Long-Term Product Line Sustainability with Planned Staged Investments

Juha Savolainen, Danfoss Power Electronics A/S

Nan Niu, Mississippi State University

Tommi Mikkonen, Tampere University of Technology

Thomas Fogdal, Danfoss Power Electronics A/S

// A model that uses planned staged investments lets developers rebuild software product lines in a sustainable way. The key idea is to use two different operational phases—investment and harvesting—to coordinate the competing, parallel needs of redesign and reuse. //

SUCCESSFUL SOFTWARE SYSTEMS tend to be long lived because they evolve when customer needs start to deviate from a system's original design. To support emerging new requirements, developers might take various shortcuts to overcome the current design's limitations. Over time, these shortcuts start to jeopardize the system's architecture, making it more complex and hindering systemwide requirements such as reliability and maintainability. Increasing complexity poses a major threat to the system's long-term viability. In this article, we propose an approach to manage software evolution by introducing two phases, investing and harvesting, that focus on long-term sustainability and short-term gains, respectively.

Planned Staged Investments

When working with a long-living software system, there are two fundamentally different ways to sustainably manage evolution. First, developers can try to keep the system up to date by continuously refactoring the design to reflect customers' changing needs. In this approach, the intent is to maintain continuous alignment between evolving requirements and the design and to prevent a gap between true user needs and the existing implementation. Although this approach can work well for small teams and systems, in a setup that relies on agile and lean approaches, scaling up the approach for very large systems is problematic.[1,2] Although researchers have proposed ways to scale agile methods,[3,4] their focus has been on practices, processes, and organization rather than on how to treat the problems that result from scaling and still achieve sustainable design.

The second approach targets larger organizations, partly reflecting Conway's law: over time, the architecture of any software system starts to resemble that of the organization.[5] Organizations that act in the software product line (SPL) mode[6] follow a software development paradigm that enables orders-of-magnitude improvements in time to

0740-7459/13/$31.00 © 2013 IEEE

market, cost, productivity, and quality (www.sei.cmu.edu/productlines). This paradigm uses two roles: application and platform engineering, both with different responsibilities and a common management function. Application engineering creates actual products, which requires a stable platform. Platform engineering creates reusable components that eventually make up the platform, which require assumptions on how future products will be built using them. As product development progresses, some assumptions will be proven wrong, but because products rely on reusable components' stability, modifying them becomes problematic: changes can range from something at the individual product level, such as adding a customer-specific extension, to the entire SPL level, such as adding a standard monitoring protocol. Moreover, in addition to products' common and individual changes, certain changes deviate from the constraints codified in the SPL's underlying architecture, such as creating a configuration with a minimal memory footprint. The challenge in practice is how to co-evolve components that constitute core assets and products so that both can endure the SPL's long-term use.

We can consider the gap between the hypothetical, ideal system and the actual existing implementation to be the system's technical debt.[7] It's made up of various issues, including shortcuts taken to implement the new functionality, lack of refactoring, and permissions given to break established architectural rules. Over time, the amount of the debt increases and the cost of fixing the growing list of issues rises. Accumulating too much technical debt becomes increasingly adverse to the system's long-term existence.

Fair, timely SPL management is necessary to sustainably develop both core assets and products.[8] We propose a model that we call *planned staged investments* to support SPL rearchitecting during evolution, where the objective is first to allow for a controlled deviation of needs and actual design and second to plan when and how to bridge that gap through redesign and rearchitecting. The overall aim is to more effectively manage SPLs when conflicting requirements simultaneously emerge from needs to redesign and reuse software. The model relies on phasing SPL operations in two different operational phases: investment and harvesting. During investment, engineering effort is put into improving reusable asset creation. During harvesting, benefits are gained from the investment in the form of simplified and faster product creation. To balance between the two phases, SPL management is obviously crucial.

> High productivity requires limiting the number of changes to enable efficient product creation on top of stable assets.

In the investment phase, SPL operations focus on developing and improving core assets; they might even partly integrate product development. For instance, so-called lead products, commonly used in SPLs, are often the first generation of products built on a revised set of core assets—think of generations of mobile phones. In contrast, the harvesting phase focuses on product development, and investments to core assets are minimized to only those that are critical for stability and robustness.

Evolving SPLs

An SPL comprises a set of software-intensive systems that share a common managed set of features. These features are designed to satisfy the specific needs of a particular market segment or mission and are created from a common set of reusable core assets in a prescribed way—take, for example, an electric motor. To create a specific electric motor, one must add product-specific parts to separate individual products from each other. These product-specific parts will be reusable only when building a new generation of similar products, if at all. However, when SPL refactoring takes place, it's possible that some of the changes will be turned into general-purpose components.

SPL evolution isn't widely considered a special form of evolution; to a certain extent, each evolution reflects that of its individual software systems. A lot of SPL research focuses on the creation of (and evolution toward) the product line but overlooks aspects related to the whole SPL and its long-term sustainability, such as technical debt.[9–11]

Adaptation Mechanisms

High productivity requires limiting the number of changes to enable efficient product creation on top of stable assets. Consequently, product-specific parts need to adapt to, or even completely reimplement, features in core assets if the features don't satisfy product requirements. Numerous adaptation mechanisms exist, and their applicability varies, depending on issues related to the product itself, productivity issues, development schedules, testing facilities, organization, and so on. For instance, in a setting where memory footprint

isn't critical, it's possible to include all system features and configure them at runtime. In contrast, when developing embedded systems, only necessary features can be included in systems, and configuration must take place at build time.

Technical Debt

Necessary updates to core assets must sometimes be postponed, because changes in the SPL infrastructure directly affect product programs that aim to derive new product variants based on existing assets. Therefore, technical debt in the system results in part from freezing the core assets so that numerous products can rely on them. Because SPLs often build on a single architecture, it's only natural that properties related to quality (such as security, scalability, and robustness) are largely determined by the core assets. These parts in turn help define the larger variability mechanisms available to product engineers. For individual functions, product-specific solutions work in cases where the common core assets don't satisfy product requirements. Consequently, SPLs have different generations, where quality properties (such as memory footprint and performance-related issues) included in core assets satisfy the needs of that particular generation; however, features of core assets can be replaced with product-specific variations. In other words, while the SPL evolves, technical debt related to architectural issues accumulates in the core assets. As long as enough products can be implemented without refactoring the underlying architecture and its common components, it's possible to harvest the core assets.

The number of product-specific variants of core assets incorporated throughout all products is a concrete measurement of the SPL's technical debt. As long as the debt is at an acceptable level, the SPL can remain in the harvesting phase while new products are built and existing ones maintained, without reworking the core assets. Eventually, maintenance costs will increase because the deviating code bases can share less and less maintenance work. Eventually, the increase in costs reaches a level where it becomes obvious that a new investment is needed to keep the SPL viable. But by that point, it's probably too late: the SPL as a whole will need major investments, in particular, for its core assets. Our planned staged investments model is simple: it always includes some level of investment, even in the harvesting phase; the operation phase defines whether the general intent is to incur technical debt or minimize it.

Finding the right balance between harvesting and investing is a system-specific approach and requires deep domain knowledge. Although we only have firsthand experience with Danfoss Power Electronics, SPLs that share its characteristics—large systems, stable demand, hard real-time requirements, safety criticality, and extensive configurability and adaptability—can benefit from what we learned about explicit renewal during investment and enforced stability during harvesting.

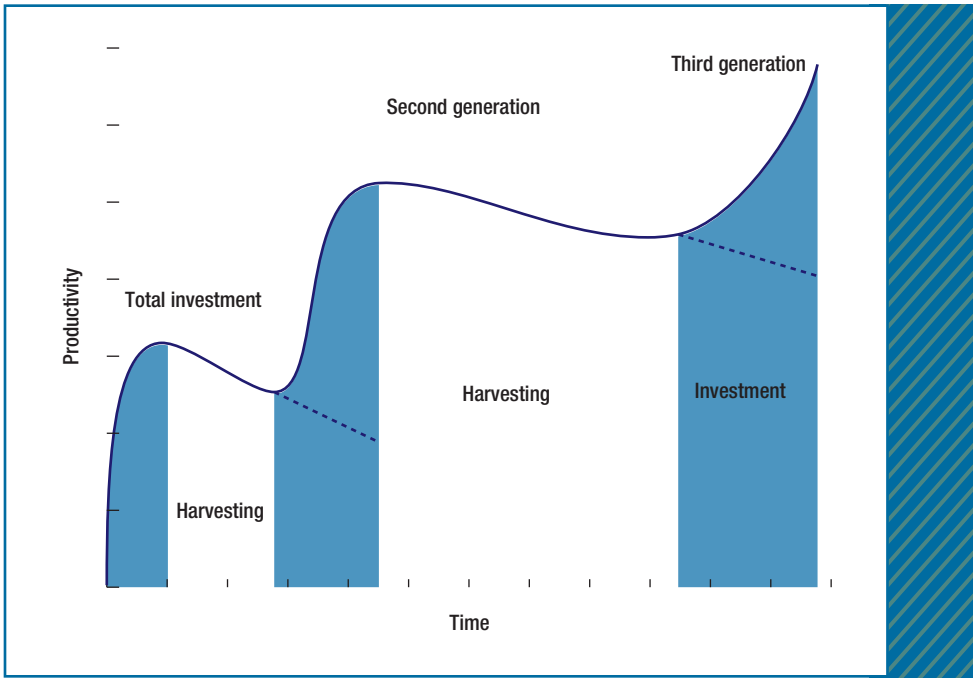

FIGURE 1. An investment plan for managing a product line. This illustrates the evolution of the Danfoss software product line.

The Danfoss Experience

Danfoss Power Electronics produces frequency converters, or *drives*. A drive is a power electronics–based device that controls an electric motor's shaft speed, or torque. The ability to control a motor is important because it allows for better process control and financial and environmental savings by reducing the amount of consumed electricity.

Because drives are used in different application domains, they must be configurable. Owing to cost and hardware

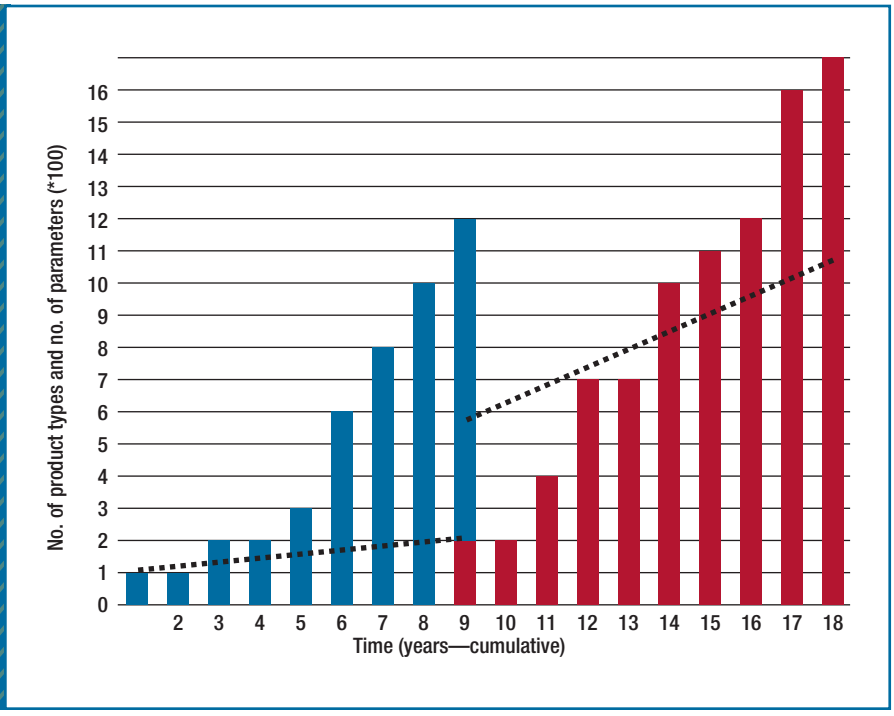

FIGURE 2. The effect of the transition from initial harvesting to the second harvesting period. The blue bars show the cumulative number of products created from the first-generation assets. The red bars indicate the same for the second generation. Black dashed lines show the increase of the number of parameters of the respective generation.

limitations, variability must be implemented in a way that only includes the relevant features and adaptations within a particular product. Figure 1 illustrates the evolution of the Danfoss SPL. The first version of the system architecture was created during the initial investment phase using the clone-and-own reuse approach,[6] in which each new product is created from a copy of the previous product. This approach allowed easy harvesting of the initial investment and rapid creation of new products. Each new product was defined as a delta, or a set of necessary modifications in software, on top of an existing product.

Evolution

Over time, the code bases in Danfoss's products started to evolve independently because their clients' requirements varied and their timeframes varied. This has led to increasing maintenance costs. After the first product, developers could have created a more sophisticated platform where mechanisms for variability were in place to prevent the increase, but creating copy products allowed for faster times to market; thus, the company entered the harvesting phase immediately after the initial investment.

As maintenance costs continued to increase (to a critical point after the third large copy product), it became apparent that changes were needed to regain productivity—moreover, a forecast predicted a sharp increase in the number of needed products. During the second investment phase, the developers merged separate code bases into a common repository, removed duplicate features, and introduced a code-level variability management approach. Because the developers found them simple and understandable, compile-time feature flags were used as the main mechanism for variability management. Although the feature flags do have scalability limitations, the developers considered the mechanism to be sufficient for the next harvesting period. Moreover, they considered risks associated with the approach to be low, which further encouraged use of this mechanism.

Figure 2 illustrates the transition from the first to second generation; black lines show the increase of the number of parameters of the respective generation. The number of parameters indicates the total amount of functionality in the drive, because the parameters make most of the new functionality user configurable. The parameter growth also correlated strongly to the growth of each product's functional requirements in the context of the second generation, convincing us that using parameters as an indication of functionality is reasonable. Because the first parameter set was copied to the second generation without modification, the change in parameter number between generations does indeed correspond to a step change in functionality.

Benefits

Claimed productivity benefits are difficult to validate. Based on the parameter count, the functionality did more than double, from 213 to 577 parameters, when transitioning from the first to the second generation. Also, the ability to increase functionality over time improved. (This is consistent with our experience that adding new features to the improved set of core assets became significantly easier.) Regarding the number of products, the change is less dramatic. However, four more

products were still released over a period of nine years than with the previous generation. All this evidence, together with the experiences of the staff that have been part of this journey, makes it plausible to claim that productivity has increased when making the transition from the first to the second generation.

Variability

The second harvesting period proved that the new approach to variability management was sufficient: an independent team could perform product-specific additions without affecting other software development tasks. Most of the software features were done independently by the product teams, allowing for fast time to market and a special focus on products' particular issues. Seventeen product types were created using this approach with relatively few developers; the total number of distinct products based on the platform is more than 30 when hardware variants are also taken into account. After the number of products increased to more than 30, feature flagging's limitations as a variability mechanism became apparent. Increasing the number of instances and types of feature flags meant that code files became difficult to understand, and introducing modifications and new variants became highly laborious.

To alleviate these problems, we planned a third investment phase that would change the variability management approach by refactoring fragmented variant code into more typical object-oriented software. Consequently, variability is now managed through subclassing, delegation, and parameterized inheritance techniques. To limit the duration of this third investment phase, we decided that we would make the change only to the higher levels of the software stack. So, the variability expressed in the application layer is solely based on new concepts, but the lower-level service and communication layers can still use feature flagging.

Critical Success Factors

For our model to work, everyone in the organization needs to understand the current operational phase and its consequences. Changing the phase of operation can be risky for an organization and its staff, so implementers always need to proceed with caution.

The investment phase is typically an exciting time for software engineering because of the new design and implementation options for examination and experimentation. Engineers might even try to prolong the investment phase to get more time to improve the platform and perfect the design choices. In contrast, during harvesting, products are developed and technical debt is accumulated. This can lead to frustration for some engineers if they feel that they could create more elegant designs or are dissatisfied with the resulting design's aesthetics in general.

In contrast, the investment phase is stressful for management because revenue isn't generated but costs still accumulate. Risks must be continuously managed to get new productivity benefits without unnecessarily prolonging the investment phase. However, the harvesting period demonstrates the productivity benefits gained from the investment and thus should lead to significant revenue gains and profits by creating a set of products on top of common assets. So, for management, there's a desire to prolong the harvesting period and postpone the need for investment to maximize profits over the short term.

Changing Mindsets

A critical concern is how the organization can switch between these two very different mindsets. Based on our experiences with SPLs in different domains, we've found this very difficult to do. During investment, the focus should be on the future: careful planning, risk management, and impact analysis. During harvesting, the focus should be on the present: matching customer needs, eliciting product requirements, and efficiently engaging in iterative development. Because the focus is radically different, the transition won't be smooth because a major part of the organization will be changed. Consequently, we believe that the only option is to make the change as explicit as possible—everyone in the organization should know when the change takes place. To this end, we believe that the biggest success factor for the approach is trust: developers must trust that management has sufficient patience to go through the investment phases, and management must trust engineering estimates of cost, duration, and scope of platform improvements. This need for trust could even culminate with the appointment of an SPL champion, who has deep insight in

For our model to work, everyone in the organization needs to understand the current operational phase.

both technical properties of the SPL as well as customer needs and the organization's business side.

Balancing the Phases

There are several potential pitfalls that any organization applying our model should know. For example, a prolonged harvesting stage will always lead to decreased productivity and lower quality while product-specific needs become increasingly difficult to meet owing to accumulating technical debt. It's possible that management will begin to look at the investment needed as an indication of poor engineering rather than as a logical consequence of the overly extended harvesting period.

However, the harvesting period must be long enough to be profitable, and the investment phase must be extensive enough to renew the system. For software engineers, it's sometimes difficult to accept that the software made during harvesting won't win any beauty contests. Engineers should only perform localized changes with low impact. This is difficult because the change in the phase of operation is driven by problems, and developers are typically inclined to try to fix everything with new solutions. During harvesting, however, overly elegant solutions should be avoided, and there should be no overengineering. Investment is a step change that requires careful planning, requirements, and technical surveys on technically feasible solutions. In contrast, product creation is iterative, rapid, and agile. Unfortunately, these two phases might also require different people to implement them, which limits the practicality of the approach in its purest form.

Combining Phases

Mixing investment and harvesting phases can be harmful. During harvesting, engineers tend to do unnecessary or premature refactoring because these tasks typically seem technically rewarding. Instead, they should wait for the investment phase, where the system as a whole can to be addressed. However, these partial fixes can jeopardize the platform's stability and robustness, which is more harmful to the SPL than an obsolete design detail. In other words, persistence with obsolete features still benefits the SPL's stability. Therefore, even as technical debt accumulates and the number of problems increases, the developer's mindset should focus on just the absolutely necessary updates.

Although we emphasize separating the investment and harvesting phases here, in practice, this isn't a simple matter: there are good reasons to have an overlap between investment and harvesting. After the initial investment phase, you will rarely be in a position to simply stop product creation. Consequently, domain engineering might change an operation phase while application engineering keeps creating products on top

ABOUT THE AUTHORS

JUHA SAVOLAINEN is a software director responsible for software R&D at Danfoss Power Electronics A/S. His research interests include embedded systems, product line engineering, software architectures, and requirements engineering. Savolainen received a PhD in computer science from Aalto University, Finland. He's a member of the IEEE Computer Society. Contact him at juhaerik.savolainen@danfoss.com.

NAN NIU is an assistant professor of computer science and engineering at Mississippi State University. His research interests include requirements engineering, product line engineering, and information seeking in software engineering. Niu received a PhD in computer science from the University of Toronto, Canada. He's a member of the IEEE Computer Society. Contact him at niu@cse.msstate.edu.

TOMMI MIKKONEN is a professor of software systems at Tampere University of Technology. His research interests include software architectures, distributed systems, and mobile and Web software. Mikkonen received a PhD in computer science from Tampere University of Technology, Finland. Contact him at tommi.mikkonen@tut.fi.

THOMAS FOGDAL is the embedded software platforms manager at Danfoss Power Electronics A/S. His research interests include bringing existing software product lines into the future. Fogdal received a BSc in electrical engineering from University of Southern Denmark. Contact him at tfogdal@danfoss.com.

of the existing platform. When pushed to the extreme, you might end up in a situation in which every modification requires a customer benefit and architecture improvement component (similar to cases studied elsewhere[12]), but managing this at the SPL level could become increasingly difficult.

Providing Resources

Supporting tools are needed to manage the SPL. Gathering metrics on many aspects of productivity can help developers understand trends and plan for new investment phases based on hard evidence. Tools operating at code level exist to identify dependencies between components, which helps in estimating the clarity of the architecture, and data from bug-tracking systems can help estimate the number of bugs in future. The tools don't need to be directly connected with software, and other metrics (such as sociotechnical congruence[13]) might be useful as indicators. For example, a large sociotechnical congruence deviation might indicate that technical debt has accumulated to an unacceptable level, thereby needing investment.

Our approach of alternating investment and harvesting requires close cooperation between development and product management. The inability to communicate and coordinate actions within an organization obviously will lead to problems, such as waiting too long to make an investment or fostering mistrust between development and management. We believe that the most important change is to simply understand the cycle of investment and harvesting. Such understanding can help align the whole organization, which will lead to better technical decisions that are justified from the business perspective.

References

1. A. Cockburn, *Agile Software Development*, Addison-Wesley, 2001.
2. M. Poppendieck and T. Poppendieck, *Lean Software Development: An Agile Toolkit*, Addison-Wesley Professional, 2003.
3. S. Ambler, "Agile Software Development at Scale," *Balancing Agility and Formalism in Software Eng.*, LNCS 5082, Springer, 2008, pp. 1–12.
4. D.J. Reifer, F. Maurer, and H. Erdogmus, "Scaling Agile Methods," *IEEE Software*, vol. 20, no. 4, 2003, pp. 12–14.
5. M.E. Conway. "How Do Committees Invent?," *Datamation*, vol. 14, no. 4, 1968, pp. 28–31.
6. P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.
7. P. Kruchten, R.L. Nord, and I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice," *IEEE Software*, vol. 29, no. 6, 2012, pp. 18–21.
8. P. Clements et al., "Project Management in a Software Product Line Organization," *IEEE Software*, vol. 22, no. 5, 2005, pp. 54–62.
9. J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, Addison-Wesley, 2000.
10. K. Schmid and M. Verlage, "The Economic Impact of Product Line Adoption and Evolution," *IEEE Software*, vol. 19, no. 4, 2002, pp. 50–57.
11. K. Pohl, G. Böckle, and F.J. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer, 2005.
12. S. Murer, B. Bonati, and F.J. Furrer, *Managed Evolution: A Strategy for Very Large Information Systems*, Springer, 2010.
13. M. Cataldo, J.D. Herbsleb, and K.M. Carley, "Socio-technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity," *Proc. ACM/IEEE Int'l Symp. Empirical Software Eng. and Measurement* (ESEM 08), ACM, 2008, pp. 2–11.