Incremental and Iterative Reengineering towards Software Product Line: An Industrial Case Study

Gang Zhang[1,3], Liwei Shen[1], Xin Peng[1], Zhenchang Xing[2], Wenyun Zhao[1]
[1]School of Computer Science and Technology, Fudan University, Shanghai, China
[2]School of Computing, National University of Singapore, Singapore
[3]Alcatel-Lucent Shanghai Bell, Shanghai, China
{09110240022, shenliwei, pengxin, wyzhao}@fudan.edu.cn
xingzc@comp.nus.edu.sg

Abstract— **It is common in practice that a Software Product Line (SPL) is constructed by reengineering a set of existing variant products. To alleviate the problems of high risks of failures and the limitations of resources and cost, incremental reengineering towards a SPL is a natural choice in many cases. However, several problems remain unaddressed properly, such as how to define increments, how to satisfy regular product delivery in parallel with reengineering, and how to achieve early successes. In this paper, we report an industrial case study on a successful SPL-targeted reengineering project conducted in Alcatel-Lucent. In this project, the project team applied the principles of agile development in the process of SPL reengineering. The key practices of the project include value-based increment definition, domain-driven responsibility alignment, iterative component refactoring and integration. We analyze the reengineering process of a major component qualitatively and quantitatively, with the focus on initial investment required, trend of investment, returns on investment and quality improvement. Our case study shows that incremental and iterative approach with stakeholder-value considerations can help to achieve steady and successful SPL reengineering in a cost-effective manner. We also find that SPL adoption can be regarded as an emergent result of the reconstruction and improvement of existing product assets.**

Keywords- software maintenance; software product line; reengineering; transition; Agile software development

I. INTRODUCTION

Software Product Line (SPL) is proposed as an economic way to develop and maintain the set of variant products in a specific domain [1]. SPL engineering emphasizes high-efficient product development by proactive reuse. However, it is common in practice that a software organization already has a collection of variant products developed with ad-hoc reuse (such as version branching) before it decides to adopt SPL. Usually, the organization may make this decision when it feels that the cost of product development and maintenance grows too fast or it becomes impossible to derive new, envisioned products based on existing legacy products [3]. In such situations, a reengineering-driven SPL adoption can be employed, which is referred to as the migration to SPL in an extractive way [10]. During the reengineering process, commonality and variability among the variant products are identified, core assets are extracted, and legacy products are reconstructed based on the shared core assets.

Extractive SPL reengineering can either employ "big bang" mode that reengineers existing legacy products as a whole, or incremental mode using a component-wise approach [3]. For reengineering large-scale legacy products, "big bang" approach requires large up-front investments and takes a long period of time to gain returns. As a result, the project team often encounters a series of difficulties such as lack of resources and high risks of failures. Therefore, in many cases, incremental reengineering by components offers a more feasible choice to alleviate the problems of large investments, limited resources, and risks of failures.

However, several challenges still remain unaddressed in incremental reengineering for SPL adoption. First, a team usually has to start a reengineering process with small initial investment and resources. Second, in most cases, the team is required to accomplish regular product delivery in parallel with the reengineering process. Third, staged success and early returns are often required to strengthen the confidence of senior managers. These challenges can be summarized as one problem, i.e. how to properly define and carry out the reengineering increments in a steady, risk-reduced and cost-efficient way.

In this paper, we report our case study on a SPL reengineering project recently conducted in Alcatel-Lucent. The reengineering process of the targeted product family (called *IXM-PF*) suffered from the challenges described above. Thus, in this project, the project team decided to combine the methodologies from SPL engineering and reengineering, and apply principles of agile development [9] to SPL reengineering. Four key practices are involved in the reengineering process. First, it involved a value-based increment definition that requires only small initial investment and thus enables early success. Second, it employed a domain-driven process to align the boundary of components and guide component refactoring. Third, the increments were reengineered with localized impact to facilitate early integration and ensure regular product delivery. Fourth, in each increment, component refactoring and integration in one product were iteratively propagated to other variant products.

The *IXM-PF* reengineering project achieved initial success after one and a half years. It should be noted that the same team was responsible for the regular maintenance and delivery of related products during this time period. After reengineering, an initial set of core assets had been established and reused in both legacy products and several new products. This result strengthened the confidence of the organization on SPL reengineering.

The result of the project has been evaluated by means of the quantitative analysis of investment, return, and product quality.

Our case study shows that incremental and iterative approach with stakeholder-value considerations can help to achieve steady and successful SPL reengineering in a risk-reduced and cost-effective manner. Moreover, we find that SPL adoption can be regarded as an emergent result of reconstruction and improvement of existing product assets by applying incremental and iterative strategy.

The remainder of the paper is organized as follows. Section II presents the background of the case study and an overview of the reengineering process. Section III details the practices adopted in the project with the problem, solution, and result analysis. Sections IV and V present the evaluation as well as the discussion respectively. Section VI reviews related work on reengineering towards SPL. Finally, Section VII concludes the paper and outlines our future work.

II. Background and Process Overview

A. Background

Our case study is based on an industrial SPL reengineering project in a telecom software product family named *IXM-PF*. *IXM-PF* has been developed and maintained for more than ten years in Alcatel-Lucent. It contains a set of variant products targeting different markets, from cost-sensitive markets to rich application markets, interleaved with different deployment scenarios and supporting of different transport standards. Over time, new products were developed by branching and adapting from selected ancestors in an ad-hoc way. Furthermore, variant products were maintained separately. Each product in the family contained more than 10M lines of C/C++ code, scattered among 300 to 400 modules.

A project team was built to take the task of reengineering *IXM-PF* towards SPL. The first author of the paper participated in the whole project from the beginning as an architect and a methodology expert.

Figure 1 depicts the evolution graph involving all the variant products whose names are modified due to commercial considerations. The products inside the dotted circle are those legacy products involved in the SPL reengineering project. In the project, the team is expected to identify and reconstruct a series of components as core assets that can not only unify the maintenance of legacy products but also support the derivation of new products.

Various kinds of stakeholders are involved in the product family. The marketing and sales persons as well as their representatives demand quick response to customer requests and high-quality product delivery. The architects advocate the reengineering towards SPL in order to avoid duplicate development and increased maintenance cost. Furthermore, senior managers hope that long-term benefits can be achieved with low risks and small initial resource requirements. Although the long-term goals of different stakeholders seem to be consistent, the project team has to balance long-term benefits and short-term returns, and achieve early success under the constraint of limited resources. A win-win solution [16] with less initial effort and earlier return was highly desired from the organization in order to achieve long-term benefits of SPL adoption, ensure regular product delivery, and provide early success to strengthen the confidence of the management.

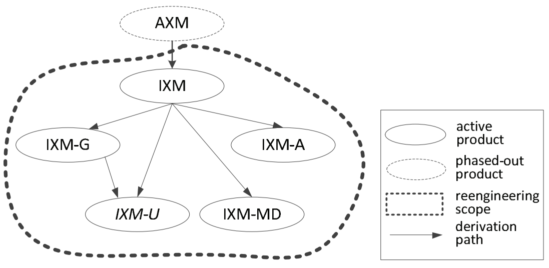

Figure 1. Evolution graph of IXM-PF

B. Process Overview

The project team has applied the Agile principles [9] to the SPL reengineering. Specifically, the project team adopted an incremental and iterative reengineering process as shown in Figure 2 in which the circles and rectangles denote activities and artefacts respectively. The process involves two layers. In the outer layer, an incremental process was conducted by components, and in the inner layer, an iterative sub-process was performed among different variant products. As a result, a set of domain components were accumulated as core assets, while the variant products were reconstructed on the shared core assets.

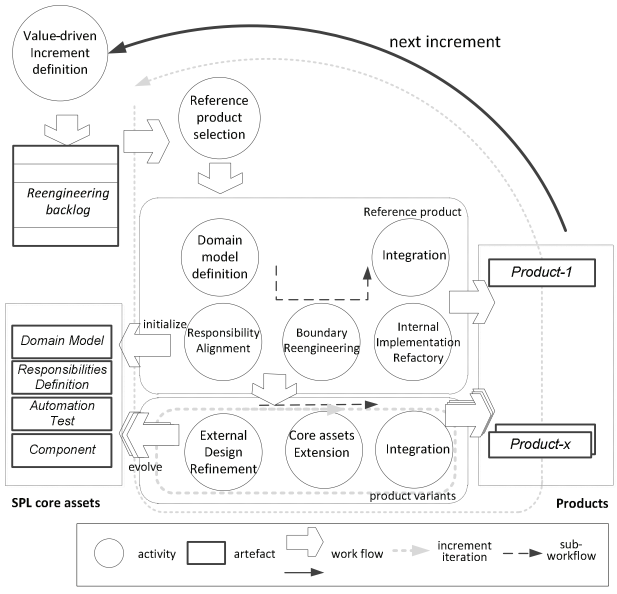

Figure 2. Process overview of the *IXM-PF* reengineering project

In the outer layer, components were identified, reconstructed and incorporated into core assets incrementally. An increment thus covers a specific component. A backlog specifying the order of component reengineering was defined as the roadmap of the incremental process (Section III.A). It should be noted that not all the components were included in the backlog. Only those relevant to maintenance problems or new market opportunities were chosen in the scope of reengineering.

In the inner sub-process for each increment, the component extraction and reconstruction was first conducted in a selected reference product and then the component was iteratively propagated to the other variant products. The reference product was selected based on its value in the domain, i.e. the product that was actively maintained and shared the most commonalities among the variant products (Section III.B.(1)). The initial implementation of the targeted component in an increment was extracted from the reference product.

A domain model represents the concepts of the component to provide high-level guidance for the extraction and reengineering of the component (Section III.B.(2)). Note that the team did not construct a complete domain model at the beginning of each increment. Instead, they gradually extended and refined it iteratively. Based on the domain model, the component responsibility definition was aligned with the domain concepts (Section III.B.(3)), and then both the component boundary and the internal implementation could be refactored accordingly. The component boundary was refactored to conform with the responsibility definition and restrict the scope of the component implementation refactoring (Section III.B.(4)). On the other hand, the internal implementation was refactored to improve the maintainability of the component and enhance its reusability and extensibility for handling variability in the SPL context (Section III.B.(5)). Next, the refactored component was integrated with the other parts of the reference product (Section III.B.(6)).

After component refactoring and integration in the reference product, the reengineered component was iteratively propagated to the other variant products. In each iteration, the domain model established for the current component and the responsibility definition were refined and extended to accommodate the differences brought in by the variant product under investigation. This was called external design refinement by the team (Section III.C.(1)). Then, at the implementation level, the component was adapted and integrated to replace the corresponding part in each variant product, which was called core assets extension and integration (Section III.C.(2)). This iterative propagation ensured that the refactored component was shared among different variant products and helped to achieve the goal of SPL reengineering in an incremental mode.

III. PRACTICES

In this section, we detail the practices involved in the reengineering process through an increment reengineering of a specific component in *IXM-PF*. Each practice is described in terms of *problem*, *solution*, and *result*. The problem identifies the difficulties the team encountered when they tried to perform the practices in the reengineering process. The solution gives out the detailed considerations towards the objectives of the relevant activities in each practice. The result is thus emergently obtained to achieve SPL adoption.

A. *Value-driven Increment Definition*

Problem: *IXM-PF*, as a large-scale legacy product family, contains 430 components in total. In an incremental reengineering process, the team anticipates that the reengineering of the selected components can bring immediate benefits to the on-going project, i.e. to maximize Return of Investment (ROI) for the organization. Thus, how to choose a suitable set of components as the current and the later increments is the first consideration.

Solution: The goal of increment definition is to identify and prioritize the candidate components that will be reengineered to the SPL core assets. All items in the backlog should be prioritized depending on the value for the organization after reengineering. In the initial phase, only selected components are put in the backlog instead of all the components, because the components will be reengineered incrementally and the backlog should be refined continuously to reflect dynamic business context. According to Lean principle [19], the backlog covering all components should be avoided.

The value of a component is evaluated from two dimensions: the project dimension and the SPL dimension. It covers the interests of both the short-term and the long-term sensitive stakeholders.

1) The project dimension focuses on the potential benefit of the reengineering of a component in an on-going or a predictable project. Therefore, a component that requires much maintenance or new feature introduction effort will be assigned with high priority because the cost can be reduced greatly after the successful reengineering. On the contrary, a component is of no value if it is not involved in any future plan.

2) The value of a component in the SPL dimension is determined by its variability. Among the product family, components can be classified into three categories: same, different, and standalone. Thus, we give higher priorities to the components that are classified into “different” because they are worth being reengineered in order to enhance the reusability and extensibility.

Result: By identifying the priority for the potential candidates of SPL core assets, the team obtained the initial backlog of components to be reengineered. *EntityManagement* appeared on the top of the backlog due to its contribution to both the project dimension and the SPL dimension.

As for the project dimension, the team already knew that several new projects would impact a component called *EntityManagement*. Based on the historical data, there were 92 faults reported in total in the last two years on the component and the total cost of solving the faults in *IXM* exceeded two Person Years (PY). *IXM-A* and *IXM-G* had the similar situations. Improving the maintainability of the component could help to decrease the maintenance effort. In addition, the architecture and strategy department forecasted that the component would be extended to support new entity types in six months. It thus showed an opportunity that the creation of SPL core asset could be integrated with feature development.

As for the SPL dimension, *EntityManagement* contained variaton points among the products. In the case study, the component was reused in each product in an ad-hoc way. It resulted in a lot of branches on the component. Thus, it was desired to be reengineered as a consistent artefact with the support of variability implementation mechanisms.

For the remainder of the paper, we will use the component *EntityManagement* as an example to illustrate the detailed steps in the corresponding increment. We have changed the names and related identifiers without impacting the reader's understanding for the commercial reasons.

B. Component Reengineering in Reference Product

After the increments were defined in the backlog, the reengineering was conducted on the components in order. In each incremental cycle, the component was first extracted from a reference product that was selected according to several value-driven principles (III.B.(1)). Then the process was carried out through a sequence of activities, from domain model definition (III.B.(2)), to responsibility alignment (III.B.(3)), boundary reengineering (III.B.(4)), implementation refactoring (III.B.(5)), and finally integration into reference product (III.B.(6)).

1) Reference product selection

Problem: There were five products involved in *IXM-PF* to be reengineered. The initial implementation of the component was extracted from one of the products (called *reference product*). After being reengineered, it was further propagated to the other products. The reengineering upon the reference product should bring benefits for the project and ease the propagation to the other products. Thus, a proper reference product for the reengineering of the initial component had to be decided.

Solution: The value-driven principles are applied to achieve the selection criteria as follows.

- The selected product should be active in regular development. The regular project benefits from the improvement of extensibility of the core assets. Meanwhile, SPL core assets benefit from regression test suites of the regular project. This criterion helps to increase ROI for the organization.
- Under the above constraint, the similarity between the selected reference product and the other products should be maximal. This criterion helps to reduce the total cost of the component propagation to the other products.

TABLE I. PROPERTY-VALUE PAIRS OF ENTITYMANAGEMENT

	IXM	**IXM-A**	**IXM-MD**	**IXM-U / -G**
Identifier strategy	static bitmap	static bitmap	static bitmap	dynamic generation
Dimension of Entities	High	High	Low	Very High
Lookup performance	medium	medium	medium	high
Distributed	Yes	Yes	No	Yes
Supported Entity types	different supported entity types between the products			
Supported Card types	different supported card types between the products			

Result: In the project, the team selected *IXM* as the reference product for the reengineering of the component *EntityManagement*. They came to this decision based on the facts that all five products were in active status but *IXM* had the maximal similarity with the other products. Table I listed the property-value pairs of the component *EntityM*anagement in the five variant products. In the last two rows, the types of supported entity and supported card are both recognized as different between the products rather than pointing out their type names. From the table, *IXM* shared the most common values with the other products, such as *identifier strategy*, *dimension of entities*, and *lookup performance*.

2) Domain model definition

Problem: The team found that *EntityManagement* contained 135 responsibilities that are the component services realizing business requirements. They were implemented as operations belonging to the source code level interfaces. However, some responsibilities were not relevant to or even conflict with the central concept of the component. Therefore, they needed to clarify the component concepts in a formal way. This helps to establish the basis for component reengineering.

Solution: Domain Driven Design (DDD) is a vision and approach to tackle the complexity mentioned above. DDD puts the domain model as the main focus, which reflects a deep understanding of the domain. DDD is introduced in Evans' book of *Domain Driven Design* [11]. The concept "domain" means "*a sphere of knowledge, influence, or activity*". This concept is critical to the clarification and refactoring of the component responsibility, boundary, and internal implementation. Furthermore, it also contributes to addressing the variability of the components across variant products. In our solution, the initial version of the domain model was mainly established for the reference product. The concept related with the other product variants (for example the optional *concreteEntityTypeX* in Figure 3) could be included, but it is not expected that the initial model contains all the information for all the product variants. This domain model will be evolved iteratively in the external boundary engineering phase that will be discussed in Section III.C.(1).

Result: The team constructed the domain model of *EntityManagement* through an intensive workshop that involved domain experts, architecture experts, and designers. The workshop analyzed the legacy assets from the reference product and distilled the core concepts. Figure 3 shows part of the domain model created in this iteration. This model adopts the notation of class model where the nodes represent domain concepts and the edges represent relationships between the concepts such as inheritance and aggregation.

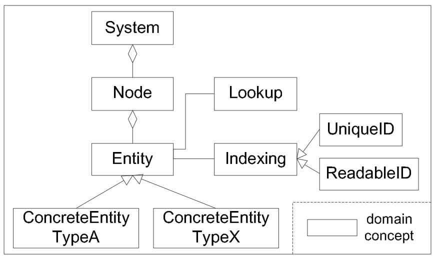

Figure 3. Part of the domain model in the first iteration

3) Component responsibility alignment

Problem: Before performing the reengineering upon the component implementation, the team had to identify the responsibilities in *EntityManagement* that deviated from the domain model defined in the preceding sub-section. The identification of responsibility deviations helps the team align component responsibility according to the domain concepts.

Solution: The domain model is used as a calibration for the identification of component responsibility deviations. According to the domain model distilled in the processes of the preceding step, the existing component responsibility definition should be re-aligned with the domain model. We define the relationships between the component responsibilities and the domain concepts into the following four types: *consistent*, *partially consistent*, *erroneous*, and *irrelevant*. The partially consistent and the consistent responsibilities should be re-aligned according to the domain model. The erroneous and the irrelevant responsibilities should be removed in order to avoid polluting the domain concepts.

Result: Table II shows the result of the responsibility alignment of *EntityManagement*. It shows that 27 out of 135 operations were consistent responsibilities and thus were retained. In addition, 48 partially consistent operations were re-aligned with the concepts defined in the domain model. According to the rules, the remaining 21+39 erroneous and irrelevant responsibilities were removed.

TABLE II. RESPONSIBILITY CONSISTENCY WITH THE DOMAIN MODEL

Consistent	Partially consistent	Erroneous	Irrelevant
27	48	21	39

A responsibility refinement was followed. In the project, the team changed the header files of *EntityManagement* from *C* to *C++* according to an organizational strategy. Meanwhile, after a detailed alignment of the component responsibilities with the domain model, they redefined the responsibilities into thirty operations clustered in four interfaces (some overlaps existed between consistent and partially consistent responsibilities, thus the number of the merged operations was decreased a lot).

4) *Boundary reengineering on reference product*

Problem: In legacy products like *IXM*, the boundary around the implementation of *EntityManagement* was vague and inconsistent with the expected domain model. The implementation was interwoven with other codes so that it was difficult to extract the component *EntityManagement* from the reference product. Therefore, the team needed to wrap the component as a relatively independent unit in order to avoid the side-effects of the component reengineering on the other parts of the product.

Solution: The component boundary in the reference product clarifies the scope for reengineering the component. A correctly defined boundary guarantees safe refactoring of a component because modifications will be limited inside the component.

At the implementation level, boundary reengineering indicates the modifications of source codes from two aspects: the internal-adapters and the external-dependencies.

The internal-adapters encapsulate existing implementation within newly defined responsibilities. Internal-adapters can be regarded as an intermediate artefacts. The key consideration of the modification is to decrease the effort for future component maintenance.

The external-dependencies represent client components that depend on the component. In order to define proper external dependencies, these clients should be modified to comply with newly defined responsibilities, e.g. invoking methods through newly introduced interfaces.

Result: In this step, the team wrote a script to find the client components that depended on *EntityManagement*. Then, the external-dependencies in 54 components were cleaned up. Furthermore, the internal-adapters were implemented to provide the expected services which were defined in Section III.B.(3). In addition, system level tests were executed continuously during the whole process to guarantee that no side-effect was introduced during the boundary reengineering.

5) *Internal Implementation Reengineering*

Problem: In legacy codes, the team found frequently that one concept in the domain model had been implemented in many places. For example, in the original component, the implementation had to be modified in 139 locations to support a variant *concreteTypeA* and 158 locations for another variant *concreteTypeB*. There were also many *switch-case* fragments for component variants. These code-smells increased difficulty in the maintenance and reuse of the component, and thus need to be refactored.

Solution: Internal implementation reengineering aims to improve the maintainability of the reference product as well as to enhance the reusability of the component to support variability that will be identified in Section III.C.(1).

There are two general techniques for implementation reengineering: a generalized refactoring concept (for any activity that improves internal quality without changing external behaviors) or a specific refactoring concept (especially by very small steps, for example using techniques mentioned in the refactoring book of Martin Fowler [12]). Since refactoring is not the main concern of this paper, we will not discuss it in detail. Readers are referred to [12] that presents the detailed refactoring steps.

Result: Figure 4 presents part of the class model after the refactoring of *EntityManagement*. This class model contains several pre-identified variants in the initial iteration. It will be further evolved in Section III.C.(2).

The refactored implementation showed some key advantages. First, it was aligned with the domain model. For example, there was a class *LookupStrategy* corresponding with the variant of indexing in the domain model. Thus, it brought great convenience for the variability binding upon the implementations based on the domain model customization. Second, the core assets and the application assets were isolated. *ConfigManager_IXM* and *EntityTypeAStrategy* were the application assets specified for *IXM* (colored in grey). The other classes were the core assets that would be reused by all variant products. Third, it was a Just In Time (JIT) [19] core assets production. For example, the team postponed the implementation of *EntityTypeXStrategy* that was only used by the other products until they needed it (Section III.C.(2)).

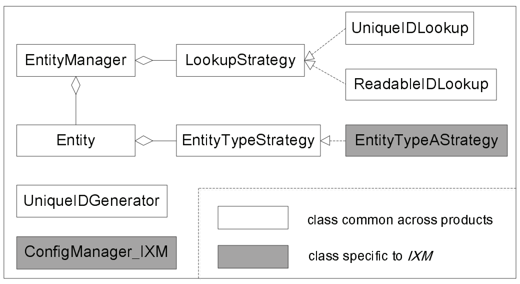

Figure 4. Part of the class model after the first iteration

In addition, Table III shows the comparison of five metrics before and after the reengineering. Three metrics (Lines of code, average McCabe complexity and average statements per operation) were optimized after the reengineering. Although the lines of code to support a variant was similar (196 vs. 140 lines), the code fragments to be modified were obviously reduced. This brought significant convenience for the reuse of the variant features.

TABLE III. SYTEM CHARACTERS COMPARISON FOR REENGINEERING

	Before reengineering	After reengineering
Lines of code	91,106	31,932
Average McCabe Complexity	5.85	2.62
Average statements/operation	15.82	7.03
Lines of code to support *concreteTypeA*	196	140
Code fragment to support *concreteTypeA*	139	2

Before reengineering on the internal implementation, the team prepared component-level automated test suites that provided a safeguard mechanism for the reengineering. The test suites covered 100% of the defined operations. For *EntityManagement*, 96 test cases were developed to cover the identified 30 operations in the four interfaces. They were executed repeatedly during the reengineering of internal implementation. The automation test suites obtained was also regarded as the important core assets for a SPL.

6) Integration into reference product

Problem: *EntityManagement* could not bring business return because it was not really reused by the reference product after the previous reengineering activities (from III.B.(1) to III.B.(5)). Only when they are applied to the product as reusable core assets, they can bring actual business value and verify the usefulness of the core assets.

Solution: Integration of a reengineered component to the reference product allows the staged feedback and increases the business value. The principle we follow is to integrate as early as possible. The reason is two folds. First, earlier integration can provide earlier feedback to identify potential problems. Second, improved implementation is more flexible, which makes it possible to catch up the requirements of other features easier when there are co-impacts on the component.

Result: In the project, the team integrated *EntityManagement* into the on-going product *IXM*. It brought immediate feedback to the project due to the extensibility of the newly obtained core assets. The detailed data will be analyzed in Section IV.

C. Iterative Core Assets Refinement

After the team obtained a set of core assets in the reference product, they employed an iterative process in the core assets refinement and the application integration. By the iterative approach, the core assets were refined. Therefore, it reduced the cost of early investment. The requirement of a complete design on all of the variant products was also lightened.

Each iteration covers one product with two basic activities. The first activity is the external design refinement, which focuses on the refinement of the domain model, the responsibility definition, the component level automated testing, and the implementation level software modification (III.C.(1)). The second activity is to extend core assets and replace the existing implementation by the core assets obtained in the previous iterations (III.C.(2)).

1) External Design Refinement

Problem: The initial core assets including the domain model, the component responsibility definition, and the automated component test suites for *EntityManagement* were already produced in the first iteration, i.e. in the reference product. However, there was only limited considerations on the other variant products in the previous iteration. They need to be refined through the comparison with other product context in order to address the variability as well as to make the core assets reusable by different variant products.

Solution: In each iteration, SPL core assets are refined by checking the similarities and differences between the existing products anticipating to reusing the core assets and the assets obtained in the last iteration.

The observation can be achieved from two inputs: the domain expert analysis and the existing source codes in the product. If new variants of the domain model or the responsibility definition are found, the corresponding assets will be adapted accordingly. The outputs of this activity include the potentially-updated domain model, the responsibility definition, and the design model (refers to the class model in Figure 4) compliant with the updated domain model. The corresponding test suites may also be adapted to keep consistent with the evolved responsibility definition.

Result: The well-defined core domain model and the responsibility definition of *EntityManagement* were kept stable and were directly reused during the propagation from the reference product to the other products in most of the iterations. It was also the reason why the team put domain model as the key asset in the previous steps. However, there were still refinements on the domain model. For example in *IXM-U*, the team found a new variant denoting that the lookup algorithm should be updated because the dimension of supported entities increased greatly in comparison with *IXM*. In addition, there was a request in *IXM-U* that the *unique identifier* strategy changed to comply with customer standards. The updated domain model is shown in Figure 5. They also added several product specific card types in the domain model, which is not displayed in the figure. A function signature was updated to change the parameter type from *int16* to *int32* to support an increased dimension in *IXM-U*.

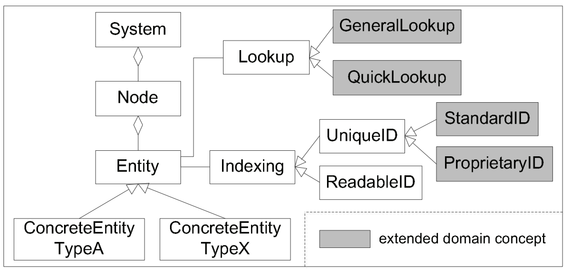

Figure 5. Part of the domain model after four iterations

2) *Core assets extension and integration*

Problem: The implementation of *EntityManagement* was refined to handle the variations between the products. Then, the team needed a solution to extend and integrate the asset into the other variant products.

Solution: Core assets extension and integration to the other variant products is an "adapt and replace" process. The two aspects are described briefly as follows:

- Adapt: First, variant implementations are added based on the code-level core assets obtained from the refactoring of the reference product. The variant implementations have been kept consistent with the updated domain model mentioned above. Similar with the previous step, if the variability implementation mechanisms have not been designed well, it is the right time to extend and complete them.

- Replace: After the adaptation of the variants has been added for the products, the old design is replaced with the newly obtained assets. The replacement of components is the integration process.

Result: By applying the "adapt and replace" activity upon *EntityManagement* in *IXM-PF*, the team found that the following points were updated which is depicted in Figure 6.

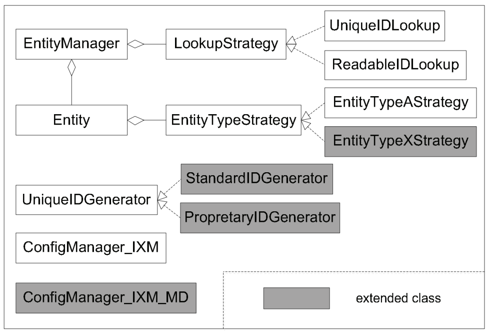

Figure 6. Part of the class model after four iterations

In the core assets, there was a request in *IXM-U* for the change of customer standard identifier strategy. First, the original *UniqueIDGenerator* was extended to two new classes: *ProprietaryIDGenerator* to support the original identification strategy and *StandardIDGenerator* to support newly requested identification strategy. Second, new *EntityTypeXStrategy* was added for request from *IXM-MD*. Finally, the application configuration was added to support the integration into other products. All of the variants in the Figure 6 are consistent with the variants in Figure 5.

During the core assets refinement, the team also run regression tests on all of the finished iterations including the reference product. Therefore, the refinement of the core assets were guaranteed not to impact the other products.

IV. EVALUATION

In the project, the team used a value-based strategy in an incremental manner. The initial investment for the reengineering was reduced, and the effort for developing new features was saved in comparison with the reengineering of the product family in "big bang" approach. This is consistent with the goal of SPL, i.e. saving total cost by reusing across multiple products and saving effort instead of requiring extra budget. In this section, we evaluate the results of the reengineering project quantitatively in the following three aspects:

- The ROI achieved when developing new features that relevant to the reengineered component.
- The extra initial investment required for adopting the incremental and iterative reengineering towards SPL.
- The benefits obtained from the reengineering after more legacy products were involved.

A. *ROI Analysis*

We performed ROI analysis in the product family *IXM-PF* involving five legacy products. *Investment* means the effort of performing reengineering upon a selected component in all variant products. *Return* denotes the saved effort when new features are to be developed based on the product line reengineered, rather than on the original product family.

Investment and return are quantified through the unit *Person Year* (PY). In particular, the efforts in investment are calculated by the sub-items according to the activities discussed in Section III. These sub-items are listed in the Investment column of Table IV (I-1 to I-6).

TABLE IV. ITEMS OF RETURN OF INVESTMENT

Return	Investment
R-1. Saved new feature development cost in all products	I-1. Domain model analysis I-2. Responsibility model definition I-3. Boundary reengineering (all related products) I-4. Component level automation test I-5. Internal implementation refactoring I-6. Total integration and faults fixing cost

The effort data of these sub-items have been recorded in the Data WareHouse (DWH) system of Alcatel-Lucent(see Table V). The table lists the sum of the efforts of I-1 to I-5 and the effort of I-6 due to the effort recording strategy in Alcatel-Lucent. From the data collected, we see that the investment for *IXM* is much bigger than those of the other products since *IXM* is selected as the reference product in the reengineering process. In addition, there is no investment for *IXM-U* because the product was directly evolved from *IXM*. The total investment (adding up cells of Table V) is 4.2PY.

R-1 is the difference between the effort to develop new features in SPL reengineered and the estimated effort if the features were developed in the original product family. Its value is calculated through the following equation:

$$R\text{-}I = \sum_{i=1}^{M} \sum_{j=1}^{N} q_{ij} \times (OC_{ij} - NC_{ij})$$

where q_{ij} denotes the variant number in the j^{th} variation point in the i^{th} product. OC_{ij} means the new feature development effort on the corresponding components before the SPL reengineering, while NC_{ij} is the effort required on the variations after the reengineering has been performed.

TABLE V. RECORDED INVESTMENT (UNIT: PY)

Product	*IXM*	*IXM-MD*	*IXM-A*	*IXM-G*	*IXM-U*
I-1 to I-5	2.5	0.1	0.1	0.3	0
I-6	0.5	0.3	0.3	0.1	0

In the case study, there are two variant points in *EntityManagement*: entity types (VP1) and card types (VP2). The involved values in the equation are listed in Table VI. Note that *NC* metrics have not been listed, because the developing effort in SPL can be ignored (The number of modified code segments is greatly reduced as listed in Table III, and it only requires efforts at the Person-Day level). In particular, there was no extension point in products where *q* equals zero so that *OC* value is meaningless (N/A).

TABLE VI. EFFORTS ON VARIATION POINTS (UNIT: PY)

		IXM	*IXM-MD*	*IXM-A*	*IXM-G*	*IXM-U*
VP1	q	2	1	0	0	4
	OC	0.8	1.2	N/A	N/A	0.8
VP2	q	2	0	0	5	0
	OC	0.4	N/A	N/A	0.4	N/A

Therefore, the return is summed as 8.8PY. As a result, ROI of the reengineering of *IXM-PF* achieved 110%. (note: ROI is calculated as *(R-I)/I*, i.e. (8.8-4.2)/4.2=110%).

B. *Initial Investment for Reengineering*

In this reengineering project, the reengineering is triggered by the new feature development on a component that is assigned a high value. Therefore, the initial investment for the reengineering can be regarded as the extra reengineering effort spent on the first identified component identified in the reference product, e.g. *EntityManagement* in *IXM* in our case study.

From Table V, the reengineering effort for the component in *IXM* reaches 3PY. Furthermore, the effort for developing a new feature (a new card type implementation in VP2 of *IXM*) is 2.4PY (2×0.8+2×0.4 from Table VI). Based on this data, the initial investment is 0.6PY. As a result, the initial investment is increased by 25% (0.6PY/2.4PY) in such a large product family. This encourages the organization.

C. *Trend Analysis*

By analyzing the data as a whole, we observed a trend on the integration costs, the faults reported, and the profits obtained upon the products whose scope extends over time.

The integration effort has already been illustrated in Table V. Figure 7 illustrates the trend of the integration effort. Since the variability has been clarified in the first several products and the variants in the following products have been covered, the integration effort spent on the variability implementation was reduced over time.

The right part of Figure 7 denotes the faults reported and solved in the integration phase. Similar with the integration effort, since the faults related with the component variants were found in the first several products and no fault emerges if no new variation was introduced, the faults reported were also reduced over time.

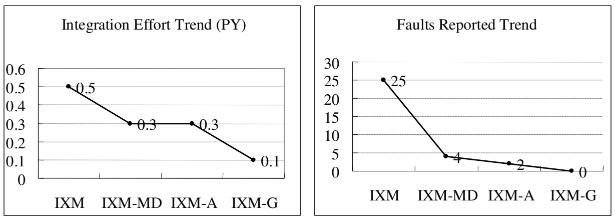

Figure 7. The trend of integration effort and faults reported

The profit obtained in each product is calculated using the data from Table V and Table VI. The profit trend is depicted in Figure 8. The figure illustrates the profit trend along with the products involved in the reengineering process. Figure 8 shows that the profit for each product as well as the accumulated profit increases rapidly. The first column identifies the initial effort of the SPL-oriented reengineering.

There are several interesting characteristics in Figure 8. First, we see that there is still initial investment (in the leftmost column) spent on the extension and integration of core assets for each product. In addition, we observe that the accumulated returns turn into positive from the second product. The reason is that the investment related to the integration and bug fixing is relatively larger in the first products as discussed in Section III.B. Therefore, the profits obtained increase rapidly over time.

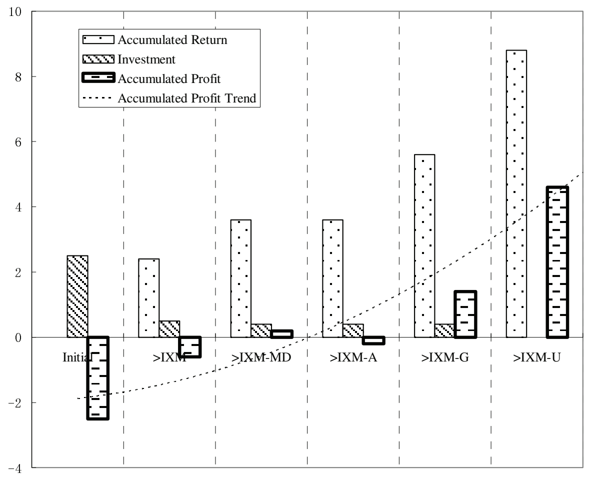

Figure 8. The trend of profit obtained

V. DISSCUSSION

A. *Relationship with Agile Development*

Our case study demonstrates an approach to reengineering an existing product family towards SPL incrementally, with a set of deployable practices to guarantee the steady reengineering process, and the early success and feedback. It followed the general Agile principles and adopted a few practices of Agile development such as refactoring and continuous integration. However, it does not require that the organization must transit towards Agile software development.

As pointing the core of our approach, we have integrated the increment and iteration practices of Agile [9] and Lean [19] software development to accelerate the feedback and to satisfy the customer requirements in an incremental way. Based on the iteration approach, we minimize the *Working In Progress* [19]. This also helps to reduce the initial investment and to generate the feedback earlier.

Since the increments are reengineered in a sequential order, it is necessary to prioritize them in a proper way. We define the priority rules by integrating the prioritization principle of user stories [17] as well as the value evaluation from value-based software engineering [16]. Thus, the increment definition is aligned with the business objectives where the components with the most benefits for the further development and maintenance will be assigned with high value and be reengineered preferentially.

B. *Generalization of Practices*

The guidelines summarized from the case study require neither organizational SPL strategy commitment nor the team structure transition. It relaxes the prerequisite of SPL adoption in comparison with other approaches. Since the incremental reengineering towards the selected components is aligned with the business objective, an increment can be reengineered along with the regular development or maintenance tasks. Furthermore, our reengineering process keeps the core asset engineering and the application engineering in the same context. It tries to remove the separation between the domain engineering and the application engineering. This practice enhances the *maximum communication* principle of Agile. Thus, both the reengineering and the integration can be done in one team.

In our case study, existing products in the same family share a similar architecture. It eases the identification of similar components. However, the reengineering process does not strictly require the full alignment of the responsibility or implementation, because the architecture and design will be evolved continuously.

C. *DSSA Unconcerned*

The process in this reengineering project does not depend on Domain Specific Software Architecture (DSSA) as the foundation to reengineer legacy products to SPL in an incremental way. Although DSSA helps to clarify the variability in the whole software systems at the granularity of component, our process works at a finer level where an architecture covering all the components is not necessary. Instead, we propose to use domain model for representing the variability in a component that will be shared across the products. Furthermore, we refine and address the variations in a component incrementally by comparing the component behavior of the reference product with that of the other variant products.

D. *Qualitive Analysis*

For different projects, the concrete profits obtained from the SPL reengineering tightly relates to the actual business context. The following key factors should be highlighted:

- The size of the selected increment: Smaller increment requires less initial investment.
- The associations between new requirement and selected component: Closer relations help to bring more benefit. For example in our case study, *IXM-A* integrated the new core assets, but there is no benefit achieved from the new feature that is not related with the reengineered component. It is not an ideal case certainly. In fact, such situation should be avoided from the viewpoint of value.
- The sequence of integration: Changing the sequences will impact the detailed revenue although the trends of accumulated profit were reserved.
- How many products and how long the observation was performed: In the case study, the team spent about one year to integrate five products. Generally, the longer the observation window last and the more the products are involved, the more benefits the reengineering process brings.

VI. RELATED WORK

The reported case study conforms to one of the basic ideas proposed by Schmid and Verlage [3]. In their paper, reengineering can either focus on packaging existing legacy system as a whole or it can aim at a component-wise approach. Our work follows the latter idea. However, our approach proposes the component-level increment for decreasing the initial investment that is not considered in [3]. In addition, we specify the activities that include a series of steps or guidelines in detail in the incremental and iterative process framework.

Kruger [10] explains three different transition models for adopting software mass customization: extractive, reactive and proactive models. It also introduces a software configuration tool, GEARS, to ease the transition. GEARS helps to consolidate legacy products into software product line. Compared with their work, we started with our investigation from the approach perspective rather than the tool perspective. Furthermore, we propose to reengineer components and integrate them into products incrementally, which forms a framework with detail steps and guidelines in the extractive reengineering.

There have been a lot of industrial case studies focusing on the SPL reengineering. Kang et al. [2] conduct a case study of home service robot system. Their work focuses on the recovery of feature model and architecture from existing assets, and their approach is used to guide assets reengineering. However, they do not address increment that is key characteristic in our approach. Ronny et al. report an industry case study in Ricoh [4]. Their work aims to refactor core assets for reuse when they migrate single system development to software product line paradigm. They apply the PulSE-DSSA method for improving the reusability of the core assets and the maintainability of the products. We take the similar consideration in our reengineering process. However, the detailed incremental activities proposed in our process are not considered in their work.

Buhrdorf et al. [18] also perform an industrial case study in Salion. The authors adopt the reactive software product line approach based on an initial product to meet dynamically changing market. They use GEARS to support the management of the addressed variability. Furthermore, the architecture or design artefacts of the initial product are retained and evolved as core assets. In contrast, our process deals with an application domain where several legacy products reside in. Furthermore, our approach emphasizes on the component increment and the product iteration that is more feasible for extractive reengineering. However, we share similar concerns to theirs such as the huge initial investment as a barrier for reengineering, and extracting domain assets by refactoring rather than predefining.

There are also works trying to combine the software product line engineering (SPLE) and Agile methodology. Hanssen et al discuss the process of the fusion of Agile and SPLE in the aspects of development strategy and product release [7]. Yaser Ghanam and Frank Maurer integrate Agile practices into SPL organizations [5][6]. In contrast, we believe that integration of Agile and SPLE is not the simple combination of both practices. We have applied and adapted the key principles from Agile [9] and lean [19], including *fast feedback* and *less working in progress* in many activities of the case study.

Mohan et al. [8] propose to integrate software product line engineering and agile development based on the theory of Complex Adaptive System. In their work, several important practices such as *control scope*, *refactor selectively*, *derive products incrementally*, *refactor for reuse*, *provide adequate autonomy* have been proposed. In contrast, our process is adopted in the area where variant products with similar architecture exist. In addition, we adopt the agile principles into our reengineering practices. As for some detailed principles, we follow *refactor for reuse* [8] to support the component reengineering. In addition, we apply automated component-level testing in order to enhance feedback, and value-based rules for the component and product selection.

VII. Conclusion

Incremental reengineering of legacy variant products towards SPL at the component level is often a practical way for reengineering-driven SPL adoption. However, how to properly define and carry out the increments in a steady, risk-reduced and cost-efficient manner still remains as a challenging problem to incremental SPL reengineering. In this paper, we demonstrate the experience of the team in one product family of Alcatel-Lucent. In the case study, we summarized an incremental and iterative process framework and a series of work steps and guidelines. Our case study shows that incremental and iterative approach with stakeholder-value considerations can help to achieve steady and successful SPL reengineering in a cost-effective manner. Furthermore, we also find that SPL adoption can be regarded as an emergent result of the reconstruction and improvement of existing product assets.

In the future, we will apply analysis of the case further from Value-Based Software Engineering (VBSE)[16] perspective to define the value formally. Furthermore, we will concentrate on developing automatic or semiautomatic techniques and tools to assist the understanding and refactoring activities involved in SPL reengineering.

Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant No. 60703092. Furthermore, the authors thank the *IXM* product family team at Alcatel-Lucent for the support during the research and evaluation of the case study.

References

[1] P. Clements and L. Northrop, Software Product Lines: Practice and Patterns, Addison-Wesley, 2001.

[2] K.C. Kang, et al., Feature-Oriented Re-engineering of Legacy Systems into Product Line Assets - a Case Study. In International Software Product Line Conference. 2005.

[3] K. Schmid and M. Verlage, The Economic Impact of Product Line Adoption and Evolution. IEEE Software 19(4), 50–57 (2002)

[4] R. Kolb et al., A Case Study in Refactoring a Legacy Component for Reuse in a Product Line. In the 21st IEEE International Conference on Software Maintenance (ICSM'05). 2005

[5] Y. Ghanam and F. Maurer. An Iterative Model for Agile Product Line Engineering. In The SPLC Doctoral Symposium. 2008

[6] Y. Ghanam et al., Reactive Variability Management Using Agile Software Development. In the international conference on Agile methods in software development. 2010.

[7] G.K. Hanssen and T.E. Faegri, Process Fusion: An Industrial Case Study on Agile Software Product Line Engineering. In Journal of Systems and Software, vol. 81, no. 6, 2008.

[8] K. Mohan, B. Ramesh, V. Sugumaran. Integrating Software Product Line Engineering and Agile Development, In IEEE Software, 2010 May/June

[9] Manifesto for Agile Software Development. http://www.agilemanifesto.org/

[10] C. Kruger, Easing the Transation to Software Mass Customization. In the 4th International Workshop on Product Family Engineering. 2002

[11] E. Evans, Domain-Driven Design - Tackling Complexity in the Heart of Software, Addison-Wesley, 2004.

[12] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[13] R.C. Martin, Agile Software Development: Principles; Patterns and Practices, Prentice Hall, 2003.

[14] M. Folwer, Continuous Integration. http://martinfowler.com/articles/continuousIntegration.html

[15] Y. Pengfei et al. A Case Study of Variation Mechanism in an Industrial Product Line. In the 11th International Conference on Software Reuse. 2009

[16] B. Boehm, A. Jain, An Initial Theory of Value-Based Software Engineering. In Value-Based Software Engineering, Springer Verlag, 2005

[17] M. Cohn, Agile Estimating and Planning. Prentice Hall, 2005.

[18] R. Buhrdorf, D. Churchett, C.W. Krueger. Salion's Experience with a Reactive Software Product Line Approach. 5th International Workshop on Software Product-Family Engineering, 2003.

[19] M.Poppendieck, T.Poppendieck. Lean Software Development: An Agile Toolkit. Addison-Wesley. 2003