rösel mentions design patterns which have provided a good payment. As another usable tool is mentioned design-by-contract, i.e. exact responsibilities for component operations with pre- and post-conditions are defined before implementation. [Rös98]

In [SLMH96] is presented an idea of reuse contracts that define a certain set of rules with associated documentation. This first article represents contracts between classes in same inheritance hierarchy of object oriented classes. The concept of reuse contracts has been later enhanced outside the single inheritance hierarchy. In [MeHo00] reuse contracts are renamed as evolution contracts and a notation based on UML is represented. At short glimpse the notation seems to be extensive but rather laborious even it probably enables a very accurate and fine-grained tracking tool and a basis for handling of software evolution. The idea of notation is especially to tackle with unanticipated evolution in contrast with variation mechanisms used in the product-line approach that are meant to solve future changes that can be foreseen. The topic is not further considered in this document but these two pointers are represented if someone wants to get more familiar with the idea. [SLMH96] is easier to tackle as it introduces the idea of reuse contracts and does not demand previous knowledge.

Svahnberg and Bosch list guidelines for software product line evolution [SvBo99]. Guidelines are derived based on case studies made in two companies. A revision of the same list is also represented in [ESAPS01] and the following summary with short excerpts is the combination of the lists.

Avoid adjusting component interfaces. This will happen sooner or later, but try to delay it. Interfaces can either expand or shrink. Expansion happens due to new requirements demand more functionality from component, but affects also components utilizing these new services. Shrinking occurs due to creation of a new component that handles the functionality handled before by some other component. Dependencies between calling components have to be tracked and the new component must be used instead. If the new functionality is well behaved, it can be added as a small module next to the previous implementations, and connect to them using the public interface. In most other cases, there is no solution but modifying the original source code.

Focus on making component interfaces general. When designing the architecture and the component interfaces, focus on how new implementations in components are added. Issues concerning this are where to keep functionality that at the design stage is perceived to be common to all implementations, and what to actually include in the interface. One way to find out if an interface is general enough is to make a thorough domain analysis and present a somewhat alternative view of how to evaluate the architecture and the interfaces for evolvability, based on maintenance scenarios prepared by developers with more or less experience.

Separate domain and application behavior. It is common that a component implements not only the domain functionality that it is supposed to support, but also a number of facilitating functionalities that are more focused on providing application structure and support. This comprises things like reference counting for memory management or dependency graphs to start up the components in the correct sequence. The problem is that such functionality has a tendency to evolve more than the domain functionality, in that frequently new features are requested to provide even better control. This kind of application functionality should be separated from the domain functionality, and that aggregate relationships are used as the only connector between the two. This will not help alleviating changes to the application functionality, but it will help in making the domain functionality more reusable, and less prone to require modifications as the application functionality changes.

Keep the software product line intact. Unless there are some very compelling reasons, for example that the new product family is written in a new language, one should strive to keep as much as possible from the other product families, i.e. to only create a branch of the product line. Often, however, even if a component does not provide all of the functionality one may wish, it will be cheaper to enhance the existing component to supply the new functionality and adopt the previous products to this, than it is to maintain two parallel versions of the same component, with all that this brings of bug fixes and synchronization problems. Sooner or later, a decision will be taken to join the two versions anyhow.

Detect and exploit common functionality in component implementations. Every domain component should have at least one library component in its tow, where common functionality between component implementations can be placed as soon as it is identified as shared between implementations. Development of a new implementation then also becomes a "reuse inspection" for the implementation used as a template. Trouble with this approach arises when it is realized that the functionality put into the library component was in fact just common for two of the implementations, and that all the other implementations use other functionality. Also, the impact of changing in the shared functionality may be hard to estimate.

Be open to rewriting components and implementations. There are situations when a particular implementation or maybe even an entire component cannot cope with a desired change. Rather than forcing the change into the component, which results in a patchy code and shortens the life of it even further, we suggest that the option to rewrite the component from scratch is considered. Even if the rewritten code does not get better than the original code it will certainly be better than the code produced by patching the previous implementation.

Avoid hidden assumptions and design decisions in the source code. The reason behind a need for rewrite is that some design decisions do not take on a first class representation in the architecture, and becomes hidden in the source code. When such assumptions or design decisions change, it is easier to rewrite the entire component, according to the previous guideline, rather than to change the assumption in the existing code. All design decisions should be given a first-class representation in the form of components or design patterns so that the effects of changing these decisions are minimized.

Keep the variation points below the architectural design phase. Variances should be defined and maintained in architectural design. When implementing a certain set of variations for a certain product, keep the variances of other product invisible. Implementation phase is not recommended place to introduce variations.

Karhinen and Kuusela present a good list about matters that cause inflexibility in large systems [KaKu98]. When modifying a system, one has to

- conform to the interface of those modules you communicate with,
- use data representation of those interfaces,
- use the data representation of all data stores,
- base your execution architecture on the existing operating system and its scheduling principles,
- use only the time slot allocated to you,
- conform to the fault tolerance mechanisms,
- respond to all other housekeeping requests,
- use only accepted development process and supported programming languages and tools.

In [BeRa00], an important notice is made about the importance of definition of common and domain specific vocabulary that will aid in all stages of software evolution. It will help communication in initial implementation and evolution phase but also during servicing phase. E.g. defect reports or requests for additional functionality are usually expressed in terms of the application domain concepts, and a substantial part of program comprehension is the location of the application domain concepts in the code. The program comprehension phase may be even more important in the servicing stage because the knowledge that team has during evolution phase is no more present, and the defined vocabulary may substantially help the comprehension.

8 Summary and conclusions

In this report was presented the basic organizational and asset responsibility models for product lines. Two organizational models, looking the product-line approach from different aspects, were presented. The importance of recording of assumptions and design decisions was discussed based on the literature background, and a couple of solutions for the recording of design decisions were presented. Some product-line specific metrics and more general metrics for measuring evolution and evolvability were presented. Two product-line oriented configuration management approaches were presented. The first one is used in practice in an industrial company and another, more academic approach, is a part of the larger software engineering model designed for model-driven software development. Three different visual formats for product-line evolution tracking were presented. The first one is used to visualize the evolution of a framework and applications derived from it. The second one is used to track asset responsibilities for different components and to visualize which software components are totally or partly reused and which components are parts of a new development. The third format, propagation graph, is used to visualize dependencies between different artifacts from requirements to implementation artifacts. Propagation graph and its subgraphs can be used to evaluate the fitness of different artifacts for reuse, after all dependencies are tracked and taken into account. In addition, miscellaneous guidelines found from literature related to software evolution were presented.

The presented organization models offer the basic framework for different type of organizations. The roles presented should exist in a one form or another in any kind of product-line organizations. A hierarchical product-line organization can compose of rather independent units with well defined contact points. One approach that was not explicitly discussed in the survey is to split product-line effort around the different kind of domain areas that require more or less different skill sets. Competence centers can be also formed around areas that are not application-domain specific, for example, around the knowledge how to integrate the assets provided by individual units to an end-product. For example, FAST approach offers a reference model for roles and their organization inside a domain specific unit and Owen model offers an approach for coordinating efforts between independent divisions.

Recording of assumptions and design decision is important in the development of product lines, as well as in the development of any large software system. The importance becomes greater as the expected lifetime of system grows. Naturally, one expects a long lifetime for product-lines, which is a prerequisite for getting return on the original investment made to a product-line.

There are not many metrics targeted for product-line development, but the presented metrics can be also used in product-line context. There is certainly a gap in research, what comes to the metrics that support development of evolving product-line.

In configuration management approaches for product lines, one must recognize the existence of variants and their handling (permanent and temporary variants). One must agree upon the rules, formats and ways of delivering reusable software packages between different units of product-line. An extremely big challenge is to ensure consistency of a framework or an application after a part of it changes. Nowadays consistency is ensured by regression testing. It is suggested that making the changes more explicit, representing them as first class assets in their own right, would enable the consistency checking.

Problems related to large-scale software reuse are rather well understood, as explained in [ScTo00]. Tool support is needed to enable efficient tracking of dependencies between different software and non-software artifacts. One should be able to track down dependencies, because they determine the reuse potential of an artifact and reveal possible conflicts between the reused artifact and a target application.

References

[ABB+02] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, Jörg Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[BeRa00] Keith Bennett, Vaclav Rajlich. *Software Maintenance and Evolution: a Roadmap.* In Anthony Finkelstein (Ed.), The Future of Software Engineering, ISBN 1-58113-253-0, ACM Press, 2000.

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and evolving a product-line approach.* Addison-Wesley, 2000.

[ClNo01] Paul Clements, Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2001.

[CJH01] Stephen Cook, He Ji, Rachel Harrison. *Dynamic and static views of software evolution*. Proceedings of IEEE International Conference on Software Maintenance (ICSM'01), pp. 592-601. IEEE Computer Sociecty Press, 2001.

[ESAPS01] *Change management and evolution support.* ESAPS Consortium Wide Deliverable, Siemens-WP3-010518, April 2001. Available from http://www.esi.es/esaps/public-pdf/CWD32-18-04-01.pdf (28.11.2002).

[Fic01] Robert G. Fichman, C.F. Kemerer. *Incentive Compatibility and Systematic Software Reuse*. Journal of Systems and Software, Volume: 57 Issue: 1, pp. 45-60. Elsevier Science, 2001.

[Har01] Maarit Harsu. *A Survey of Product-Line Architectures*. Tampere University of Technology, Software Systems Laboratory, Report 23, 2001.

[ISO91] International Standardization Organisation. *Information Technology - Software Product Evaluation – Quality Characteristics and Guidelines For Their Use.* First ed., ISO/IEC 9126. ISO/IEC, Geneva, Switzerland, 1991.

[JRL00] Mehdi Jazayeri, Alexander Ran, Frank van der Linden. *Software Architecture for Product Families – Principles and Practice*. Addison-Wesley, 2000.

[KaKu98] Anssi Karhinen, Juha Kuusela. *Structuring Design Decisions for Evolution.* Proceedings of Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, February 1998. Springer-Verlag, 1998.

[Kiv00] Kari Kivistö. *A third generation object-oriented process model: Roles and architectures in focus*. Ph.D. Thesis. Department of Information Processing Science, University of Oulu, 2000. Available from http://herkules.oulu.fi/isbn9514258371/ (7.11.2002).

[KuSa00] Juha Kuusela, Juha Savolainen. *Requirements Engineering for Product Families*. Proceedings of the 22nd International Conference on Software Engineering, pp. 61 - 69. ACM Press, 2000.

[Leh96] Manny Lehman. *Laws of Software Evolution Revisited.* Proceedings of EWSPT96, pp. 108-124. Springer, 1996.

[Leh98] Manny Lehman. *Software's future: managing evolution.* IEEE Software, pp. 40-44, January-February 1998.

[Leh00] Manny Lehman. *Rules and Tools for Software Evolution Planning and Management*. Position Paper, FEAST 2000 Workshop, Imperial College, July 2000. Available from http://www.doc.ic.ac.uk/~mml/feast2/ papers.html.

[MeHo00] Tom Mens, Theo D'Hondt**.** *Automating Support for Software Evolution in UML*. Published in Automated Software Engineering Journal, February 2000. Available from http://prog.vub.ac.be/cgi-bin/perform.cgi?action= person &id=prs_8.

[Mey98] Bertnant Meyer. *The role of object-oriented metrics*. IEEE Computer, Volume: 31 Issue: 11, Nov. 1998, pp. 123 -127. Also available as an enhanced version from http://archive.eiffel.com/doc/manuals/technology/ bmarticles/computer/metrics/page.html (7.11.2002).

[Omm01] Rob van Ommering. *Configuration Management in Component Based Product Populations*. In Tenth International Workshop on Software Configuration Management (SCM-10), 2001. Available from http://www.ics.uci.edu/~andre/scm10/papers/ommering.pdf.

[Par94] David Lorge Parnas. *Software aging.* Software Engineering, 1994. Proceedings of 16th International Conference (ICSE-16), pp. 279-287. IEEE, 1994.

[SLMH96] Patrick Steyaert, Carine Lucas, Kim Mens, Theo D'Hondt. *Reuse Contracts: Managing the Evolution of Reusable Assets*. Proceedings, of OOPSLA '96, ACM SIGPLAN Notices, pp. 268-285. ACM Press, 1996. Available from http://prog.vub.ac.be/cgi-bin/perform.cgi?action=person&id=prs_3.

[Rös98] Andreas Rösel. *Experiences with the Evolution of an Application Family Architecture*. In Frank van der Linden (ed.): Development and Evolution of Software Architectures for Product Families. Proceedings of the Second international ESPRIT ARES Workshop Las Palmas de Gran Canaria, Spain, February 1998, pp. 39-48. Springer-Verlag, 1998.

[Sav00] Juha Savolainen. *Tools for Design Rationale Documentation in the Development of a Product Family*. First Working IFIP Conference on Software Architecture, 22-24 February 1999, San Antonio, TX, USA, A position paper. Available from http://www.ece.utexas.edu/~perry/prof/wicsa1/final/savolainen.pdf.

[SJW+02] S.R. Schach, B. Jin, D.R. Wright, G.Z. Heller, A.J Offutt. *Maintainability of the Linux kernel*. IEE Proc.-Softw., Volume: 149 Issue: 1, pp. 18 -23, February 2002.

[Sch02] Stephen R. Schach. *Object-Oriented and Classical Software Engineering, Fifth Edition*. McGraw-Hill, 2002 (published in July 2001). [Lecture notes for the book are available as Powerpoint presentation from http://www.vuse.vanderbilt.edu/~srs/transparencies/index.html (7.11.2002).]

[ScTo00] Stephen R. Schach, Amir Tomer. *Development/Maintenance/Reuse: Software Evolution in Product Lines*. From Patrick Donohue (Ed.): Software Product Lines Experience and Research Directions. Proceedings of the First Software Product Lines Conference (SPLC1), pp. 437-450. Kluwer Academic Publishers, 2000.

[SvBo99] Mikael Svahnberg, Jan Bosch. *Evolution in Software Product Lines: Two Cases (not final version)*. Journal of Software Maintenance, Volume: 11 Issue. 6, pp. 391-422, 1999. Available from http://www.cs.rug.nl/~bosch/papers/Evolution_in_PLA.pdf (not final version).

[SWK01] *SWEBOK – Guide to the Software Engineering Body of Knowledge*. From 'A project of the Software Engineering Coordinating Committee'. Trial version 1.00, May 2001. Available from http//:www.swebok.org/documents/Trial_1_00/Trial_Version1_00_SWEBOK_Guide.pdf.

[TCO00] Peter Toft, Derek Coleman, Joni Ohta. *A Cooperative Model for Cross-Divisional Product Developmentfor a Software Product Line*. In Patrick Donohue (Ed.): Software Product Lines Experience and Research Directions. Proceedings of the First Software Product Lines Conference (SPLC1), pp. 111-132. Kluwer Academic Publishers, 2000.

[WeLa99] David M. Weiss, Chi Tau Robert Lai. *Software Product-Line Engineering – A Family-Based Software Development Process*. Addison-Wesley, 1999.