Variability Design and Customization Mechanisms for COTS Components*

Soo Dong Kim, Hyun Gi Min, and Sung Yul Rhew

Department of Computer Science,
Soongsil University,
1-1 Sangdo-Dong, Dongjak-Ku, Seoul, Korea 156-743
{sdkim, syrhew}@comp.ssu.ac.kr, hgmin@otlab.ssu.ac.kr

Abstract. Component-Based Development (CBD) is gaining popularity as an effective reuse technology. Components in CBD are mainly for inter-organizational reuse, rather than intra-organizational reuse [1]. One of the common forms of reusing commercial-off-the-shelf (COTS) components is to acquire and customize them for each application. Therefore, components must be developed with consideration of commonality and variability in a domain in order to increase the reusability and applicability [2]. One effective factor in determining the quality of components is how precisely the variability is modeled and how effective customization mechanisms are provided. COTS components often come in binary and blackbox form, therefore modifying the source code or re-linking object code with library are forbidden. However, much of current approaches to component customization are directed towards tailoring whitebox components, i.e. source code is modified. In this paper, we present a comprehensive set of techniques to realize variability into blackbox components and to provide effective interface-based customization mechanisms. Maintanbility, applicability and reusability can be enhanced by using the mechanism.

1 Introduction

CBD is gaining popularity in both industry and academia as an effective reuse technology. Components in CBD are mainly for inter-organizational reuse, rather than intra-organizational reuse. One of the common forms of reusing COTS components is to acquire and customize them for each application. Therefore, components must be developed with consideration of commonality and variability in a domain in order to increase the reusability and applicability. One effective factor in determining the quality of components is how precisely the variability is modeled and how effective the customization mechanisms are.

COTS components often come in binary and blackbox form to minimize the coupling between components and applications and to protect intellectual design assets. For blackbox components, modifying the source code or re-linking object code with library are forbidden. However, much of current approaches to component

* This work was supported by Korea Research Foundation Grant. (KRF-2004-005-D00172).

O. Gervasi et al. (Eds.): ICCSA 2005, LNCS 3480, pp. 57 – 66, 2005.

customization are directed towards tailoring whitebox components. Tailoring whitebox components involves understanding the internal design of the component, modifying source code, and rebuilding the component with any necessary library. While tailoring whitebox component is more effort and time consuming, tailoring blackbox component only involves invoking the customization interface to set appropriate variants and so it is more efficient and less time consuming.

However, it is challenging to be able to customize blackbox components without accessing their internal design and source code. In this paper, we present a comprehensive set of techniques to design components with variability so that blackbox form of components can be customized effectively only through interfaces. We focus on practical applicability of customization techniques which can be implemented in popular CBD platforms.

We survey representative works on variability design in section 2, and present a foundation on variability types and interfaces. The main customization techniques proposed are presented in section 4, 5 and 6. The proposed work is compared with other works in section 7.

2 Related Work

In this section, we present a survey of representative customization methods for whitebox and blackbox components.

Keepence and Mannion's Work suggests three patterns for variability design; *single adapter*, *multiple adapter* and *options* patterns [3] as in figure 1. In *single adapter*, generic features are modeled in a base class and specific features are modeled in subclasses. Only one subclass can be instantiated in any single system. *Multiple adapters* are similar to single adapter, but more than one subclass can be instantiated in any single system. In *options* pattern, two associated peer classes are created to realize a variation. Keepence's work suggests three types of variability mechanism. However, this research specifies range of variant. It doesn't include detailed implementation techniques.

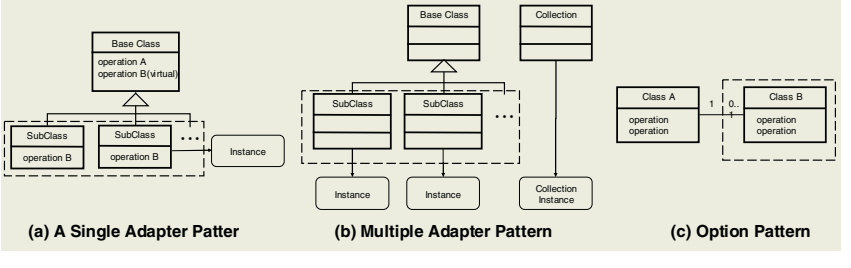

Fig. 1. Keepence's Patterns for Variability

Anastasopoulos and Gacek's Work identifies several customization methods; aggregation/delegation, AOP, conditional compilation, dynamic class loading, dynamic link libraries(DLL's), frames, inheritance, overloading, parameterization, properties and static libraries [4]. *Aggregation/delegation* method enables objects to

delegate requests to other objects which provide a customized behavior. *Conditional compilation* method enables control over the multiple code segments; including or excluding selected code segment from a program compilation. *Dynamic class loading* is a feature in Java where all classes are loaded into memory as soon as they are needed at runtime. *Frame* is used to specify adaptable behavior. *Parameterization* is used to pass variants. *Static library* contains a set of external functions that can be linked to an application after it has been compiled. Among the proposed techniques, only aggregation/delegation, dynamic link library and parameterization methods can be effectively applied to blackbox components.

Svahnberg and Bosch's Work suggests five customization techniques; inheritance, extensions, parameterizations, configuration and generation of derived components [5]. *Extensions* mechanism is used when parts of a component can be extended with additional behavior, selected from a set of variations. *Configuration* enables selection of source code segment and files from a code repository to form a customized product. *Generation* of derived components is hard coded to a particular set of parameters. However, inheritance and generation methods cannot be used for customizing blackbox component.

3 Foundation

In this section, we summarize fundamental concepts and terms used for presenting our methods. We first define terms related to variability; *Variation Point (VP)*, *Variant*, and *Variability*. *Variation Point* is a place in software where a minor difference occurs among family members [6]. It is possible for a function to have more than one variation point. *Variant* is a value or instance that can validly fill in variation points, i.e. a variant resolves a variation point. A variation point typically has more than one variant. *Variability* is characterized by various variations within common requirement, and it consists of variation points and a set of their valid variants. Therefore, variability is a comprehensive description of variations occurring in a family.

There are four types of variability in CBD; variability on *Attribute*, *Logic*, *Workflow*, or *Persistence* [6]. Attribute is defined as an abstract storage to store values, and it is realized as constants, variables, or data structures. *Attribute variability* denotes occurrences of *variation points* on attributes. The typical forms of variations are the different number and/or data types of attributes for a given function. Logic describes an algorithm or a procedural flow of a relatively fine-grained function. *Logic variability* denotes occurrences of *variation points* on the algorithm or logical procedure. Workflow describes a sequence of method invocations to carry out a coarse-grained function. *Workflow variability* denotes occurrences of *variation points* on the sequence of method invocations. Persistency is maintained by storing attribute values of a component in a permanent storage so that the state of the component can alive over system sessions. Typically a component contains several classes, and classes contain persistent attributes. These attributes must be mapped to a representation on a secondary storage such as files on hard disk and relational database tables. *Persistency variability* denotes occurrences of *variation points* on the physical schema or representation of the persistent attributes on a secondary storage.

4 Selection Technique

The selection technique is to define classes and an customize interface for clients to select one of the variants realized inside the components. Once a variant is selected, the value is stored and remembered so that further invocations can refer to the selected variant. The selection mechanism works in four steps.

4.1 Step 1. Defining Variable Functions

This step is to define functions to handle the selection process for the classes which have variation points. As shown in figure 2, variation points are realized as functions which include a *switch* statement, and variants are specified as *case* clauses within the switch statement.

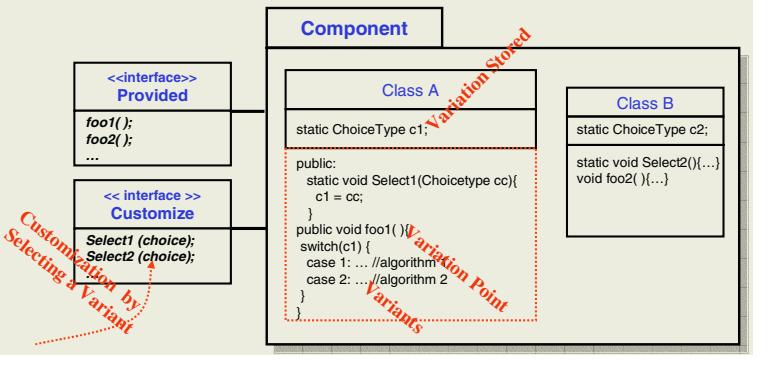

Fig. 2. Mechanism of Selection Technique

In the figure 2, the function *foo1 ( )* of *class A* realizes a variation point and each *case* clause of *foo1( )* realizes an associated variant. If *foo1( )* is a function computing a temperature, the unit of temperature can be a variation point with two variants; Centigrade and Fahrenheit. Consequently, one *case* clause is an algorithm using Centigrade and the other *case* clause can be an algorithm using Fahrenheit.

4.2 Step 2. Defining Static Attributes and Operations for Customization

This step is, for each class in a component, to define attributes for storing selected variants and to define *customize* operations that read selections made by clients and store the selected variants in these attributes. Once the selections are (possibly persistently) stored in these attributes, further invocations of *foo1( )* operation will refer to the values stored in the attributes.

In this way, one-time customization has a persistent effect on the variability. In order to keep the value of *c1* for a long period of time or persistently, the value must be stored in a secondary storage such as a file or a database. If the value isn't stored, it should be customized at a re-installation time such as Web Application Server (WAS) rebooting. The customization activity ends at this moment.

4.3 Step 3. Defining Customize Interface

This step is to collect *customize* operations in various classes in a component into a single *customize interface*. In figure 2, *Select1 (Choicetype cc)* and *Select2 (Choicetype cc)* are *customize* operations; Therefore, they are included in the *customize interface* of the component. The component consumer sets the variants using this interface to customize components.

4.4 Step 4. Setting Variants

This step is to customize components using *customize* operations. For example, a component consumer invokes a customize operation, *Select1(choice),* with a parameter setting on an appropriate variant. The actual argument of the method is now stored in a static attribute *c1* by the assignment statement of *Select1( )* method. As a result, further invocation on *foo( )* will refer to this attribute. Now, customization on components is completed, and the operations in the *Provided* interface are invoked.

4.5 Remarks for Realizing Attribute Variability

Selection technique can be applied to various types of variability; attribute, logic, workflow, and persistence variability. The mechanism requires special attention when realizing attribute variability.

Selection mechanism for realizing attribute variability utilizes the mechanism of generic classes, which capture common behavior and instantiate concrete classes of specific data types. For example, the *template* construct of C++ provides the mechanism of generic classes.

The variation points are realized as functions which include a *switch* statement, and variation on data type is specified as *case* clauses within the switch statement. The objects which have different data type are created by the case clauses. Each *case* clause includes a statement to create an object that has attributes of an appropriate data type.

As shown in figure 3, the component includes two classes; class *A* and class *Account* which have variation points, and *accountID* and *createAccount( )*, of the attribute variability type. Class A has a attribute variability on a variation point *createAccount( );* this variation point has already implemented possible types of *accountID* using a switch-case statement. Furthermore, class *Account* has attribute variability on a variation point *accountID;* this variation point has been implemented using a parameterized type concept.

A static int attribute *c* in *Class A* stores variants. A *createAccount( )* method in class A performs two things according to variation storage *c*. The first thing is to instantiate template class Account conforming to *c*, the second thing is to create a new account object with particular type of *accountID*. If it can be an integer type or string type, the attribute is instantiated by “new Account<int> (12932)” and “new Account<string> (“AZ129”)” statement.

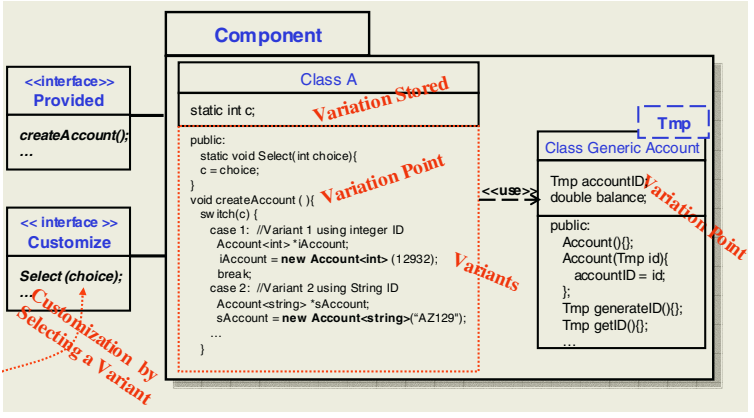

Fig. 3. A Mechanism of Attribute Variability using Selection Technique

5 Plug-in Technique

Plug-in technique is used to assign an external variant to a variation point of a component through a *customize* interface. By passing references of objects to components and setting these functions are objects invoked inside components, application-specific functionality is defined and supplied to the components. In this way, components can be customized for each application.

The effects of customizing components should be persistently stored in and around the components. With the plug-in technique, this is done by persistently maintaining the references, functions or objects passes as parameters.

The *plug-in* technique can be applied to various types of variability; attribute, logic, workflow, and persistence variability.

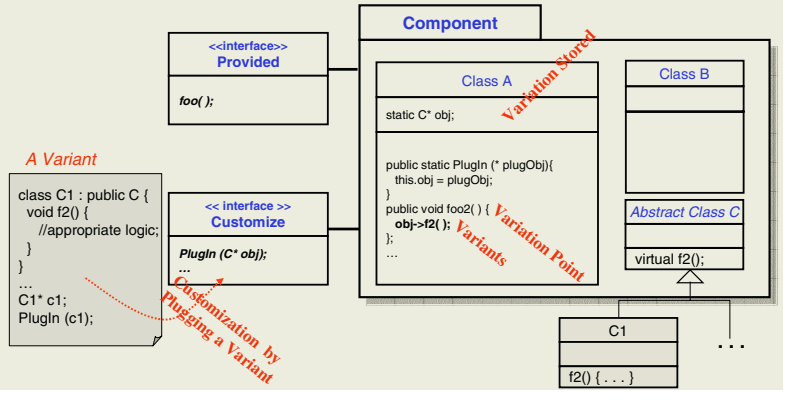

Fig. 4. Mechanism of Passing Pluggable Objects

5.1 Step 1. Defining Variable Functions

This step is to define the functions which handle the plug-in process for the classes which have variation points. The functions have hot spots for unknown variants. The hot spots will be filled by external functions and objects.

As shown in figure 4, an object as a variant can be plugged into a component. Variation points include object pointers that are hot spot. The object pointers will be plugged by a component consumer.

5.2 Step 2. Defining Static Attributes and Operations for Customization

This step is, for each class in a component, to define attributes for storing references of plugged functions and objects, and to define *customize* operations that read functions or objects made by clients and store the plugged functions or objects in these attributes. Once the references of functions and objects are (possibly persistently) stored in these attributes, further invocations of *foo1( )* operation will refer to the external functions. It is shown how pluggable functions and objects can be passed on to a component as a component customization technique.

To show an example of pluggable objects, In figure 4, the method *foo2( )* has a logic variability and the variation point is the method *f2( )* inside *foo2( )*. Hence, each family member may supply its own implementation of *f2( )*.

5.3 Step 3. Defining Customize Interface

This step is to collect *customize* operations in various classes in a component into a single *customize interface*. A component has a *customize* interface which takes the value or references of external elements and assigns it to its corresponding variation point.

In figure 4, an external object *c1* must be plugged into *obj* inside the component. To instantiate this variation point with an appropriate object, a customize interface is defined which contains *PlugIn (Classname* Obj)* method. Now, a family member can pass a reference to its own object that is extended by abstract class C. Class A stores the variant object that has the appropriate logic. The component client invokes the *foo( )* function. The *foo2( )* function invokes the *f2( )* method in a variant object by the dynamic binding of the object-oriented technique.

The method of *PlugIn(void (*fn)())* and *PlugIn (C* Obj);* are *customize* operations; thus, they are included in the *customize interface* of the component. Component consumer sets pluggable functions and objects using this interface to customize components.

5.4 Step 4. Setting Variants

This step is to customize components using *customize* operations. A component consumer makes an appropriate object that should be inherited from abstract class. The consumer invokes *PlugIn (C* Obj)* with the object. Therefore, further invocation on *foo1( )*and *foo2( )* will run the plugged parts.

6 External Profile Technique

External Profile technique is used to assign an external variant to a variation point of a component through an external profile such as a XML file. The external profile describes variants for customization. This can be done by storing the values of external profile. If a variant is changed, the component consumer only modifies the profile.

6.1 Step 1. Defining Variable Functions

The step for customization is similar to the technique of *passing pluggable object*. Variation points are realized as functions which include a sentence to read the variant. They will be read by XML profiles.

How variants in external profile can be passed on to a component as a component customization technique are exhibited. Figure 5 shows a component which contains class A. The class has the method *foo()* that is a variation point which is resolved by an external profile.

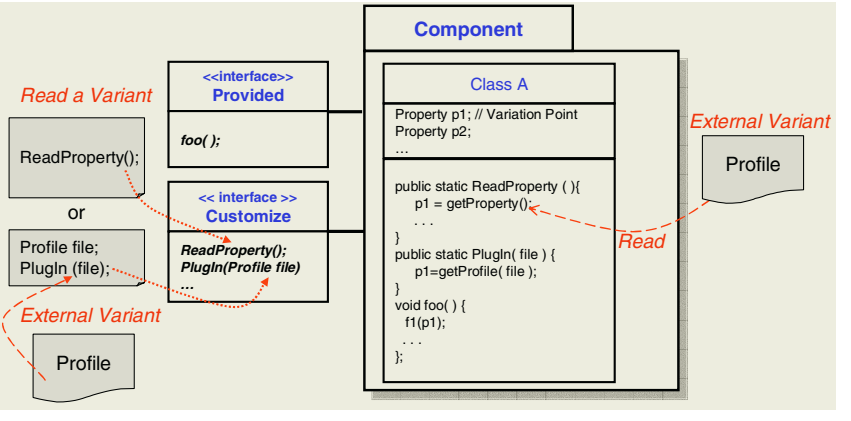

Fig. 5. Mechanism of 'Read the External Profile'

6.2 Step 2. Defining Static Attributes and Operations for Customization

The variable function reads variants from an external profile that includes variants for a family member. The component consumer invokes the *ReadProperty( )* in the *Customize* interface, the *ReadProperty( )* in *Class A* reads and analyses the external profile that should have a fixed location. The method *ReadProperty()* knows the position of the external profile.

In a different other mechanism, a component consumer invokes the *PlugIn (profileFile)* in *Class A*. The function reads and analyses the external profile. The profile does not have a fixed location. This technique is used to assign an external profile to a variation point of a component through a *customize* interface. These customize operations read variants from the external profile and store variants to the

attributes. Therefore, the customize operations read XML files. Using Simple API for XML (SAX) rather than Document Object Model (DOM) is preferred.

However, the customize mechanism may read a XML profile once when a component consumer tailors components. The whole document is not needed to be fed into memory using a tree structure. There is no control over the order. The customize mechanism prefers to SAX rather than DOM. The *External Profile* technique can be implemented using Java and SAX API.

6.3 Step 3. Defining Customize Interface

This step for customization is similar to the technique of *Selection* and *Plug-In*. This step is to collect *customize* operations in various classes in a component into a single *customize interface*.

6.4 Step 4. Setting Variants

This step is to customize components using *customize* operations. The *External Profile* can be described by a XML file. For example, variability attributes are grouped thus; *Attr*, *Logic,* and *Workflow* in figure 6. If the requirement of the target system is changed, then the component consumer only modifies the XML profile.

```
<?xml version="1.0">
<Variability>
  <AttrType variantType = "var1" >…</AttrType>
<AttrType variantType = "var2" >… </AttrType>
<LogicType> … </LogicType>
<WorkflowType> ...</WorkflowType>
. . .
</Variability>
```

Fig. 6. Example of Variant Profile using XML

7 Assessment

In this paper, we propose variability design and customization mechanisms for COTS components. Our mechanism addresses variation types and scopes for variability mechanism. It is to increase component reusability and maintanability. We now compare our work to other representative works.

The catalysis presents two types of variability mechanism using inheritance. Keepence's work suggests three types of variability mechanism. However, these researches do not include detailed implement technique. Anastasopoulos' paper presents variability and feature type. However, the research is used to whitebox component. Svahnberg's research suggests five types of variability type. However, the research does not include inner detailed mechanism.

Our mechanisms are challenging to be able to customize blackbox components without accessing their source code. If new requirements that were covered by customization mechanism are discovered, we only select or plugin each variant by

customizing interfaces to maintain components. Therefore, maintanbility can be enhanced by using the mechanism.

8 Concluding Remarks

As components are more for inter-organizational reuse, we need to model variability as well as commonality. One of the common forms of reusing COTS components is to acquire and customize them for each application. Therefore, components must be developed with consideration of commonality and variability in a domain.

In this paper, we present a comprehensive set of techniques to design components with variability so that blackbox form of components can be customized effectively only through interfaces. The COTS component is usually blackbox component. We focus practical applicability of customization techniques which can be implemented in popular CBD platforms.

The four types of component variability are covered by our customization mechanism. We proposed three techniques for variability implementation; *Selection, Plug-In* and *External Profile* technique. The techniques were presented more detailed customization methods. Through the three customization mechanism, we believe that the applicability, reusability, and maintainability of components can be greatly increased.

References

[1] Kim, S., "Lesson Learned from a Nationwide CBD Promotion Project," *Communications of the ACM, Vol. 45, Issue. 10, Oct.*, 2002.

[2] Kim, S., and Park, J., "C-QM: A Practical Quality Model for Evaluating COTS Components," *Proceedings of IASTED International Conference on Software Engineering*, Innsbruck, Austria, Feb., 2003.

[3] Aanastasopoulos, M., and Gacek, C., "Implementing Product Line Variabilities," *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*, Toronto, Cananda, May, 2001.

[4] Keepence, B., and Mannion, M., "Using patterns to model variability in product families," IEEE Software, Vol. 16, Issue. 4, July-Aug., 1999.

[5] Svanhnberg, M., and Bosch, J., "Issues Concerning Variability in Software Product Lines," *Lecture Notes in Computer Science 1951, Proceedings of the Third International Workshop on Software Architectures for Product Families*, 2000.

[6] Choi, S., Chang, S., and Kim, S.,"A Systematic Methodology for Developing Component Frameworks," *Lecture Notes in Computer Science 2984, Proceedings of 7th Fundamental Approaches to Software Engineering (FASE'04) Conference*, 2004.