Feature Oriented Programming: a Step towards Flexible Composition of Modular Programming

Mahua Banerjee, Sushil Ranjan Roy
Department of Information Management
Xavier Institute of Social Service
Ranchi, India
banerjee.mahua@rediffmail.com,
sushilranjan@yahoo.com

Chiranjeev Kumar
Department of Computer Science & Engineering
Indian School of Mines
Dhanbad, India
k_chiranjeev@yahoo.co.uk

Abstract: **Feature Oriented Programming is the study of feature modularity, a way of implementing a product lines. Software product lines (aka program families) generate families of programs and not monolithic software. In FOP a feature is a unit of functionality that satisfies a requirement. From a set of features, many different software systems can be generated that share common features and differ in other features. This paper focuses on the technical aspects of FOP by modifying the reusability concepts through interfaces, objects and lifters. Lifters are generalization of inheritance comprising the reusability of features.**

Keywords: Feature; product lines; interfaces; lifters

I. INTRODUCTION

Software is subject to frequent changes in order to react to altering and evolving requirements. The process of continuous adaptation, extension, and customization is known as software evolution. The idealized goal of software engineers is to reuse as much as possible code from previous development stages to build a new version of the software. To achieve this, software must be designed reusable, extensible, and customizable. Doing so involves one difficulty. The more code one reuse, the less it fits the different contexts of deployment. Software product lines (aka program families) promise a solution by generating families of programs and not one single monolithic piece of software. A product line is used by two different categories of persons. One who generates a program, known as deployer and another who implements. The deployer creates a program by choosing from a set of features. Usually, many combinations of features are allowed, resulting in a high variance of the generated programs. Feature-oriented programming assists in modular system product lines.

Program families, also known as software product lines, group programs with similar functionalities in families. The key idea is to arrange the design and implementation as a layered stack of functionalities. Different programs consist of different layers. Thus, implemented layers can be reused in multiple programs. A fine-grained layered architecture leads to reusable, extensible and customizable software. Motivated by these observations
several studies are made in the domains of databases [1], middleware [2], avionics [3] and network protocols [1] to show that Feature-Oriented Programming (FOP) [4] is appropriate to implement such layered, step-wise re-fined architectures.

The contribution of this paper is to explain the feature concept from abstract definition to a technical one (by arranging the definitions provided by different authors). Firstly we explained this from a problem space and solution space point of view. Secondly we refined the concepts of Mixinlayers in order to support the technical aspects. And finally we explained the significance of lifters, which is a modification of mixinbased inheritance, with an example. The overall objective of this paper is to make FOP user friendly for modular programming.

II. CONCEPTS AND TERMINOLOGY

A. *Feature*

A feature is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option. Typically, from a set of features, many different software systems can be generated that share common features and differ in other features. The set of software systems generated from a set of features is also called a software product line [5, 6].
There are several definitions of a feature (ordered from abstract to technical):

1. Kang, Cohen, Hess, Novak and Peterson [7]: "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"

2. Kang et al. [8]: "a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained"

978-1-4577-0697-4/12/$26.00 ©2012 IEEE

3. Czarnecki and Eisenecker [9]: "a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept"
4. Bosch [10]: "a logical unit of behaviour specified by a set of functional and non-functional requirements"
5. Chen, Zhang, Zhao and Mei [11]: "a product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements"
6. Batory, Sarvela and Rauschmayer [4]: "a product characteristic that is used in distinguishing programs within a family of related programs"
7. Classen, Heymans and Schobbens [12]: "a triplet, f = (R,W, S), where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification"
8. Zave [13]: "an optional or incremental unit of functionality"
9. Batory [14]: "an increment of program functionality"
10. Apel, Lengauer, M¨oller and K¨astner [15]: "a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option"

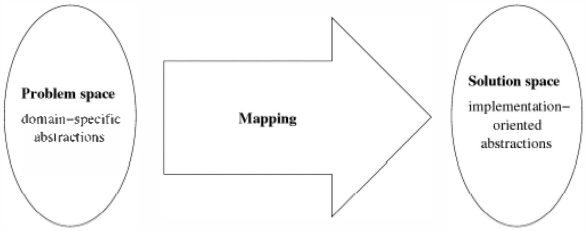

Figure 1: Problem and solution space

The definitions become less abstract and more technical if we analyze them from top to bottom. The first seven definitions reflect that features are abstract concepts of the target domain, used to specify and distinguish software systems, whereas the last three definitions capture the fact that features must be implemented in order to satisfy requirements. This switch over between abstract and implementation view is due to the different uses of features. For example, in feature modeling, features are used to describe the variability of a software product line for communication with stakeholders, independent of any implementation. In feature-oriented programming, a feature is a first-class language construct for structuring source code. Moreover the distinction between problem space and solution space, as illustrated in Figure 1, is useful for the classification of the definitions presented above. The problem space comprises concepts that describe the requirements on a software system and its intended behavior. The solution space comprises concepts that define how the requirements are satisfied and how the intended behavior is implemented. The first seven definitions describe the feature concept from the perspective of the problem space. Here, features are used to describe what is expected from a software system. The last three definitions describe features from the perspective of the solution space, i.e., how features provide/implement the desired functionality.

B. Feature Oriented Programming

Feature Oriented Programming (FOP) is a design methodology and tools for program synthesis. The goal is to specify a target program declaratively in terms of the features that it offers, and to synthesize an efficient implementation that meets these specifications. FOP studies the modularity of *features* in product lines, where a feature is an increment in program functionality [4]. The idea of FOP is to synthesize software (individual programs) by composing features (a.k.a. *feature modules*). Typically, features refine the content of other features in an incremental fashion. Hence, the term *re-finement* refers to the set of changes a feature applies to a code base. Adding features incrementally, called *stepwise refinement*, leads to conceptually layered software designs.

Mixin layers is one of the approaches to implement features [16, 4]. The basic idea is that features are seldom implemented by single classes. Typically, a feature implements collaboration, which is a collection of roles represented by mixins that cooperate to achieve an increment in program functionality. FOP aims at abstracting and explicitly representing such collaborations. A mixin layer is a module that encapsulates fragments of several different classes (roles) so that all fragments are composed consistently. Figure 2 depicts a stack of three mixin layers (L1 – L3) in top down order. These mixin layers crosscut multiple classes (C_A – C_C). White boxes represent classes or mixins; gray boxes denote the enclosing feature modules; filled arrows refer to *mixinbased inheritance* for composing mixins.

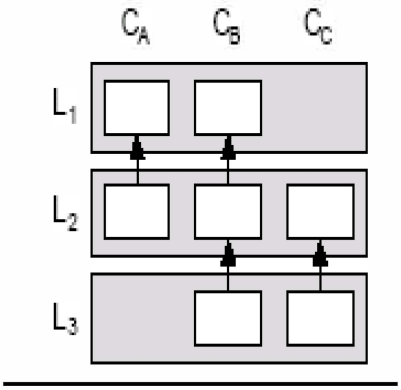

Figure 2 Stack of three mixin layers

FOP has been used to develop product-lines in widely varying domains, including compilers for extensible Java dialects, fire support simulators for the U.S. Army, network protocols, and program verification tools.

C. Product Line Architecture

Product line architecture is software architecture for a family of software products. In the software product family, individual products share common parts and functionality, but some parts are different and must be customized. Product line architecture defines the concepts, structure, and texture necessary to achieve variation in features of variant products while achieving maximum sharing parts in the implementation.

III. HANDLING OF FEATURE ORIENTED PROGRAMMING

In the software product family, individual products share common parts and functionality. This sharing of common parts and functionality is provided by the mixin based inheritance. This mixin based inheritance can further be modified by introducing lifters which replace method overwriting in subclasses. Lifters are separate entities and always handle two features at a time, which is important for flexible composition. The following section introduces the steps for handling FOP using lifters followed by an example of FOP.

A. Steps for Handling a FOP

The major steps required for working in an FOP environment are:

1) Define interfaces.
2) Create new features, for base implementations from the existing interfaces.
3) Create objects with all the features for interaction handling in a particular order.
4) Provide lifters (which are separate entities capable to handle two features at a time) to replace method over writing in subclasses.

Interfaces are defined to have several implementations for one interface. Furthermore, they ease translation into a high level language, like Java/FeatureC++ in which a class can implement several interfaces.

Codes are written for base implementations of individual features. Objects are composed of several features in a particular order for interaction handling. Finally the lifters are required to adapt the functions of an interface which is essential for flexible composition.

B. An Example of Feature Oriented Programming

In this section we introduce a feature oriented programming based on modeling stacks (partially) with the following features:

Stack: providing push and pop operations on a stack.
Counter: adds a local counter (used for the size of the stack)
Lock: adding a switch to allow/disallow modifications of an item (here used for the stack)
Bound: which implements a range check, used for the stack elements
Undo: adding an undo function, which restores the state as it was before the last access to the item

The undo and bound features are not shown in this paper. We only represent the pseudo code for stack handling through FOP with the features stack, counter and lock.

1. Defining interfaces for features

```
interface Stack {
void empty();
void push(char a);
void push1(char a);
void pop();
char top();
}

interface Counter {
void reset();
void inc();
void dec();
int size();
}

interface Lock {
void lock();
void unlock();
}
```

2. Defining features through interfaces

The code below provides base implementations of the individual features. The notation feature SF denotes a new feature named SF which implements Stacks. Similarly the new features CF and LF implements Counter and Lock respectively.

```
feature SF implements Stack {
String s = new String();
void empty() { s = ""; }  // Use Java Strings ….

void push(char a)  { s =  String.      valueOf(a).concat(s); };
void pop() {s =  s.substring(1); };
char top()  {  return (s.charAt(0) ) ; } ;
void push1(char a) { this.push(a) ;  this.       push(a); } ;
}
feature CF implements Counter {
```

```
int i  = 0;
void reset()  { i = 0; } ;
void inc() { i =  i + 1; } ;
void dec() { i = i – 1; } ;
int size() { return i ; } ;
}

feature LF implements Lock {
boolean l =  true;
void lock() { l =  false; } ;
void unlock() { l =  true; } ;
boolean is_unlocked() { return l; } ;
}
```

Objects are created like Java. Mainly features are used as new constructor to create object. In our example, SF is used as a new constructor in which we use other two feature implementations CF and LF for creating an object with all three features as follow:

```
new LF (CF (SF) )
```

The above one is creating an object with all three features. For interaction handling it is important that features are composed in a particular order e.g. the above first adds CF to SF and then adds LF.

3. Introducing Lifters

In addition to the base implementations we need to provide lifters which replace method overwriting in subclasses. Such lifters are separate entities and always handle two features at a time. In the following code, features(via interfaces) are lifted over concrete feature implementations.

```
feature CF lifts Stack {
void empty() { this.reset(); super.empty() ; };
void push(char a){this.      inc();  super.push(a); };
void pop(){this.dec();super.            pop(); } ;
}

feature LF lifts Stack {
void empty()
 {if this.is_unlocked())
 { super.empty(); }} ;
void push(char a)
  { if (this.is_unlocked() ) { super.push(a) ; } };
void pop()
 { if (this.is_unlocked() ) { super.pop() ; } } ;
}

feature LF lifts Counter {
void reset()
 { if (this.is_unlocked() )
 { super.reset(); } } ;
void inc()
 { if (this.is_unlocked() ) { super.inc(); } } ;
void dec()
 { if (this.is_unlocked() ) { super.dec(); } } ;
}
```

For instance the code below *feature CF lifts Stack* adapts the functions of *Stack* to the context of CF, i.e. the counter has to be updated accordingly. When composing features, this lifter is used if CF is added to an object with a feature with interface *Stack* and not just directly to a stack implementation. This is important for flexible composition.

Methods which are unaffected by interactions are not explicitly lifted e.g. top and size. The modular specification of the three features, separated from their interactions allows the following object compositions:

_ Stack with counter
_ Stack with lock
_ Stack with counter and lock
_ Counter with lock

For all these combinations, the three lifters shown above adapt the features to the combinations. The resulting objects behave as desired. In addition, we can use each feature individually.

The composition of lifters and features is shown in Figure 3 for an example with three features. To compose stack, counter and lock we first add the counter to the stack and lift the stack to the counter. Then the lock feature is added and the inner two are lifted to lock. Hence the methods of the stack are adapted again using the lifter from stack to lock.

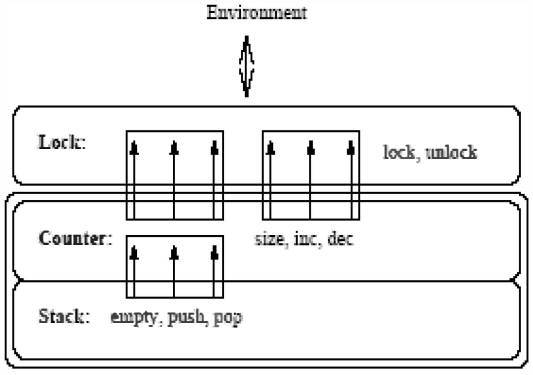

Figure 3 Composing features (rounded boxes) by lifters (boxes with arrows)

The composed object provides the functionality of all selected features to the outside, but for composition we need an additional ordering. In particular the outermost feature is not lifted, similar to the lowest class in a class hierarchy whose functions are not overwritten.

IV. Feature Composition and Related Work

The composition of features in the above FOP technique assumes that features are composed in a particular order. Only from external interface it is possible to view an object as composed of a set of features. Ordering with lifters helps in modifying the concepts of inheritance of Object oriented programming. Although inheritance can be used for such feature combinations, all needed combinations, including feature interactions have to be assembled manually. In contrast we can reuse features by simply selecting the desired ones when creating an object.

Mixins [16, 4] have been proposed as a basic concept for modeling other inheritance concepts. The main difference is that we consider interactions and separate a feature from interaction handling. If mixins are used also as lifters then the composition of the features and the number of lifters has to be done manually in the appropriate order. Instead we can just select features here.

Composition filters [17] have been proposed as another approach to compose objects in layers similar to the feature order in our approach. Messages are handled from outside by each layer. The main difference is that we consider interactions on an individual basis and separate a feature from interaction handling.

Several other approaches allow changing class membership dynamically or proposing other compositions mechanisms. None of these consider a composition architecture which is a refinement of inheritance.

V. Conclusions

Feature oriented programming is an extension of object oriented programming. Our approach of making FOP user friendly does not limit the programmer in modularizing software in terms of OOP. It suggests for much higher modularity and flexibility. Reusability is simplified, since for each feature, the functional core and the interactions are separated. This difference encourages to write independent, reusable features and to make the dependencies to other features clear. In contrast inheritance with overwriting mixes often leads to highly entangled (sub-) classes. Hence the feature model assists in separating the core functionality from interaction resolution, object creation by composing features and a composition architecture which generalizes inheritance. The pseudo code provided in the example can be implemented in FeatureC++ as a future work.

References

[1] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions on Software Engineering and Methodology (TOSEM), 1(4), 1992.

[2] S. Apel and K. B¨ohm. Towards the Development of Ubiquitous Middleware Product Lines. In Proceedings of the ASE Workshop on Software Engineering and Middleware (SEM), volume 3437 of Lecture Notes on Computer Science. Springer, 2005.

[3] D. Batory et al. Creating Reference Architectures: An Example from Avionics. In Proceedings of the Symposium on Software Reusability (SSR), 1995.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering (TSE), 30(6), 2004.

[5] P. Clements and L. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, 2002.

[6] K. Pohl, G. B¨ockle, and F. van der Linden. Software Product Line Engineering. Foundations, Principles, and Techniques. Springer-Verlag, 2005.

[7] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90- TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[8] K. Kang, S. Kim, J. Lee, K. Kim, G. Kim, and E. Shin. FORM: A Feature-Oriented ReuseMethod with Domain-Specific Reference Architectures. Annuals of Software Engineering, 5(1):143–168, 1998.

[9] K. Czarnecki and U. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[10] J. Bosch. Design and Use of Software Architectures: Adopting and Evolving aProduct-Line Approach. ACM Press / Addison-Wesley, 2000.

[11] K. Chen, W. Zhang, H. Zhao, and H. Mei. An Approach to Constructing Feature Models Based on Requirements Clustering. In Proceedings of the International Conference on Requirements Engineering (RE), pages 31–40. IEEE CS Press, 2005.

[12] A. Classen, P. Heymans, and P. Schobbens. What's in a Feature: A Requirements Engineering Perspective. In Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE), volume 4961 of Lecture Notes in Computer Science, pages 16–30. Springer-Verlag, 2008.

[13] P. Zave. An Experiment in Feature Engineering. In Programming Methodology, pages 353–377. Springer-Verlag, 2003.

[14] D. Batory. Feature Models, Grammars, and Propositional Formulas. In Proceedingsof the International Software Product Line Conference (SPLC), volume 3714 of Lecture Notes in Computer Science, pages 7–20. Springer-Verlag, 2005.

[15] S. Apel, C. Lengauer, B. M¨oller, and C. K¨astner. An Algebra for Features and Feature Composition. In Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST), volume 5140 of Lecture Notes in Computer Science, pages 36–50. Springer-Verlag, 2008.

[16] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Transactions on Software Engineering and Methodology (TOSEM), 11(2), 2002.

[17] Lodewijk Bergmans and Mehmet Aksit. Composing synchronization and real-time constraints. Journal of Parallel and Distributed Computing, 36(1):32-52, July 1996.