

Reactive Variability Management In Agile Software Development

Yaser Ghanam, Darren Andreychuk, and Frank Maurer

Department of Computer Science

University of Calgary

Calgary, AB, Canada

{yghanam, djandrey, fmaurer}@ucalgary.ca

Abstract — Agile organizations focus on developing software systems that satisfy their current customer base, without worrying about best practices to handle variations of requirements in the system. Scaling agile methods up to adopt variability management practices in their traditional form is challenging. In this paper, we discuss the challenges and we contribute a lightweight, iterative approach that enables agile organizations to manage variability on demand in a reactive manner. The approach relies on agile practices like iterative development, refactoring, and continuous integration and testing. We present a case study to show how the approach was used to handle variability arising from technical and usability issues, and we provide a discussion of the advantages and limitations of the approach.

I. INTRODUCTION

A. Variability Management

Variability management plays an important role in defining and handling the parts of the system that may vary. This is often needed when a number of similar – yet not identical – systems are to be derived from a common platform to satisfy different needs. This software paradigm is called Software Product Line (SPL) engineering [5]. Companies consistently report that SPLs yield significant improvements. Some reported reductions in the number of defects in their products and cuts in costs and time-to-market by a factor of 10 or more [15]. Commonality between systems is what makes SPLs economically effective; whereas variability is what makes mass customization possible. SPLs deal with similar systems as a family of products sharing a library of core assets. But since customer requirements are rarely exactly the same, shared assets have to accommodate a certain degree of variability. For instance, the customer of an intelligent home system should be able to choose a subset of components that fulfills his wants. It should also be possible for customers to tailor certain aspects of these components to meet their specific needs. A security module, for example, offers different techniques to secure access control such as PIN protected locks, access by magnet cards and finger print authentication. When choosing to have a security system component, customers may select one or more of these options.

B. Problem Statement

Traditionally in SPL engineering, variability analysis is conducted upfront during a phase called domain engineering. A comprehensive analysis is conducted to specify the commonalities and variations in the prospective SPL. Commonality and variability analysis is concerned with determining the requirements of the members of the software family, and defining how these requirements may vary. This includes determining all sources of variation (i.e. variation points) as well as the allowed values (i.e. variants). After the domain engineering phase comes the application engineering phase. As a starting point, application engineers use the reference architecture, the reusable artifacts, and the variability profile – that were all defined in the domain engineering phase. Based on the specific requirements of a certain product, application engineers make decisions on what variants should be selected for each variation point. The outcome of this phase is an instance of the system that represents a specific product. Ideally, application engineers should provide feedback to domain engineers pertaining to problems and limitations of the current architecture or variability definition.

For agile organizations, the focus has been to develop software systems that satisfy their current customer base, without worrying about best practices to handle variations of requirements in the system. Recently, the agile community has been investigating ways to scale agile up to the enterprise level rather than the team level like in [12] and [16]. This will eventually require that agile organizations find a way to adopt SPL practices to manage variability in customer requirements in a more effective way. However, adopting SPL practices in their traditional form is challenging. For one, agile organizations foster a culture of minimalism in upfront investment and process overhead including documentation. This is in direct conflict with traditional approaches to SPL engineering where a whole phase, namely domain engineering, is dedicated for domain and requirement analysis upfront. Moreover, especially during domain engineering, documentation is deemed essential to communicate knowledge to application engineers. Secondly, agile organizations depend heavily on fast delivery as a mechanism for quick customer satisfaction and feedback, which is too difficult to achieve when a domain engineering phase is to occur before delivering any products. Thirdly, the flexibility to accommodate changes

in requirements and new customer requests is an important characteristic of agile teams. This characteristic will be compromised if two separate processes – namely domain engineering and application engineering – are introduced, because it may slow down the feedback loops between teams.

C. *Goal and Contribution*

Our goal is to reconcile conflicts between traditional SPL engineering and agile software development. We argue that for agile organizations to adopt a SPL approach, a reactive – as opposed to proactive – framework is more befitting. This paper contributes a framework that shall allow agile organizations to incrementally and reactively construct variability profiles for existing and new systems. The framework leverages common agile practices such as iterative software development, refactoring, continuous integration and testing to introduce variability into systems only when it is needed.

The rest of this paper will be structure as follows. First we review the literature around the topic of this paper in Section II. Section III describes the proposed approach. In Section IV we evaluate our approach using a case study of a real experience. Finally, Section V discusses the advantages and limitations of our approach.

II. LITERATURE REVIEW

A. *Incremental and Reactive Variability Management*

In our research we stress that for an approach to fit well with agile principles and practices, being incremental and reactive is key. By “incremental”, we exclude big-bang transitional approaches. And by “reactive”, we exclude proactive approaches in which a great amount of upfront speculation is required. The quest for an incremental and reactive approach to establishing and managing product lines is a relatively new phenomenon. For one, organizations did not want to throw away their investments in legacy systems and start all over again. Also, for many organizations the transition to systematically managed variability in their systems was too big a change if they were to follow the strict domain-then-application engineering model.

Kruger [11] contributed ideas and commercialized a tool to ease the transition to software mass customization. The main idea is that domain engineering and application engineering should not be separate. Their tool utilizes the concept of separation of concerns to realize variability in software systems. The tool is closed source and not available for academic evaluation. Reactive approaches, with the support of tools like the one in [11] has been reported to require orders of magnitude less effort compared to proactive approaches [2]. Clegg et al. [4] proposed a method to incrementally build a SPL architecture in an object-orientated environment. The method provides useful insight into realizing variability in an incremental manner, but does not discuss how to communicate variability from the requirement engineering phase to the realization phase. The aim of our work is somewhat similar to the abovementioned efforts. However, we differ in that we are not only concerned with realizing variability in a system.

Rather, we are interested in the process of managing variability as it evolves in an agile context, as will be detailed later.

B. *Agile Product Line Engineering*

Agility in product lines is a fairly new area of research. In 2006, the 1st workshop on agile product line engineering was held as part of the 10th international SPL conference [6]. The workshop aimed at bringing practitioners from the agile community and the SPL 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. One of the presented efforts was the iterative approach proposed by Carbon et al. [3]. This approach is based on PuLSE-I [1] 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. The approach does not discuss the role of agile methods in the domain engineering phase. In a different venue, Hanssen et al. [9] described how SPL techniques can be used at the strategic level of the organization, while agile software development can be used at the medium-term project level. Also, Paige et al. [14] proposed building SPLs using Feature Driven Development. They assert the method worked well when giving special considerations for the product line architectural and component design. While these efforts are interesting attempts to combine concepts from agile software development and SPL engineering, their goal is different from that of our research. While their goal is to find ways to introduce or enhance agility in existing SPLs, our goal is to enable agile organizations to incrementally and reactively build and manage SPLs by adopting frameworks that align well with agile principles and practices. Our goal goes hand in hand with the recommendations of McGregor [13] who presented an interesting theoretical attempt to reconstruct a hybrid method. He concluded that competing philosophies of the two software paradigms make their integration difficult. But he asserts that the two can be tailored under the condition that both should retain their basic characteristics. In our research, we try to tailor variability management to fit within an agile context such that the advantageous characteristics of SPL practices are attained and the agility of software development is not deteriorated.

III. THE PROPOSED APPROACH

This section will present the proposed approach to manage variability in a reactive manner using agile practices. The recommended process involves a number of steps, namely: eliciting new requirements, conducting a variability analysis, updating the variability profile, refactoring the architecture, running the tests, realizing the new requirements, and finally running the tests once again. This is an iterative process that repeats whenever new requirements are available. Each one of the steps is discussed in detail in the following subsections.

A. *Eliciting New Requirements*

This is the first natural step in any software development process. Traditionally – and especially in the case of SPL engineering – this is a fairly heavyweight process, because it involves domain analyses to predict what requirements may be

needed in the future. In agile software development, it is sufficient to get only the available set of requirements and divide them into work items that can be achieved in 2- to 4-week iterations. Speculation is to be avoided as much as possible. In our approach, we adopt the agile way of requirement elicitation. We also use a customer-driven elicitation process. This means that unless something is actually requested (or needed) by a known customer, we do not invest into incorporating it in the development process.

B. Variability Analysis

Variability analysis is traditionally conducted upfront in the domain engineering phase. Elicited requirements are analyzed in terms of what they share in common, and in what aspects they may vary. Sources of variations are determined, and they are called variation points. The allowed values for these variation points are also determined, and they are called variants. In our approach, we avoid a one-shot upfront variability analysis, simply because it does not fit within the iterative nature of requirement elicitation in agile methods. Rather, we conduct a variability analysis every iteration between the current requirements in the system and the newly elicited requirements.

During variability analysis, we use lightweight techniques to determine the commonalities and variations between the new requirements and the existing ones. Although we do not specify a certain technique to conduct this analysis, we recommend the use of a simple issue-implication table that lists all the issues that may cause variability in the system, and their implications in terms of variability. In each iteration, the expected outcome of this step is a list of changes to the variability profile. This includes new variation points, new variants for existing variation points, and new abstraction of common aspects. In Section IV, we use a case study to illustrate in detail how this is done in a real setting.

C. Updating the Variability Profile

By variability profile we refer to the list of all variation points in the system and their variants. They are usually expressed in a formal representation or using a feature model [10]. In this paper, we use this simple notation to illustrate the idea:

Variation Point X = {Variant A, Variant B}

Variant A = [feature1, feature2, feature3]

Variant B = [feature1, feature2, feature3]

Where: {} implies OR grouping; [] implies AND grouping.

After the variability analysis step in each iteration, we update the variability profile with any new variation points or variants arising due to the new requirements (in cases where there are no changes to the variability in the system, we may not need to do that). It is important to keep a variability profile for the system to ensure that all aspects of variability are traceable to code artifacts and that they are communicated well to all stakeholders through and after the development process. Variability profiles are also used to explicate any dependencies and constraints between variation points and variants. In [8], we explain in great detail how to maintain variability profiles using feature models and executable acceptance tests.

D. Refactoring the Architecture

Using the refactoring techniques described in [13], the architecture has to be refactored in order to accommodate the new variability. For example, new architecture layers can be introduced to abstract common aspects, and other layers can be specialized to handle variable aspects. It is important to note that the goal of this step is to refactor the architecture to be ready to accommodate the new version of the variability profile, and not to realize this variability. The actual realization of that variability happens at a later step. For example, suppose a feature x existed in the system before the current iteration. If feature y in the new requirements is just another variation of feature x, then a new variation point is defined. Although we have two different variants x and y, at this point we only consider the existing, not the new, variant. Thus, the architecture is refactored to accommodate a variation point with the variant x. This is important because we would like to separate the side effects of refactoring from those of adding new functionality.

E. Running the Tests

To make sure the refactoring process in the previous step did not have any side effects, we run all the tests in the system. This includes executing automated unit tests and acceptance tests as well as running all manual regression tests (usually used to test user interfaces and hardware related functionality). If a test fails, this means the refactoring process needs to be fixed, undone or redone to make this test pass again. We should not proceed to the next step until all tests are in a passing state.

F. Realizing the New Requirements

Having refactored the architecture to be able to realize the new variability (if any), in this step developers implement the new functionality. The developers should produce test artifacts either before (using test-driven development) or after writing the production code.

G. Running the Tests (again)

This step is similar to step E. All tests for the new functionalities as well as the older ones have to be run in order to make sure the new changes are actually verified and validated, and that the old functionality is not impacted by these changes. When all tests pass, a new iteration of the process can take place when needed.

IV. A CASE STUDY

A. Experience Context

System Overview:

The application we will discuss throughout this paper is called eHome. It is a software system to monitor and control smart homes. Generally, the interface of the application consists of a floor plan representing the smart environment to be controlled, a number of items that can be dragged and dropped on the floor plan, and a set of graphical user interface (GUI) controls. A screenshot is shown in Figure 1.

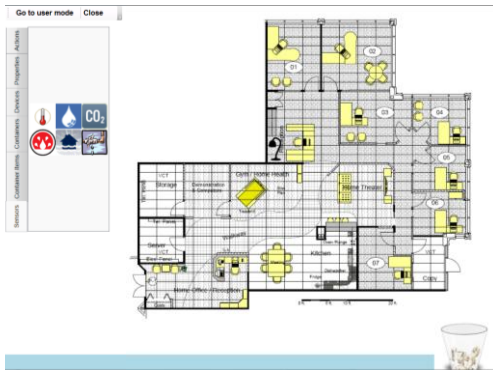


Figure 1. eHome application

Interacting with eHome occurs in two modes, namely:

(a) user mode which allows the dwellers to obtain information about climate variables in the home such as temperature, humidity, CO₂ levels and other sensory information, check the current status of certain devices in the home such as lights being on or off, change the status of devices such as turning lights on and off, keep track of items in containers such as a fridge or a medicine cabinet using RFID.

(b) designer mode which allows the users to add devices to be monitored and controlled, drop an icon of the device onto the floor plan and attach it to the actual device, add sensors to get climate information, add containers (e.g. medicine cabinet) and add items to the containers (e.g. pill bottles).

Initial Development:

The abovementioned features were all requested by an industrial partner we have been working with for the past year. The initial request was to deploy eHome on an HP TouchSmart PC¹ which has a single-touch vertical display. However, actual development of eHome was done on normal PCs with different screen dimensions and no touch capabilities. When we deployed eHome on the HP machine (which happened frequently because we had a testing HP PC onsite), we often needed to adjust certain scaling factors to fit the HP wide screen. We also realized that some decisions that had been made during development on the normal PCs needed to be revisited. Examples are:

- The size and design of some GUI elements made it challenging to interact with eHome using a finger touch because the latter is much thicker and less accurate than a mouse pointer.
- One event in eHome was triggered by a right-click which, on a touch-screen, did not make sense.

New Technologies:

As we went along, we wanted to deploy eHome on a large-scale SMART DViT Table² with an older version of the SMART SDK. A later request from our partner was to deploy

eHome on a digital tabletop they had recently purchased. Specifically, it was the New SMART Table³ which supported multi-touch input and had a newer version of the SMART SDK. Later on, we obtained a Microsoft Surface and we decided to include it within the hardware platforms that we should support. As more platforms were supported, more decisions were revisited and the software design underwent drastic yet incremental changes. These changes were mainly driven by the two factors we mentioned in Section 1: technical issues and usability issues. Examples of such issues include:

- Three different SDKs that dealt with touch point input, one for each hardware platform.
- Conventional GUI elements like menus and tabs assumed a single orientation (vertical).

Sources of Variability in eHome:

The technical and usability issues were not the only sources of variability in eHome. In fact, the first source of variability was business-driven. Smart homes vary widely with regards to what smart devices exist in the home, and what kind of monitoring and controlling is requested by a given customer. This variation in requirements often results in delivering a different application for each smart home. However, in spite of the differences between these applications, they share a lot of underlying functionality and business logic. Therefore, it is better to think of these applications as a family of systems that are somewhat similar yet not identical – which is the general understanding of what a Software Product Line (SPL) is. In this paper, we will not discuss SPLs in terms of business-driven variability – but we will focus on technical- and usability-driven variability due to the utilization of vertical and horizontal displays.

B. Using the Approach

When dealing with a new and fast-changing technology like digital tabletops, uncertainty about the future can be too high. This in turn might render useless any efforts to speculate these needs. In the development of eHome, we avoided huge investments in upfront work. Instead, we followed a bottom-up, evolutionary approach to develop and maintain the SPL. We incrementally embraced new variations as needed, and allowed our common platform to evolve gradually. The following sections will discuss this matter in more detail.

In the discussion to follow, each section talks about one variability aspect. For each aspect, we analyze the issues we encountered and their implications on our system, and then we describe our approach to contain them. Although the examples we provide are specific to our system, this does not deteriorate the generality of the analysis or the proposed approach – because we believe that researchers and practitioners in this field will encounter similar issues and implications that can generally be resolved using the same approach.

¹ HP TouchSmart IQ770 PC datasheet, available at: http://www.hp.com/hpinfo/newsroom/press_kits/2007/ces/ds_pc_touchsmart.pdf, last accessed June 18, 2009.

² DViT Technology, available at: <http://smarttech.com/DViT>, last accessed June 18, 2009.

³ SMART Table datasheet, available at: www2.smarttech.com/st/en-US/Products/SMART+Table, last accessed June 12, 2009.

Variability within Vertical Displays:

By vertical displays, we refer to the normal PCs that were used by developers to develop eHome as well as the HP TouchSmart PC on which eHome was initially deployed. The differences between these two groups were issues related to the mouse-versus-touch input. Table 1 describes these issues and their implications.

TABLE 1. VARIABILITY BETWEEN A NORMAL PC AND AN HP TOUCHSMART PC

Issue	Implication
Right-click events do not make sense on a touch screen.	An alternate way (provided by the HP machine) to capture the right-click event on the touch screen was 'press-&-hold'.
The tip of the mouse cursor is tiny and accurate compared to the tip of a finger.	All GUI objects have to be larger to accommodate the finger touch more precisely.
When applying a touch on the vertical surface, the body of the finger covers some content on the screen (Figure 2a).	A vertical slider that was used to control the intensity of a light was changed into a horizontal slider (Figure 2b).

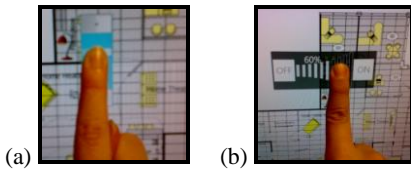


Figure 2. (a) part of the vertical slider is blocked by the body of the finger. (b) the horizontal slider solves this issue.

As mentioned earlier, the development for normal PCs and HP TouchSmart PCs was the initial stage in the evolution of eHome. At that stage, the architecture of eHome looked like the one in Figure 3a. The Presentation layer included all the view-related elements, whereas the UI Controller managed the communication between the Presentation layer and the Data Object Model. The Hardware Controller was responsible for communication between the actual hardware devices with the Model or the UI Controller. External Resources included the hardware devices, XML configuration files, and web services.

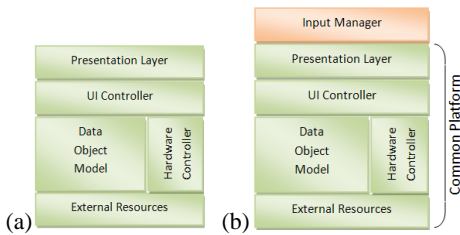


Figure 3. eHome architecture (a) before and (b) after considering variability at the Input Manager layer.

At first when we only considered the first issue (right-click vs. press-&-hold) as a source of variability, a conceptual layer was added to reflect this variability as shown in Figure 3b (previously, input was managed within the Presentation layer). The common platform included everything but the Input Manager where variability occurred. One variation point was defined as "input mechanism" with the two variants "mouse" and "touch." Later, when the other two issues were to be managed, variability penetrated down to the Presentation layer as shown in Figure 4. That is, the variability profile we had so far could be described as:

Input Mechanism = {mouse, touch}

Mouse = [scale factor x, vertical slider, right-click]

Touch = [scale factor y, horizontal slider, press-&-hold]

Variability between Vertical & Horizontal Displays

To migrate eHome from a vertical surface to a horizontal one, we initially deployed eHome on a horizontal display without any modification to understand the differences. After a number of usability observations and going back and forth between the vertical and horizontal settings, we realized a raft of issues. Table 2 (on page 7) lists these issues and their implications on the migration process. In this paper, we do not argue that these implications improved usability as this is yet to be appraised. The point, however, is that usability issues introduced new sources of variability. At this stage, we realized new variability occurring at the same two layers of the architecture. Not only did we have to go back and modify the variability we had previously defined in the Input Manager, but we also needed to explicate more variability in the Presentation layer. All the other layers were left intact. The updated variability profile included the following:

Input mechanism = {mouse, single touch, multi-touch}

Mouse = [right-click], Single-touch = [press-&-hold],

Multi-touch = [press-&-hold, two-touch-zooming, gesture support]

Layout = {normal PC, TouchSmart PC, digital tabletop}

Normal PC = [scale factor x, vertical slider, conventional GUI controls, textual feedback]

TouchSmart PC = [scale factor y, horizontal slider, conventional GUI controls, textual feedback]

Digital tabletop = [scale factor z, circular slider, redundant GUI controls, text-less feedback]

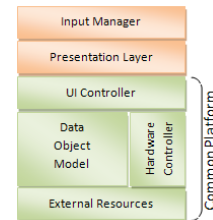


Figure 4. Architecture after considering variability at the Presentation layer.

Variability within Horizontal Displays

In the previous sections, we discussed variability due to differences between vertical displays. We then discussed variability due to the migration of eHome from a vertical display into a horizontal one. This section will discuss variability that was due to differences between horizontal displays. By horizontal displays, we namely refer to three hardware platforms: SMART DVIT Table, New SMART Table, and Microsoft Surface. As illustrated in Table 3 (on page 7), we dealt with three different SDKs, two of which were different versions from the same vendor. The first tabletop on which eHome was deployed was the SMART DVIT Table. We utilized the dual-touch capability of this table by adding a feature that allowed the user to place two touch points on the floor plan to zoom in and out. This kind of interaction required the hardware platform to support at least two simultaneous touches, which made the interaction irrelevant to the previous hardware platforms. For this reason, we chose not to include

this interaction with the rest of the interactions in eHome that were common to all platforms.

Rather, a specialized controller was introduced in the UI Controller layer to manage all communication between eHome and the touch handlers in the SMART SDK, as shown in Figure 7 – A. By this separation, it was easier to plug this feature in and out. The new controller was responsible for managing three events, namely: TouchDown, TouchUp and TouchMove. In case the touch events were part of a zooming interaction, the specialized controller will handle the zooming. Otherwise, the touch events were rerouted to mouse events we had previously defined in the UI Controller for the previous platforms in order to maximize code reuse and avoid code redundancy.



Figure 5. eHome on a horizontal display has redundant GUI elements to support multiple orientations.

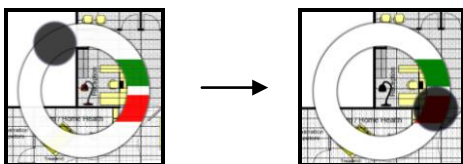


Figure 6. Circular slider to control light intensity

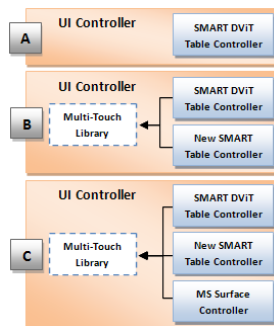


Figure 7. The evolution of variability due to differences in the SDKs

The second step was deploying eHome on the New SMART Table. The New SMART Table came with its own SDK, and the technology was different from the older table. Therefore, a new specialized hardware controller was also created to manage communication between eHome and the touch handlers in the new SMART SDK. At this stage, we had two different controllers one for each table. These controllers, however, shared common aspects such as the main triggering events and the zooming interaction. These common aspects were abstracted in a new layer we called “Multi-Touch Library” as shown in Figure 7 – B. The new layer was abstracted in a way so that it was completely agnostic to the target hardware platform – all specificities were kept in the specialized controllers.

Later on, this abstraction served well in accommodating the new digital tabletop – MS Surface. That is, it only took about one day worth of work to deploy eHome on MS Surface, because all we needed to do was create a new specialized controller to communicate with the Surface SDK, while all other aspects were managed by the Multi-Touch Library. Figure 7 – C shows the final organization. As was done before, variability was evolved to include a new layer, namely the UI Controller layer. The following variation point was added to the variability profile:

Multi-Touch SDK = {SMART DVIT Table, New SMART Table, MS Surface}

SMART DVIT Table = [old SMART SDK], New SMART Table = [new SMART SDK], MS Surface = [Surface SDK]

V. DISCUSSION

In the previous sections, we discussed an approach to reactively manage variability in systems using agile practices. We also reported a case study where we used the approach to manage variability in an application that was to be deployed on a number of different hardware platforms. In this section, we discuss the advantages and limitations of our approach as learned from the case study.

A. Opportunistic Reuse of Code and Test Artifacts

In the case of eHome, about 60% of the code (production and testing) is reused amongst all platforms. This figure could even be higher for systems that have a thinner presentation layer than the one in eHome. Maximizing reuse is desirable because it lessens the time and effort to produce new products and maintain existing ones. For instance, if the underlying technology for a certain feature (e.g. item tracking) changes, we need to make the proper modification in the common platform only once. Then we re-instantiate different products for the five different platforms we support. Also, say a vendor produced a new digital tabletop technology. All the work we need to do is at the UI Controller layer. The common platform can be used as is without any changes. However, this flexibility to change, adapt and reuse is achieved through a good understanding of the variability profile of the product line – which makes explicating and managing variability essential.

B. Explicating and Managing Variability

Adopting a SPL practice provides a systematic approach to think about and handle variations in the family. That is, before deciding to support a new digital tabletop platform, we need to know what is different about the new platform that cannot be supported by the existing product line. If there is any difference, then decisions need to be taken on where in the architecture this variation should be accommodated and what impact it will have on other platforms in the family. Without having an explicit variability profile of the SPL, taking such decisions becomes more difficult and is accompanied with higher risks. More importantly, with the variability profile the instantiation process of different products can be formalized by looking at each product in the product line as a function of the variation points. That is, any product P in the family is formalized as:

$$P = f(vp_a, vp_b, \dots) = f(\{v_{1a}, v_{2a}, \dots\}, \{v_{1b}, v_{2b}, \dots\}, \dots)$$

Where vp : variation point, v : variant, $\{\}$: OR operator

For instance, let's consider the variability profile of eHome. To produce a product that is specific to the HP TouchSmart




PC, we need to specify the variants as: Input mechanism > Single-touch = [press-&-hold]

Layout > TouchSmart PC = [scale factor y , horizontal slider, conventional GUI controls, textual feedback]

TABLE 2. ISSUES LEADING TO VARIABILITY BETWEEN VERTICAL AND HORIZONTAL DISPLAYS

Issue	Implications
Horizontal displays are, typically, physically larger than vertical ones.	A new scaling adjustment factor is defined for UI objects to make them bigger, and hence easier to interact with, on larger displays.
Horizontal displays deal with multiple touch points not only single touch points or mouse clicks.	This new input mechanism needs to be incorporated into the Input Manager layer as a new variant.
Conventional GUI elements like buttons, menus and tabs were oriented in a top-down fashion, which for a horizontal surface did not seem natural because people sit on different sides of the table.	The conventional GUI elements were replaced by panels available on each of the four sides of the tabletop, as shown in Figure 5. Instead of one Exit button on the top left corner of the screen, an Exit button was added on each corner of the tabletop. The "change mode" button (user/designer) was removed. Instead, the change of mode on the digital tabletop happens automatically.
Feedback to the user was provided using a status bar at the bottom of the screen, which was not suitable for a multi-oriented surface (i.e. horizontal display).	Alternative ways to provide feedback were used. For example, when a certain operation executes successfully, the corresponding icon on the surface glows.
When using a slider control, vertical and horizontal sliders seemed counterintuitive if there were people sitting around the table (e.g. if you go up in a vertical slider, it seems as if you are going down for a person sitting opposite to you).	A circular slider was used with clearly flagged ON/OFF positions, as shown in Figure 6. Regardless of where you sit around the table, if the handle of the slider is moving towards the ON button, then the intensity is increasing and vice versa.
Some features were not readily easy to use for everybody around the table because the UI controls were closer to a certain part of the screen.	For deleting an object, instead of a single trash can on the bottom right corner of the screen, if the user touches an object while in the designer mode, the user has the option to drag it to any of the trash cans distributed on the corners of the screen.
Readability of text on the horizontal display was limited because of the presumed top-down orientation.	The horizontal interface includes far less text than the vertical one. Descriptive icons and UI controls, animations, as well as visual cues like pulsation or glowing are used to replace text.
With multi-touch capabilities, horizontal displays provided new interactions that were not possible on vertical displays (This was specific to our case – new versions of the HP TouchSmart PCs support dual-touch interactions).	On horizontal displays, it was made possible to zoom in and out of the floor plan using two finger touches.
On a big scale tabletop, drag-and-drop became difficult due to the physical limitations on the reach of an arm.	Gestures were made available as additional (not substitutional) ways of executing certain features. For example, to delete an object, one can use a scratch gesture.

TABLE 3. DIFFERENCES BETWEEN THE SMART DVIT TABLE, NEW SMART TABLE, AND MS SURFACE

	Picture	Issues			Implications
		Dimensions <i>L x W cm</i>	Touch Points	SDK	
SMART DVIT Table		240 x 100	2	SMART SDK old version	<ul style="list-style-type: none"> - The aspect ratio of the SMART DVIT Table is 2.4 (compared to 1.33 for the New SMART Table and 1.56 for MS Surface). This introduced challenges in treating all four sides of the table equally. For example, instead of four panels, we only put two panels, one on each long side of the SMART DVIT Table, because the floor plan could not rotate with its full size except for a full 180 degrees. - An abstraction layer was introduced to embrace the different ways the SDKs deal with touch points. - Although we did not encounter this problem, we anticipated that when we add features that should support collaborative work, the limitation of 2 touch points might be a source of variability.
New SMART Table		55.9 x 41.9	40	SMART SDK new version	
MS Surface		108 x 69.0	Large number - exact number unknown	Surface SDK	

$$\begin{aligned} \text{Or: } P_{\text{TouchSmart PC}} &= f(\text{input mechanism, layout}) \\ &= f(\text{single-touch, TouchSmart PC}) \end{aligned}$$

This formal representation is then fed to the SPL through a configuration file or any other mechanism in order to start the instantiation of a specific product.

C. The Ability to Form Combinations

One more advantage of the systematic treatment of variability is the ability to combine different variants to come up with diverse products. For example, suppose we want to support the new HP TouchSmart PC that enables two simultaneous touches. We can come up with a new combination of variants to add the zooming behavior:

Input mechanism > Multi-touch = [press-&-hold, two-touch-zooming, gesture support]

Layout > TouchSmart PC = [scale factor y, horizontal slider, conventional GUI controls, textual feedback]

$$\text{Or: } P_{\text{New TouchSmart PC}} = f(\text{multi-touch, TouchSmart PC})$$

That is, by choosing a different variant for a given variation point, we ended up with a different product for the new platform. Constraints are usually defined to filter out invalid combinations.

We understand that some of these advantages are inherited from the SPL practice itself. However, it is imperative to point out that using our iterative approach allows organizations to realize the same advantages in a way that is more cost effective (because it is lightweight) and less risky (because it minimizes speculation), and with a faster return on investment (because systems are continuously delivered as opposed to waiting until the application engineering phase).

D. Limitations

The main limitation of our approach is that there is no clear definition of the roles needed in the different steps. For example, who in a typical agile organization should conduct the variability analysis? Can developers assume the responsibility of updating the variability profile? This is vital because variability analysis and profiling require a wide knowledge of existing requirements in the system. Therefore, a developer who only worked on a certain aspect of the system may not be qualified for this role. A second concern we had about the proposed approach is the amount of discipline needed to implement the approach successfully. For example, the approach relies on the premise that tests are written for all features in the system and that sufficient test coverage is available. In our case, eHome had an automated testing coverage as high as 90% of the model code. We also defined a suite of regression tests to be conducted manually to test UI and hardware related issues. We are not sure what the consequences are if good testing practices are not present in the organization. We intend to conduct a more systematic evaluation in the fall of 2010 in order to draw more reliable conclusions on the advantages of our approach as well as its limitations.

VI. CONCLUSION

The general goal of our research is to reconcile conflicts between traditional SPL engineering and agile software development. This paper contributes a framework that allows agile organizations to reactively construct variability profiles for existing and new systems. The framework leverages common agile practices such as iterative software development, refactoring, continuous integration and testing to introduce variability into systems only when it is needed. We showed, by example, how to use the proposed approach, and we discussed the advantages that can be realized, and the limitations that may hinder successful adoption of the approach. Future work includes evaluating the approach in an agile organization to form a better understanding of the practicality and feasibility of the approach.

REFERENCES

- [1] 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.
- [2] Buhrdorf, R., Churchett, D., and Krueger, C., Salion's Experience with a Reactive Software Product Line Approach, Revised Papers of the 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003.
- [3] Carbon, R., Lindvall, M., Muthig, D., and Costa, P., Integrating PLE and agile methods: flexible design up-front vs. incremental design", The 1st International Workshop on APLE, 2006 - SPLC.
- [4] Clegg, K., Kelly, T., McDermid, J., Incremental Product-Line Development, International Workshop on PLE, Seattle, 2002.
- [5] Clements, P., and Northrop, L., *Software Product Lines: Practice and Patterns*, Addison-Wesley, 2001.
- [6] Cooper, K., and Franch, X., "APLE 1st International Workshop on Agile Product Line Engineering", SPLC, 2006.
- [7] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts D., *Refactoring: improving the design of existing code*. Addison-Wesley, 1990.
- [8] Ghanam, Y., and Maurer, F., Linking Feature Models to Code Artifacts using Executable Acceptance Tests, to appear in the proceedings of the 14th International Software Product Line Conference (SPLC 2010), South Korea, September 2010.
- [9] Hanssen, G., and Fægri, T., Process Fusion: An Industrial Case Study on Agile Software Product Line Engineering, special Issue of Journal of Systems and Software (JSS), 2008.
- [10] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A., FODA Feasibility Study, SEI Technical Report, 1990.
- [11] Kruger, C., Easing the Transition to Software Mass Customization, in Proceedings of the 4th International Workshop on Product Family Engineering, Germany, 2002.
- [12] Leffingwell, D., *Scaling Software Agility: Best Practices for Large Enterprises*, Addison-Wesley Professional, 1st edition, 2007.
- [13] McGregor, J. Agile Software Product Lines, Deconstructed, *Journal of Object Technology*, 7(8), 2008
- [14] Paige, R., Xiaochen, W., Stephenson, Z., and Phillip J., Towards an Agile Process for Building Software Product Lines, XP 2006, 198 - 199.
- [15] Schmid, K., and Verlage, M., The Economic Impact of Product Line Adoption and Evolution, *IEEE Software*, 19 (4), pp. 50-57, 2002.
- [16] Shalloway, A., Beaver, G., and Trott, J., *Lean-Agile Software Development: Achieving Enterprise Agility*, Addison-Wesley Professional, 1st edition, 2009.