Integrating Product Line Engineering and Agile Methods: Flexible Design Up-front vs. Incremental Design

Ralf Carbon[1], Mikael Lindvall[2], Dirk Muthig[1], Patricia Costa[2]
[1]*Fraunhofer Institute for Experimental Software Engineering (IESE)*
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
{carbon, muthig}@iese.fraunhofer.de
[2]*Fraunhofer Center for Experimental Software Engineering, Maryland (FC-MD)*
{mlindvall, pcosta}@fc-md.umd.edu

Abstract

Today's market expectations force organizations to invent and provide new products in short time and thus to speed up their product and software development. On the one hand, Product Line Engineering (PLE) is a promising approach that is believed to shorten time to market and increase high quality of products. On the other hand, agile methods aim at reducing time to market and increasing quality. However, different philosophies underlie PLE and agile methods. PLE requires flexible, up-front design to set-up a reference architecture for families of products. Agile methods propose a simple, incremental design that only designs for the product at hand. Nevertheless, combining PLE and agile methods is a must to further increase both time to market and quality of products. In this paper we present an approach how to combine PLE and agile methods. To better understand the effects of a combination of PLE and agile methods, we performed an experiment. We also investigate the Return on investment (ROI) of the activities related to PLE and agile methods.

1. Introduction

Today's market expectations in domains such as mobile devices, digital cameras, and office devices force organizations to invent and provide new products in short time and to speed up their product development. Software is the crucial part in most of such products. Hence, Software Engineering has to cope with such challenges.

On the one hand, Product Line Engineering (PLE) [5];[12] has shown to shorten time to market and guarantee high quality of products [5]. The key concept of PLE is proactive, strategic reuse. On the other hand, agile methods like Extreme Programming, Scrum, Feature-driven Development [4];[8], and the agile principles they are based on [9] have shown in several success stories that they can reduce time to market and increase quality [10].

The underlying philosophies of PLE and agile methods differ, for instance, the design strategies: PLE requires flexible, up-front design to set-up a reference architecture for a family of products. Agile methods propose a simple, incremental design that only designs for the product at hand. The focus of PLE is to build an architecture that is flexible in respect to a class of anticipated changes, but real-world experiences show that it is not feasible (neither from a technical nor a economical standpoint) to build-in flexibility for all possible anticipated changes. A combined PLE approach would thus build-in flexibility for the most probable anticipated changes (proactive), while being agile enough to quickly incorporate changes that were not anticipated and for no flexibility was built in (reactive).

We strongly believe that the two approaches must be combined. PLE as a proactive, strategic reuse approach forms the basis to develop new products in less time and with higher quality because they can rely on proven parts. Agile methods are then used in application engineering to perform the customization or calibration of a product for a specific customer.

The use of agile methods in a product line context has to be evaluated. We especially need to understand the return on investment (ROI) for various activities. As a first step, we conducted an experiment where one half of the subjects applied incremental design for the implementation of five new features, and the other half

applied flexible design. Both groups started with a flexible reference architecture typical for a product line context. The results from this experiment have not yet been analyzed in detail, but we believe the experimental design in itself has value and is interesting for this workshop. The intention of this paper is to discuss our proposed framework for how to combine PLE and agile methods. We would like to use the results from the workshop discussions to further improve our combined approach and the analysis of the current results and to be able to design and conduct additional experiments to guide the improvement of the suggested combined approach.
The remainder of this paper is structured as follows. In Section 2 we give our definition of agility. Then we introduce PLE and agile methods and point out their contributions to agility in the light of our definition in Section 3. In Section 4 we show the integration of agile practices and principles in the PLE processes by means of several examples. Section 5 gives an overview of our experiment on flexible design up-front vs. incremental design. We also define ROI calculations for a series of change requests. We conclude with a summary and an outlook on future work in Section 6.

2. Agility beyond Agile Methods

In Software Engineering the notion of agility is often equaled to applying agile methods like Extreme Programming, Scrum, etc. But dictionaries define agility as “the capability to move with surprising ease and speed.” Thus, agility is an external property of an organization. Organizations are seeking to become agile because of customers’ demand and general market pressure. If an organization is not agile it cannot prosper in today’s competitive market environment. The fact that agility is related to “moving” implies that customers must get a sense that the work on producing software artifact is moving forward. Processes make artifacts move, i.e. people of an organization apply processes and produce or evolve artifacts thus moving them closer and closer to the customer. If the processes people of an organization apply are capable of producing artifacts with surprising ease and speed then the people and the whole organization appear agile to the customers.
In Figure 1, we further detail our definition of agility. We associate the ease of moving with the quality of produced or evolved artifacts and the speed with the time to market of the end product. The most important goal for an agile organization is to minimize the time it takes its products to reach the market, but a short time to market is only valuable if the product meets the markets ever changing quality needs both in terms of functional and non-functional requirements. The key is the organizations’ processes since they enable the agility of an organization, i.e. the agility is visible in an organization’s processes. We distinguish between engineering processes and management processes in Software Engineering. Engineering processes transform the requirements via several intermediate products to an executable system. They are the primary source of the quality of the development artifacts. In an optimal case, they increase the quality of development artifacts. Management processes steer a project and keep it on track. They are the primary source for efficiency or short times to market.

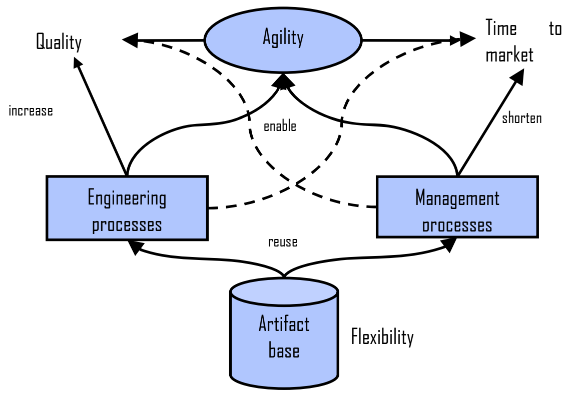

Figure 1: Illustrating the Definition of Agility

Figure 1 also illustrates the relationship of agility and flexibility. Reuse of artifacts from similar projects has proven to be a promising solution to increase the quality of new products and to shorten their time to market. Thus, reuse can directly be related to agility and must be considered as a key success factor towards agility.

3. Introduction of PLE and Agile Methods

3.1. Product Line Engineering

Product line engineering splits the overall development activities into two main parts (Figure 2).
On the one hand, there is application engineering (AE) that develops particular products as requested by customers; on the other hand, there is family engineering (FE) that designs the product family (i.e., creates a product-line-specific reuse infrastructure) after identifying its common characteristics and predicted variability that make a product line.
This identification is typically conducted through scoping activities that systematically define an

economically useful scope for a product line. The quality of the scope definition is crucial for the ultimate success of a product line. Success thereby means that the reuse infrastructure covers most of the features requested by application engineering projects and thus most customer requirements can be fulfilled through reuse. In several cases, more than 90% of the applications were reused [2]. Consequently, development that is based on a reuse infrastructure is much faster than the development of products that are developed from scratch, i.e. the time to market of new products is significantly decreased.

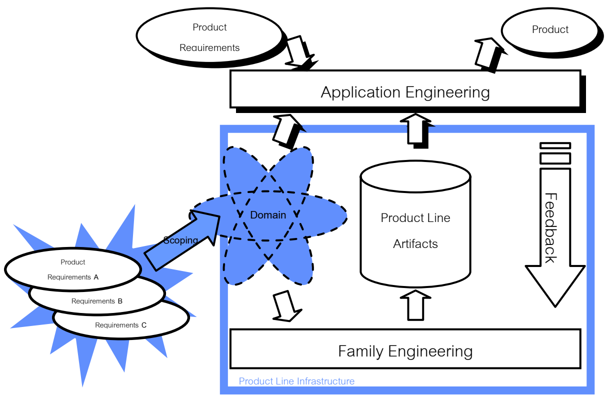

Figure 2: Product Line Life-Cycle Model

However, PLE requires an initial up-front investment to set-up a product line infrastructure with reusable, flexible artifacts. The key artifact for successful PLE is a flexible reference architecture. Additional effort is required to maintain the flexibility of the product line artifacts over time since it has a tendency to degenerate as an effect of the changes it undergoes over time. A question for developers of such infrastructures that never will completely disappear is whether the right change scenarios have been anticipated and whether the maintenance of the flexibility will pay off in the future.

3.2. Agile Methods

According to the Agile Manifesto [9] it is of highest priority to satisfy the customer through early and continuous delivery of valuable software. The key concepts in agile methods to quickly develop running software and shorten time to market are short, time-boxed iterations, short feedback cycles within the development team and with the customer, continuous integration, and automated regression tests. The development is code-centric and does not invest in design up-front, instead the design emerges from the code. Developer documentation is reduced to a minimum and mainly remains in the code; user documentation is only produced on demand. Test-driven development and pair programming, for instance, ensure and increase the quality of the products.

Agile methods explicitly do not develop flexible artifacts for reuse. They assume that future change scenarios cannot be anticipated anyway and the investment in flexibility will not pay off. Instead, the focus is on the requirements at hand. Thus, agile methods might fail at producing reusable, flexible artifacts. But most organizations produce similar products for several customers. Thus, the problem with agile methods in such a context is that commonalities between several products are not leveraged but products are always built more or less from scratch. At best products are copied and adapted to a new context in an unplanned ad hoc fashion.

4. Integration of PLE and Agile Methods

In Section 3, we showed the potential of PLE and agile methods to increase the agility of an organization according to our definition. In this section, we introduce possible combinations of the approaches that aim at empowering organizations to reach a level of agility that could not be accomplished by using only one of them. We first sketch an application engineering process called PuLSE-I (I stands for instantiation) that is part of Fraunhofer PuLSE™ (Product Line System and Software Engineering)[1] that is successfully used by Fraunhofer IESE since years. Then we show how organizations that aim at getting more agile can use PuLSE-I and how agile practices like the planning game help in this context.

4.1. PuLSE-I

PuLSE-I is a reuse centric application engineering process. A new instance of a product line is developed in five steps as follows:

1. Plan for a product line instance: First, the match between the product specific requirements and the scope of the product line is identified. If features are requested that are out of the scope, a request to FE is sent to decide if the scope should be extended,

[1] PuLSE™ is a registered trademark of the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern, Germany (for more information visit www.iese.fhg.de/PuLSE).

otherwise the new feature will become product specific. The result of the planning step is a project plan that contains effort estimates for constructing new product line assets (FE) and for implementation of product specific items (AE).

2. Instantiate and validate product line model: The product line model represents the requirements for the whole product line. It contains a decision model for resolving the included variabilities. All decisions related to assets reused in the product at hand have to be made at this point. If new features are added to the scope of the product line FE has to update the generic product line model. Product specific features out of the scope of the product line have to be specified in addition. The result of the instantiation step is a specification of the product at hand.
3. Instantiate and validate reference architecture: The generic reference architecture is instantiated in this step. All decisions related to reused components of the architecture are made at this point. If the scope of the product line is extended for the product at hand FE must extend the generic reference architecture. Product specific components and connectors are added to the instantiated reference architecture by AE.
4. Construct product: Based on the product specific instance of the reference architecture, lower level design, implementation, and test of the product is performed. The components already existing in the product line asset base can be instantiated. New components that are required in the product line asset base but do not exist so far are constructed by FE. Product specific parts are implemented by AE. All components are integrated and tested.
5. Deliver system: After construction, the product can be installed in the customer's environment. Experiences from the delivery of previous instances of the product line at the sites of other customers can be reused.

A detailed description of PuLSE-I can be found in [3].
The reuse centeredness of PuLSE-I helps organizations quickly develop new products. If many features out of the current scope of the product line are required by the customer, the time to market of a new product will increase. Most likely, AE has to implement product specifics and FE eventually has to evolve existing product line assets or construct new ones. In this case, the time to market is directly influenced by the collaboration between AE and FE and by the processes used in FE. A tight coupling between AE and FE is necessary. In the next section, we will show how agile principles and practices can be used in the context of PuLSE-I.

4.2. PuLSE-I and Agile Methods

Agile principles and practices can only be leveraged if an iterative and incremental development process is used. Hence, we assume that after the main reuse step in the beginning of PuLSE-I, an iterative process is used. Only if a product is developed and delivered incrementally, frequent feedback can be given, which is a crucial principle in agile software development.
By means of PLE, an organization is capable of producing a first version of a product for a specific customer including the core functionality faster than what is possible with any agile software development method. If PuLSE-I is used iteratively, we can produce a version of the requested product with all the functionality already in the current scope of the product line in one or several days. Because of the approved quality of the reusable assets, the customer directly gets a high quality product that can be used and evaluated to give feedback. In further iterations, new functionality can be added to the scope of the product line or product specific features can be implemented. As we have seen in the previous section, the steps from 1 to 4 of PuLSE-I potentially require requests to FE. At this point, a thorough coordination between AE and FE is crucial to leverage the full agility potential of a product line organization. If interfaces of new reusable components to be constructed by FE are specified together with the AE teams, for instance, parallel work can be increased during product construction (step 4). This can lead to a reduced time to market of products, if the integration of AE and FE results works out well.
Now, we will show by means of two examples, how agile practices can be used in this context. We chose the planning game [8] as a management practice and incremental design [4] as a development practice.
In the planning game, the customer prioritizes the requirements and the developers estimate the effort needed to implement those requirements. The end dates of iterations are specified, i.e. the customer states when executable versions of the product must be delivered. Based on this information, the requirements can be assigned to iterations. In a product line context the developers from AE and FE contribute to the product instance. Hence, developers from AE and FE should participate in the planning meeting besides the customer. Especially, the developers from FE can give

immediate feedback, if requested features are within the scope of the product line, and if not, how much effort is needed to integrate them. The direct communication and negotiation between the customers, developers of AE, and developers of FE can bring the following benefits, for instance:

- The reuse rate can be increased. FE gets information on newly requested features early in an AE project and can evolve the product line assets proactively. Thus, in upcoming iterations AE can build upon the "right" reusable assets.
- Redundant implementations of product specifics in several AE projects can be reduced. FE developers participate in planning games of all AE projects and thus can coordinate the work done in parallel in the AE projects.

The increased reuse rate and the avoidance of redundant implementations directly influences the time to market of new products in a positive way, because implementation effort can be saved.
Incremental design only takes into account the requirements of the iteration at hand and does not aim at building in flexibility that could be leveraged by future requirements or change requests. The underlying assumption is that it will not pay off to build in flexibility because future requirements and changes cannot be anticipated anyway. If this is true, we could save effort in FE in a product line context that is spent on building in flexibility for future requirements, because most likely this effort will not pay off. Instead, FE could fully concentrate on supporting AE projects in their current iterations by increasing the reusability of the generic components already chosen for reuse.
In contrast to incremental design, up-front design aims at building in flexibility into architectures and designs to prepare for future requirements and change scenarios. Up-front design and the resulting build-in flexibility have proven to be a crucial success factor in PLE.
We conducted an experiment to learn more about the effects of up-front design and incremental design and how to combine them. We give an overview of this experiment in the following section.

5. Experiment: Flexible Design Up-Front vs. Incremental Design

In order to develop a better understanding of the effects of built-in flexibility as compared to incremental (or evolutionary, emerging) design, we conducted an experiment. The results from the experiment are still being analyzed, but we think the experimental setting provides an interesting framework that can lead to a fruitful discussion on how to study these issues.

In this experiment, we wanted to understand the impact of up-front design and incremental design on the quality of the resulting design and the required effort. We focused on the following research questions:
1. How much overall design and implementation effort is needed for a series of changes when performing up-front design compared to incremental design?
2. How do design and implementation effort for one specific change evolve over a series of changes when performing up-front design compared to incremental design?
3. What is the quality of the design after a series of changes when performing up-front design compared to incremental design?

The experiment took place in the first two weeks of the lecture "Foundations of Software Engineering" at the University of Kaiserslautern. 67 students participated in the experiment. The students worked in pairs. Even though we could not pair students up, we had good reasons to believe that the teams would be relatively balanced, i.e., inexperienced students would pair-up with more experienced partners. Thus, there were 38 groups of which 19 were assigned up-front design, and 19 incremental design.

All students received exactly the same basic information throughout the experiment. The following material was handed out at the beginning of the experiment:

- Documentation and Code of TSAFE 2b [11].
- One concrete change request
- A document describing desirable future features without giving too much detail and without indicating whether any of these features would be addressed in a future change request.

All student groups designed and implemented the same 5 change requests. The only variation was the assigned process. All students groups were requested to hand in the same artifacts (including design documents, source code, and output from test runs) for every change request. In addition, they reported the effort to produce each artifact.

In the following sections we will describe our view of the two design processes up-front design and incremental design.

Up-front design

In the beginning of the first iteration the students design the system up-front with respect to the first concrete change request and all the high-level descriptions of further change requests. This means they should build in flexibility that allows for further changes. After the extensive design phase of the first iteration they implement the first change request. The upcoming four iterations start with a short up-front design phase where the design is extended according to the concrete change request of that iteration. We assume that the up-front design in the first iteration facilitates this extension. On the short design phase of iteration 2-5 follows an implementation phase.

Incremental design

In the beginning of the first iteration the students design with respect to the first concrete change request. They are told that it doesn't make sense to take into account the high-level descriptions of possible further change requests. After this design phase they implement the first change request. The upcoming four iterations are handled in the same way. First the design of the system is evolved according to the concrete change request of the actual iteration, second the change request is implemented.

Design of change requests

In order to study the effect from change requests (CR) on built-in flexibility, one has to carefully design the change requests so that they affect the area of the system that is subject for study. In this case, we also wanted to study how a set of change requests would affect the system and its built-in flexibility, which implies that the whole set of change requests needed to be carefully designed. We addressed this problem by reasoning about the architecture and the design of the software and how change requests could be formulated so that they would affect the desired areas of the system. This turned out to be a complicated problem since we also needed to predict how the different processes most likely would implement the changes differently. In the end, we defined a set of change requests that have the following characteristics.

1. They all mainly affect the computation part of the system. Since the architecture specifies that the server is responsible for all computations, the "correct" place to put the new computations would be in the server.
2. They all build upon each other to some extent and could not have been implemented in a different order with the exception that CR1 and CR2 are relatively independent. CR3 is dependent on CR1 and CR2. CR4 is an extension of all previous change requests while CR5 also adds new functionality that is somewhat independent of previous change requests.

Test cases

Since we wanted to measure the time it takes to implement the CRs according to two different strategies, we wanted to avoid students spent time on other activities. Therefore, we provided a set of test cases for each CR. The CR is considered implemented when all test cases pass. This would also drive simple groups to only implement exactly what is necessary to pass the test case.

Reasoning about ROI

In order to reason about the return on investment for built-in flexibility, we created the following framework. Because of the limited space, it only focuses on two change requests (CR_1 and CR_2), but can easily be extended as necessary.

The $Base_0$ represents the initial system. Since we are dealing with change requests, we assume that the initial system has some functionality already.

In order to implement a change request CR_n, the software has to be changed. I.e. new code needs to be added, existing code needs to be deleted or changed. This change is represented by ΔCR_n. This change activity will require some effort $e = f(\Delta CR_n)$.

For each CR, flexible groups will strive to build-in flexibility to prepare for what might be required by any set of future CRn. Thus additional code needs to be added, existing code needs to be deleted or changed. This additional change is represented by ΔCR_x. This activity will require some effort $e = f(\Delta CR_x)$.

The changes the simple groups perform can be expressed as $Base_{s1} = Base_0 + \Delta CR_1$. The changes the flexible groups perform can be expressed as $Base_{f1} = Base_0 + \Delta CR_1 + \Delta CR_{X1}$.

For the next change request, CR_{n+1}, flexible groups will gain from the flexibility built into the base:
-flexibility($Base_{fn}$), while Simple groups will pay a penalty for the built in inflexibility:
+ inflexibility($Base_{sn}$).

	CR1	CR2
Simple	$Base_{s1}= Base_0+ \Delta CR1$	$Base_{s2}=Base_{s1}+\Delta CR2+ inflexibility(Base_{s1})$
Flexible	$Base_{f1}= Base_0+ \Delta CR1+ \Delta CRX_1$	$Base_{f2}=Base_{f1}+\Delta CR2– flexibility(Base_{f1})+\Delta CRX_2$

Table 1: Work required to implement CR1 & CR2

Assuming that the solutions from two groups (one simple and one flexible group) are comparable in terms of functionality and quality, we can calculate the initial extra work (could be zero if the flexible group didn't follow the process) that was conducted to prepare for future changes:

$$\Delta CRX_1 = Base_{f1} - Base_{s1}$$

We can calculate, for each CR, the relative return on the investment to do flexible vs. simple (could be negative) by comparing the gain from (-) previous extra work and the cost for (+) current extra work for the future that was conducted to prepare for future changes (-) inflexibility from previous extra work. Since we can calculate the diff between two different implementations, we can determine the amount of change between them. This amount of change is an indication of the work needed to implement the change request.

$$Base_{f2}=Base_{f1}+\Delta CR2–flexibility(Base_{f1})+\Delta CRX_2$$
$$Base_{s2}=Base_{s1}+\Delta CR2+ inflexibility(Base_{s1})$$

$$Diff(Base_{f2,}Base_{f1)}=\Delta CR2–flexibility(Base_{f1})+\Delta CRX_2$$
$$Diff(Base_{s2,} Base_{s1)}= \Delta CR2+ inflexibility(Base_{s1})$$

ROI relative to any one increment is calculated as:

Work for flexible groups: $WF =\Delta CR_{n+1}– flexibility(Base_{fn})+\Delta CRX_{n+1}$		
Work for flexible groups: $WS =\Delta CR_{n+1}+ inflexibility(Base_{s1})$		
$WF > WS$	$WF \approx WS$	$WF < WS$
Bad because more work is devoted to adding flexibility for future CRs compared to the benefits from previously built-in flexibility (negative ROI)	Not good because about the same work is devoted to adding flexibility for future CRs compared to the benefits from previously built-in flexibility (zero ROI)	Good because less work is devoted to adding flexibility for future CRs compared to the benefits from previously built-in flexibility (positive ROI)

Table 2: Relative ROI for any one increment

We suggest applying this ROI framework in the analysis of the data that we collected throughout the experiment. We suggest using it in order to determine whether the investment the flexible groups made in their solutions paid off over the five change requests in comparison to the other groups.

6. Conclusions

Agility is the capability of an organization to cope with today's customer demands for short time to market and high quality of products. A combination of Product Line Engineering and practices from agile methods leads to a higher level of agility in this sense compared to applying only one of these approaches in isolation. In our proposed combination approach, PLE forms the basis of each agile organization. Reuse is leveraged to build new products that rely on proven parts. Agile methods are used to perform the tailoring of a product for a specific customer during application engineering. The combination is supported by Fraunhofer PuLSE^{TM} and especially based on PuLSE-I. Agile practices and principles can be integrated in the context of PuLSE-I, the product instantiation component of PuLSE^{TM}. The planning game, for instance, includes great potential to increase the agility of a product line organization, but also other practices like continuous integration or automated regression testing can contribute.
Since PLE and agile methods are based on different philosophies, more research is necessary to understand the effects of the approaches and their combination in detail. With a better understanding of what the strengths and weaknesses from these two approaches are, we can develop an even more powerful combined approach. Ideally, this analysis is based on the return on investment from them so that the practices that have highest return can be combined and the ones that have yield low return can be avoided.
We discussed an experiment that we conducted in order to develop a better understanding of the integration of PLE and agile methods. By applying the proposed combined approach and by analyzing experimental results using our ROI framework, and by conducting more experiments, we believe that over time we will be able to further improve our combined approach.
It is important to note another aspect that affects ROI for building a flexible design: the fact that the built-in flexibility in the up-front design can accommodate the future changes. In the experiment, the students received a high-level description that the ROI framework, so they could create a design that would accommodate those changes. In real projects, we

cannot predict the future changes. So, how can we decide when it is more suitable to use a flexible design or an incremental design?
There are contexts in which anticipated changes can be predicted with certain accuracy and you know that changes will inevitably happen with a high frequency. In those cases, it pays off to do a flexible design. In those cases, you can design your system with the "right" flexibility in place, since your prediction of chances is fairly accurate. Doing a flexible design might be more time consuming at first, but it will pay off in the long run. If the design is flexible so that new changes can be easily incorporated, not only the effort to implement the change will be less, but also the system will contain fewer defects. Since changes will be easily accommodated in the design, fewer existing components will need to be changed. Fewer defects will be introduced in the existing components and existing components that were not changed don't need to be re-tested.
Examples of situations when this is true are:

- products that undergo several changes because of similar technology changes (e.g. different lens in digital camera)
- products which require that new releases are done in a short period of time because of market demands (e.g. cell phones and digital cameras)
- long-life systems (e.g. telecom, ground control, air traffic control etc) where you know that the next release will have to build upon the previous one and that the software will live forever.

The opposite situation when you don't know how long the product will live and you cannot predict accurately anticipated changes would be better off using an incremental approach.

Acknowledgements

We would like to thank J. Prof. Dr. Rausch at the University in Kaiserslautern, Germany for letting us conduct the experiment in his Software Engineering class. We also acknowledge that this work was supported by a NSF grant as part of their Science of Design program.

References

[1] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. PuLSE: A Methodology to Develop Software Product Lines. Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), 1999, pp. 122–131.

[2] P.G. Bassett. Framing Software Reuse. Lessons From the Real World. Upper Saddle River: Yourdon Press, 1997.

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

[4] K. Beck, C. Andres. Extreme Programming Explained – Embrace Change, 2nd edition. Addison-Wesley, 2005.

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

[6] P. Clement, J.D. McGregor, S.Cohen. The Structured Intuitive Model for Product Line Economics (SIMPLE). SEI Technical Report CMU/SEI-2005-TR-003, 2005.

[7] J.M. DeBaud, K. Schmid. A systematic approach to derive the scope of software product lines. In Proceedings of the 21st International Conference on Software Engineering, pages 34–43, 1999.

[8] J.Hunt. Agile Software Construction. Springer-London, 2006.

[9] The Agile Manifesto: http://www.agilemanifesto.org.

[10] M. Lindvall, D. Muthig, A. Dagnino, C. Wallin, M. Stupperich, D. Kiefer, J. May, and T. Kähkönen. Agile Software Development in Large Organizations. IEEE Computer 37[12], pp. 26-34, 2004.

[11] M. Lindvall, I. Rus, F. Shull, M. Zelkowitz, P. Donzelli, A. Memon, V. Basili, P. Costa, R. Tvedt, L. Hochstein, S. Asgari, C. Ackermann, and D. Pech. An Evolutionary Testbed for Software Technology Evaluation. Innovations in Systems and Software Engineering - a NASA Journal, vol. 1, no. 1, pp. 3-11, 2005.

[12] Weiss, D.M. and Lai, C. Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley, 1999.