Extreme Product Line Engineering – Refactoring for Variability: A Test-Driven Approach

Yaser Ghanam and Frank Maurer

Department of Computer Science
2500 University Dr. NW, Calgary
Alberta, Canada T2N 1N4
{yghanam, fmaurer}@ucalgary.ca

Abstract. Software product lines - families of similar but not identical software products - need to address the issue of feature variability. That is, a single feature might require various implementations for different customers. Also, features might need optional extensions that are needed by some but not all products. Software product line engineering manages variability by conducting a thorough domain analysis upfront during the planning phases. However, upfront, heavyweight planning approaches are not well-aligned with the values of minimalistic practices like XP where bottom-up, incremental development is common. In this paper, we introduce a bottom-up, test-driven approach to introduce variability to systems by reactively refactoring existing code. We support our approach with an eclipse plug-in to automate the refactoring process. We evaluate our approach by a case study to determine the feasibility and practicality of the approach.

Keywords: variability, software product lines, agile methods, refactoring.

1 Introduction

1.1 Software Product Line Engineering

Software product line (SPL) engineering enables organizations to manage families of products that are similar but not identical [1]. The idea is to target a certain market domain, but be able to satisfy the various requirements of different market sectors (i.e. customers) in that domain. This is achieved by developing a common platform from which different instances of the system are derived. Being able to customize the products during the instantiation process is due to the concept of variability [2].

Variability refers to the notion that the components that constitute the system may exist or behave differently in different instances of that system. Moreover, some components may be optional and, hence, may not exist in every instance of the system. Every product line has a variability profile that encompasses a number of variation points and a set of variants for each variation point. A variation point is an aspect in a certain requirement that can have multiple states of existence in the system. Each state of existence is called a variant.

Consider the example in Figure 1a. The weather module has two variation points. The first is of type “option” – the weather trend analyzer. Options are either selected (variant 1) or discarded (variant 2). That is, the weather trend analyzer is optional depending on whether the customer would like to have it or not. The second variation point is of type “alternatives” – the weather UI panel. Alternatives are mutually exclusive – if one is selected, the other alternatives cannot be selected. That is, the weather UI panel can have one of two different formats depending on whether the application is to run on a handheld device (variant 1) or a normal PC (variant 2).

When the customer purchases a smart home system with the weather module, a decision has to be made on what variants should be selected for every variation point in the system. Based on these decisions, different instances can be produced such as the ones shown in Figure 1b. Decision making is usually governed by the wants and needs of the customer, as well as any technical or business constraints.

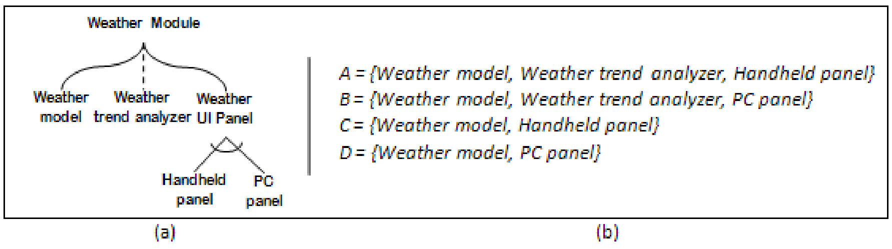

Weather Module
Weather model
Weather trend analyzer
Weather UI Panel
Handheld panel
PC panel
A = {Weather model, Weather trend analyzer, Handheld panel}
B = {Weather model, Weather trend analyzer, PC panel}
C = {Weather model, Handheld panel}
D = {Weather model, PC panel}
(a)
(b)

Fig. 1. Example of variability in a module.

Organizations that use product line practices to manage their software production have a competitive advantage. Some organizations 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 [3]. This is because dealing with a single configurable entity that is considered the base for all products in a given category provides for many benefits such as:

a. Software Reuse: Reuse of the same source code to produce a number of systems provides an economic advantage.
b. Increased software stability: Reused components can be better tested as the effort amortizes over a larger number of software systems.
c. Mass customization: Mass customization means that a wider range of customers can be satisfied even though their needs might be different.
d. Reduced maintenance effort: Maintaining a single base is more efficient than maintaining separate products. For example, if a bug needs to be fixed, the change should only happen within the single base, and the change will take effect in all individual products as soon as they are re-instantiated.

1.2 Software Product Lines and Agility

In the previous section, we explained what a SPL is and why it is a useful practice. This section discusses SPLs in the context of agile software development.

Traditionally, SPL engineering favors upfront, proactive approaches. Planning for variability is done during the domain engineering phase in order to determine what may vary and how. The variability profile is then taken into account to build a flexible reference architecture for all the instances to be built in the future. During the next phase, namely application engineering, customized instances of the system are derived and delivered to the customers.

For agile organizations, adopting a SPL approach in its traditional form is challenging. This is due to the common values and principles [4] in the agile culture that can be seen as being in direct conflict with essential aspects of SPL engineering. For one, agile methods encourage working on what is currently needed and asked for rather than what might be needed in the future. Therefore, dedicating an entire domain analysis phase to make predictions about variability might be a risky option. Furthermore, in the agile culture, it is deemed important to deliver working software on a regular basis – which in a SPL context is too difficult to achieve especially in the early phases.

In this paper, we argue that for agile organizations to adopt a SPL practice, a reactive – as opposed to proactive – approach is more befitting. We focus on the notion of variability considering that it is one of the most important aspects of SPLs. We describe a bottom-up, reactive approach to gradually construct variability profiles for existing systems. The approach relies on common agile practices such as refactoring and test-driven development to introduce variability into systems only when it is needed. The approach can provide the reuse and maintainability benefits of SPLs while keeping the delivery-focused approach coming from agile methods.

The next section describes the proposed approach in detail. In Section 3, we present an evaluation of the approach followed by a discussion of limitations. Section 4 discusses relevant literature and underscores our distinct contribution.

2 The Extreme Software Product Line Approach

2.1 Overview

The fundamental verdict of our approach to variability in the agile context is that variability should be handled on-demand. That is, unless requirements about variations in the system are available upfront, the agile organization should not proactively invest into predicting what might vary in the system. Rather, the normal course of development should take place to satisfy the current needs of the customers. Later on, should a need to introduce a variation point arise – whether during development or after delivery – agile teams shall have the tools to embrace this variability reactively. When a product line practice is not adopted, embracing variability is usually done by: a) clone-and-own techniques where the base code is copied and then customized to satisfy the new variation, or b) ad-hoc refactoring, where it is left up to the developer to refactor existing code to satisfy both the new as well as the existing variation. In the first case, the organization will have to maintain and support more than one base code, which is highly inefficient and error prone. In the second case, there is neither a systematic way to refactor the code nor a way to convey knowledge about the existence of variation points – which may cause variability in the system to become too cumbersome and expensive to maintain, and may render the instantiation process vague.

The approach we propose here is different in that it enforces systematic and test-driven practices to deal with variability in order to ensure consistency and efficiency.

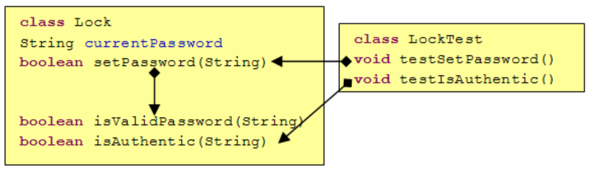
2.2 The Role of Tests

In agile approaches like Extreme Programming [5], automated tests are deemed essential. There usually exist two types of tests: unit tests and acceptance tests. Our approach makes use of both types; however, this paper focuses on the use of unit tests only. A unit is defined as the smallest testable part of a system. A unit test verifies the correctness of the behavior of an individual unit, or the interaction between units. In test-driven development, unit tests are written before writing production code. Tests are automated to be executed frequently and help in refactoring of the code base.
In our approach, unit tests are relevant in three different ways:

1. Unit tests are used as a starting point to drive the variability introduction process. This point will be discussed further in the upcoming sections.
2. When a variation point is introduced along with its variants, unit tests ought to exhaust all the different variants, and therefore they are part of the variability introduction process.
3. Unit tests serve as a safety net to make sure the variability introduction process did not have any destructive side effects.

2.3 The Variability Introduction Process

To illustrate the proposed approach, we use a very simple example. Say, within a smart home security system, we have an electronic lock feature on every port (door or window). The diagram in Figure 2 illustrates the current state of the system. The Lock class is tested by the LockTest class. Arrows show the call hierarchy. E.g. LockTest.testSetPassword() calls the public method Lock.setPassword(), which in turn calls the private method Lock.isValidPassword(String).



```
class Lock
String currentPassword
boolean setPassword(String)


boolean isValidPassword(String)
boolean isAuthentic(String)
```

```
class LockTest
void testSetPassword()
void testIsAuthentic()
```

Fig. 2. Current state of the Lock feature.

Currently, the system allows the user to set a 4-character password for the locks. The password criteria are checked in the Lock.isValidPassword() method below. Say we need to introduce a variation point to the lock feature to allow customers to choose the desired security level needed on the locks before they purchase the system. Variants include a 4-char lock (the original lock), an 8-char lock (a new alternative), or a non-trivial 8-char lock (another new alternative - characters cannot be all the same and cannot be consecutive).

```
class Lock {
      String currentPassword="";

      public boolean setPassword(String password) {
              if(isValidPassword(password)) {
                      this.currentPassword = password;
                      return true;
              }
              return false;
      }

      boolean isValidPassword(String password) {
              if (password.length()==4) return true;
              return false;
      }


      public boolean isAuthentic(String password) {
              if(password == currentPassword) return true;
              return false;
      }
}
```

One way[1] to design this is to use an abstract factory pattern [6] to reach the configuration shown in Figure 3. We can abstract the method that is responsible for password validation (i.e. Lock.isValidPassword(String)) and provide three different implementations for it.

Our tool supports this refactoring process: when a developer wants to introduce a variation, the tool helps in the required refactoring of the code base. It creates an abstract factory, corresponding concrete classes, and the required test classes.

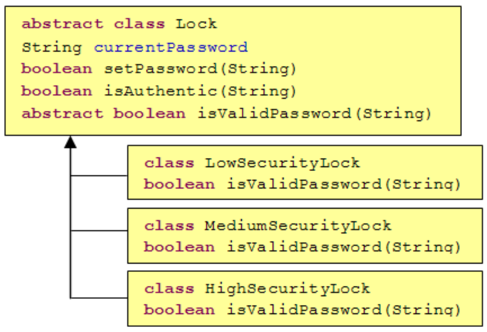

abstract class Lock
String currentPassword
boolean setPassword(String)
boolean isAuthentic(String)
abstract boolean isValidPassword(String)
class LowSecurityLock
boolean isValidPassword(String)
class MediumSecurityLock
boolean isValidPassword(String)
class HighSecurityLock
boolean isValidPassword(String)

Fig. 3. The new state of the Lock feature

[1] There are other techniques and patterns to realize variability. Choosing the appropriate technique for each case is beyond the scope of this paper. Our approach, however, should work fine with any code-based variability implementation technique.

Refactoring the code to reach the configuration in Figure 3 has consequences. First of all, we need to write unit tests that reflect the new changes. Also, we need to change all instantiations of the old Lock class to use the new factory instead. And we need to determine which implementation we want before every instantiation of the Lock class. For this to work, we also need to provide an implementation selector class to return the proper implementation.

To take care of the refactoring process and all its consequences, we take advantage of the existing traceability between unit tests and production code. We provide a formalization of the variability introduction process in order to automate it:

There exists in the system a set of unit test classes C_u. Each testing class c_{ui} has a set of testing methods.

$$C_u = \{c_{u1}, c_{u2}, c_{u3}, \dots, c_{ui}\}$$
$$c_{ui} = \{m_{ui1}, m_{ui2}, m_{ui3}, \dots, m_{uij}\}$$

Each testing method m_{uij} may instantiate or use a number of classes in the system - set C. Each class c_k consists of a set of methods.

$$C = \{c_1, c_2, c_3, \dots, c_k\}$$
$$c_k = \{m_{ck1}, m_{ck2}, m_{ck3}, \dots, m_{ckn}\}$$

Each method in c_k may instantiate or use a number of other classes in the system.

Over the previous sets, we define a process to introduce a variation point to the system. The process consists of the following functions:

1. A Variation Initialization function to determine:
   a. The unit test of interest as a starting point $m_{uij} \in c_{ui}$. This feeds into the next step.
   b. One of two variation types: *alternatives*, or *options*. This determines the kind of refactoring needed to embrace the variation point.

   These two attributes should be determined by the developer. The developer chooses the unit test that tests the scenario where variability needs to exist. In the example above, it is the LockTest.testSetPassword() method shown below[2], because this is where we mainly test for the setting password part of the feature.

```
public void testSetPassword() {
     Lock lock = new Lock();
     Assert.assertFalse(lock.setPassword(""));
     Assert.assertFalse(lock.setPassword("Hello"));
     Assert.assertTrue(lock.setPassword("Helo"));
}
```

   Then the developer decides whether the new variability is due to the need to provide alternative implementations or to add options to the feature at hand. In the example, we choose alternatives.

[2] This unit test is for illustration only. It is understood that in real life best practices like one-assert per test should be observed.

1. A Call Hierarchy function to determine the transitive closure of the unit test m_{uij} selected in step 1. This includes all methods in the system that are invoked due to the invocation of m_{uij}.

   In the example above, the call hierarchy of LockTest.testSetPassword() includes the methods: Lock.setPassword(String) and Lock.isValidPassword(String).

   At this stage, developer's input is needed to identify where in the call hierarchy the variation point should exist. This determines the method that is causing the variation to happen. For example, because the variation point we need to inject is pertaining to the validation of the password criteria, we choose Lock.isValidPassword(String).

2. A Variability Trace function that – given the method m_{ckn} chosen in 2 – determines all the classes and methods that can potentially be affected by introducing the variation point.

   In the example above, say there is a class Port that instantiates the Lock class. This instantiation needs to be updated to use the new factory.

3. A Refactoring and Code Generation function to perform the code manipulations needed to introduce the variation point and the variants based on the variation type determined in 1b. Given the method $m_{ckn} \in c_k$ from the previous step, the following code manipulations will take place:
   a. Refactoring c_k so that m_{ckn} is abstracted as a variation point.

```
abstract class Lock {
      String currentPassword="";
      public boolean setPassword(String password) {
              if(isValidPassword(password)) {
                      this.currentPassword = password;
                      return true;
              }
              return false;
      }
      abstract boolean isValidPassword(String password);
      public boolean isAuthentic(String password) {
              if(password == currentPassword) return true;
              return false;
      }
}
```

   b. Generating implementations for the variants:

```
  class LowLock extends Lock {
       boolean isValidPassword(String password) {
              if (password.length() == 4)
                     return true;
              return false;
       }
  }
  class MediumLock extends Lock {
       boolean isValidPassword(String password) {
              // TODO Auto-generated method stub
              return false;
       }
  }
  class HighLock extends Lock {
       boolean isValidPassword(String password) {
              // TODO Auto-generated method stub
              return false;
       }
  }
```

c. Declaring a new enumeration to explicate the variation point and its variants:

```
public enum VP_SECURITY_LEVEL { V_LOW, V_MEDIUM, V_HIGH }
```

d. Creating/updating a configurator class:

```
public class VariantConfiguration {
       public static VP_SECURITY_LEVEL securityLevel =
                                      VP_SECURITY_LEVEL.V_LOW;
   }
```

The configurator serves two purposes. For one, it enables easy configuration and instantiation of products. Every variable in this class represents a variation point. The value assigned to each variable represents the variant we need to choose. Secondly, the configurator helps explicate the variability profile of the system so that it is visible to the stakeholders (later, we plan to link this profile to feature modeling tools to provide a better visualization).

e. Generating an implementation selector:

```
public class LockFactory {
       public static Lock createLock() {
              if (VariantConfiguration.securityLevel ==
VP_SECURITY_LEVEL.V_LOW) return new LowLock();
              if (VariantConfiguration.securityLevel ==
VP_SECURITY_LEVEL.V_MEDIUM) return new MediumLock();
              if (VariantConfiguration.securityLevel ==
VP_SECURITY_LEVEL.V_HIGH)     return new HighLock();
              else return null;
       }
   }
```

f. Updating affected code segments found in step3 to use the new factory:

```
Lock lock = LockFactory.createLock();
```

4. A Test Update function to update affected unit tests and generate unit tests for the new variants. This not only makes sure the new changes did not have a destructive effect on the original system, but also encourages test-driven development because it generates failing tests for developers to write before writing the logic for the new variants.

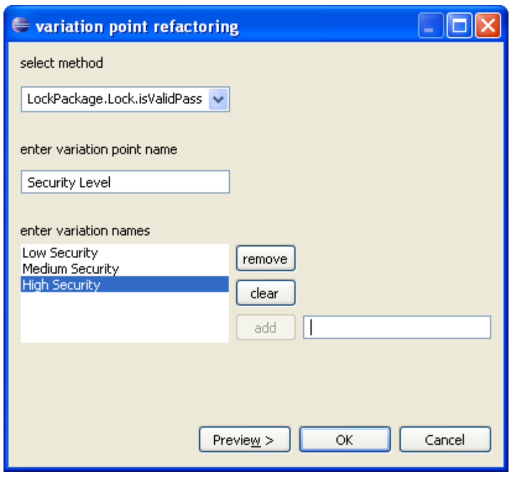
   In the example above, the LockTest.testSetPassword() method is refactored to LockTest.testSetPassword_Low() as a test for the first (original) variant. Two more tests are added to test the other two variants. In each test, the first statement selects the variant to be tested. In the case of *options*, we generate tests for all combinations of options. This of course has to go through a validation engine to filter invalid combinations but this is yet to be implemented in the tool. Generated tests initially have the same body of the original test (where the process started) as a template for the developer to change as required. However, these tests initially are forced to fail to remind the developer to edit the test and its corresponding production code.

```
        @Test
        public void testSetPassword_Low() {
                VariantConfiguration.securityLevel =
                                        VP_SECURITY_LEVEL.V_LOW;
                Lock lock = LockFactory.createLock();
                Assert.assertFalse(lock.setPassword(""));
                Assert.assertFalse(lock.setPassword("Hello"));
                Assert.assertTrue(lock.setPassword("Helo"));
        }
        @Test
        public void testSetPassword_Medium() {
                // TODO Auto-generated method stub
                VariantConfiguration.securityLevel =
                                        VP_SECURITY_LEVEL.V_MEDIUM;
                Lock lock = LockFactory.createLock();
                Assert.assertFalse(lock.setPassword(""));
                Assert.assertFalse(lock.setPassword("Hello"));
                Assert.assertTrue(lock.setPassword("Helo"));
                org.junit.Assert.fail();
        }
        @Test
        public void testSetPassword_High() {
                // TODO Auto-generated method stub
                VariantConfiguration.securityLevel =
                                        VP_SECURITY_LEVEL.V_HIGH;
                Lock lock = LockFactory.createLock();
                Assert.assertFalse(lock.setPassword(""));
                Assert.assertFalse(lock.setPassword("Hello"));
                Assert.assertTrue(lock.setPassword("Helo"));
                org.junit.Assert.fail();
        }
```

2.4 Automation

The abovementioned process to introduce a variation point in a system entails a number of steps that can be error prone and time consuming. We built an eclipse plug-in that automates the whole process assisted by input from the developer. The tool is open source and is available online [7]. When a variation point is to be refactored into the system:

1. The developer navigates to the unit test corresponding to the aspect of the feature where the variation point should be added.
2. The developer chooses to add a variation point of a certain type.
3. The tool finds the transitive closure of all objects and methods used in the chosen unit test. The developer selects the method that is considered the source of variation.
4. The developer specifies a name for the new variation point, and specifies names for all different variants. Figure 4 shows a snapshot of a expected input.



select method: LockPackage.Lock.isValidPass

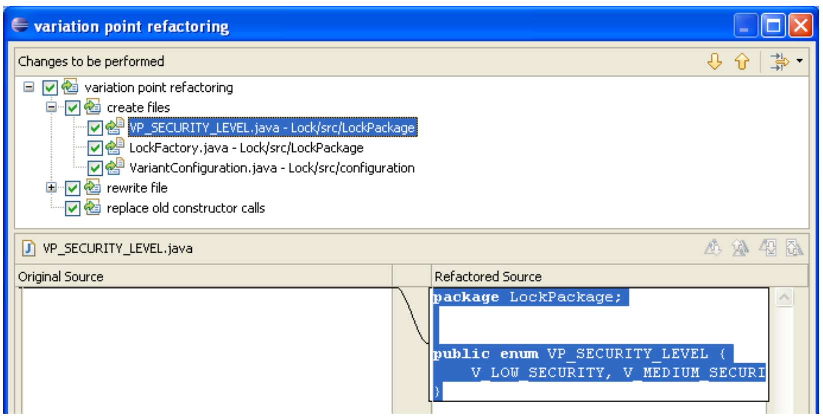
enter variation point name: Security Level

enter variation names:
Low Security
Medium Security
High Security

Fig. 4. Expected input from the developer

5. The tool will do the proper refactoring and code generation as described in the previous section. In fact, all the refactored and generated code we provided at each step in the previous section was the output of our eclipse plug-in. Namely, the tool will:
   - Abstract out the source of variation.
   - Provide an implementation class for each variant.
   - Provide a factory to select the proper implementation.
   - Define an enumeration to enable easy configuration of the system at instantiation time. All variation points will be packaged nicely in a configuration file to convey knowledge about variability in the system and the decisions that need to be made.

As shown in Figure 5, before any refactoring takes place, the developer is made aware of all the changes.



- variation point refactoring
  - create files
    - VP_SECURITY_LEVEL.java - Lock/src/LockPackage
    - LockFactory.java - Lock/src/LockPackage
    - VariantConfiguration.java - Lock/src/configuration
  - rewrite file
  - replace old constructor calls

VP_SECURITY_LEVEL.java

```
package LockPackage;


public enum VP_SECURITY_LEVEL {
    V_LOW_SECURITY, V_MEDIUM_SECURI
}
```

Fig. 5. The developer is made aware of the refactoring steps

6. The tool will update all references of the old class to the correct object instantiation technique.
7. The tool will provide unit tests for every variant. In case of options, unit tests will be provided to test all possible combinations of extensions.

3. Evaluation

So far in this paper, we described a bottom-up, test-driven approach to construct variability profiles for software systems. The previous sections provided an overview of what the variability introduction process entails and how to formalize it for automation purposes. We also presented an eclipse plug-in that we built as an enabling tool for developers to automate the process.

3.1 Goals and Setting

The goal of the evaluation was to check whether the proposed approach is feasible and practical with the help of the plug-in we developed. We conducted a limited case study to investigate these two issues.

To check for feasibility, we used the simple example we presented earlier in this paper as a starting point. Then, we extended the example in a number of ways, namely: adding more classes, adding more unit tests, applying various object-oriented techniques like subclasses and interfaces, and using different types of constructors like parameterless constructors, parameterized constructors and super constructors. After each extension to the mockup system, we tried to apply a number of operations such as adding variation points of different types, and adding variants. The main aim was to find how feasible it is to inject variability into a system through systematic refactoring given all the complications of object-oriented design.

On the other hand, the aim of the practicality check was to elicit an initial insight on how practical this approach is in terms of introducing variability to an arbitrary system that was not originally developed with variability in mind. For this part of the evaluation, we used a real open source system available at SourceForge. We selected this system by listing all projects in sourceforge.net, sorted by the number of downloads in a descending order, and with a Java as a programming language filter. Then we examined the projects one by one looking for those projects that already had unit tests and were easy to import by Eclipse Ganymede 3.4.2. The rationale behind these criteria was that we needed a system that is actually in use (to avoid experimental projects). Also, because our plug-in is written for Eclipse, it only supports Java. The availability of unit tests was also important, because we did not want to write our own tests to avoid any biases.

The system we chose is called Buddi [8]. It is a simple financial management program targeted for users with little or no financial background. It allows users to set up accounts, create spending and income categories, record transactions, and check spending habits. Up till the end of November, Buddi has been downloaded 647,354 times. It has about 24,775 lines and 227 classes.

Buddi was originally intended for personal use. In order to introduce variability to Buddi, we asked the question: what do we have to do to make Buddi suitable for small businesses? We developed a number of scenarios such as:

1. Buddi usually accepts credit payments (transactions). To use this feature to support store-credit, we need to provide the possibility of assessing risks. This yields the first variation point *Risk Assessment* with the variants (alternatives): *None*, *Flexible* and *Strict*. The *None* variant represents the original implementation of Buddi that did not have the notion of risk assessment. The *Flexible* variant puts some restrictions on the credited amount, and checks balance status for the past few months. The *Strict* variant adds more restrictions on any store-credit such as considerations of when the account was opened.
2. Buddi updates the balance of a customer after each transaction. For fraud-detection, we might need to take some security measures. This yields the second variation point *Security Measures* with the variants (options): *Log Balance Updates*, and *Send SMS to Customer*.

Following a product line approach, we can honor these requests without having to maintain a personal version as well as business version of Buddi. In our evaluation, the goal is to see how practical it is to use our approach to inject this variability using the Eclipse plug-in.

3.2 Results and Limitations

The feasibility evaluation enabled us to refine our approach in an iterative manner before we use our plug-in on a real system. We found some issues that we managed to fix such as dealing with instantiations where parameterized constructors are used, resolving call hierarchies and refactoring instantiations where there already is a design pattern in use, and dealing with classes that already are abstract. We also faced complications that are yet to be resolved such as handling inheritance. For example, if the original code was: Lock c = new SubLock() where Lock is not necessarily abstract, it is not possible for the static code analyzer we are using to detect at compile time which subclass is being used (if there is more than one). After a number of rounds, we came to the conclusion that the approach is indeed feasible, but more experiments need to be set in order to detect cases where systematic refactoring is tricky.

Checking for practicality, we used our plug-in to refactor the code systematically as per the proposed approach. The plug-in was in fact able to handle both types of variations (i.e. alternatives and options) without any human interference – except for input from the developer wherever it was prescribed in the approach. The output of using the approach for each variation was as expected. That is, relevant code was refactored and new code was generated as prescribed. The process did not create any compilation errors or cause a test to fail.

However, we noticed that the Call Hierarchy function caused a noticeable delay of about 9 seconds on average. Although this delay might not limit the usefulness of the tool, it might, for larger projects with millions of lines of codes, limit its practicality. We intend to investigate this issue further to find more efficient ways to obtain only relevant units rather than the whole hierarchy. Also, we realized that the transitive closure for some tests might be too large to navigate. This is a tool specific issue that we hope to deal with in the future where we can visualize the call hierarchy in a better fashion. A real limitation to the practicality of the current implementation of our plug-in was the inability to combine variation points in a hierarchical manner. For example, currently we do not support scenarios where a variation point of type alternatives need to be defined, and one or more of these alternatives have a number of options to select from. Nonetheless, we believe that this issue can be resolved with more sophisticated code analysis and refactoring.

Currently, the tool does not support dependencies between variation points and variants. For example, multiplicity constraints between alternatives and options are not taken into account. We are at this time working on extending the tool to handle constrains and support more complex variation scenarios.

3.3 Threats to Validity

The cases we chose to test in our feasibility evaluation might be subjective. Although we tried to cover a wide range of object-oriented configurations, we understand that our evaluation for feasibility did not exhaust all possible cases, which made it hard to claim that the approach is completely feasible for any project and under any circumstances. Moreover, the practicality evaluation is limited to one project only, which might threaten the validity of the generalization that the tool is practical for other projects. Also, the scenarios we generated might be subjective which might have biased the results.

Another obvious threat to validity is that we ourselves conducted the evaluation. Confirmation bias might exist. And the sample size is too small to even consider statistical validity.

We intend to collect a larger sample of projects of different scales and architectural styles in order to evaluate our approach more thoroughly. Furthermore, we think a controlled experiment will be useful to evaluate the approach from a developer's perspective.

4. Related Work

There is a large body of research on SPL engineering. Main topics of interest in this domain include scoping, domain engineering, reuse, and variability management. In our work, we deal with variability management, but we also focus on incremental introduction of product line practices to software organizations. Similar efforts in the literature include Kruger's work [9] on easing the transition to software mass customization. Kruger built a tool that utilizes the concept of separation of concerns to manage variability in software systems. Moreover, Clegg et al. [10] proposed a method to incrementally build a SPL architecture in an object-orientated environment. The method does not discuss how to extract and communicate variability from the requirement engineering phase to the realization phase. O‟ Brien et al. [11] discussed the introduction of SPLs in big organizations based on a top-down mining approach for core assets. They assume the organization has already developed many applications in the domain. None of these approaches follows test-driven practices.

Agility in product lines is a fairly new area. Carbon et al. [12] proposed the use of a reuse-centric application engineering process to combine agile methods and SPLs. The approach suggests that agile methods take the role in tailoring products for specific customers (during application engineering). The approach does not discuss the role of agile methods in the domain engineering phase. Hanssen et al. [13] described how SPL techniques can be used at the strategic level of the organization, while agile software development is used at the medium-term project level. While these efforts are interesting attempts to combine concepts from agile software development and SPL engineering, their focus is different from the focus of our research. They suggest using agile approaches in the application engineering/product instantiation phase but assume upfront domain engineering. Our work is focused on a lightweight, bottom up approach to start product lines in a non-disruptive manner using agile methods. To the best of our knowledge, this research focus is original and has not been previously discussed in the literature. The use of a test-driven approach for incremental SPLs was initially proposed in one of our earlier works [14].

5. Conclusion

SPL engineering provides organizations with a significant economic advantage due to reuse and mass customization opportunities. For agile organizations, there is a considerable adoption barrier because of the upfront, heavyweight nature of SPL practices. In this paper, we contribute a novel approach to introduce variability to existing systems on-demand. We propose a test-driven, bottom-up approach to create variability profiles for software systems. Systematic refactoring is used in order to inject variation points and variants in the system, whenever needed. We also contribute an Eclipse plug-in to automate this process.

A limited evaluation of the feasibility and practicality of the approach was presented. The approach, supported by the plug-in, was found to be feasible and practical, but suffered some limitations that we are currently trying to address.

Our next research goal is to find out how acceptance tests can play an effective role in this process. Since acceptance tests are usually used at the feature level, we hope to be able to use them as anchor points for variability in a given system.

Acknowledgment

This work is partially funded by iCore (Alberta Ingenuity Fund). Also, we would like to thank Steffen Salbinger for his great contribution to the development of the tool.

References

1. Clements, P., and Northrop, L. (2001) *Software Product Lines: Practice and Patterns*, Addison Wesley.
2. Gurp, J., Bosch, J., and Svahnberg, M., "On the Notion of Variability in Software Product Lines," *Working IEEE/IFIP Conference on Software Architecture (WISCA'01)*, 2001.
3. Klaus Schmid, Martin Verlage, "The Economic Impact of Product Line Adoption and Evolution," *IEEE Software*, vol. 19, no. 4, pp. 50-57, July/August, 2002.
4. "Manifesto for Agile Software Development," http://agilemanifesto.org/principles.html
5. Beck, K. and Andres, C. (2004) *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional.
6. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) *Design Patters: Elements of Reusable Object-Oriented Software*, Addison-Wesley.
7. https://fitclipse.svn.sourceforge.net/svnroot/fitclipse/trunk/ProductLineDesigner.
8. SourceForge, http://sourceforge.net/projects/buddi
9. Kruger, C., "Easing the Transition to Software Mass Customization", in *Proceedings of the 4th International Workshop on Product Family Engineering*, Germany, 2002.
10. Clegg, K., Kelly, T., and McDermid, J., Incremental Product-Line Development, International Workshop on Product Line Engineering, Seattle, 2002.
11. OBrien, L., and Smith, D., MAP and OAR Methods: Techniques for Developing Core Assets for Software Product Lines from Existing Assets, CMU/SEI-2002-TN-007, 2002.
12. Carbon, R., Lindvall, M., Muthig, D., and Costa, P. Integrating PL Engineering and Agile Methods: Flexible Design Up-front vs. Incremental Design, 1st International Workshop on Agile Product Line Engineering, 2006.
13. Hanssen, G., and Fægri, T., "Process Fusion: An Industrial Case Study on Agile Software Product Line Engineering", Journal of Systems and Software, 2008.
14. Ghanam, Y., Park, S. and Maurer, F., A Test-Driven Approach to Establishing & Managing Agile Product Lines. Proceedings of the 5th Software Product Line Testing Workshop (SPLiT 2008) in conjunction with SPLC 2008, Limerick, Ireland..