An Iterative Model for Agile Product Line Engineering

Yaser Ghanam
University of Calgary
yghanam@ucalgary.ca

Frank Maurer
University of Calgary
frank.maurer@ucalgary.ca

Abstract

Agile software development (ASD) and software product line engineering (SPLE) seem to be two rewarding yet disparate schools of thoughts in software engineering. ASD encourages strong business involvement in development activities, focuses only on the requirements at hand, and deems huge investment in requirement and design upfront unjustifiable. On the other hand, SPLE considers intensive domain analysis and flexible & detailed software design as prerequisites to any development effort. SPLE plans for potential future projects, and dedicates considerable resources for preplanning efforts. Integrating ASD and SPLE, although is challenging, has a huge potential of magnifying enhancements in quality, cuts in cost and reductions in time-to-market. In this paper, we present our research on this integration. We propose a model that enables agile organizations to establish product lines without disturbing the agility of their practices. The model is a bottom-up application-driven approach that relies on automated tests to derive core assets from existing code.

1. Introduction

Agile software engineering is a collection of methodologies that, according to the Agile Manifesto [1], give customer involvement and satisfaction the highest priority. Agile practitioners preach an iterative development approach that encourages values and practices such as stakeholder communication, early feedback from customer, test-driven development, short iterations, just-in-time design and continuous integration. The field of software engineering has matured enough to realize that getting the customer requirements right is key to the success of any software project. This is why traditional software engineering approaches invest so much time at the beginning of the project life cycle to elicit these requirements, clarify any vagueness around them, document them and produce designs that attempt to satisfy them.

On the other hand, given the high level of uncertainty of customer requirement at the beginning of the project, agile methods discourage large investments in upfront analysis and design. Big-design-up-front (BDUF) is seen by many agile practitioners as the antithesis of agility. Agile methods tackle requirements in a different manner. While a project vision and rough scope are usually developed by agile teams, they do not spend more than a few weeks on this effort before iterative development starts. Detailed requirements are only determined during development iterations and only for features that are part of this increment. Requirements are elicited from the customer in the form of user stories and made more concrete by defining acceptance tests [2]. A short development iteration (two to four weeks) implements these user stories and produces a working version of the system. At the end of each iteration, the customer gets the final say on how well those requirements are satisfied and what needs to be done in the next iteration. The architecture of the system evolves gradually bottom-up as the project needs become clearer. Design decisions are agreed upon by the members of the development team who talk to each other in their daily scrum meetings. Agile methods demonstrate an outstanding flexibility to accommodate change requests. This responsiveness to change, although might be expensive, can still be more economically justified (by the not-so-much investment upfront in predicting future changes) than other models which deem any change request during the implementation phase very expensive (because of the so-much investment upfront).

While scientific data on agile methods is not yet conclusive, they seem to work well according to a growing number of case studies, experience reports and controlled experiments investigating individual agile practices (e.g. business organizations are reporting success in adopting agile practices like Test Driven Development [3]).

Another successful practice in industry is software product line engineering. By planning for families of products as opposed to single products, SPLE offers opportunities for cost minimization, time reduction, and quality improvement. This is achieved through emphasizing flexibility of the reference architecture of the product line and focusing on the reusability of development artifacts. According to [4], the ultimate goal of product line engineering is to enable organizations to produce systems of higher quality, with less cost and in shorter time. Nonetheless, for a software product line to enjoy success, current approaches require a huge amount of effort to be invested upfront in domain analysis, application analysis, flexible architectural design, documentation and core asset development. Interestingly enough, agile methods, on a high level, aim to achieve the same goals (higher quality, shorter time, less cost), while not spending any extra time or money in early stages where uncertainty is high. SPLE seems to focus on improving organizational efficiencies while ASD focuses on increasing the effectiveness of individual teams.

Since pivotal aspects of SPLE sound off-putting to the agile community, there is a need to investigate whether and how ASD and SPLE can work together to achieve their common goals. *Our research goal is to build a bridge between ASD and SPLE to combine the advantageous characteristics of both. Our ultimate objective is to come up with a model that enables agile organizations to establish software product lines without affecting the agility of their practices.*

The rest of this paper will be structured as follows. Section 2 is a literature review on this research topic. Section 3 talks about why it is challenging to make agile methods capable of establishing product lines. Section 4 discusses our proposed model to bridge agile and SPLE. Section 5 discusses the methodologies we are using to study the issue. Section 6 talks about possible ways to evaluate our work. Section 7 summarizes our research motivation, goals and the progress made so far and gives a glance on the contribution of this ongoing research.

2. Related Work

Literature on SPLE in general is extensive. With specialized conferences like Software Product Line Engineering, efforts to enhance the practice are numerous. Most of these efforts seem to rely on common bases cited from books like “Software Product Lines - Practices and Patterns” [4]. But when we look for literature on agile product line engineering, one can barely find a paper dedicated to that topic, or a paper that has “agile” and “product line” in its title. At SPLC 2006, an encouraging initiative was the 1st international workshop on agile product line engineering [5]. The workshop aimed to bring practitioners from the agile community and the SPLE community to discuss commonalities and points of variation between the two practices. The theme of the discussions in that workshop was around how feasible it is to integrate the two approaches. In conjunction with this workshop, one of the presented efforts was the iterative approach proposed by Carbon et al. [6]. This approach is based on PuLSE-I [7] which is a reuse-centric application engineering process. The proposed approach gives agile methods the role of tailoring a product for a specific customer during the application engineering process. While an interesting first step, this effort does not consider the introduction of the product line practice in an agile organization in a way that does not disturb agility nor does it discuss how the product line architecture is derived in the first place.

Also, Paige et al. [8] proposed building software product lines using Feature Driven Development. They assert the method worked well when giving special considerations for the product line architectural and component design. Another noticeable effort was by Hanssen et al. [9] who presented a success story of integrating SPLE and ASD. In this experience, SPLE was used for long-term planning at the strategic level of the organization. On the other hand, ASD was used at a medium-term project level to serve tactical processes.

3. Issues

This section discusses some of the issues that agile methods will have to address in order to be able to integrate product line engineering approaches.

3.1 Requirement analysis

In sequential software engineering models (like Royce’s Waterfall model), it is essential to conduct domain or/and application analysis prior to any design or implementation in order to get the right requirement specifications. This is why in traditional software engineering models, practitioners advise that projects allocate a high amount of resources to requirement engineering [10]. When we talk about developing a family of products, investment in domain analysis becomes even more crucial and resource demanding. This is because it is no longer sufficient to analyze the project at hand; but there is a need to plan for future projects that are potential members of the product family. Such a long-term planning requires deep insights into future market opportunities and is surrounded by a high level of uncertainty.

This issue is a real challenge for agile methods that deem huge investments in domain analysis economically unjustifiable because of the high level of vagueness at the beginning of the project and the uncertainties of where the market will be going over the next years. Agile methods encourage focusing on immediate needs that can be delivered to the customer at the end of a development iteration. This approach limits uncertainty (as forecasting what is needed in a few weeks is easier than predicting what might be useful a few years down the road) and increases net present value (as payback for an investment will start to come within weeks of development effort). Introducing a new phase where lengthy domain analysis has to take place before starting any implementation to address current problems would affect the agility of the practice and, thus, might not be appealing to agile practitioners.

Another issue with domain analysis is the detailed documentation of the findings. While it is of high importance for existing product line models to produce formal requirement documents as a reference for future projects in the same domain, agile methods do not provide documentation unless it is essential to satisfy a customer's need. Rather, documentation of agile-produced systems often consist mainly of the code itself accompanied by acceptance and unit tests as well as high-level overview documents.

3.2 Preplanned reuse

Agile methods develop systems based on current needs and not based on predictions of what is going to be useful in the future: current needs are concrete, future needs are more like options. Detailed planning includes only a small set of clear requirements that are requested directly by the customer and discussed amongst the different stakeholders. After this set of requirements has been satisfied (accepted by the customer), another set is to be implemented taking into consideration all available information. No time is allocated at the beginning of the project to produce detailed requirement documents given their frequent changes. According to [11], it is not sufficient to tailor and reuse artifacts whenever they are needed to satisfy a product line requirement of reuse. Reuse has to be preplanned. There has to be a specific set of artifacts that are to be reused. Special considerations in the design and implementation are given to these core assets. Designing these core assets requires knowledge of what the architecture of the system will encompass, and thus, this has to be done after the architecture has been defined.

In agile methods, however, the architecture is supposed to evolve over time. And because of this expected evolution, it makes no sense to produce and continuously spend time and effort in updating lengthy design documents for an architecture that is continuously changing.

Giving up investment in requirement and design upfront does not come without a cost. Starting the development process focused on concrete needs likely results in producing artifacts (acceptance tests, design modules, pieces of code … etc) that are application-specific and hard to reuse for other applications. Not producing reusable assets effectively costs nothing when developing single systems. In fact, it saves money as SPLE literature indicates that it takes 2-3 product development efforts to recover the initial investment into the product line approach. But opportunity costs become high when thinking about developing families of products where reuse is pivotal.

3.3 Core asset management

Needless to say, managing available core assets is an essential success factor in software product lines. It is not sufficient to develop these assets and make sure they fit within the platform of a specific product line. Once the product line starts to mature, the number of core assets also starts to increase. Therefore, there should be formal mechanisms through which core assets can be described and identified for reuse and maintenance purposes. It should also be possible to trace these assets across the different phases in the product line so that a quality enhancement of a given asset can regressively affect all instances of this asset in the product line.

These issues have been already addressed in traditional SPLE approaches through detailed documentation and means of tracking and associating artifacts at different variations of the product line. However, this is a challenge for agile methods that, again, do not value spending time on producing detailed requirement and design documents. So the question is whether it is possible to keep track of core assets and manage them with practices compatible to agile approaches.

3.4 Flexible Design

To make customization possible in a software product line, flexibility is to be introduced at two different levels. One level is the reusable artifacts. These artifacts need to be flexible enough to be plugged into or/and interfaced with different members in the family. This consequently implies the other level at which flexibility is to be introduced which is the system architecture. The architecture needs to be flexible to accommodate variations in the product line. Variation points are to be defined in order to incorporate potential variants into the architectural design. Defining variability in advance is essential to the success of any product line practice and helps define the scope of the product line.

Putting a flexible system architecture in shape before starting any implementation seems to be a reasonable, or more precisely expectable, requirement in traditional software models that finalize the architectural design before the implementation phase starts. But for agile teams, the story is different. In agile practices, the architecture is derived bottom up from concrete requirements and evolves over time. The question is: how can we create variation points in the product line if the architecture is not to be worked on prior to the implementation?

4. The Proposed Model

4.1. Overview

Our proposed model to integrate agile methods with SPLE takes into consideration the issues mentioned in the previous section. On a high level, this model differs from other proposed models in a number of ways. For one, as opposed to other models like the one in [6] and [9] that attempt to utilize agile methods as an enabling development model within SPLE, the model we are presenting here emphasizes agile as the key player within which SPLE techniques will be integrated at the enterprise level. Differently put, while previous works target SPL-based organizations that want to make their product development more agile, our model targets agile organizations that wish to establish an SPL.

Secondly, current SPLE models depend heavily on platform requirement and architectural design as prerequisites to instantiating product instances. In this model, however, we adopt a bottom-up approach through which the product line is built iteratively from existing product instances. The platform will evolve progressively in an iterative approach at the project level.

4.2. The Corner Stone of our Approach

It is true that agile software development values working software over comprehensive documentation [1]. But this does not mean that agile-produced artifacts are untraceable. Assuming a healthy practice of agile methodologies, test-driven development necessitates that all feature development be driven by acceptance tests defined by the business stakeholders. We look at acceptance tests as the corner stone of the bridge between agile methods and product line engineering. Acceptance tests are core assets that will be reused and will, consequently, drive reuse of development artifacts in the product line. The rest of this section explains what an acceptance test is and how it is produced.

Usually in the planning meeting, the customer defines a set of user stories that are effectively a subset of the system requirements. These requirements are translated (by the customer and other stakeholders) into human-readable test cases that define acceptance criteria for the feature in form of examples [12]. These test cases are called acceptance tests and are usually organized in a tabular format. Figure 1 is a simple example of an acceptance test for a course registration system.

ID	3
Description	Adding an offered course.
Preconditions	CPSC 688 and SENG 615 are offered only in Fall08.
Operation	a. Add CPSC 688 in Fall08 b. Add SENG 615 in Spring09
Expected outcome	a. Course is added successfully. b. Course is not added. Message: “SENG 615 is not offered in Spring 09”

Figure 1 - Example of an acceptance test

Using frameworks like Fit or GreenPepper [13], the development team automates these tests before they start the actual coding of the intended features. These tests will initially fail. With progress being made in the development process, test cases start to gradually pass. Ideally, when a new developer joins the team and is asked to enhance a certain feature, the associated acceptance test should be sufficient to understand what the objective of the feature is.

4.3. The Iterative Model

Let’s assume an organization is developing two independent systems; yet these systems are in the same or very similar domain. Each of the systems has a number of developers who, in the worst case scenario, do not communicate. Development of both systems is going on in parallel and is following a test-driven development approach. As shown in Figure 2, at the end of the development stage, we expect the following to be available for each of the systems: a set of acceptance tests, an implemented architecture, and a set of code modules.

The organization realizes that more systems in the same domain will be in demand soon and decides to work on establishing a product line. System C is the first system to benefit from the product line approach.

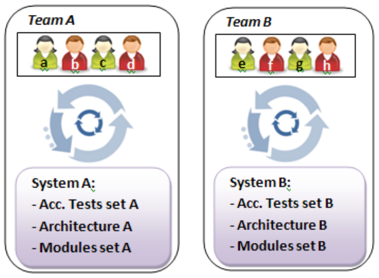

Figure 2 - Development of two independent systems

4.3.1. Core Assets Team. Prior to starting the development of system C, the first step would be to form a team of developers who will be responsible for creating and maintaining core assets.

The responsibilities of this team include mining the existing systems A and B (on demand) for modules that can be reused in products under development, extract a generic layer of these modules, and define variation points and variants to these generic artifacts. The core asset team also helps the product teams to refactor their products to use the core assets.

The mining process will depend primarily on acceptance tests that are associated with code modules (or features). This process requires that the members of this team are familiar with existing systems in the organization. Thus, the core asset team will consist of senior developers from Team A, B and C. This also ensures that the core asset team is grounded in the actual needs of the teams that will use its work results.

4.3.2. Evaluation & Extraction. When the new team starts the development of system C, they first obtain a set of user stories from the customer. These user stories are converted to acceptance tests. The acceptance tests are discussed with the core asset team. The evaluation process entails finding similarities between the acceptance tests at hand and those that already exist from the previously developed systems. If the team decides the level of similarity is above a certain threshold value alpha (α), then a match is found and the tailoring process follows[1].

The tailoring process follows and encompasses more sophisticated procedures to refactor acceptance tests and produce a final artifact that has two layers: a generic layer and a variability layer. In this process, all relevant acceptance tests from previous systems (say $A.T_{1A}$, $A.T_{1B}$) in addition to the acceptance test in hand (say $A.T_{1C}$) are used to extract an acceptance test $A.T_1$\` that has a generic fixed component $A.T_{1G}$, a variable component $A.T_{1V}$, a variation point such as X, and a number of variants $A.T_{1A}$\`, $A.T_{1B}$\` and $A.T_{1C}$\` as shown in Figure 3. For example, $A.T_{1G}$ defines a generic acceptance test for a door locking system. A variation point X would be the print type with three different variants: $A.T_{1A}$\` (finger print), $A.T_{1B}$\` (voice print) and $A.T_{1C}$\` (eye print).

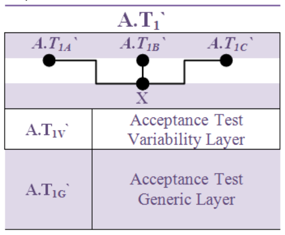
A.T1`
A.T1A`
A.T1B`
A.T1C`
X
A.T1V`
Acceptance Test Variability Layer
A.T1G`
Acceptance Test Generic Layer

Figure 3 - $A.T_1$\` consists of two layers

The extraction process of the generic A.T is conducted in three steps:

1) Define the intersection of all relevant acceptance tests to come up with a generic layer.
2) Specify how and why non-intersecting parts differ to define variation points.
3) Specify how acceptance tests in hand relate to the newly specified variation points to specify variants.

4.3.3. Refactoring. Once the acceptance test model has been defined as in Figure 3, the refactoring process is conducted. We mentioned before that every feature (user story) is developed against a prewritten test that defines acceptance criteria. This is why the refactoring process will be steered by the newly generated acceptance test model. That is, test cases that have been classified within the generic A.T will drive the generation of a generic module (M_G), whereas those that have been categorized within the variability layer will drive the generation of specialized modules (M_V). Figure 4 illustrates the anticipated object model resulting from the direct mapping of the acceptance test model ($A.T_1$\`in Figure 3) to a code module (M1). While it is the responsibility of the core assets team to produce the generic layer of this module along with interfacing points, the generation of the variability layer will be mainly the responsibility of the development teams.

[1] The value α is a predefined percentage at which it is more economical to reuse than develop from scratch. Initially, α can start low (say 30% - 3 out of each ten cases already exist) to give the product line an opportunity to grow. Later α can be increased gradually as the scope of the product line gets more restricted. If the level of similarity between what is required and what already exists does not exceed alpha, then the request cannot be honored, and it would be the responsibility of the requesting team to develop that specific module.

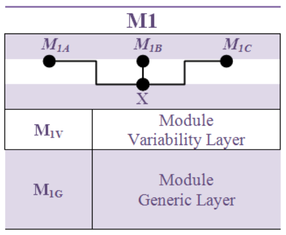

Figure 4 - Refactoring process produces Module 1

4.3.4. Managing Core Assets. Once the definition of the new module has been finalized and the refactoring process to build this module from existing code has been achieved, the core assets team adds the refactored module as a self-contained component (class, interface or package) along with all associated variants into a repository of reusable modules for future uses. Reusable modules in the core asset repository are referenced by their corresponding acceptance tests. In early stages, these acceptance tests can be looked at manually, but with the number of assets increasing over time, there might be a need to develop an automated search mechanism. Using automated acceptance tests as a core asset provides solutions for a number of issues:

1. *Documentation*: with the lack of traditional documentation of the current status of the product line, there needs to be an alternative to make it possible for newcomers to understand what is already there, what needs to be maintained, and what needs to be produced.
2. *Traceability*: whenever an application instance makes use of existing modules, the corresponding acceptance tests in the repository links this usage to the reusable component. This tracking is done so that when core asset in the repository is modified (e.g. to fix a bug), all application instances can be traced back and, hence, can be notified of this change.
3. *Maintainability:* when receiving new reuse requests from future projects, new information about the domain might be available. This will affect already existing reusable artifacts. They will be continuously maintained so their flexibility and fitness for future reuse increases every time. Maintaining an artifact includes different options such as expanding the generic layer, defining new variation points or identifying new variants. As more requests are made to the repository of reusable components, these components get polished until they reach a stage where they become pluggable without further modifications.

4.3.5. Core Asset Incorporation. The development efforts to achieve the model in Figure 5 will mainly be spent by the core assets team with a possible (and encouraged) cooperation of the development team who originally initiated the request. This cooperation already exists considering the participation of team members who belong to the development team of System C. However, more communication is always encouraged in agile settings for everyone to be up-to-date on the progress.

The resulting module is streamed back to be incorporated in the system under development while other requests (possibly from different teams) are progressing in the same process.

4.3.6. Architecture Evolution. The process described above consists of a number of steps including: forming the core assets team, evaluating reuse requests, refactoring existing code, adding newly developed modules to the repository, and finally incorporating the new module into the system under development. This multi-step process is to be conducted iteratively at the project level. That is, every project in the same domain will initiate requests to the core assets team asking for reusable artifacts. Handling schedules of different teams involved in the product development using a configuration management system is an open research question. As more projects are developed, the number of reusable artifacts increases resulting in a higher level of reuse and refinement of pluggable components. As more components in the architecture become stable (API less frequently changed), the generic platform will consequently stabilize and the variability layer will gradually separate. To illustrate, look at Figure 5 and notice how the two architectural levels of the product line evolve as more projects are realized.

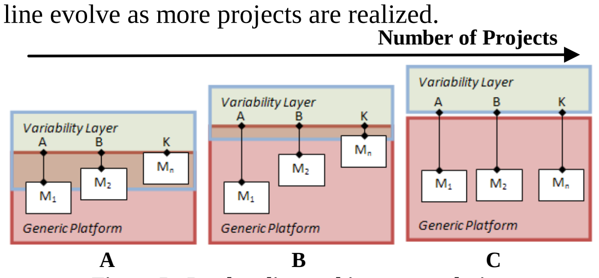

Figure 5 - Product line architecture evolution

At point A, the product line is still in the first stages where existing modules (M_1 and M_2) are continuously experiencing considerable alterations and new reusable modules are introduced (M_n). At this point, there is high coupling between the variability layer and the generic framework.

At point B, we start to notice signs of maturity since existing modules are starting to move towards the generic framework (i.e. experiencing less alteration). Also, at this point the interfaces of repeatedly used modules are getting more polished and thus the variability layer is getting thinner as the generic modules start to directly provide the required functionality to the applications.

When the product line reaches point C, it starts to stabilize since most of the modules now belong entirely to the generic framework with a very thin layer of variability containing instances of variation.

It is true that in this iterative approach, the first few projects that feed into the iterative process may not economically benefit from the product line practice. But according to [11], even in traditional product line models, at least three software systems need to be developed before the break-even point is realized (getting back the upfront investment). Following this bottom up approach initially avoids the upfront cost (and the associated risk) of the traditional SPL approach. The first two to three systems developed by our approach will (combined) not cost more than the first two to three systems using traditional SPL. The company can start the SPL initiative with much less risk as it already has the experience from building a few similar systems. It has learned a lot about the domain during development and is now more able to predict where the market is going. It avoids investing into reusable assets that nobody wants to use (as it does not have to anticipate what will be used but simply respond to reuse requests from application development teams).

5. Methodology

We will combine empirical studies in industry with a practical implementation of the proposed agile product line approach with our own development team. To begin, we will look carefully into case studies of agile-based organizations that do development of systems in a specific domain. These organizations may not claim to have a product line, but when we look at the development processes of some of our industrial contacts, we can easily see core asset development activities in some of the larger agile teams.

Furthermore, to be able to study the feasibility and practicality of the proposed model, we decided to actually implement a small-scale product line for a specific application domain. The domain we picked in collaboration with one of our industrial sponsors is software for monitoring and controlling intelligent homes. These systems will be designed for use on touch sensitive displays and digital tabletops. Establishing a product line in such an application domain makes an excellent case study for our research. For one, the field of tabletop applications is a fairly new field that is increasingly emerging as a future core technology. This implies that a learning curve is to take place before the development teams become familiar with the domain. This is an advantage since we are interested in knowing how the learning curve (when introducing a new domain) will affect the effort of establishing a product line. The other advantage this application domain offers is the numerous opportunities for customization (which we see as a precondition for establishing a product line).

Members of the development teams are interns with good background about software engineering in general and agile practices in particular. They will be working in teams in a software engineering lab in settings that encourage face-to-face communication and facilitate daily scrum meetings. This lab will also have a digital tabletop for testing as well as a small simulation environment for the intelligent home.

We will use ideas from action research and implement the iterative model for establishing the product line by giving direct guidance to the developers in our team. Team members will be involved in release & iteration planning with our industrial partners. They will also be involved in the decision making process at every stage of the development. Their performance will be closely monitored and their output will be evaluated by the customer for quality. These measures are taken in order to make the environment as real as possible so that we can get precise feedback on the model. Through daily scrum meetings and informal discussions with the developers, we aim to have a good idea on how well the model is working. More specifically, we are interested in knowing what factors affect the effectiveness of the core assets team, whether there are any communication issues between the development teams and the core assets team, and whether there are any coordination or synchronization problems between teams working in parallel on different projects. We will also monitor and pursue feedback on the generation of the reusable artifacts, their quality, their evolution and their flexibility. Moreover, the evolution of the architecture itself across different projects in the product line is one of the major foci of our study.

6. Evaluation

The results of the study shown in the previous section will be incorporated in a refinement process of the proposed model. After that, we will evaluate the model through two different phases.

6.1. Hypothetical Systems

Since we have industrial partners who are interested in the applications we will be developing for intelligent homes, these partners can be a valuable source of realistic requirement specifications for a few hypothetical systems. The requirements of these systems are to be evaluated against the capabilities of the product line that was established during our study. For each of these systems, if 1) the product line is capable of accommodating a certain percentage of the requirements through existing reusable artifacts, and 2) the architecture shows flexibility to interface with newly developed artifacts, then our model has successfully produced a product line that has a business value. If one of these two conditions was not met, further investigation is to be done to identify sources of problems before proceeding to the next step.

6.2. Empirical Evaluation

When the first evaluation phase has been successfully accomplished, we will be looking for industrial partners who are willing to validate the model by implementing it in their business organizations. Getting partners who are willing to risk their time and man power to validate the model is a difficult task. But once the request is honored, feedback from these organizations will significantly contribute to the enhancement of this model.

7. Conclusions: Progress & Contributions

In this paper we presented an iterative, acceptance test driven model for agile product line engineering. The model aims for a seamless integration between ASD and SPLE. It follows a bottom-up approach where reusable artifacts are extracted from existing assets on demand. The corner stone of our model is the use of automated acceptance tests for replacing traditional documentation for SPL purposes. Currently, we are in the initial stages of our research where the model is being continuously revised and refined.

One of the issues that we need to study more is whether test-driven-development is the most effective agile method in achieving the objectives of the proposed iterative model. Many other questions remain open. We would like to answer questions like: would it be more practical to rely on unit tests as opposed to acceptance tests or maybe both? How can these tests be written for reuse? What is the best way to track and maintain core assets across different application instances? How do we control the scope of the product line so that it does not go out of control? What are the financial implications of a bottom-up product line approach in an agile context, and how does it compare to more traditional SPL approaches? Also, a major question is how to improve communication amongst different teams working on different projects in the same product line?

We strongly believe that investigating this topic will have a significant impact on both research in software engineering and the software industry. Not only will the integration of ASD and SPLE provide substantial benefits to software development organizations, it will also open the doors for new research areas in the field of agile software engineering.

8. References

[1] “Manifesto for Agile Software Development,” available at http://agilemanifesto.org. Last accessed April 20th, 2008.

[2] Reppert, T., “Don’t Just Break Software, Make Software: How Story-Test-Driven-Development is Changing the Way QA, Customers, and Developers Work”, *Better Software*, 6(6): 18–23, 2004.

[3] Melnik, G., Jeffries, R., “Test-Driven Development – The Art of Fearless Programming”, *IEEE Software*, 24(3): 24-30, 2007.

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

[5] Cooper, K., and Franch, X., "APLE 1st International Workshop on Agile Product Line Engineering", *SPLC, 2006.*

[6] Carbon, R., Lindvall, M., Muthig, D., and Costa, P., “Integrating product line engineering and agile methods: flexible design up-front vs. incremental design”, *The 1st International Workshop on APLE, 2006* - SPLC.

[7] Bayer, J., Gacek, C., Muthig, D., and Widen, T., “PuLSE-I: Deriving Instances from a Product Line Infrastructure”, *Proceedings of the 7th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2000*, pp. 237 - 245.

[8] Paige, R., Xiaochen, W., Stephenson, Z., and Phillip J., “Towards an Agile Process for Building Software Product Lines”, *LNCS: XP 2006*, pp. 198 – 199.

[9] Hanssen, G., and Fægri, T., "Process Fusion: An Industrial Case Study on Agile Software Product Line Engineering", accepted for special Issue of *Journal of Systems and Software (JSS)*, 2008.

[10] Hofmann, H., Lehner, F., "Requirements engineering as a success factor in software projects", *IEEE Software,* 18(4), pp.58-66, 2001.

[11] Pohl, K., Böckle, G., and Linden, F, *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, Germany, 2005.

[12] Perry, W., *Effective Methods for Software Testing*, John Wiley & Sons, New York, 2000.

[13] GreenPepper Software, www.greenpeppersoftware .com. Last accessed May 1st, 2008.