Feature-Driven and Incremental Variability Generalization in Software Product Line

Liwei Shen, Xin Peng and Wenyun Zhao

School of Computer Science, Fudan University, Shanghai 200433, China
{061021062, pengxin , wyzhao}@fudan.edu.cn

Abstract. In the lifecycle of a software product line (SPL), incremental generalization is usually required to extend the variability of existing core assets to support the new or changed application requirements. In addition, the generalization should conform to the evolved SPL requirements which are usually represented by a feature model. In this paper, we propose a feature-driven and incremental variability generalization method based on the aspect-oriented variability implementation techniques. It addresses a set of basic scenarios where program-level *JBoss-AOP* based reference implementations respond to the feature-level variability generalization patterns. It also provides the corresponding guidance to compose these patterns in more complex cases. Based on the method, we present a case study and related discussions.

1 Introduction

Software product line (SPL) engineering [1] promises to improve time-to-market, cost, productivity, and quality in a proactive mode, i.e. predicting and implementing all the application variations in advance through domain engineering. However, the proactive approach only suits organizations that can predict their product line requirements well into the future and have the time and resources for a long waterfall-like development cycle [2]. Most of the real SPLs are developed and maintained in a mixed mode of proactive and reactive approaches. Usually, an initial SPL platform involving one or several product variations will be developed first. After that, the SPL will evolve in an incremental and iterative mode. In each iteration, more product variations are recognized and included, and the SPL platform is extended and improved.

In the SPL lifecycle, incremental generalization is usually required to extend the variability of the existing core assets to support the new or changed application requirements. The term *generalization* named by Thum et al. [4] indicates the case when new products are added and no existing products are removed. It is a part of the term *refactoring* defined by Alves et al. [3] as “a change made to the structure of a SPL in order to improve its configurability, make it easier to understand, and cheaper to modify without changing the observable behavior of its original products”. In our work, we focus on variability increments, e.g. introducing new variation point or variant. It is the generalization that usually aims to add more variations based on the existing core assets. On the other hand, the generalization should conform to the evolved SPL requirements, which are usually represented by a feature model [4, 5, 6, 12, 17].

In this paper, we propose a feature-driven and incremental variability generalization method based on the aspect-oriented variability implementation techniques. We identify a set of basic scenarios which address the generalization patterns on the feature model, representing the requirement-level variability evolution, and provide program-level reference implementation based on *JBoss-AOP* for each of the pattern. Then, we can obtain incremental generalization guidance for more complex variability evolution by composing the basic feature model generalization patterns and deriving the integrated reference implementations.

The paper is structured as follows. Section 2 presents the background of variability generalization in the SPL. Section 3 introduces the basic generalization scenarios and their composition guidance. Section 4 demonstrates the method with a case study. Later in Section 5 and Section 6 we present discussions and related work respectively. Finally we conclude the paper and plan the future work in Section 7.

2 Background: Variability Generalization in SPL

In our method, we use the standard feature model as shown in Figure 1(a) to capture the feature-level variability evolutions. The three types of variation points are optional, alternative and OR features. On the program level, we employ *JBoss-AOP* as the reference implementation technique for the variations, since aspect-oriented programming (AOP) [7] has been proven an efficient way for the variability implementation [16]. Moreover, the obliviousness characteristic of AOP [8], which implies that the developer of the initial core assets need not to be aware of and prepare for the future variability evolutions, supports stepwise variability increment without intruding the existing programs.

JBoss-AOP [9, 10] is one of the most popular tools in the AOP community. In *JBoss-AOP*, an aspect is defined by an *interceptor* and a *pointcut*. The interceptor is the same as the advice and it is programmed as a Java class (implements *org.jboss.aop.advice.Interceptor*). The pointcut is defined in XML (*jboss-aop.xml*) and related interceptors are bound to it. The advantage of applying *JBoss-AOP* as the implementation technique will be discussed in section 5.1.

Feature tangling and scattering complicate the traceability from features in a feature model all along to the program level units, usually classes or methods in an object-oriented language. It means there may be several program units, especially methods, contributing to a single feature, while a single program unit, especially a class, may contain implementations for several features. In our paper, we assume that each feature is implemented by one or more methods in the same class or in different classes, i.e. a one-to-many relationship between features and methods.

The variability implementation with *JBoss-AOP* can be seen in Figure 1(b). For the sake of simplicity, we assume further that one feature corresponds to one method. A mandatory feature is connected through the normal method and constructor invocations. An optional feature is implemented by weaving the method into the base program through an interceptor. The implementation for an alternative feature adopts the inheritance approach, i.e., each method corresponding to a variant should reside in a class that inherits the base class containing the base method and all of them ought to keep the same method signature. Then the feature can be implemented by one of its variants through instantiating the variant-related class when customized. We call it *object replacement* mechanism, which is handled by an aspect. The implementation for an OR feature also follows the object replacement mechanism but it is replaced at runtime. Therefore a selecting program is necessary in the interceptor which helps to determine the variant. In addition, the interceptor codes for implementing alternative and OR features are undetermined at design time since the variant to be chosen and the variants included cannot be decided until the product derivation phase. So, these two kinds of interceptors are suggested to follow the certain implementation templates and they will be reified to support the object replacement when the SPL is customized.

Feature model generalization is a transformation performed to increase the variability of a SPL. In [3], Alves et al. propose twelve types of unidirectional feature model refactorings. Their work concentrates on feature-level variability refactoring only, while we focus on feature-driven program-level variability generalization. On the other hand, in this paper we only consider the generalizations which are part of the feature refactoring types that add the variability incrementally, i.e. improve the configurability by introducing new variation points or variants.

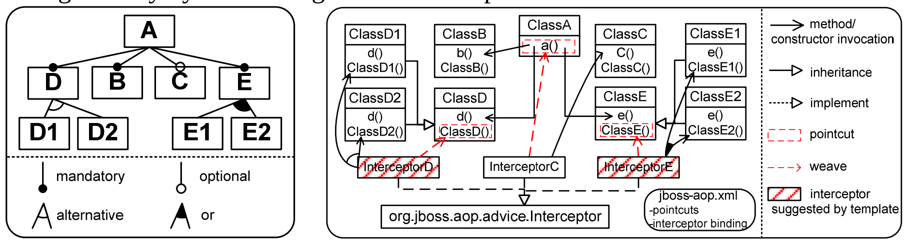


Fig. 1 (a). Standard feature model **(b)**Variability implementation with JBoss-AOP

3 Basic Generalization Scenarios and Their Composition

We identify six basic generalization scenarios, each of which includes a feature generalization pattern and the corresponding reference implementation. Based on the basic scenarios, we provide guidance for scenario composition to handle the more complex cases. Furthermore, we still follow the assumption that one feature corresponds to one method in describing the basic scenarios in Section 3.1, while feature tangling and scattering are taken into consideration in Section 3.2.

3.1 Basic Generalization Scenarios

The scenarios are illustrated in Figure 2, where the legends follow those in Figure 1. In particular, the squares filled with color indicate new or modified entities. Besides the graphs, the snippets of the reference implementations when deriving a product are illustrated in Figure 3. The snippets include the method body of method *invoke* in the interceptor class as well as the XML segment of the aspect binding in *jboss-aop.xml*.

3.1.1 *AddOptional*

This scenario describes adding a new optional feature *B* into the feature model. In the program level, a new aspect is introduced. The pointcut is the method (*ClassA.foo()*) which intends to invoke the new method implementing the added feature. The interceptor thus includes the invocation to the new method which can only be executed before or after the base method. The implementation snippets when feature *B* is included can be seen in Figure 3(a).

3.1.2 *MandatoryToOptional*

This scenario indicates transforming an existing mandatory feature *B* to an optional one. It's another situation to implement optional features besides *AddOptional*. In the program level, an aspect is introduced to weave into the existing programs. The pointcut points to the method (*ClassB.bar()*) which corresponds to the optional feature. The interceptor then overrides the target method (not really modify it) by writing an empty method body or simply returning a nonsensical value if the method should have a return value. As a result, the aspect takes effect when the new optional feature is removed in a product, and vice versa. The implementation snippets when feature *B* is removed are shown in Figure 3(b).

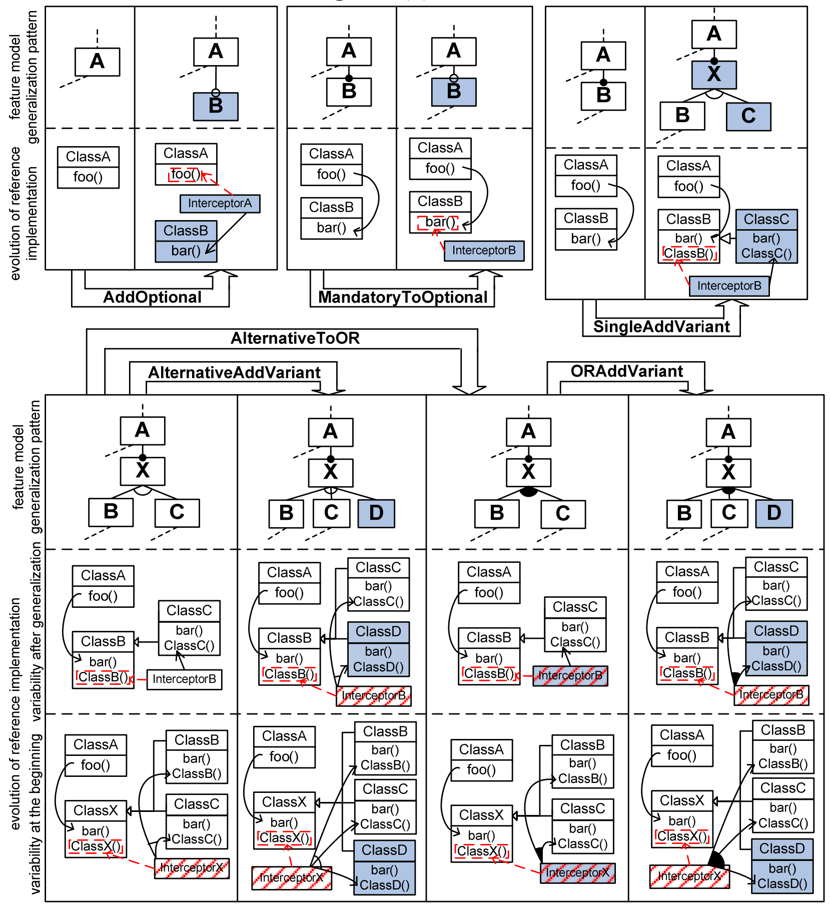


Fig. 2. Six basic generalization scenarios

3.1.3 *SingleAddVariant*

This scenario describes adding a variant to an existing single feature, i.e. the feature without variants, to make it alternative. We cannot implement the new alternative feature using the method in Section 2 where there is a super-class acting as a placeholder at the beginning. Explaining the reason using the graph, we can see that the invocation from *ClassA* to *ClassB* is fixed, so it's not practical to alter *ClassA* to invoke a class created as a super-class and then make *ClassB* inherit the new class. Therefore, we adopt a new method based on the object replacement mechanism. *ClassC* is firstly defined as a subclass of *ClassB* where the method *bar()* should be overridden. Secondly an aspect is created. Its pointcut is *ClassB*'s constructor while its interceptor instantiates *ClassC* to get an object taking the place of *ClassB*. Once feature *C* is selected, the aspect should be bound in *jboss-aop.xml*. Contrarily, the aspect won't be woven if the original feature *B* is selected. The interceptor code and xml segments when feature *C* is bound are shown in Figure 3(c).

3.1.4 *AlternativeAddVariant*

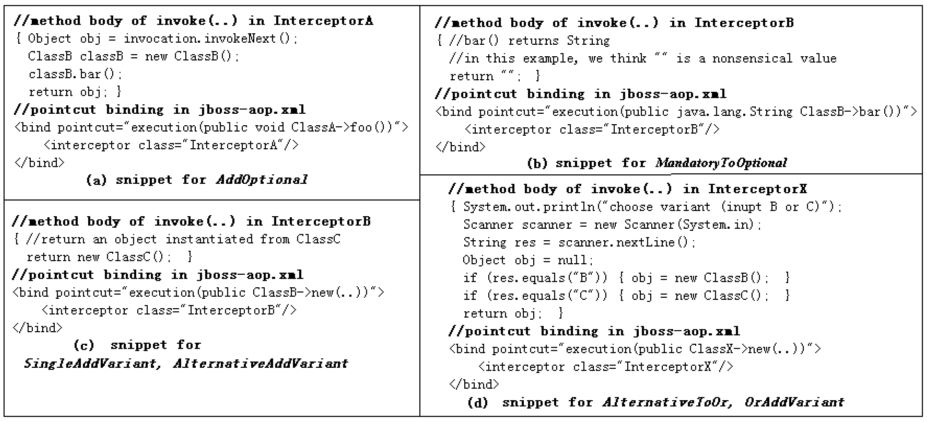
This scenario is captured when a new variant is added to an existing alternative feature. There are two approaches to implement an alternative feature: *SingleAddVariant* (it becomes alternative after generalization) and the idea from section 2 (it is alternative in the beginning). They are different in the pointcut definition and are illustrated in the second and third rows respectively. However, the underlying generalization method for the two situations is unique, both following the object replacement mechanism. We define the new class (*ClassD*) as a subclass of the one (*ClassX* and *ClassB* in two styles respectively) invoked by *ClassA*. The pointcut remains the same, pointing to the constructor. The interceptor codes are undetermined since the variant choice decision is unknown at design time. However, the generalization pattern provides the implementation template for the interceptor, which is reified when the SPL is customized. The interceptor template for the reference implementation can be concluded from Figure 3(c). The intercepted class and the chosen variant are the variable points in the template.

3.1.5 *AlternativeToOR*

This scenario transforms an existing alternative feature to an OR feature, i.e. more variants can be included in a product and the choice is made at runtime through a selecting program. Similar with *AlternativeAddVariant*, two approaches can implement the alternative feature. However, no matter which approach is adopted, the only thing is to modify the interceptor following a new template which contains the selecting logic for the future included variants. There is a set of *if* clauses each of which has a condition and a corresponding object instantiation sentence. Sometimes the application requires the human interaction to do the decision, so the interceptor may have to provide a user interface to acquire the human's choice. The reference implementation snippets after SPL customization are shown in Figure 3(d) and the template can be concluded from them.

3.1.6 *ORAddVariant*

This scenario indicates adding a new variant to an existing OR feature. The reference implementation containing the pointcut definition and the interceptor template keeps the same with *AlternativeToOR*. If the new variant is to be included in the derived product, we just need to ensure that the selecting program in the interceptor embodies the *if* clause involving the new variant.



```
//method body of invoke(..) in InterceptorA
{ Object obj = invocation.invokeNext();
  ClassB classB = new ClassB();
  classB.bar();
  return obj; }
//pointcut binding in jboss-aop.xml
<bind pointcut="execution(public void ClassA->foo())">
    <interceptor class="InterceptorA"/>
</bind>
```

(a) snippet for *AddOptional*

```
//method body of invoke(..) in InterceptorB
{ //bar() returns String
  //in this example, we think "" is a nonsensical value
  return ""; }
//pointcut binding in jboss-aop.xml
<bind pointcut="execution(public java.lang.String ClassB->bar())">
    <interceptor class="InterceptorB"/>
</bind>
```

(b) snippet for *MandatoryToOptional*

```
//method body of invoke(..) in InterceptorB
{ //return an object instantiated from ClassC
  return new ClassC();  }
//pointcut binding in jboss-aop.xml
<bind pointcut="execution(public ClassB->new(..))">
    <interceptor class="InterceptorB"/>
</bind>
```

(c) snippet for *SingleAddVariant, AlternativeAddVariant*

```
//method body of invoke(..) in InterceptorX
{ System.out.println("choose variant (inupt B or C)");
  Scanner scanner = new Scanner(System.in);
  String res = scanner.nextLine();
  Object obj = null;
  if (res.equals("B")) { obj = new ClassB();  }
  if (res.equals("C")) { obj = new ClassC();  }
  return obj;  }
//pointcut binding in jboss-aop.xml
<bind pointcut="execution(public ClassX->new(..))">
    <interceptor class="InterceptorX"/>
</bind>
```

(d) snippet for *AlternativeToOr, OrAddVariant*

Fig. 3. AOP snippets for the atomic generalization patterns

3.2 Guidance for the Composition of Basic Generalization Scenarios

In reality work, we will come up against situations that cannot be solved based on a single basic generalization scenario but their integration. This is caused by the continuous SPL variability evolution as well as the feature tangling and scattering.

In this section, we will put forward an incremental generalization guidance for more complex generalization scenario cases by composing the basic feature model generalization patterns in order and deriving the integrated reference implementation. The phases are as following:

(1) Identify the set of ordered basic feature model generalization patterns for each variation point. Under the circumstances of a complicated generalization scenario, the variability evolution may emerge in the different points, while in a certain point the evolution continues. Hitherto, it is an ad hoc phase for the developers to address them.

(2) For each variation point, adopt the corresponding reference implementations. In this phase, the pointcuts as well as the interceptors are defined in the program level. One feature model generalization may cause the implementation evolution in several parts due to the feature tangling. Furthermore, the implementations for the last three scenarios involve the same pointcut (the constructor) and the same interceptor, whereas the interceptor implementation is of different templates. In this situation, the interceptor should follow the last template within the set and the pointcut is preserved.

(3) Find out the conflicts in reference implementations. Conflicts may appear when the reference implementations of different variant points involve the same pointcut and they cannot be arranged in order. The typical situation is that two features with alternative or OR variability map to the methods in the same class, then the object replacement will bring mistakes when the two interceptors are both woven, i.e., the object can only be replaced by the later bound one. Under such circumstances, the implementations cannot be carried out and the developers should be informed to do the additional undesirable work such as modifying the base programs.

(4) Customize the variability. When there is no conflict in the reference implementations, the binding states of the feature model variability can be determined,

e.g. the optional feature is included or not, which variant is chosen for the alternative feature, etc. In the program level, it indicates the decision whether an interceptor will be bound to a pointcut (first three scenarios), or determine the implementation codes of an interceptor (last three scenarios).

4 Case Study

In this section, a simplified case study is presented based on the composition guidance. The typical generalization case is described in the following sentences.

Before generalization: an initial library management system provided the free service of borrowing books for the campus students.

After generalization: the system was applied for the social use. Public readers were charged and they had to refill money to their accounts. In the beginning, the system received cash only. Along with the diversity in payment manner, the library system allowed readers to refill by credit card or check. Thus readers could choose one of the three kinds of payment when refilling.

The generalization scenario is explicit then the feature model generalizations are easy to identify. Figure 4(a) represents the generalization related part in the feature model. We can see that a new optional feature with three variants has been added to the domain. The whole transformation is combined with a set of basic feature model generalization patterns in the following path: *AddOptional* + *SingleAddVariant* + *AlternativeAddVariant* + *AlternativeToOR*.

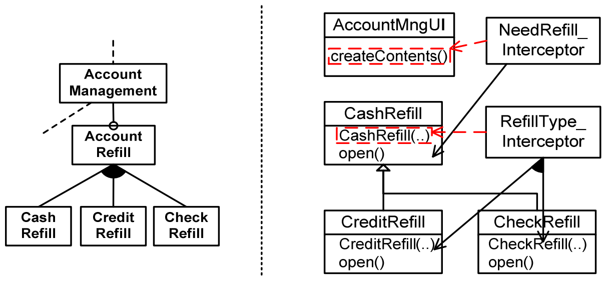
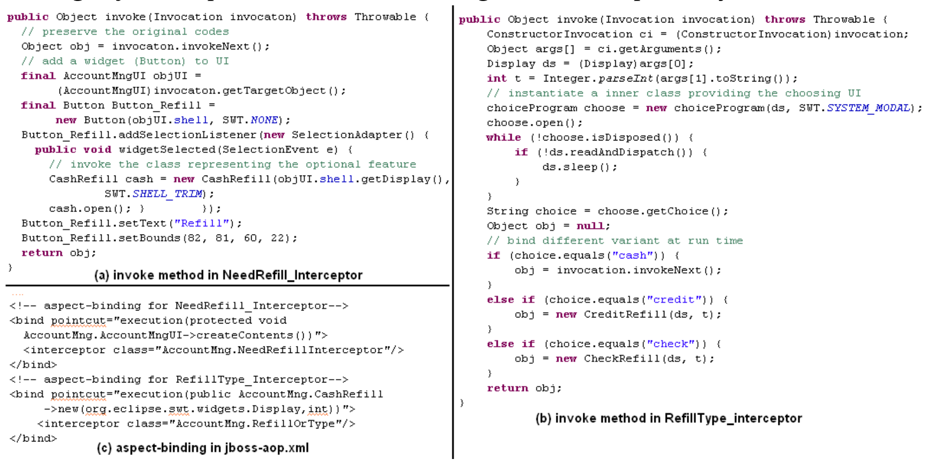


Fig. 4. (a) Feature model generalizations (b) reference implementation

In the program level illustrated in Figure 4(b), a new class *CashRefill* and a new interceptor *NeedRefill_Interceptor* are introduced first according to *AddOptional*. The interceptor not only helps to connect *AccountMngUI* and *CashRefill*, but also adds a widget in the UI that triggers the method invocation (codes in Figure 5a). In the process of the next three reference implementations, another two classes are introduced and both inherit *CashRefill*. In addition, the interceptor *RefillType_Interceptor* is initially introduced by *SingleAddVariant* and at last its implementation code follows the template in *AlternativeToOR*. In particular, the interceptor includes an inner class providing the UI for selecting. Since the simple traceability in the example, there is no conflict found in the implementations.

The variation points are going to be customized. We assume that the feature *AccountRefill* is bound in a product, and the three variants are all included. Thus the codes of *NeedRefill_Interceptor* and *RefillType_Interceptor* as well as the aspect binding in *jboss-aop.xml* are illustrated in Figure 5a/b/c respectively.



```
public Object invoke(Invocation invocaton) throws Throwable {
  // preserve the original codes
  Object obj = invocaton.invokeNext();
  // add a widget (Button) to UI
  final AccountMngUI objUI =
       (AccountMngUI)invocaton.getTargetObject();
  final Button Button_Refill =
       new Button(objUI.shell, SWT.NONE);
  Button_Refill.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
      // invoke the class representing the optional feature
      CashRefill cash = new CashRefill(objUI.shell.getDisplay(),
              SWT.SHELL_TRIM);
      cash.open(); }         });
  Button_Refill.setText("Refill");
  Button_Refill.setBounds(82, 81, 60, 22);
  return obj;
}
```

(a) invoke method in NeedRefill_Interceptor

```
public Object invoke(Invocation invocation) throws Throwable {
    ConstructorInvocation ci = (ConstructorInvocation)invocation;
    Object args[] = ci.getArguments();
    Display ds = (Display)args[0];
    int t = Integer.parseInt(args[1].toString());
    // instantiate a inner class providing the choosing UI
    choiceProgram choose = new choiceProgram(ds, SWT.SYSTEM_MODAL);
    choose.open();
    while (!choose.isDisposed()) {
        if (!ds.readAndDispatch()) {
            ds.sleep();
        }
    }
    String choice = choose.getChoice();
    Object obj = null;
    // bind different variant at run time
    if (choice.equals("cash")) {
        obj = invocation.invokeNext();
    }
    else if (choice.equals("credit")) {
        obj = new CreditRefill(ds, t);
    }
    else if (choice.equals("check")) {
        obj = new CheckRefill(ds, t);
    }
    return obj;
}
```

(b) invoke method in RefillType_interceptor

```
....
<!-- aspect-binding for NeedRefill_Interceptor-->
<bind pointcut="execution(protected void
  AccountMng.AccountMngUI->createContents())">
  <interceptor class="AccountMng.NeedRefillInterceptor"/>
</bind>
<!-- aspect-binding for RefillType_Interceptor-->
<bind pointcut="execution(public AccountMng.CashRefill
     ->new(org.eclipse.swt.widgets.Display,int))">
    <interceptor class="AccountMng.RefillOrType"/>
</bind>
```

(c) aspect-binding in jboss-aop.xml

Fig. 5. Interceptor codes and XML segments

5 Discussion

5.1 JBoss-AOP versus AspectJ

The adoption of *JBoss-AOP* as the reference implementation technique rather than *AspectJ* [11], which is another widely used AOP tool, includes three reasons. Firstly, aspects in *AspectJ* are software entities that define pointcuts and advice codes which have their own syntaxes and keywords. Contrarily, the interceptor (advice) of *JBoss-AOP* is written in regular Java classes which are easily coded and reused. Secondarily, the pointcuts of *JBoss-AOP* are defined and centralized in an XML file. Operating on this unique file can be regarded as a central configuration mechanism for the SPL. However, the pointcuts of *AspectJ* are defined in the aspects and they are dispersed in the programs. Thus they cannot be managed efficiently. Thirdly, *JBoss-AOP* supports the dynamic AOP while *AspectJ* doesn't. It allows modifying the bindings (the removal of aspects and the weaving of new aspects) at runtime without recompiling. It will be a crucial character in our later work that the SPL generalization can be automatically performed at runtime.

5.2 Limitations

Limitation of the work can be found in the program-level reference implementation based on *JBoss-AOP*. Firstly, the last three scenarios cannot handle the static method invocation. Since the classes embodying the static methods don't need to be instantiated, the object replacement cannot take effect. Secondly, we assume that a class which embodies the methods representing a variant should be either a super-class or a subclass since it’s the premise for object replacement at runtime. Thus, the variant-related classes have to be predefined with inheritance when the variants are introduced into the domain. Thirdly, we have noticed that there are two implementation approaches in the last three scenarios. It will make the generalization difficult to perform, especially when it grows larger. Hitherto, we have taken note to them and plan to work out solutions in the future work.

6 Related Work

Alves et al have explored SPL adoption strategies at the feature model level [3] and at the implementation level [13]. Our work is inspired from [3] while the difference lies in the fact that we only focus on the generalizations which add variability to a SPL incrementally. In [13], the authors propose a method and a tool for extracting a product line and evolving it in the implementation level. The work includes a set of simple programming laws that adopt *AspectJ* to handle with the variations. Compared with their mechanism, we utilize *JBoss-AOP* to implement variability. On the other hand, the reference implementations are associated with the feature model generalization patterns in our paper. However, their papers are independent.

A feature-oriented refactoring (FOR) approach [14, 15] is proposed to decompose a program into a set of features, which are considered as the increments in program functionality. The purpose of their work is to specify the relationships between features and their implementing modules, especially to describe the SPL variability as program refinements added to the base program. Our work applies generalization to address the configurability increments in a SPL, which is different from the idea of FOR. Howsoever, their formal theory on the program level is believed to be complementary to our future work on automatic SPL generalization.

7 Conclusion and Future Work

Software product lines cannot be stable all the time, thus incremental generalization is indispensable to support new or changed application requirements. In this paper, we propose a feature-driven and incremental variability generalization method. In it, a set of basic generalization scenarios are introduced which contain the feature model generalization patterns and program-level reference implementations based on *JBoss-AOP*. Composition guidance is also presented to handle more complex generalization cases. In addition, a case study and some discussions are posed.

It’s our elementary research so we have concluded our future work to improve the method. Firstly, we will try to solve the limitations mentioned in section 5.2. Secondly, we will improve the flexibility and scalability of our method to handle with other software artefacts like components under a comprehensive traceability support. Thirdly, an automatic SPL generalization is desired to replace the ad hoc approach. A formal basis for the variability generalization is critical. In addition, a supporting tool is expected that it can support the feature modeling for a SPL and perform the program-level evolution driven by the generalizations in feature model automatically.

Acknowledgments. This work is supported by National Natural Science Foundation of China under Grant No. 60703092, National High Technology Development 863 Program of China under Grant No. 2007AA01Z125, and Shanghai Leading Academic Discipline Project under Grant No. B114.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
2. Krueger, C.: Eliminating the Adoption Barrier. IEEE Software, Vol 19. (2002) 29-31
3. Alves, V., Gheyi, R., Massoni, T.: Refactoring Product Lines. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06). (2006) 201-210
4. Thum, T., Batory, D., Kastner, C.: Reasoning about Edits to Feature Models. In: Proceedings of the 31th International Conference on Software Engineering (ICSE'09). (2009)
5. Kang, K.C., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. (1990)
6. Kang, D.C., Kim, S., Lee, J., Kim, K., Kim, G.J., Shin, E.: FORM: A feature-oriented reuse method with domain-specific architecture. Annals of Software Engineering, Vol 5. (1998) 143-168
7. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997). (1997) 220-242
8. Filman, R., Friedman, D.: Aspect-oriented programming is quantification and Obliviousness. In: Proceedings of the Workshop on Advanced Separation of Concerns, in conjunction with OOPSLA'00 (2000)
9. Khan, K.: JBoss-AOP. (2008) Available at URL: http://www.jboss.org/jbossaop/.
10. Pawlak, R., Seinturier, L., Retaille, J.: Foundations of AOP for J2EE Development. Apress (2005)
11. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: Getting Started with AspectJ. Communications of the ACM, Vol 44. (2001) 59–65
12. Batory, D.: Feature models, grammars, and propositional formulas. In: Proceedings of the 9th International Conference of Software Product Lines (SPLC'05). (2005) 7-20
13. Alves, V., Matos, P., Cole, L., Borba, P., Ramalho, G.: Extracting and evolving mobile games product lines. In: Proceedings of the 9th International Software Product Line Conference (SPLC'05). (2005) 70–81
14. Liu, J., Batory, D., Lengauer, C.: Feature Oriented Refactoring of Legacy Applications. In: Proceedings of the 28th International Conference on Software Engineering (ICSE'06). (2006) 112-121
15. Trujillo, S., Batory, D., Diaz, O.: Feature Refactoring a Multi-Representation Program into a Product Line. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06). (2006) 191-200
16. Peng, X., Shen, L.W., Zhao, W.Y.: Feature Implementation Modeling based Product Derivation in Software Product Line. In: Proceedings of the 10th international conference on Software Reuse (ICSR'08). (2008) 142-153
17. Peng, X., Zhao, W.Y., Xue Y.J., Wu, Y.J.: Ontology-Based Feature Modeling and Application-Oriented Tailoring. In: Proceedings of the 8th International Conference on Software Reuse (ICSR'06). (2006) 87-100