SPL Migration Tensions: An Industry Experience

Antony Tang[1], Wim Couwenberg[2], Erik Scheppink[2],
Niels Aan de Brugh[2], Sybren Deelstra[2] and Hans van Vliet[1]
[1]VU University Amsterdam, [2]Océ Technologies
[atang, hans]@cs.vu.nl, [wim.couwenberg, erik.scheppink, niels.aandebrugh, sybren.deelstra]@oce.com

Abstract

In a software development environment where legacy software systems have been successfully deployed, there are tensions that deter the organization from moving towards software product line engineering (SPLE). An example is the effort required to develop a product line architecture versus time-to-market pressure or the lack of evidence to justify the benefits of SPLE. In this experience report we discuss the tensions that exist in Océ Technologies. We report that a reactive software reuse approach has not yielded the desired long-term benefits of reusability. A proactive approach requires knowledge exchange and coordination between software management and technical staff. We describe how such knowledge sharing can ease the tensions and facilitate a SPLE migration process.

Categories and Subject Descriptors D.2.11 [**Software Architectures**] Domain-specific architectures. K.6.3 [**Software Management**] Software selection.

General Terms Management, Design, Economics.

Keywords *Industry Case Study, Software Product Line Engineering, Architecture Management, Agile Development Process.*

1. Introduction

Software reuse has, for many years, been an area of interest in software engineering research and in the software industry, and many companies have successfully used techniques to implement software product line engineering (SPLE) to achieve large-scale software reuse. Although it is intuitive to recognize that SPL has many benefits, it is often difficult to estimate and justify its costs and benefits. There are a number of reasons for this. Firstly, the application of product line techniques requires staff to acquire knowledge in the new practice. Secondly, management may not be convinced or be aware of the extent of the benefits that SPL engineering could bring. Thirdly, it is difficult to estimate and justify the costs and benefits of applying SPL to a legacy system where some core assets already exist but their reusability effectiveness cannot be gauged.

In an industrial setting with serious time to market pressure it is often easier to scavenge the existing code base to develop a new product. This situation is the more likely to happen in a technology-dominated environment where the engineers take pride in developing new and challenging features, rather than following the masterplan as dictated by a product line architecture. This situation is exacerbated when Agile development practices are used and meeting short-term delivery goals dominates the thinking of the engineers.

Under these circumstances, a tension is created between the proper, top-down, product line engineering approach that emphasizes reuse across products, and the bottom-up development of new products under market pressure [1, 2]. Naturally, a question is how to combine these two seemingly opposing forces to improve the effectiveness of producing software in an environment where software assets and practices already exist.

We conjecture that both development processes may co-exist and co-evolve. There may be situations where quick development of new products is what matters. There are also situations when systematic re-engineering of products towards a proper product line is beneficial. Gathering data on different performance indicators of both processes, and instituting a knowledge feedback loop in the organization can provide support to make decisions that balance between building SPL assets and building software that satisfies current requirements.

In this paper, we describe an SPLE experience in Océ Technologies. Océ Technologies produces printers that incorporate sophisticated software and the print software systems have been successfully deployed for many years. With the many printer products that the company is selling the reusability of software is an important issue. In a recent case of design for software reusability, we study the tensions in migrating to SPLE. In this paper, we first report the experience of a SPL implementation (Section 2 and 3). We analyze the underlying tensions in the development process, the architecture planning process and the software development culture that deter SPLE migration (Section 4). The theme of this paper is how various types of knowledge can be used to resolve the tensions and overcome the inertia to enable an incrementally move towards SPLE (Section 5). We propose a knowledge sharing approach that allows both software management and technical staff to exchange knowledge to improve decision making in SPL implementation.

2. Background of the Industry Case

Océ Technologies produces high-end printers to serve the business markets for high-volume printing, wide-format printing and office printing. Printer software is one of the main components in a printer. The software renders images and controls the print engine. The company is in a highly competitive market place where providing new features to the market in a timely fashion is important to product success. For this reason, software development in

Océ Technologies has adopted an Agile development process with governance by the architectural and engineering councils.

In its software development organization, developers are typically grouped into teams of between three to eight people. A major product development would require a considerable number of software development teams, plus architects, integrators and testers. The software development cycle is typically short, an increment takes 8 weeks, with sprints of 2 weeks. Developers are highly motivated and knowledgeable. Documentation is minimal. The development culture can be described as innovative where the best people are assigned to the problems, and as such job rotation is common.

Océ Technologies has long recognized that software reuse is important in terms of leveraging existing software assets across different products. Architects have formed a council to examine how to leverage reuse of existing software components across products.

Over the last few years, Océ Technologies has refactored modules and interfaces several times to try to achieve software reusability amongst products. Even though the end-product was developed and deployed successfully, software developers had to redesign and reprogram key features. Software reuse was achieved by scavenging design and code from a previous project, and adopting it to the new requirements by affixing new design and code. After a couple of such development iterations, the architectural design gets eroded and the software becomes sensitive to new design changes. In 2010, a team of designers and software developers used SPLE principles to redesign a major architectural component within the printer system.

The software component under consideration is a part of the printer controller. A printer controller manages all data that enters and leaves the print system. This can be print jobs received via a network, scanned documents that are exported to a desktop PC, settings that are entered via a web client, and so on. The controller offers different ways to handle jobs. Jobs can be printed immediately upon reception, or stored to allow them to be edited on the print system and then printed repeatedly at a later time. Keeping up with new demands and diversity is one of the major challenges in controller software development.

Océ's strategy is to support this increasing variety of printers and configurations with a single controller software code base. This paper reports the experience we had with the redesign of the architectural component we call the "Printer Controller Component" (PCC). PCC manages print jobs that are sent to the print system. This is an interesting component from an SPL point of view since, although many components technically are the same for each controller, the PCC has specific behavior per product. Hence it is a component where variation points can be expected to play an essential role.

While the PCC has specific behavior per product it is still expected to exhibit consistent behavior towards the engine and operator. Two main aspects are how it handles run-time contradictions (RTC) and how it deals with error situations. An RTC occurs when the print engine temporarily stops printing because it is out of resources. Typical examples are: input tray empty, output tray full, toner empty, out of staples etc. The printer controller is responsible to detect and report the RTC as early as possible and respond to situations according to an engine's capabilities. Errors can occur at any time during printing and can range from paper jams to engine errors. The printer controller must report the errors to the operator, stop printing until all errors are resolved and the print system has to recover from the error situation before resuming. Error recovery can consist of many steps and may require operator inputs to deal with the error.

Certain differences between products are so fundamental that the PCC must exhibit different behavior. For instance, the way print engines handle inputs can be by a print job or a print page. So variation points not only exist for specific communication but also for perceived behavior of the overall print system.

In the past four years four redesigns of PCC took place. PCC-1 was redeveloped from scratch entirely. PCC-2 reused part of the PCC-1 code base and was developed in parallel with PCC-1. PCC-3 reused the PCC-1 code base with additional features and was developed after PCC-1 had been completed. PCC-4 was developed based on SPLE principles and its intention was to accommodate all the commonalities of previous PCCs.

PCC-1 aimed to better define and separate the responsibilities of PCC from the rest of the controller software so that a dedicated team could work on it, without impacting other subsystems too much. The introduction of a new printer family instigated the PCC-2 and PCC-3 redesigns. It was decided to keep the main PCC structure intact. There were two reasons for this decision. Firstly, architects thought that a different architecture redesign would not meet the product release time frame. Secondly, there was an inertia to move to SPLE.

Shortly after work had started on PCC-3, two crucial events happened. Firstly the business priorities and market focus changed. Secondly, the approach to redesign the software for reuse only added to the complexity of the component and that in turn had severe implications on its maintainability. So it was decided to start a fourth, major, redesign of PCC (i.e. PCC-4). The business driver in the PCC-4 redevelopment was to support a product line to lower the recurring costs of maintenance and to support new product features. By this time, the inertia and the mindset towards SPLE had changed. This latest redesign employs SPLE principles. It has centered on four major concepts.

1. A new interface was introduced between PCC and its external components so that there is uniformity in their communication. The responsibilities of PCC were reconsidered and the parts that were functionally specific were isolated and relocated to a separate component using this new interface.

2. Explicit variation points were created in all areas within PCC where different products need them. A strict separation was made between the core framework that must remain identical for all products and plugins that should support specific product behavior.

3. The existing module test framework was replaced by a fine grained unit test environment. Module testing checks the response of the entire module (i.e. component) whereas unit testing tests a targeted area within a component, reducing test overhead and improving test efficiency.

4. The scripting of runtime contradictions was enabled using a scripting language. Experience showed that adding an RTC to the existing PCC costs too much for such a common variation point. Therefore we designed explicit RTC interfaces and made those available in a scripting environment, reasoning that a simple concept should only require simple code.

This SPLE initiative to redesign PCC was taken by an engineering team in a self-directing manner, endorsed by management. However, the resulting architecture design may also erode as time progresses, especially when architectural design governance with a mid-term or long term focus is absent. In order for SPLE to gain traction, it is necessary to understand the fundamental issues such as cost-benefits and other tensions that contribute to the inertia to change. This is where knowledge sharing about costs and benefits of such an approach come into play.

3. Costs and Benefits of a Domain Specific SPL

The inertia to adopting SPLE can be attributed to many reasons, one of which is the lack of understanding of the benefits that SPLE can provide. So it is necessary to analyze (a) the potential benefits of using SPLE; (b) the way we could measure, predict and justify which parts of the software developed under SPLE would yield a positive ROI; (c) implications on extending and sustaining SPL architecture governance.

There are different methods to compute the cost-benefits for SPL, all of these methods consider SPL in terms of its product life-cycle [3-5]. The general idea amongst these metrics is similar. All of them consider a common software platform that can be reused many times in different products. Then there is a part which is unique to individual products and its redevelopment is unavoidable. A discussion of these related works is in Section 6. For now, we assume to use the Structured Intuitive Model for Product Line Economics (SIMPLE) [6]. We chose to use this model because we are able to gather the data that relate to this model.

The SIMPLE model splits software product line development costs into four basic cost functions: (a) Organizational costs are the costs of reorganizing teams and adapting the development process to SPLE. (b) Core-Asset Base (CAB) is the cost of developing the reusable core asset or framework; (c) Unique cost is the cost for developing each unique feature in a product that is on top of the CAB; (d) cost of reuse is the cost to check if and which CAB works, the tailoring of CAB for reuse, additional testing costs and the learning curve etc.

In our case, we see that a reactive investment approach has been taken where assets are built incrementally without key strategies to building an SPL architecture [7]. There was no organization dedicated to SPL design and development, therefore the organization cost cannot be measured. However, we can measure CAB costs, unique costs and reuse costs. These are discussed in Section 3.1. In addition to the costs of building the software, there are other costs and benefits that are important to an SPL implementation: ease of maintaining software, time-to-market and the software quality. McGregor describes some of these factors in [8]. We report on these aspects in Sections 3.2-4 to further support our analysis.

3.1 Software Refactoring Cost

The redesign that resulted in PCC-1 was based on three new concepts. Firstly, a formal interface (called PCCI) was introduced between PCC and the job management function in the controller. Secondly, this formal interface was implemented in a CM library to simplify communication with the print engine. And thirdly, PCC functions are segmented, so that each function is responsible for a single behavioral aspect. These design concepts have proved to be valuable and have persisted over the years. In particular the PCCI interface and CM library remain almost unchanged till PCC-4.

This design worked fine for a single product but there are major limitations when the software was to be reused in other products because many new product features did not fit well with the existing design. These are new and unforeseen requirements that required flexibility in the PCC-1 architecture to support. But PCC-1 lacked that flexibility. Retro-fitting these new requirements led to the interlocking of responsibilities between functions, which in turn led to high complexity and maintenance cost. The original architecture design eroded over time and software reusability had not been fully achieved.

There were other architectural design features that hindered software reuse. Abstractions were introduced where none were required which led to higher complexity and costs. Some critical assumptions about engine behavior were made at the core of the PCC-1 design that reduced the reusability of that component.

The redesign of PCC-1 and PCC-2 took place in the context of their own respective projects. In both cases, the controller development was seen as part of that project. Team leaders reported primarily to their project and the architectural board. The architectural board was informed about both projects but did not pay attention to aligning software developed in the two projects to achieve reusability. The two projects had separate designers who made their own design decisions without mutual alignment. Reconciling these design differences to come up with a general PCC was impossible. When it came to point where a common platform to support future products is needed, none of the existing software can fulfill that. A totally new redesign was required.

	Core-asset Base	Unique Cost	Reuse Cost	Total Cost
PCC-1	41.5	26.5	32	100
PCC-4	23	10.5	2.5	36

Table 1. SIMPLE relative Costs of PCC developments

In PCC-4 development to cater for all previous PCC functions, the SPLE principles were used, and the separation of responsibilities of common and variable functions is clear. Table 1 shows the relative cost of the implementation of PCC-4, compared to PCC-1. The cost unit in Table 1 is expressed relative to PCC-1 total development cost, i.e. 100 point as the basis. PCC-4 has a much lower development cost. No doubt, some experiences had been gained from the previous three rounds of design that partly explains the reduction in PCC-4 development costs. It is also true that careful architecture planning and redesign has created a more elegant software design, which lowers the cost of programming and testing, and has lowered the total cost of PCC-4 development.

The development history of PCC shows that long-term architecture planning can be a worthwhile investment, especially the planning towards SPL when the software can be used in multiple products. Such investments in SPL architecture design can only be coordinated at the management level, and guided by an architectural board with architectural governance.

3.2 Software Maintenance Cost

At least two factors make software maintenance cost for the latest PCC design lower than that of its predecessors. The internal design is refined further into functionally independent modules. Moreover, this internal structure is strictly enforced by placing interfaces between the separate modules. Because of this, bugs can mostly be localized and fixed within a single module.

Some runtime contradictions are implemented in scripts and are totally separated from any other code in PCC. This has the advantage that the code is easy to understand and bugs can be fixed without the need for any recompile and install cycles.

3.3 Time-to-Market

In implementing one typical feature for a printer product using PCC-4, it was found that the development time took is 50% as compared to the development time of the same feature using PCC-1. Some of the features are developed in lesser time. Because these features can be added and tested as plugins without touching the PCC framework or other plugins, a shorter time-to-market to deliver new product features can be achieved.

3.4 Software Quality

The finer subdivision of PCC in modules and the introduction of internal interfaces made PCC much easier to test. The generic framework and product specific plugins can be tested separately. Since tests are easier to develop, more tests are developed to achieve better test coverage. In contrast, the test sets for PCC-1 are intertwined and the test cases are harder to develop, making it difficult to understand the test cases and cover all scenarios.

4. Tensions in Migrating to SPL

The decisions on the software development processes are made primarily by software (SW) management, explicitly or implicitly, and these decisions have profound influence on the efficiency and the quality of software products over time. As shown in this industry case,software reuse as a stand alone directive is inadequate. Architects and engineers who were under pressure to deliver products in a sprint would choose to satisfy short-term project needs instead of the long-term product needs as our example has shown. This is a tension in which SW management must play a role in resolving. In order to understand the forces that influence PL implementation in a case such as this, we show the causal relationships between software management, software architecture and its outcomes (Figure 1).

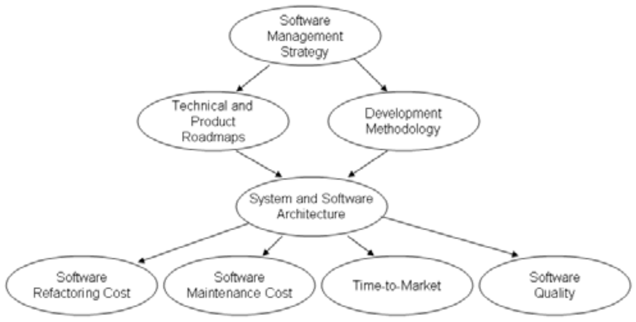
Software Management Strategy
Technical and Product Roadmaps
Development Methodology
System and Software Architecture
Software Refactoring Cost
Software Maintenance Cost
Time-to-Market
Software Quality

Figure 1. Causal Relationships between Software Management and Product Results

A *software management strategy* influences the *development methodology* such as the use of Agile development and the process of architectural planning. Agile development process focuses on immediate implementation and that affects, mostly implicitly, architecture planning and design that is about long-term planning of software implementation.

Software management and architectural council influence *roadmaps planning* where software reuse strategy and software flexibility to adopt new features are a part of. If the SW management strategies for SPLE are present, and architectural principles for reuse are maintained, the *system and software architecture* realization would be the result of the planned activities rather than happenstance. This relationship hints at the maturity of software reuse in an organization, whether it is taking an opportunistic approach or a systematic approach [4].

A well designed architecture is important to SPL because the way a software system is structured influences how software variability can be catered for and how software commonalities can be reused. A good SPL architecture design can contribute positively to reducing refactoring cost, reducing maintenance cost, and improving the time-to-market and software quality [9]. The reverse can also be true where poorly designed architecture does not provide these benefits. These potential benefits need to be balanced with the costs of realizing the SPL architecture. However, such knowledge to help justify and make the decisions is often unavailable, especially when the industry domain is specific and unique.

As Océ manufactures more printer products for different markets, sharing the code-base for common product features becomes more important. The redevelopment of PCC in a SPLE fashion is a natural progression. However, a number of tensions exist in the current culture and development process that inhibit the SPL initiative. In this section, we examine the different tensions using the causal relationships model in Figure 1.

4.1 Sprinting to Software Implementation

Océ Technologies uses the Agile development process in which different teams of developers would make increments every eight weeks (with sprints of 2 weeks) to deliver software products. The way Agile development is practiced, designers and developers take pride in their success on delivering the required product features at the end of each sprint. This practice focuses on reaching the targets of each sprint, the reuse strategy is adversely influenced by this dominating force.

Naturally, designers and developers scavenge existing software for reuse because they are familiar with the existing software and can estimate how and how much to adapt existing software in a new product. Overtime, any reusable architecture feature can be eroded as ad-hoc modifications are made to the software.

The habits of achieving short-term goals and doing things quickly with what can be conveniently used would be so entrenched in the development culture that it may have priority over alternative long-term solution where major architectural redesign is required.

4.2 Architecture Planning and Software Implementation

As mentioned earlier, Océ Technologies has set up a program and an architectural council to manage software reuse. Such initiatives have not been entirely successful as evidenced by the PCC example. There are tensions why this might be the case.

Firstly, a tension exists between the top-down architectural design approach and the bottom-up development approach. From the top down, architectural design at Océ currently stops short at defining the behavior and the interfaces of and between subsystems, architects loose in-depth implementation knowledge. From the bottom-up, the developers have the freedom to carry out the detailed design and the implementation. The development of a SPL architecture requires the architects to have a long-term vision to create software structures that are reusable. The architects must also have in-depth knowledge of the software components to design the system structures and their interfaces so that the architecture platform is reusable. With this top-down design, some of the freedom that developers currently have enjoyed would be taken away.

Secondly, implementing non-functional requirements such as modifiability and flexibility requires a long-term vision of a software product. It also requires architectural governance to ensure its successful implementation.

SPL architecture design cannot be planned at the implementation level as each implementation team focuses on local features and functions. The design for non-functional requirements that overarches a system must be exercised and coordinated at the architectural level. It has created a tension with Agile development method where long-term planning clashes with short-term implementation demands. On the other hand, some designers and developers can be visionary and see opportunity for software improvements and they seize that opportunity, as the Agile development process encourages. The opportunistic approach may improve software reusability in local software modules but they would not be systematically planned for the entire system.

4.3 Reusing Legacy Software

When there is a complete set of legacy software that works for the existing products, it is difficult to justify why a new SPL architecture is a good idea, especially when product deadlines are looming. Even though SW management and designers agree in principle that the reusability of software can be improved and SPLE is a nice idea, there is an inertia to redevelop working software. At the management level, they are concerned with managing the costs, benefits and the risks of change. Some people would argue why change what already works, and face uncertainties that SPLE may not deliver the benefits. Sufficient justifications are required to support such a business case.

At the technical level, a concern is about the scope of the change, i.e. what must be redesigned, and the reusability from such a design; another concern is about the implications of the technical design, i.e. what is the impact of a SPL architectural design on the rest of system. How much software needs to be redeveloped? The tension is about why change is needed.

4.4 Knowledge of Costs and Benefits

Despite many economic models that exist, the costs and benefits of SPL implementation is application domain dependent, and requires in-depth knowledge on how to estimate the costs and the benefits accurately. In an organization that is transitioning from legacy software to SPL, it is difficult for architects to estimate the ROI because such information is typically unavailable.

As SPL architecture development is a long term investment. The technical soundness and feasibility of such an endeavor is often challenged in terms of the potential benefits that it may bring. An organization that is starting with SPLE may know its current costs, but the future benefits can be difficult to quantify. These tensions can inhibit a software organization to make sound decisions in adopting SPLE.

4.5 It Is About Knowledge

The tensions to transit from a legacy process to a SPLE process exist primarily because important knowledge is lacking and the knowledge is not communicated between the right parties. General knowledge of SPLE is widely available in industrial and research reports. However, their applicability to specific application domains is largely unknown.

At the technical level, architects and designers need to be aware of SPL principles and if those principles are applicable and beneficial to their existing software products. They have to learn and experiment with these principles to explore their applicability. The costs and benefits would need to be measured to support decision making for both technical and SW management.

SPL architecture planning is an integral part of SPLE. Architects and engineers must share the knowledge if a SPL architecture design is to work. Architects also need to obtain in-depth implementation details in some parts of a system in order to analyze software commonalities and variability.

On the other hand, SW management needs to know the technical feasibility of software changes and their potential ROI. SW management also needs to learn about managing SPLE and how SPLE impacts on the architecture design, the current development organization and on the existing engineering programs.

5. Dual Level Knowledge Sharing in SPLE

Venturing into SPLE by a well-established software development organization is about managing the risks and the relative benefits of the changes. It is also about resolving various tensions in the development process, in the investment process and in the planning process. We suggest that sharing appropriate knowledge at the right time between the technical and SW management staff can alleviate some of the tensions during SPL migration. This is an iterative and incremental process where knowledge is exchanged in a feedback loop (see Figure 2). In the exchange, SW management benefits from the technical inputs, and, on the other hand, technical staff receives directives from well-considered SW management decisions and can work in an environment that is conducive to SPL architecture planning.

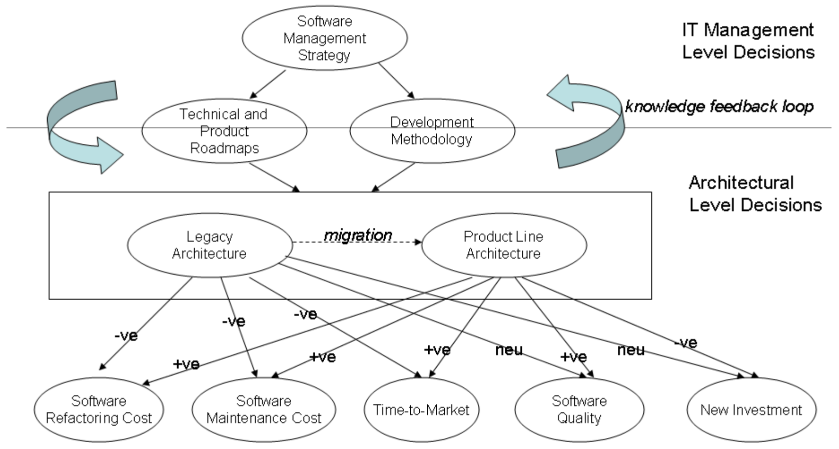
Software Management Strategy
IT Management Level Decisions
Technical and Product Roadmaps
Development Methodology
knowledge feedback loop
Architectural Level Decisions
Legacy Architecture
migration
Product Line Architecture
-ve
-ve
-ve
+ve
+ve
+ve
neu
+ve
neu
-ve
Software Refactoring Cost
Software Maintenance Cost
Time-to-Market
Software Quality
New Investment

Figure 2. Knowledge Feedback Loop in Software Product Line Management

5.1 Justifying SPL Costs and Benefits

One of the major challenges of migrating to SPL is to systematically explore opportunities to improve software reuse. The opportunistic approach of stumbling across the "low-hanging fruits" of software reuse is not sustainable as a business model. SW management needs facts to systematically justify and implement SPL reengineering.

In section 3, we have illustrated the cost savings of PCC implementation and its potential productivity gains. Further development and maintenance of a SPL architecture platform requires budget allocation. Additionally, the qualitative analysis has shown that the software quality and the time-to-market can also benefit from this initial investment. SW management can use this knowledge to revise its development strategies where budgets need to be allocated to projects to investigate the costs and benefits of SPL designs. Additionally, investigation should take place to assess applicability of SPLE in other areas. The costs of SPLE adoption are represented by the SIMPLE model:

- Organizational cost (C_{ORG}) – maintain a team or a group of architects and developers to support this initiative; continue to maintain PCC-4 as a CAB; to prevent the CAB from architectural erosion; to create sufficient documentation to retain the knowledge of using and maintaining this CAB.
- Reuse cost – the current reuse cost of this case study is calculated based on designers and developers who have been involved in the development of the CAB. The reuse cost should be higher than what is reported here when other developers start to use it. This is because the other developers have to learn the SPL architecture platform, understand its design, analyze the variability points and the testing regime.

These two costs can now be estimated but more data are required to improve their accuracy. They represent the partial knowledge that needs to be gathered as SPL implementation progresses. With the initial success, SW management would need to maintain the foothold that has been gained and extend such gain to other potentially beneficial areas.

If an incremental SPLE approach is taken to migrate the legacy software to a SPL platform, a collaborative process between SW management, architects and developers to make an objective assessment of the ROI is required. The engineers, designers and architects, on one hand, can present SPL opportunities and technical assessments for management's considerations. SW management, on the other hand, needs to have in-depth understanding of the short-term and long-term costs and benefits of each SPL migration.

Such a balance is about the cost of implementing a upcoming product release and the long-term benefits that may be gained if a SPL architecture is used. When implementing certain software feature, it may be possible to align these two objectives or to "kill two birds with one stone". If the cost of implementing a legacy solution is CL_{UNIQUE} and the cost of implementing a SPL solution to achieve the same requirement is (CP_{UNIQUE} + CP_{CAB}), then the question is how much can (CP_{UNIQUE} + CP_{CAB}) exceed CL_{UNIQUE} before we decide to choose a SPL design. This question is not applicable if CL_{UNIQUE} has a higher or equal cost to (CP_{UNIQUE} + CP_{CAB}) because one would then choose a SPL solution.

The answer to this question depends on knowing a few factors: (a) the cost difference of software refactoring, using the SIMPLE model in this case, and the amount of reuse of this SPL architecture in the future as a basis of calculating the ROI, the time frame used in the estimation should not be unrealistically short or long; (b) the maintenance cost improvements from the SPL architecture design; (c) the time-to-market factor, that is if the company has to react to market competition quickly, what would be the opportunity cost of not having this SPL architecture; (d) improvements in software quality.

Figure 2 shows a comparison of these factors in terms of continuing with the legacy software or migrating to a SPL platform. A migration to SPL would require additional development cost, so it has a negatively (-ve) impact on investment, whereas legacy software requires no new investment and so it is neutral. In this case study, we have found that the SPL implementation (PCC-4) has positive (+ve) impacts on future refactoring cost, maintenance cost, time-to-market and software quality. When this is compared to PCC-1, all these aspects are either negative (-ve) or neutral (neu). This knowledge that has been learned has enabled SW management and architects to justify the use of SPLE and continue with this success.

In the future, software refactoring and maintenance costs will be measured using SPLE and software engineering practices. The other two factors are both quantitative and qualitative in nature, and they require measurements and judgments from both the architects and SW management. This knowledge will be used to support decision making for incremental SPL practices.

5.2 SPL Architecture Planning Discipline

In migrating from legacy architecture to a SPL architecture. There are different approaches in architectural implementation: the revolution approach is to reengineer the entire architecture; the reactive approach is to reengineer an architecture when opportunity arises; the proactive approach is to investigate the potential gains of SPL and prioritize the reengineering.

Which approach to take depends on the situation. Whilst a revolution approach may be suited for start-up product lines, it would be costly and risky when many legacy software are in place and all of them are reengineered at the same time.

A reactive approach is dictated by the circumstances. It relies on opportunities showing up by themselves. As in our case, if the resource and the market situations were different, the opportunity of reengineering PCC-4 might not have been present. So a reactive approach does not provide the company the timely competitive edge that it can gain from SPLE. Additionally, the architecture design resulting from this approach may not be coherent because of the piecemeal and unplanned design activities.

The proactive approach requires investments in a SPL architecture organization to do planning and scoping. Architects should actively analyze the commonalities and variability of past, existing and future products to seek improvements. Past and existing designs can be obtained from design documentation and the codebase. Future features can be gathered from technical and product roadmaps, and stakeholders. Then decisions need to be made as to which part of the system (i.e. scoping) can be implemented in a SPL to gain the benefits. The cost of the SPL architecture investigation represents the startup and organization investment.

The planning and execution of any SPL architecture approach requires SW management commitment and architecture planning and governance. SW management must have a strategic plan on what it intends to invest and why. This knowledge must be communicated to the technical staff to align expectations and objectives. Architects must ensure that reducing cost and improving software reusability, time-to-market and high software quality are the goals in the strategic plan. The architecture roadmaps and architecture design must be aligned, investigated and updated to reflect what and how these goals can be achieved. Architectural planning and governance would address the software reusability issue similar to what has happened in PCC-4 development.

5.3 Adapting SPLE to Agile Development Culture

The tension between SPL architecture planning and the Agile development method is about the freedom of making decisions and by whom. SPL architecture constrains, to a certain extent, how developers can implement software. Innovative developers typically do not want those constraints. We suggest that this tension can be eased if the reasons of the SPL architecture design are communicated clearly. A well-considered architecture design should have the following information: the context and the considerations of the design, the key architectural decisions and the tradeoffs that have been made, and the resulting design is the logical outcomes of those considerations [10]. The knowledge of such design rationale is important to convince developers to adhere to the SPL design and its principles.

Although SPL architecture design is a top down approach, sharing design rationale is an important avenue to gain the trust and buy-in from developers. Under a SPL architecture framework, the innovative forces of the developers can continue to work. Developers should be encouraged to explore and strengthen SPL opportunities. Additionally, developers should be encouraged to find ways to improve the benefits, quality and the architectural design of the product. The additional contributions to improving the SPL architecture design and the economics of development is a shift from merely delivering to the short-term goals. It requires developers to think about the long-term benefits their work would bring to the organization.

6. Related Work

Software reuse in specific domains, or SPL, has been studied for a long time. It is well recognized that software architecture is an important consideration in SPLE [7, 11]. The key idea is to differentiate between commonalities and variability in an application, and implement the architecture in a way that the application domain knowledge can be encapsulated in a reusable application architecture platform common to the different products [12]. Different methods have been suggested to scope out which parts of the system should become a SPL [13, 14]. The key considerations in SPL scoping include asset scoping, product portfolio planning, product line analysis, domain potential assessment, release planning and asset scoping.

Successful SPL implementations have been reported. For instance, HP's Owen approach is reported to have been successful in the inkjet product family [9, 15]. In the last 10 years, whilst staff has grown by a factor of 5, the code-base has increased 10 folds and the number of products per year increased 7 folds. This productivity gain is attributed to SPLE. HP's success is not only a result of the engineering ingenuity and architecture design. It is fundamentally important to manage the engineering process appropriately to achieve such results.

A scavenger approach [16] is obviously not the most effective method for planning software reuse. Kircher et al [17] suggest that no strong rules can be derived from the success stories in SPL yet. They believed that managers should shape the organization whilst the software engineers recognize the need to change. This view is also shared by us in terms of the collaboration through knowledge exchange in the feedback loop.

A particular area of interest is how Agile development process works with SPLE since Océ Technologies has adopted this development method and its culture is entrenched in it. The Agile method is about anticipating new requirements and changes, it is suited for meeting short term software development goals where project customers are small and discrete, it is suited to delivering quick results [18]. On the other hand, SPLC is about building assets, and planning the software architecture so that reuse can be maximized for the current products sets as well as for the future.

The fundamental philosophy of these two approaches is opposite to each other. The terms of reference in Agile is short versus SPLE's long term investments in software asset development. The motivations of the developers, architects and management are also different under these models. A challenge to the migration from legacy system to SPLE lies in the reconciliation of these two methods. A suggestion is to strengthen architectural practice to mitigate the differences [19]. Another suggestion is to emphasize the commonalities of the two approach: collaboration, changing requirements and maximizing outputs [20]. A further suggestion is to adopt organizational and management changes in a coordinated way to handle the changing environment [2].

The eventual success of a SPL implementation is measured by the benefits that it delivers. There are a number of economic models to address this issue. Boehm et al. suggested the Constructive Product Line Investment Model (COPLIMO). The model measures initial development cost and the annualized post-development life cycle, the calculation requires estimating the number of lines of code [3]. However, one could argue that the code that is built for the SPL architecture would have higher complexity and therefore more expensive to build than the custom application code, a case where the cost is dependent on the type of codes in question [4].

The SIMPLE model proposed by Clements et al. classifies SPL costs into organization, core asset, reuse and unique costs. By measuring the costs of setting up SPLE organization and constructing core assets, and measuring the benefits that result from it, they work out the ROI [6, 21]. We find that this economic model can measure our data and so we choose to use it. In addition to a quantitative economic model, we need to also consider time-to-market and quality. There are suggestions in [8] about a qualitative analysis method that evaluates the costs and benefits based on qualitative information.

These works have inspired our vision in SPLE migration. As far as we know, we have not found any reports or papers that show SPL migration in a similar situation as ours.

7. Conclusions

SPLE engineering has been studied for many years, and successes have been reported. In a situation where legacy software has been used in products successfully and the Agile development process is working well, the inertia to migrate to a SPL architecture is great despite the obvious benefits that SPLE may bring.

In a recent industry case, we report and analyze the costs and benefits of a successful SPL reengineering of a major architecture component. From the analysis, we recognize that there are multiple tensions that restrict an organization from embracing SPLE. The choice of continuing with the existing software development approach or moving towards SPLE is multi-faceted. It is concerned with the approach to software reuse – opportunistic or systematic; the organizational approach to delivering software under the Agile environment; the approach to architectural planning; and the benefit measurements for justifying change.

A transition to a SPLE development environment requires technology and organizational changes. It requires coordinated efforts between SW management and technical staff. In view of the tensions, new knowledge such as architectural plans, costs and benefits must be gained and shared to help both groups to manage change. This paper discusses the sharing of essential knowledge to help SW management create a development environment conducive to innovations and long-term software reuse, and help technical staff align their design and innovations towards SPLE.

Acknowledgments

This research has been partially sponsored by the Dutch "Regeling Kenniswerkers", project KWR09164, Stephenson: Architecture knowledge sharing practices in software product lines for print systems.

References

[1] G. K. Hanssen and T. E. Fægri, "Process fusion: An industrial case study on agile software product line engineering," *Journal of Systems and Software,* vol. 81 (6), pp. 843-854, 2008.

[2] K. Mohan, B. Ramesh, and V. Sugumaran, "Integrating Software Product Line Engineering and Agile Development," *IEEE Software,* vol. 27 (3), pp. 48-55, 2010.

[3] B. W. Boehm, A. W. Brown, R. J. Madachy, and Y. Yang, "A Software Product Line Life Cycle Cost Estimation Model," in *International Symposium on Empirical Software Engineering (ISESE)*, 2004, pp. 156-164.

[4] W. B. Frakes and C. Terry, "Software Reuse: Metrics and Models," *ACM Computing Surveys,* vol. 28 (2), pp. 415-435, 1996.

[5] G. Böckle, P. Clements, J. D. McGregor, D. Muthig, and K. Schmid, "A cost model for software product lines," in *Software Product-Family Engineering*, 2004, pp. 310-316.

[6] P. C. Clements, J. D. McGregor, and S. G. Cohen, "The structured intuitive model for product line economics (SIMPLE)," CMU/SEI-2005-TR-003, 2005.

[7] W. B. Frakes and K. Kang, "Software Reuse Research: Status and Future," *IEEE Transactions on Software Engineering,* vol. 31 (7), pp. 529-536, 2005.

[8] J. McGregor, "Qualitative SIMPLE," *Journal of Object Technology,* vol. 7 (7), pp. 7-16, 2008.

[9] H. Mebane and J. T. Ohta, "Dynamic Complexity and the Owen Firmware Product Line Program," in *SPLC,* 2007, pp. 212-222.

[10] A. Tang, J. Han, and R. Vasa, "Software Architecture Design Reasoning: A Case for Improved Methodology Support," *IEEE Software*, vol. Mar/Apr 2009 (pp. 43-49, 2009.

[11] P. Clements and L. Northrop, *Software product lines*: Addison-Wesley Reading MA, 2001.

[12] K. C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Project Line Engineering," *IEEE Software,* vol. 2002 (July/August), pp. 58-65, 2002.

[13] J. Savolainen, J. Bosch, J. Kuusela, and T. Männistö, "Default values for improved product line management," in *Proceedings of the 13th International Software Product Line Conference (SPLC)*, 2009, pp. 51-60.

[14] I. John and M. Eisenbarth, "A decade of scoping: a survey," in *Proceedings of the 13th International Software Product Line Conference (SPLC)*, 2009, pp. 31-40.

[15] P. Toft, D. Coleman, and J. Ohta, "A cooperative model for cross-divisional product development for a software product line," in *Software product lines: experience and research directions: proceedings of the First Software Product Lines Conference (SPLC1)*, 2000, pp. 111-132.

[16] H. v. Vliet, *Software Engineering: Principles and Practice*, 3rd ed.: John Wiley & Sons, 2008.

[17] M. Kircher, C. Schwanninger, and I. Groher, "Transitioning to a Software Product Family Approach - Challenges and Best Practices," in *Proceedings of the 10th International Software Product Line Conference (SPLC)*, 2006, pp. 163-171.

[18] M. Ali Babar, T. I. and, and M. Pikkarainen, "An industrial case of exploiting product line architectures in agile software development," in *Software Product Line Conference (SPLC)*, 2009, pp. 171-179.

[19] P. Abrahamsson, M. Ali Babar, and P. Kruchten, "Agility and Architecture: Can They Coexist?," *IEEE Software,* vol. 27 (2), pp. 16-22, 2010.

[20] J. McGregor, "Agile Software Product Lines, Deconstructed," *Journal of Object Technology,* vol. 7 (8), pp. 7-19, 2008.

[21] G. Böckle, P. C. Clements, J. D. McGregor, D. Muthig, and K. Schmid, "Calculating ROI for Software Product Lines," *IEEE Software,* vol. 21 (3), pp. 23-31, 2004.