REUSE STRATEGIES IN SOFTWARE ENGINEERING

Hannu Jaakkola, Boštjan Brumen[1], Jyrki Kukkonen[2]
Tampere University of Technology, Pori, Finland
P.O.Box 300, FIN 28101 Pori, Finland

Abstract

REUSE STRATEGIES IN SOFTWARE ENGINEERING

Reuse must be seen within a broad scope in an organization. Independently on the application level, there is always question on the changes in the existing processes, never on the technology itself. We can separate different levels: the *organizational* level, *process* level and *practices* level. The lowest level – *practices* – includes generally accepted and adopted ways of working in an organization. The view of reuse is technical and component oriented. The *process level* specifies the usage of these practices in an organization. In this case the processes are improved to support the reuse approach in different *life cycle phases of the product*. Higher-level abstractions than single components are the objects of reuse. The highest level in the hierarchy – *organizational reuse* – builds organizational culture and at the same time integrates processes and practices into daily routines. To be successful a company is expected to be a *learning organization*. Learning in this context means an ability to accept *best practices* both from inside and outside the organization. A company needs a *reuse infrastructure* to apply and process the use of these practices at an organizational level. Reuse is supported by an infrastructure that provides processes of *early detection* of the reusability (*with reuse*) and reuse opportunities (*for reuse*). This paper includes a discussion on the role of reuse in software development organizations. The discussion covers at first general aspects in the topic and the reuse strategy development. Examples of different levels of reuse are introduced.

Izvleček:

Na ponovno uporabo znotraj organizacije je vedno potrebno gledati z več vidikov. Vprašanje sprememb se vedno nanaša na obstoječe procese, nikoli na tehnologijo samo, neodvisno od aplikacijskega nivoja. Ločimo lahko tri nivoje: organizacijskega, procesnega in rešitvenega. Najnižji nivo – rešitveni – obsega splošno sprejete in uporabljene načine dela znotraj organizacije. Pogled na ponovno uporabo je tehnično in komponentno orientiran. Procesni nivo določa uporabo rešitev v organizaciji. Procesi se izboljšujejo tako, da se omogoči ponovna uporaba v različnih življenjskih ciklih proizvoda. Višji nivo abstrakcije kot posamezna komponenta predstavljajo objekti ponovne uporabe. Najvišji nivo v hierarhiji – organizacijska ponovna uporaba – goji organizacijsko kulturo ponovne uporabe in hkrati uvaja procese in rešitve v dnevno rutino. Od uspešnega podjetja se pričakuje, da je v procesu učenja. Učenje v tem kontekstu pomeni zmožnost sprejeti najboljše rešitve tako znotraj kot izven organizacije. Podjetje potrebuje infrastrukturo ponovne uporabe, da lahko na organizacijskem nivoju uporablja najboljše rešitve. Ponovna uporaba naj bo podprta z infrastrukturo, ki omogoča procese zgodnjega odkrivanja možnosti ponovne uporabe (t.i. »with reuse«) in priložnosti za ponovno uporabo (t.i. »for reuse«). V članku predstavljamo diskusijo o pomenu ponovne uporabe v podjetjih, ki razvijajo programsko opremo. V začetku so prikazani splošni pogledi na ponovno uporabo, nato sledi strategija razvoja ponovne uporabe. Predstavljeni so primeri različnih nivojev ponovne uporabe.

■ ■ ■

1. Introduction

The terms *object oriented* and *reuse* are usually tightly connected to each other. In code reuse this connection is reasonably clear. Object-oriented architecture implements solutions needed for effective reuse: encapsulation, controlled service interfaces, inheritance, dynamic binding and polymorphism, etc. In *code reuse* these provide a way of *standardizing* implementations of different modules, to *force* (inherited modules) to follow predefined implementation details, to *separate dynamic and static parts* of the system from each other, and also a mechanism for specifying higher level *system abstractions* (patterns, design patterns, application frameworks). In a wide interpretation the term "reuse" covers other objects also except code; in principle *all deliverables* of software (project) can be the objects of the reuse. This article concentrates on code reuse. In code reuse different levels, like code components, patterns and application frameworks are distinguished.

[1] *Visiting Researcher in Tampere University of Technology. On leave from University of Maribor, Slovenia*
[2] *Tampere University of Technology and ICL Invia, Finland.*

It is also worth of recognizing, that the two directions of the reuse - *for and with* - are not reached by the same methods. In *for reuse* the component experts adopt disciplinary development methods. It is also question of the processes encountering which deliverables, or parts of them, are useful for reuse, and the right way of organizing the development phase to support the future reuse activities. The *with reuse* approach expects to focus on methods for easy retrieval of reusable components and deliverables to be further adapted to become useful parts of a software product under development. So, it is easy to agree that reuse is not a technique but firstly a *change in processes* and finally in the *organization*. The overall purpose and objective of software reuse is to improve software engineering productivity, quality and time-to-market.

The decision to increase reuse in an organization starts from occasional, not organized reuse (figure 1), which in principle appears in some form or other in all organizations. The evolution toward a *"reusing organization"* is long and needs adoption of *tools and techniques* supporting reuse at first. Techniques first change individual processes, which finally are organized and integrated at an organizational level to become a part of the organization's culture.

This paper discusses on the role of reuse at an organization. In the beginning the role of reuse in an organization is discussed. Based on the discussion some successful experiences are reported. The paper is based on the earlier publication of the author (Jaakkola et al. 2001).

2. The Framework - Organizational Reuse

The framework originates from an assumption that all the situations in software development are not equally suitable and beneficial for systematic reuse, and /or rather that systematic reuse can occur in several different forms in different situations. The following approach suggested in figure 2 is adopted from strategic management (Johnson&Scholes 1997).

According to this framework some factors assessed to be the most relevant from the reuse and reusability point of view are pointed out below.

2.1. Reuse Strategy Analysis

Reuse potential exists when similar, but not identical, software systems are developed. McClure defines the concept of *selective reuse strategy* (McClure 1997), where reuse efforts are and will be focused where the potential reuse benefits are the greatest, i.e. where there is most commonality between systems, sufficient knowledge and a mature and stable application

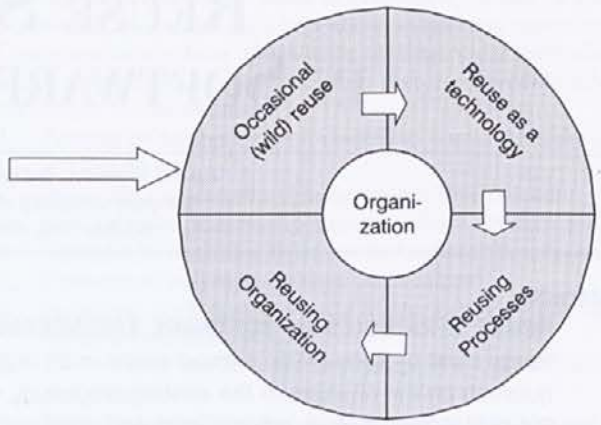

Figure 1. **Reuse Evolution from Random to Organized Reuse**

domain. This research goes deeper into these factors. Based on these requirements, three aspects are considered to be the most relevant from a reuse point of view: the *nature of software business* and certain *external and internal factors* considering software development and reuse, i.e. *reuse potential* and *reuse capability*.

The nature of software business

The nature of software business where reuse is practiced is considered to be one of the most meaningful factors that rules reuse activities. Adapting Jacobson et al's (1997) classification, three different types of software business areas are identified:

1. **Internal development**. Software developments for internal use, typical examples are banks and insurance companies that develop software-based products and services for their customers.
2. **Tailor-made producers**. System integrators that provide tailor-made software for large and diverse customer bases.

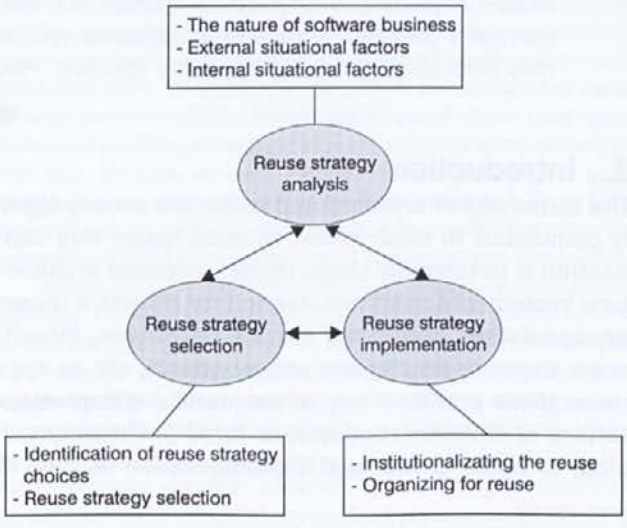

Figure 2. **Reuse Strategy Development**

3. **Software houses.** Either COTS-producers (Commercial Off-The-Shelf) or embedded software producers.

The purpose of a business type analysis is to determine

- How much commonality can be found and even can be expected to be found between different software development projects and
- To what extent a company itself can affect the architectural and technical infrastructure definition and design of software development.

In *internal development* the environment is beneficial to reuse software assets on all abstraction levels from code and components to design and architectures. The company itself, e.g. its information systems management department can determine and standardize common architectures, tools, policies and guidelines for data and its processing (Zmud et al. 1986) for different domains. The situation is more or less similar in *software houses* that can design their software production independently according to decisions of their own.

In *tailor-made software* production environment reuse is less beneficial. In many cases a company's customers want to define development tools, runtime platforms and even the internal software structure to be used in a project according to customer's *own* interests and guidelines, in order to ensure interoperability with their other information systems. In a case like that it's difficult to practice reuse at least on physical asset level. This leads to defining two different basic reuse strategy alternatives:

1. Reuse of *content-independent* software assets
2. Reuse of domain-specific *content-aware* software assets.

External situational factors

External situational factors refer most of all to an *industry and technology life cycle* concept where the maturity of industry or technology is examined in different phases from emergence through accelerating and decelerating growth to maturity and eventually decline (Kotler 1994). When it comes to the software industry, typical for the emergence phase is that tools, methods and standards are not yet formed and established. Thus, it's hard to know what the mainstream tools and technological platforms will be eventually and what the best practices software to do are on these development platforms. Furthermore, the versions and their properties and services can also change radically even within the already selected infrastructure. And eventually it's possible that totally new technologies and tools will replace the former ones.

Software development means *commitment* to some selected technologies, e.g. programming languages, software development environments including class libraries, architectural platforms that provide services for developed software etc., and even their versions. If the expected life cycle of technologies to be used is short and turbulent, the potential for reuse can be small because of risk of investing in rapidly outdated software assets.

Internal situational factors

Internal situational factors refer to evaluation of an *organization's reuse capability*. Then one area of interest is to determine how mature overall software development practices, consisting of both management process factors and development process factors are. The maturity of overall software development (e.g. SW-CMM, SPICE) mirrors expected reuse capability, but the relationship between software engineering maturity models and reuse is not straightforward (Lim 1998) and reuse can be practiced on several levels of improvement models. However, in order to formalize the practice of reuse and make it repeatable between different software development projects, it requires at least a second (repeatable / managed) level defined in the models.

2.2. Reuse Strategy Selection

Reuse maturity models have been developed to illustrate an idea that reuse is spread in an organization as a learning process from less mature to the more mature levels. Most of them inherit the basic structure from CMM and are usually built on five maturity levels from ad hoc to disciplined or optimized reuse practices (Sodhi 1999; McClure 1997; Lim 1998; Karlsson 1996). The models will likely be used in assessing the current practice and level of reuse in organization. Like in CMM successive levels of reuse maturity are based on previous levels. From a software asset point of view, reuse on lower levels occurs in the form of code reuse and higher levels of maturity leverages higher abstraction levels of software assets like designs and architectures.

If used in reuse strategy formulation, these maturity models may lead to too linear reuse thinking and its implementation. Although some preconditions and practices on higher levels of reuse (according to these reuse maturity models) are built on lower levels, the hypothesis of this paper is, that *reuse is not that one-dimensional and does not necessarily start from lower levels of maturity and evolve through higher levels towards optimized reuse.* Instead, it is suggested that appropriate reuse strategy be formed and selected *based on external and internal situational factors* when, in some cases, informal, minimum cost, project level reuse may be

the most viable strategy, while in some cases strategic level reuse should be pursued straight away.

Thus, depending on situational factors, one of the following *intentional* emphases of reuse could be suitable:

- **Informal reuse**. This approach pursues opportunistic reuse benefits e.g. between parallel or similar successive projects. The scope of reuse is on a project level where reuse opportunities are sought beyond the boundaries of individual projects by being aware of what has been previously built and what similar projects are going on simultaneously. No specific reuse investments are needed; reusable assets are documented, stored and brokered via normal version- and configuration management infrastructure used in software development.
- **Operative reuse.** An operative approach pursues cost reduction by increasing productivity and quality improvements via common and tested software assets. Since the scope of reuse is broader than at a project level, but more at an organization unit or software engineering process level, specific reuse support mechanisms beyond normal software development infrastructure are needed. These include common software asset storage, component catalogues and documenting that take into consideration that the component user and producer may not be in close interaction between each other.
- **Strategic reuse.** On a strategic level, reuse has deeper effects on the way an organization operates and pursues not only getting *better*. but getting *different* (Hamel 2000). This results in not only cutting the costs of current software production, but gaining strategic advantages to produce software better, different or more effectively than competitors (Johnson&Scholes 1997), e.g. improving the time-to-market when producing new software-based services, or developing totally new software products starting with designs from a farm of well-known and reusable assets. The reuse scope is broader than in previous cases and reuse decisions are made on at a corporate or business unit level as a part of business and production strategies.

In *informal reuse* the emphasis is on utilizing previously developed parts of software in new projects. Then all kind of suitable assets can be used either as-is but more probably *adapting* them to a part of that new software product. Informal approach can be suitable e.g. when the external environment (tools, technologies architectural platforms etc.) is still at a turbulent and unstabilized phase of life cycle and the risk for an obsolete asset base is substantial.

On an *operative level* of reuse, emphasis is not mostly on asset utilization but more on asset *production* and effective brokering. A suitable form of practicing reuse depends on the nature of the business. In the case of internal development, reuse can be context-aware vertical and domain-specific reuse, covering all life cycle products from architecture and documentation to code fragments. For system integrators context-independent reuse covering higher abstraction level life-cycle products like duplicable concepts, designs and best practices, i.e. assets above customer-specific dependencies, can be more applicable. This approach could also be suitable in a turbulent technology environment, provided that enough similar projects are running in a time window, because higher level life-cycle products absorb less work than code components (Jacobsson et al. 1999) and therefore it reduces the risk of obsolete asset libraries.

On a *strategic level* of reuse, emphasis is not only on separate software assets but also more on concepts or whole approaches to software production. Meyer and Seliger (1998) define a software *product platform* as a set of subsystems and interfaces that form a common structure from which a stream of derivative products can be efficiently developed and produced. Then the software product platform is both architecture and an implementation of architecture that propel a family of software products or internal corporate applications. Sääksjärvi (1998) in turn, defines the term *product skeleton* to mean a core product common to certain product families and that can be varied in order to produce several separate but similar software products.

Depending on the nature of business one of these approaches can be applicable. By definition, the product platform could be suitable for *internal developers*. Subsystems and interfaces unavoidably handle and process data that makes them content-aware. Then strategic reuse in internal development could mean well-defined vertical product platforms consisting of a set of core components, both physical and abstract, on which new software products for that domain are intentionally designed and developed. For *software houses* a similar approach can be suitable too. *Product line* reuse is defined as a form of vertical reuse where reuse capitalizes on commonalties between product lines (Lim 1998). Thus, software houses can develop common product platforms, or develop product skeletons, from which new products are developed tailoring these core product platforms into new products.

For *systems integrators* these kinds of domain specific approaches are not as well suited. In the case of integrator developed software for a defined customer segment, e.g. for banks and insurance companies, where certain core functionalities are conceptually

more or less equal (maybe even according to legislation), there are usually too many customer-specific features, both in data and processing, that are too hard to isolate in order to make reuse meaningful in the same way as in domain specific reuse. Rather, reuse potential could be found from context-independent parts or levels of software systems. Thus, strategic reuse can involve product skeletons, i.e. core products for certain content and context independent purposes that can be copied and adapted for several customer cases. One example of this kind of reuse is ICL Invia's mCastor channel adapter that formats and adapts content-independently the output of information systems for different types of terminals from mobile handsets to digi-tv and traditional web-terminals (ICL 2000).

2.3. Reuse Strategy Implementation

Reuse strategy implementation covers *institutionalising* of reuse entirely. It involves both "static" issues like organizing and resourcing reuse and "also "dynamic" dimension, i.e. transforming software development from the current situation to reuse-exploiting practices according to the selected strategy. The amount of effort needed is dictated by the strategy selected.

If *informal reuse* is pursued, very heavy investments are not required, but software projects that build applications both produce and consume reusable assets too, either in an intraproject manner or between different projects. Because the assets usually become part of the software that projects produce, no separate reuse specific asset storage is needed. Furthermore, reused parts are maintained as a part of the project software when no special *reuse asset maintenance group* is needed.

If strategic or operative reuse was selected, implementation probably requires, in the long run at least, new *reuse-specific organizational units* and *support roles* in the organization. Especially if the objective is to reuse assets as-is, i.e. *black-box or grey-box reuse*, the separation of asset development and asset usage and adaptation to software projects can be necessary in order to maintain and further develop assets beyond the projects' limited life cycles.

Many authors on the subject suggest that implementation of major change initiatives should be done using a number of smaller projects (Kotter 1996; Hamel 2000) instead one big change. Suitable way to proceed toward a selected reuse strategy could be via selected *pilot projects* the purpose of which is to demonstrate and test that the ideas of reuse really works (Jacobsson et al. 1997). That holds true independent of the selected reuse strategy.

3. About Reuse Infrastructure

Improved reuse is based on a well-organized *reuse culture* in a company. The reusable assets may vary case by case (figure 3).

In addition it is also question on the approach to the design

- design philosophy and
- the management and organization of the reuse.

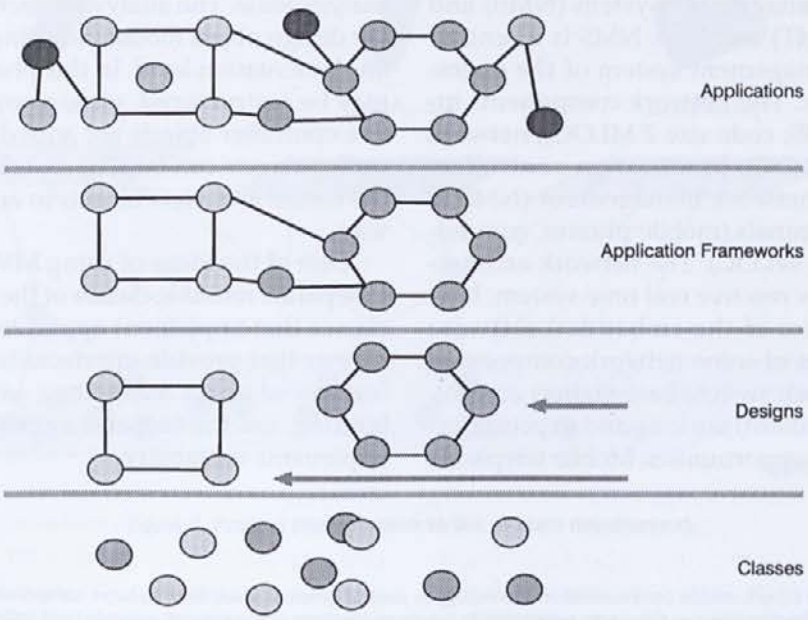

Figure 3. **Different abstraction levels of reusable assets.**

The following subchapters discuss a systematic approach to the reuse developed by an organization adopted reuse as a *strategic level solution* in their products.

The practical implementation of this culture can be called the *reuse infrastructure*, which includes well-defined processes and technical support. In this chapter the reuse infrastructure of Nokia[3] will be introduced briefly. The presentation is based on published material (Jaaksi et al. 1999; Kuusela 2000) and concentrates on solutions affecting software structure and implementation principles. The discussion on technical solutions is excluded and can be studied in more detail in the references.

3.1. Organizational and Product Aspects

Nokia has been a fast growing organization, having a large amount of *new employees* all the time. The products are usually large (embedded) software products developed *incrementally* and based on *evolutionary* development cycles in the long run. In this kind of organization, clear *guidelines* and generally accepted architectural solutions are needed. From the software point of view, the architecture includes both static and dynamic structural elements. Static parts are modeling static reality; dynamic parts have interface-oriented features, usually representing communication with the end-users. A common architectural pattern improves *understandability*, helping others to understand the architecture and functionality of the application. The guidelines also make it easy to *maintain* components in the long run, because all the components resemble each other in their internal structure.

The applications on the background of the discussion are Network Management System (NMS) and Mobile Telephone (MT) software. NMS is a centralized control and management system of the operator's cellular network. The network components include base stations (BS; code size 2 MLOC), network switch (MSC; 10 MLOC), base station controllers (BSC; <10 MLOC), network management (NMS; 3 MLOC) and user terminals (mobile phones, communicators, etc.; 0,5 –1,5 MLOC). The network architecture itself is a complex reactive real time system, having the characteristics of the embedded software mainly. The life cycles of some network components (base stations, network switch, base station controllers, network management) are long and expect effective system evolution opportunities. Mobile telephone *generations* instead are introduced at an *accelerating speed* creating a primary competitive advantage to telephone producers. In both cases (network components, telephones), there are also several variations of the same product made for different markets (e.g. currently 32 different phones are manufactured covering six protocol standards). As a conclusion, there are two different approaches to software architecture to be discussed. In the case of long life cycle products, the architecture solution must support an effective maintenance and evolutionary (and incremental) development culture. If it is a question of fast entrance to the market (new features or advanced properties of the equipment), the ability to benefit from existing software components is important.

3.2. MVC++ as a Software Architecture Style

MVC++ (Jaaksi et al. 1999, 55-60) is an application architecture developed based on some earlier comparable models (MVC, PAC, ...). According to MVC++, three types of objects are separated: model, view and controller. The *model layer (M)* corresponds to a real world and "static" problem domain. The *view layer (V)* is the outer software layer visible to the end user. Typically there is one view class for each dialog box and window of the user interface. The *controller layer (C)* controls the interaction between the model and the view. The model layer objects usually appear in the analysis class diagram and the view components are derived from the user interface specification. Controller classes are needed to connect the "dynamic" view part of the system to the "static" model part. According to the software life cycle model *OMT++* the analysis class diagram (model layer) is produced in the *analysis phase*. The analysis object model is the basis of the design object model including classes closer to the implementation level. In this phase the class diagram may be restructured, view components and respective controller objects are added to the model. Controller objects can be seen as adapters that integrate the model and view objects in an application specific way.

One of the ideas of using MVC++ architecture is to separate reusable classes of the application from the classes that implement application-specific functionality or that provide interfaces to the real world. The features of object technology – inheritance, dynamic binding, association and aggregation – are used to implement reusability.

[3] *Nokia's product spectrum covers mobile communication technology in several different levels from network components to network management and end user terminals. The discussion in this paper references to the reuse solutions of the network management software and mobile telephones. Partially the solutions are developed in "line organizations " (Nokia Telecommunications, Nokia Mobile Phones), partially as a separate activity (Software Architecture Group) of Nokia Research Center.*

3.3. Platform – Product Line - Product

In the case of large systems, effective production of product variants and new features in existing products is difficult. The fundamental problems caused by the large size of the software can be avoided by using the system structure enabling effective project work and helping in development and delivery of new functionality in a short span of time.

In a solution by Jaaksi et al. (1999, 198-210), the *system* is considered to be a family of (reasonable independent) sub products. The sub products can have their own release schedules; dependencies on other similar subsystems must be managed, of course. The deliverable grouping closely related features is called an *application product* and a tested configuration of these application products, a *system product*. An example could be a system product "office automation" including "application products" like text processing, spreadsheet, and presentation graphics. Different configurations of the application products for different market segments all constitute system products also.

Rational usage of the application product idea is the best solution for reaching the benefits of large monolithic system development (e.g. high reuse of common design solutions) and releasable highly independent application products. The common parts of the applications are organized into one independent high-level subsystem of its own. This subsystem is called an *application platform*. Application products depend on it (but not vice versa). Whereas application products provide applications to users, the application platform provides reusable components, frameworks and design guidelines to software designers.

In addition, to manage reusable assets, the *application platform* also enables a *product line approach*[4] to help system development in the future. The leading principle is to release a line of closely related products and product variants cost effectively over time (short time to market). The products are built on a common application platform that holds common software assets. The motivation to collect reusable software assets in an application platform is to make future variation easy and economical by using the results of projects in the past when creating new products. The difference to the application framework approach is that the new products are also created using the assets of the application platform. The approach is illustrated in figure 4.

The product line approach has been adopted by large-scale projects that evolve for several years. Development of the application platform needs investments which will be paid back only by effective use (adopted in processes) of it. In practice however, platforms grow with systems: the specific solution implemented at first will generate the solution to platform services. For this a mechanism detecting reusable assets must be built into the processes.

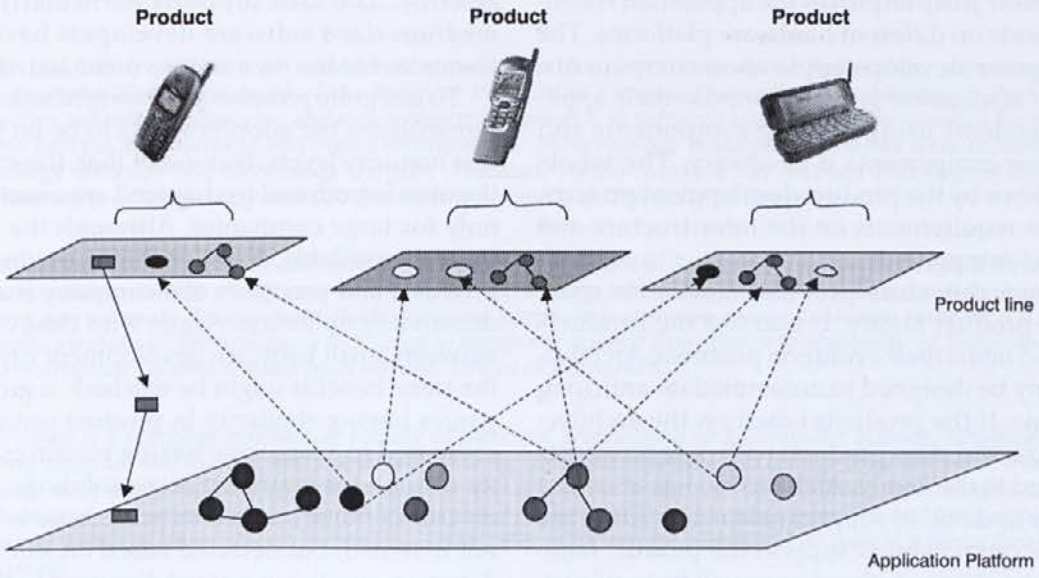

Figure 4. **Product line approach to the product development**

[4] *SEI defines product line as a group of products sharing a common, managed set of features that satisfy specific needs of a selected market or mission. The term product family is used for a group of systems built from a common set of assets. Although a product family may be developed without product line coordination and product line may be developed independently, most product lines are also product families*

3.4 Case NMP

Reuse experience of Nokia Mobile Phones (NMP) has been reported by Kuusela (2000). NMP adopted a *product line approach* originally developed for large-scale system development in Nokia Telecommunications. The application for NMP was developed by the Software Architecture Group at Nokia Research Centre, which is (was) analyzing, assessing and modeling the product architectures of business units to be able to give suggestions on improvements. NMP currently has 32 different phones manufactured for six different protocol standards. The variations of the phones are also produced for different customer segments and cultures. Functionality of the phones also depends on the fashion (e.g. user interface design) and advances in technology.

The software architecture provides the basis for reuse within a product family. Originally the software product family was addressed by only the basic requirements variation in hardware, the communication standards and the user interface. The need to have several alternatives for same functionality has driven the architecture to implement the "client-server" idea: it separates service identity from the identity of its provider and makes service usage and provision location independent. Combined with dynamic configuration management the system supports several providers for the same service and the providers can be plugged in or taken out without restarting the system.

With the architecture even the *structure of the development organization* had to be changed. The *infrastructure development group* improves the application framework and ports on different hardware platforms. The *component group* develops application components. The *product development projects* compose their application subsystems using existing components and develop new components if necessary. The whole process is *driven* by the product development projects, which place requirements on the infrastructure and request new components.

Software architecture provides a basis for reuse within the product family. It also ties the products together and limits their evolution potential. Architecture can only be designed to accommodate anticipated variations. If the products based on the architecture are successful, new products with new properties will be added to the family. Architecture has also to be periodically updated to support new needs. Once the architecture can no longer support the product family, it has to be changed; the change will be very costly and will cause a need to redesign large parts of the system.

Wide scale reuse is expected to be economical. This is not always true. When independent products continue their evolution, new requirements are faced. These requirements are tackled in the product development project. Later some of the new features may prove to have wider scope and they can be tackled on a family level; however the changes on the family level are very costly. A product family approach also limits modifiability. In practice reuse and modifiability must be balanced, and variation management and reuse must be closely connected. If variation management runs into trouble, reuse must be decreased. The grey area between perfectly organized product lines and complete independent development projects is wide. (Kuusela 2000).

4. Summary

The paper concentrates on organizational reuse solutions in software development organizations. The main message of the paper is that reuse must be a planned and well-organized part of an organization's processes. There is no single best practice to organize reuse. Reuse *strategy development* depends on the nature of the industry as well as on several internal and external factors.

"Process reuse" can be seen as an organized activity helping companies to exchange experiences in process improvement activities. In practice it is question of organized and managed co-operation between a groups of companies. This can be supported by a *"process practice platform"*, analogically to product lines and product platforms. This platform could be maintained and organized by a support organization – *center of expertise*. The idea supports particularly small and medium-sized software developers having limited resources for the own improvment activities.

To integrate *processes and improved software architecture solutions* the adopter needs to be on higher process maturity levels. Because of that, the solutions like the ones introduced in chapter 3 are usually potential only for large companies. Although the models are publicly available, to tailor the basic ideas to fit the products and processes of a company is a resources-demanding operation. Again, with close co-operation between small software development organizations, the same benefits might be reached: a group of companies having similarity in product concepts can be compared to a large one having resources and capacity enough for improved reuse solutions.

One of the sayings concerning reuse is "Models are not invented, but detected based on similarity". This detection question is not yet discussed in this paper. A proposed solution is the establishment of a *reuse team*. This is an (informal) organization of co-operation above product development teams organizing interaction between them. The product development teams have responsibility for introducing their product plan at an

early phase of the project. The aim is both to detect the components suitable for later reuse and propose the usage of already available (reusable) components in the product under development.

References

Hamel Gary (2000).
Leading the Revolution. Harward Business School Press 2000.

ICL Invia Oyj (2000).
ICL Invia offering a solution for multi-channel mobile. Press release 6.9.2000.
URL: http://www.iclinvia.com/icl_pages/pressreleases_frames.htm. Downloaded 2.4.2001.

Jaakkola H., Kukkonen J., Varkoi T.,
Best Practices as Reuse Infrastructure. In Koloumdjian J., Mayr H., Erkollar A. (editors), Proceedings of the ReTIS'2001 - Data and Document Re-engineering for the Web. Osterreichische Computer Gesellschaft, Vienna, 2001. pp. 9-31.

Jaaksi A., Aalto J-M., Aalto A., Vättö K. (1999),
Tried & True Object Development. Industry Proven Approach with UML. Cambridge University Press.

Jacobson Ivar, Booch Grady, Rumbaugh James (1999).
The Unified Software Development Process. Addison Wesley Longman 1999.

Jacobson Ivar, Griss Martin, Jonsson Patrik (1997).
Software Reuse. Architecture, Process and Organization for Business Success. Addison Wesley Longman 1997.

Johnson Gerry, Scholes Kevan (1997).
Exploring Corporate Strategy. 4th edition. Prentice Hall 1997.

Karlsson Even-Andre' (edited by) (1996).
Software Reuse. A Holistic Approach. John Wiley & Sons 1996.

Kotler Philip (1994).
Marketing Management. Analysis, Planning, Implementation and Control. Prentice Hall 1994.

Kotter John P. (1996)
Leading Change. Harvard Business School Press 1996.

Kuusela Juha,
Architectural Evolution. Nokia Mobile Phone Case. Nokia Research Center, 2000.

Lim Wayne C.(1998)
Managing Software Reuse. A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components. Prentice Hall PTR 1998.

McClure Carma (1997).
Software Reuse Techniques: Adding Reuse to the System Development Process. Prentice Hall PTR 1997.

Meyer Marc H, Seliger Robert (1998).
Product Platforms in Software Development. Sloan Management Review, Fall 1998, Volume 40, Nr. 1.

Sodhi Jag, Sodhi Prince (1999).
Software Reuse. Domain Analysis and Design Process. McGraw Hill 1999.

Sääksjärvi Markku (1998),
Tuoterunko. Uusi ajattelu ohjelmistotuotteiden strategisessa kehittämisessä. Teknologiakatsaus 62/98. Teknologian kehittämiskeskus Tekes, Helsinki 1998. In Finnish.

Zmud Robert W, Boynton Andrew W, Jacobs Gerry C. (1986),
The Information Economy: A New Perspective for Effective Information Systems Management. Data Base.

◆

Dr. Hannu Jaakkola is professor of software engineering in Tampere University of Technology, director of Center of Software Expertise (CoSE) and head of the Regional Institute of Tampere University of Technology in Pori. His research interests cover software engineering and technology management. In software engineering the research focus is especially in software process improvement and object technologies. In technology management he has wide research in the area of technology diffusion and technology transfer. Professor Hannu Jaakkola has received PhD degree (engineering) in Tampere University of Technology and BSc (business economics) in University of Tampere.

◆

Mr. Jyrki Kukkonen is a manager of software project business in Financial Services department of Fujitsu Invia Group. He is also a postgraduate student at Tampere University of Technology where his primary research interests include software engineering and software development methodologies, particularly software reuse. Kukkonen received a B.S (mech.eng) from Helsinki Institute of Technology, a M.S (eng) in computer science from Tampere University of Technology and a M.S (econ) in information systems science from Helsinki School of Economics.

◆

Boštjan Brumen is a teaching assistant at Faculty of Electrical Engineering and Computer Science, University of Maribor. His teaching areas are Databases I and II and Data Security, on both university and college level. Research interests include data mining, data analyses, data security and data reusability. As a member of Database Technologies Laboratory he actively participates in several international and national projects, related to data issues. He has been cooperating with researchers at Tampere University of Technology since 1999, with results, published at several international conferences and in journals.

◆