Feature-Oriented Programming with Family Polymorphism

Fuminobu Takeyama Shigeru Chiba
Tokyo Institute of Technology, Japan
http://www.csg.is.titech.ac.jp/~{f_takeyama,chiba}

Abstract

In feature-oriented programming (FOP), code clones are also important issue. Although an approach called a software product line (SPL) enables to implement products efficiently by reusing most of their code, SPLs implemented by FOP contain a lot of code clones. Code clones are often caused by alternative features and we also found clones in derivatives among alternative features. To resolve this problem, we propose a new FOP language named FeatureGluonJ, which supports family polymorphism with revisers. The code clones among alternative features are separated into another feature and can be shared among the alternative features by extending that feature. Furthermore, clones in derivatives can be removed as well.

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features

General Terms Languages

Keywords Feature-Oriented Programming, Software Product Lines, Family Polymorphism, Java

1. Introduction

A software product line (SPL) is a widely used approach to develop a variety of products from a single set of artifacts, especially, a source code. Feature-oriented programming (FOP) [2, 11] has been developed in research community. FOP provides separation of concerns to implement SPLs; a SPL is decomposed into modules with respect to each feature. Programmers can develop products by selecting a subset of features.

In spite of the aim of SPL, it seems that FOP does not provide reusability to implement features. It is known that a lot of code clones exists in SPLs developed by FOP [13]. The code clones often occur in alternative features because they provide similar behavior and structures consisting classes. It is difficult to remove them because the clones are not identical and scatter over revisers, which are construct to modify an existing class. Clones are also found in derivatives, which is a module connecting multiple features, among alternative features. This problem must be resolved because these code clones might make a SPL unmaintainable and, as a result, reduce variability of SPLs.

This paper propose a new feature-oriented programming language, FeatureGluonJ. In our language, inheritance of feature module enables to remove such code clones to super feature like class inheritance of object-oriented programming. Derivatives can be considered as alternative features in our language. Thus it also can be implemented by extending a generic derivative for them.

2. Feature-oriented code clones

A significant aim of modular programming is to eliminate code clones. A design-level approach for this aim is a software product line (SPL) and feature-oriented programming (FOP) is known as programming paradigm useful for implementing SPLs. Although FOP enables code reuse among products, a large number of code clones still scatter over feature modules, which are selectable components of a SPL [13]. These code clones are often found in alternative features.

We below illustrate code clones found in MobileMedia [15], which is an SPL of multimedia viewers for mobile devices. Although the original version of MobileMedia is written in AspectJ,[1] this paper uses a version that we rewrote in

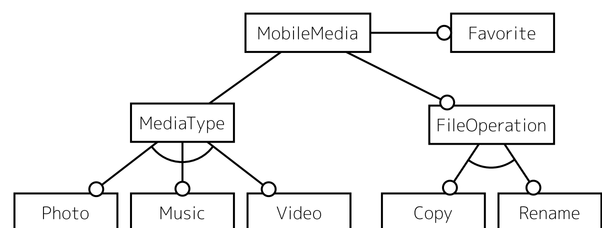

Figure 1. A feature model of MobileMedia SPL

[1] It is available from
`http://mobilemedia.sourceforge.net/`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VariComp'12, March 26, 2012, Potsdam, Germany.
Copyright © 2012 ACM 978-1-4503-1101-4/12/03. . . $10.00

```
class PhotoController {
  boolean handleCommand(Command command) {
    if (command == OPEN) {
      String selected = getSelected();
      Display.setCurrent(new PhotoViewScreen(selected));
    } else if (...) { ... }
}}

reviser PhotoTypeInitializer extends Application {
  private PhotoListScreen screen;
  private PhotoController controller;
  void startApp() {
    screen = new PhotoListScreen();
    controller = new PhotoController();
    super.startApp();
}}
```

Listing 1. Classes for the Photo feature

```
class MusicController {
  boolean handleCommand(Command command) {
    if (command == OPEN) {
      String selected = getSelected();
      Display.setCurrent(new MusicViewScreen(selected));
    } else if (...) { ... }
}}

reviser MusicTypeInitializer extends Application {
  private MusicListScreen screen;
  private MusicController controller;
  void startApp() {
    screen = new MusicListScreen();
    controller = new MusicController();
    super.startApp();
}}
```

Listing 2. Classes for the Music feature

GluonJ [3] and added extra features. MobileMedia has features shown in Figure 2. A feature indicated by an edge ending with a circle is an optional feature. Programmers can select only necessary features among optional features to customize a product. If a feature is not selected, it is not implemented in a resulting product. If programmers must select one of several features, the set of those features are called alternative features. They are indicated by edges with an arc. In this paper, furthermore, we extend the meaning of alternative features to cover a case in that programmers can select more than one feature among them. For distinction, we say that such alternative features are *combinatorial*. In MobileMedia, Photo, Music, and Video are alternative features. They implement a different type of medium.

For example, Listing 1 and 2 show a code clone between Photo and Music features. The bodies of PhotoController and MusicController classes are almost identical except the class name in the new expressions. This is natural since both features must support a similar set of commands such as open and close.

In Listing 1 and 2, a class-like construct starting with the reviser keyword is unique to GluonJ. It is called a reviser, which corresponds to a class with refines in AHEAD and an intertype declaration or an advice with an execution pointcut in AspectJ. It adds a new field or method to an existing class as open class [5] does. It can also replace an existing method with a new implementation as AspectJ's advice does.

```
reviser AddCopyToPhoto extends PhotoListScreen {
  void initMenu() {
    //add a menu item labeled "Copy" to a screen.
    addCommand(new CopyCommand());
}}

reviser AddRenameToMusic extends MusicListScreen {
  void initScreen() {
    //add a menu item labeled "Rename" to a screen.
    addCommand(new RenameCommand());
}}
```

Listing 3. The derivative features for Photo–Copy and Music–Rename

Another code clone is found in revisers PhotoTypeInitializer in Listing 1 and MusicTypeInitializer in Listing 2. Both add screen and controller fields (with different types) to the Application class. They also replace the startApp method in Application. The two implementations of startApp are identical except class names. Note that the two revisers are applied to the Application class one by one. When two reviser replace the same method, super.startApp() in a reviser invokes another implementation as Super().startApp() in AHEAD and proceed() in AspectJ. Due to space limitation, we do not mention further details in this paper.

A code clone is also found in a special feature called *derivative* [8, 9]. It is a feature that is needed to implement extra behavior only when two or more optional features are selected. Listing 3 shows a derivative for Photo and Copy and a derivative for Music and Rename. The revisers in these derivatives are clones of each other. For example, the AddCopyToPhoto reviser appends a command for copying a photo. This command must be effective only when the Photo and Copy features are selected. Note that MobileMedia contains several derivatives for every combination of media type and editing operation, such as Video–Copy and Photo–Rename.

A workaround to eliminate these clones shown above is to declare a super class of Photo and Music and put it into the Media feature. Then we can move code clone to that super class. Differences in instantiated class names can be addressed by using the factory method pattern. However, this approach will cause a large number of factory methods and annoying downcasts, which degrade maintainability. Furthermore, this approach does not work for code clones in revisers. How to design inheritance of revisers is still an open question. This is also true in AspectJ. In AspectJ, an aspect can inherit another aspect but the super aspect must be abstract. An abstract aspect itself does not change the behavior of a base program at all; it is only used as a code template. In the case of MobileMedia, we have to declare an abstract aspect in the Media feature and also declare, for each media type such as Photo and Music, a concrete aspect inheriting from that abstract aspect. This would be annoying. Another serious problem is that AspectJ provides only limited capability to customize an inherited code template. It allows a sub-aspect only to give concrete pointcut definitions. This

```
abstract feature MediaType {}

// classes below belong to MediaType
abstract class MediaController {
  boolean handleCommand(Command command) {
    if (command == OPEN) {
      String selected = getSelected();
      Display.setCurrent(new MediaViewScreen(selected));
      return true;
    } else if (...) { ... }
}}

class MediaListScreen {
  void initMenu() {}
  : // common codes among media types
}

abstract class MediaViewScreen extends Screen { ... }

reviser MediaTypeInitializer extends Application {
  MediaListScreen screen;
  MediaController controller;
  void startApp() {
    screen = new MediaListScreen();
    controller = new MediaController();
}}
```

Listing 4. The MediaType feature

```
feature PhotoMedia extends MediaType {}

class PhotoListScreen overrides MediaListScreen {
  : //photo-specific codes
}

class PhotoController overrides MediaController {
  : //photo-specific codes
}

class PhotoViewScreen overrides MediaViewScreen { ... }
```

Listing 5. The Photo feature implemented in our language

is not sufficient to absorb differences between PhotoType-Initializer and MusicTypeInitializer shown in Listing 1 and 2.

3. Modular FOP with family polymorphism

Since the code clones mentioned above cannot be eliminated by object-oriented programming technique in a satisfactory way, we need a new mechanism for FOP-specific code reuse. In this section, we show a new FOP language that supports inheritance of features. As code clones in classes can be removed to a super class in Java, clones in features can be eliminated in our language. We first show how clones in the alternative features in MobileMedia are removed. Then, we refactor the derivatives among the alternative features.

3.1 Family polymorphism with revisers

We propose a new FOP language named FeatureGluonJ, which support family polymorphism to implement SPLs more modularly. Family polymorphism is a traditional approach to reuse structure consisting of classes by extending multiple classes at once. In our language, a feature module, which is a unit for implementing a feature, is considered as a family. Besides feature module contains classes and revisers like existing FOP languages, programmers can define a new feature by extending another. The classes owned by a feature module are virtual classes [10]. A virtual class can override another virtual class defined in its super feature like a virtual function (method) overriding. A real class bound to a virtual class depends on what feature its host class belong to. See Listing 4. The feature declaration on top of the source represents MediaType feature owns classes described in that source. Photo is defined as shown in Listing 5 and have virtual classes that override the classes in MediaType. Instead of defining classes with same name, our language uses overrides to specify an overridden class. PhotoController derive methods, fields and constructors from MediaController while overriding. Since the MediaViewScreen is overridden by PhotoListScreen, the following code creates instance of PhotoListScreen and show it on the display:

```
PhotoController c = new PhotoController();
c.handleCommand(OPEN);
```

We adopt *copy semantics* for the notion of *select*, which is important in FOP. When a feature is selected for a product, its virtual classes and revisers can be used in the product; the revisers modify their target class. In our semantics, the classes in a super feature are copied to its sub feature. If programmers want to create a product that supports only photo viewer, they select thereby only Photo and does not select MediaType. Features marked abstract cannot be selected but the classes and revisers can be used to implement sub features. Programmers can try to create instance of an abstract virtual class in an abstract feature and such classes must be overridden in its sub features. Even if both of Photo and Music are selected, copied classes from MediaType are encapsulated within each feature and do not conflict.

Our polymorphism is unique in that a family contains revisers. A virtual reviser in a super feature is also copied to sub features. Although it appears that Photo in Listing 5 does not have any revisers, the feature module actually owns the MediaTypeInitializer reviser in Listing 4 derived from MediaType. A virtual reviser is polymorphic; its behavior depends on virtual classes in itself. The reviser derived by Photo creates fields and instances of components for it. Virtual revisers are applied when its feature is selected. If Photo and Music are selected, two MediaTypeInitializer in Photo and Music are applied. The remaining Music and Video can be implemented in the same way to Photo. Thus code clones among the alternative features are removed to MediaType.

Note that our family polymorphism is simplified in the similar way to lightweight family polymorphism [12]. Virtual classes and revisers appear only in direct children of a feature and there are no subtype relation between virtual classes from different features. Some readers might think the semantics of a reviser are similar to those of a virtual class. A reviser modifies target class globally and destructively. If a class is modified by two revisers, both of revisers might be executed everywhere code of the class is executed. On the other hand, virtual class overriding affects only inside of

```
feature PhotoCopy {
  import feature p: Photo;
  import feature c: Copy;
}

reviser AddCmdToPhotoList extends p::PhotoListScreen {
  void initMenu() {
    addCommand(new c::CopyCommand());
}}
```

Listing 6. Naive implementation of the PhotoCopy feature

```
abstract feature FileOperation {}

abstract class FileOperationCommand extends Command {
  FileOperationCommand(String lbl) {
    super(lbl, ITEM, 1);
}}
```

```
feature Copy extends FileOperation {}

class CopyCommand overrides FileOperationCommand {
  CopyCommand() {
    super("Copy"));
}}

class CopyController {
  ... // code to copy
}
```

Listing 7. The Copy feature without clone

```
abstract feature MediaTypeAndFileOperation {
  abstract import feature m: MediaType;
  abstract import feature f: FileOperation;
}

reviser AddCmdToMediaList extends m::MediaListScreen {
  void initMenu() {
    addCommand(new f::FileOperationCommand());
}}
```

```
feature PhotoCopy extends MediaAndFileOperation {
  import feature m: Photo; //override
  import feature f: Copy;  //override
}

// the reviser is derived
```

Listing 8. Implementation of PhotoCopy by extending a generic derivative

```
abstract feature MediaAndFileOp defines forevery(m, f) {
  abstract import feature m: MediaType;
  abstract import feature f: FileOperation;
}

reviser AddCmdToMediaList extends m::MediaListScreen {
  void initMenu() {
    addCommand(new f::FileOperationCommand());
}}
```

Listing 9. Definition of all derivatives by defines

its feature module. Even if Photo and Music are selected, multi-version of MediaController, which are named PhotoController and MusicController, exist in the product.

3.2 Implementing derivative modules by polymorphism

FeatureGluonJ enables to implement derivatives without clones as well. We first show how a naive derivative is implemented in our language. Since a derivative represents connection between features, programmers have to access classes from different features. Our language requires two steps to access external classes. First, programmers have to declare features used in the derivative by import feature. It is described in a body of a feature declaration and given the imported feature and an alias of the feature. Then virtual classes in the imported feature are accessible with feature qualified access, the :: operator. For example, we have derivative modules between Photo and Music in Listing 6. PhotoListScreen of Photo is described like p::PhotoListScreen.

Code clones among derivatives can be removed to a generic derivative and they can share the clone by using inheritance. Now we have the Copy feature implemented by extending the FileOperation feature in Listing 7. Rename is not shown but implemented as well. Now we can implement a derivative between two groups, sub features of MediaType and sub features of FileOperation, as shown in Listing 8. This derivative imports the super features of each group and its reviser is implemented by using virtual classes defined in those feature. Then, a concrete derivative, PhotoCopy, is defined by extending this generic derivative. abstract import feature is overridden by import feature with the same alias defined in sub features. With regard to PhotoCopy, since m and f are overridden with Photo and Copy, m::MediaListScreen and f::FileOperationCommand are bound to PhotoListScreen and CopyCommand, respectively. abstract import feature must be described in abstract features and overridden by concrete import feature.

Finally, programmers can define derivatives mentioned above at once if concrete derivatives contain only import feature like Listing 8. Listing 9 is the same as the generic derivative in Listing 8 except for the defines clause of the feature declaration. This defines automatically creates and select sub features extending this generic reviser for every pair of selected features that extends MediaType/FileOperation.

4. Discussion

Where code clones occurs in FOP? In MobileMedia, code clones seems to be found among alternative features. However some alternative features are defined not in domain level but in implementation level. Programmers might notice that two or more features have similar structures at implementation or design stage. These features can be considered as alternative features and implemented efficiently by family polymorphism. Derivatives among alternative features are also implementation level alternative features.

Family polymorphism versus obliviousness Although family polymorphism does reduce obliviousness a little, it is not so critical as code clones reduce maintainability and variability. For example, since pure obliviousness is preserved in Listing 1 and 2, while programmers implement Photo, they need not take care of Music. Thus, in Listing 3, menus of

each screen are initialized in methods with different names: initMenu() in Photo and initScreen() in Music. This might cause, however, troublesome clones, which are semantically similar but not syntactically. This type of clone is difficult to be found by tools and implement generic derivatives. On the other hand, by introducing their super feature, programmers have to know about the existence of other types of medium through it; They must use initMenu() to initialize menus.

Why code clones in derivative modules have not been revealed? Although there are surveys on derivative modules and code clones of software product lines, code clones among derivative modules are not mentioned. This is because the size of alternative feature is too small although there are huge alternative features such as file systems or device drivers in Linux kernel, in SPLs which are not implemented in FOP. Since the number of derivatives between two sets of alternative features might be $n \times m$ where n and m are the sizes of alternative features, the impact of these code clones becomes more serious in larger systems.

5. Conclusions and future work

We proposed a new feature-oriented programming language, named FeatureGluonJ. It supports family polymorphism; a feature can be defined based on an existing feature. Our reviser is polymorphic and its behavior changes depending on its host feature as well as virtual classes. The alternative features in MobileMedia can be implemented more modularly by using inheritance.

Related work includes CaesarJ [1], which is a programming language supporting both AOP and family polymorphism. Although it enables programmers to implement a generic and family polymorphic aspect for alternative features like our language, it does not support intertype declarations and implementing a generic derivative because of access limitation to virtual classes from aspects. On the other hand, programmers can also implement SPLs by mixin composition of classes that implement selected features [6]. However, since a super class (feature) is mixed only once, combinatorial alternative features cannot be implemented without clones mentioned in this paper.

Another approach to implement SPLs and remove code clones is performed by IDEs. CIDE [7] is IDE developed on Eclipse and support feature decomposition by assigning colors to code snippets. Backgrounds of snippets belonging to a certain feature are filled in the same color. Programmers can remove and revert snippets assigned in the same color from source code like #ifdef directives of the C preprocessor. However, code clones among combinatorial alternative feature occurs also in SPLs developed by CIDE. On the other hand, we adopt an approach using syntactical constructs. It is important to search solutions from both side. For example, clones with minor differences can be managed by synchronized editing [4]. It can be extended to support grouping clones, which corresponds to our family inheritance.

Our future work includes another type of code clones caused by language constructs such as using of GluonJ and resolvers of Airia [14]. These constructs are used for controlling scope and precedence order of revisers and described in derivatives. We need formal definition of type system of our language. This can be defined based on type systems of GluonJ and lightweight family polymorphism.

References

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, pages 135–173. Springer Berlin / Heidelberg, 2006.

[2] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *Software Engineering, IEEE Transactions on*, 30 (6):355–371, 2004.

[3] S. Chiba, A. Igarashi, and S. Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In *OOPSLA*, pages 539–554. ACM, 2010.

[4] S. Chiba, M. Horie, K. Kanazawa, F. Takeyama, and Y. Teramoto. Do we really need extending syntax for advanced modularity? In *AOSD Modurality Visions*, 2012. to appear.

[5] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA*, pages 130–145. ACM, 2000.

[6] V. Gasiunas and I. Aracic. Dungeon: A case study of feature-oriented programming with virtual classes. In *Proceedings of the 2nd Workshop on Aspect-Oriented Product Line Engineering*, 2007.

[7] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, pages 311–320. ACM, 2008.

[8] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the impact of the optional feature problem: analysis and case studies. In *SPLC*, pages 181–190. Carnegie Mellon University, 2009.

[9] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE*, pages 112–121. ACM, 2006.

[10] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA*, pages 397–406. ACM, 1989.

[11] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Aksit and S. Matsuoka, editors, *ECOOP*, pages 419–443. Springer Berlin / Heidelberg, 1997.

[12] C. Saito, A. Igarashi, and M. Viroli. Lightweight family polymorphism. *Journal of Functional Programming*, 18:285–331, 2008.

[13] S. Schulze, S. Apel, and C. Kästner. Code clones in feature-oriented software product lines. In *GPCE*, pages 103–112. ACM, 2010.

[14] F. Takeyama and S. Chiba. An advice for advice composition in AspectJ. In *Software Composition*, pages 122–137. Springer Berlin / Heidelberg, 2010.

[15] T. Young and G. Murphy. Using AspectJ to build a product line for mobile devices. AOSD 2005 Demonstrations, 2005.