

# Implementing Feature Interactions with Generic Feature Modules

Fuminobu Takeyama<sup>1</sup> and Shigeru Chiba<sup>2,3</sup>

<sup>1</sup> Tokyo Institute of Technology, Japan

[http://www.csg.ci.i.u-tokyo.ac.jp/~f\\_takeyama](http://www.csg.ci.i.u-tokyo.ac.jp/~f_takeyama)

<sup>2</sup> The University of Tokyo, Japan

<http://www.csg.ci.i.u-tokyo.ac.jp/~chiba>

<sup>3</sup> JST, CREST, Japan

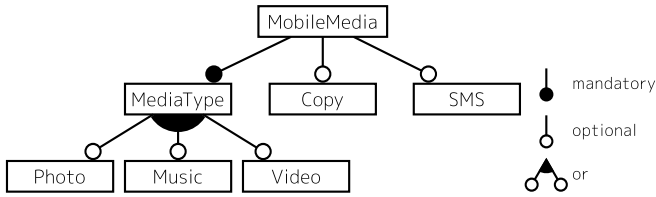
**Abstract.** The optional feature problem in feature-oriented programming is that implementing the interaction among features is difficult. Either of the modules for the interacting features cannot contain the code for the interaction if those features are optional. A modular approach for implementing such interaction is separating it into a module called derivative. However, as the number of derivatives increases, it does not scale. This paper shows how derivatives for combinations of features from each group are efficiently implemented. A group of features are implemented by using the inheritance of feature modules. A super feature module works as a common interface to members of that group. It thereby allows to describe a generic derivative applicable for the groups. This paper also presents a feature-oriented programming language, FeatureGluonJ, which provides language constructs for this approach.

## 1 Feature-Oriented Programming

Feature-oriented programming (FOP) [26] is a programming paradigm where source code is decomposed for each feature. Although it was originally an approach for implementing similar classes, it now refers to an approach for implementing similar software products; such a family of products is called a software product line (SPL). This allows developers by just selecting the features for that necessary product.

In FOP, the code for each feature is separately described in a module called a feature module. A feature module is a collaboration of the classes needed for the feature and/or extensions to the classes belonging to other features. The extensions can be aspects in AspectJ; advices can attach code for the feature to existing code; inter-type declarations can add new fields to an existing class. Several product lines such as the feature-oriented version of Berkeley DB [18] and MobileMedia [30] have been developed in AspectJ. AHEAD Tool Suite [5] has a language construct called a refinement for the same purpose. It enables overriding existing methods and add fields from outside.

A challenge in FOP is the optional feature problem [21]. If multiple optional features interact with each other, any of feature modules for those features should



**Fig. 1.** The feature model of MobileMedia

not contain the code for the interaction since the code must be effective only when those interacting features are selected. Although a possible approach is separating such code into independent modules called *derivatives* [21,22], the number of derivatives tends to be large as the numbers of features increases.

This paper proposes a design principle to reduce the effort in implementing derivatives. A group of features are implemented by inheriting their common super feature module. This module works as an interface common to its sub-features and allows implementing a derivative in a reusable manner for every combination of sub-features. To demonstrate this principle, we developed a new FOP language, FeatureGluonJ, which provides a language construct called a generic feature module for reusable implementation of derivative as well as a feature-oriented module system supporting inheritance.

## 2 The Optional Feature Problem

This section explains a difficulty in feature-oriented decomposition known as the optional feature problem [21,22]. We can see this problem in the MobileMedia SPL. MobileMedia is a family of multimedia-management application for mobile devices and widely used in the research community of SPLs. This paper uses the six features taken from MobileMedia: *MediaType*, *Photo*, *Music*, *Video*, *Copy*, and *SMS* in Fig. 1. The representation in Fig. 1 is called a feature-model diagram [17]. In this diagram, a node represents a feature, and an edge represents dependence between features. A feature-model diagram also represents constraints on selecting a feature when building a product. A feature indicated by an edge ending with a white circle is called an optional feature. Developers select features from optional features to customize a product. If a feature is not selected for the product, that feature is not implemented in the resulting product. In Fig. 1, *Photo*, *Music*, and *Video* are children of *MediaType*. The arc drawn among the children represents a *or*-relation and developers must select at least one feature from them if their parent is selected. This paper considers such features as being optional features as well.

In FOP, a feature should be implemented as an independent feature module. If a feature is selected, the corresponding module is compiled together with other selected feature modules. In case of the original MobileMedia implemented in

AspectJ if a feature is selected, aspects belonging to the feature are compiled and linked. If a feature is not selected for a product, its aspects do not affect the product.

However, a group of optional features may interact [9] with another group. Which feature module should implement that interaction? For example, the `Photo` feature interact with the `Copy` feature in `MobileMedia`. If both features are selected for a product, then the command for copying a photo is displayed in the pull-up menu of the screen, i.e., window, showing that photo. This command should not be implemented in a feature module for `Photo` or `Copy` since the command is not activated unless `Copy` is selected. It should not be in a feature module for `Copy` since `Copy` may be selected without `Photo`. If so, the `Copy` feature module must not add the command to the menu. No matter where the command is implemented, `Photo` or `Copy`, the resulting code would cause undesirable dependence among optional features and lower the variability of a product line.

A more modular approach is to implement such interaction into an independent module called a *derivative*. A derivative is a specialized feature module for interaction, which is selected only when all interacting features are selected.<sup>4</sup> A derivative is described as a normal feature module. We show a derivative for the combination of `Photo` and `Copy` in List. 1. This code is a part of the `MobileMedia Lancaster`<sup>5</sup>, a `MobileMedia` implementation in AspectJ [12]. The `CopyAndPhoto` aspect implements the derivative. It has an advice executed after a constructor call for the `PhotoViewScreen` in order to add the command for copying a photo.

The scalability of derivatives is, however, still under discussion in the research community. Suppose that  $n$  optional features interact with each other. Naively, each of the  $2^n - n - 1$  combination of features requires its own derivatives. The composability may reduce the number of derivatives. The paper [26] advocates that if developers provide *lifters*, which can be regarded as derivatives, for every *pair* of interacting features, then the lifters for any combination of the features can be composed by those lifters; the number of necessary lifters are thereby  $\frac{1}{2}(n^2 - n)$ . In practical product lines, although all features do not interact with each other, a number of derivatives are still required. For example, in Berkeley DB refactored in FOP [18], 38 features have 53 dependencies, which must be separated into derivatives. The paper [20] concludes that the difficulty in implementing features is mainly due to the interaction among the features.

Note that feature interaction is often observed between feature groups. Suppose that two feature groups have  $n$  and  $m$  features. If a feature from one group interact with one from the other, other pairs between the two groups will also interact with each other due to the similarity of features. Such interaction will require  $n \times m$  derivatives in total. Furthermore, these derivatives will be similar

---

<sup>4</sup> In the original definition in [22], a derivative is a refinement of a method introduced by another feature module.

<sup>5</sup> We show simplified code for explanation. The original code is available from: <http://mobilemedia.sourceforge.net/>.

---

```

public aspect CopyAndPhoto {
    after(Image image) returning (PhotoViewScreen f):
        call(PhotoViewScreen.new(Image)) && args(image); {
        f.addCommand(new Command("Copy", Command.ITEM, 1));
    }
}

```

(a) CopyAndPhoto.aj

---

```

public aspect CopyAndMusic {
    pointcut initForm(PlayMediaScreen mediaScreen):
        execution(void PlayMediaScreen.initForm()) && this(mediaScreen);

    before(PlayMediaScreen mediaScreen): initForm(mediaScreen) {
        mediaScreen.form.addCommand(new Command("Copy", Command.ITEM, 1));
    }
}

```

(b) CopyAndMusic.aj

---

**List. 1.** The derivatives for Copy and Photo/Music written in AspectJ

to each other. They are redundant and should be merged into a single or only a few derivatives.

A group is often represented by a parent-children relation in a feature-model diagram. In Fig. 1, `MobileMedia` contains a group consisting of `Photo`, `Music`, and `Video`. We call this group the `MediaType` group named after the parent node. There is another group that the developers of the original `MobileMedia` did not recognize. It is a group consisting of `Copy` and `SMS`, which enable the users to send a photo shown on the screen by SMS. The two groups involve close interaction. `Copy` interacts with `Music` as well as `Photo`. The derivative for `Copy` and `Music` in List. 1 (b) is similar to `CopyPhoto` in List. 1 (a). `SMS` also interacts with `Photo`, `Music` and `Video`<sup>6</sup>; if these features are selected, a command to send each medium must be added to the menu. Thus, `MobileMedia` requires 6 derivatives for the two groups.

The `MediaType` group is an extension point of `MobileMedia`. One of the goal of FOP is step-wise, *i.e.* incremental, development of large-scale software [5], and hence one of realistic development scenarios is adding a new media type as a new feature. Suppose that developers add plain-text documents as a new medium. Then they will have to implement derivatives for the combination of the plain-text feature and `Copy` and `SMS`. The effort to implement derivatives will increase as the size of the product line grows up.

In this paper, the optional feature problem means not only the difficulty in separating feature interaction but also the maintainability problem due to a huge number of derivatives. This paper addresses this optional feature problem by reducing redundancy of derivatives among feature groups. The interactions discussed in this paper are intended ones. Although there are unintended interactions caused by unanticipated advice conflicts, for example, at shared join points [1], discussing unintended interactions is out of scope of this paper.

---

<sup>6</sup> The original `MobileMedia` does not support to send a music or a video by SMS. It is not clear that this limitation is caused by the optional feature problem.

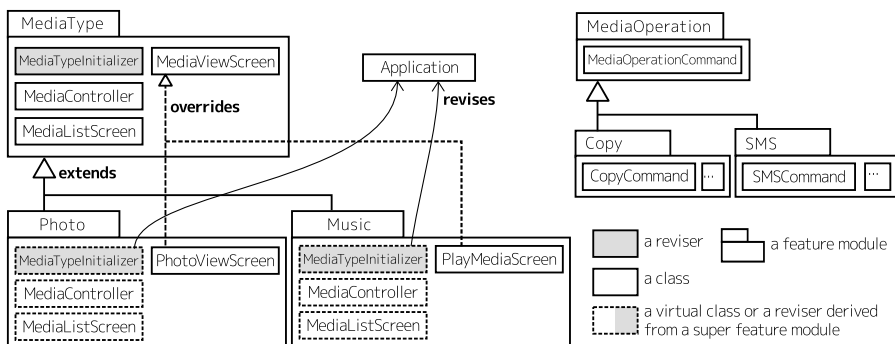


Fig. 2. An overview of feature modules consisting MobileMedia in FeatureGluonJ

### 3 Implementing Feature Interactions in FeatureGluonJ

Our feature-oriented programming language named *FeatureGluonJ*<sup>7</sup> provides language constructs to reduce redundancy of derivatives among feature groups. FeatureGluonJ is an extension of GluonJ [10], which is a aspect-oriented language based on Java. While GluonJ adds a new language construct called *reviser* to Java as a construct like AspectJ's advice. FeatureGluonJ also adds a generic feature module as a feature-oriented module system.

First, FeatureGluonJ provides an inheritance mechanism for feature modules. Features often make is-a relations [16]. In MobileMedia, the *Photo* feature is a *Media* feature. Thus, in FeatureGluonJ, the *Photo* feature module, which is the implementation of *Photo*, is a sub feature module of *Media* as shown in Fig. 2. It can not only add new classes but also redefine the classes contained in the *Media* feature module. The *Media* feature module works as a common interface to this feature group including *Photo* and *Music*. The interface represents which classes are commonly available in the feature group. This inheritance mechanism is not novel; it is provided by Caesar [24,25], CaesarJ [4], and Object Teams [14,16]. However, they are not studied in the context of modularity of feature modules [20,29]; this paper focuses on how to use this inheritance mechanism to efficiently implement derivatives.

Another unique mechanism in FeatureGluonJ is a generic feature module. It is a feature module taking feature modules as parameters. Suppose that there are two feature modules. Then the derivatives for combinations of their sub feature modules are often almost identical. For example, in Fig. 2, the derivative for *Photo* and *Copy* is almost identical to the derivative for *Music* and *SMS* since they are for combinations between *Media* and *MediaOperation*. A generic feature module enables to describe such derivatives in a generic manner by using the interfaces specified by *Media* and *MediaOperation*. Note that the task of a typical derivative is to modify the classes in the feature modules that the

<sup>7</sup> The FeatureGluonJ compiler is available from:

<http://www.csg.ci.i.u-tokyo.ac.jp/projects/fgj/>

derivative works for. These classes are often ones specified by the interfaces of the super feature modules such as `MediaType` and `MediaOperation`.

### 3.1 FeatureGluonJ

This section describes the overview of FeatureGluonJ to show how developers can implement an SPL<sup>8</sup>. FeatureGluonJ provides a module system called feature modules. A feature module implements a feature and a derivative. It is represented by two constructs, a feature definition and a feature declaration. A feature definition is described in a separated file, and it defines a feature name and its relation to other features. List. 2 (a) defines the `MediaType` feature, which is an abstract feature for other features that are to support a media type. The body of this feature is empty in this example, but it may contain `import feature` declarations, as shown later.

A feature declaration is similar to a `package` declaration in Java. It is placed at the beginning of a source file and specifies that the classes and revisers in that source file belong to the feature modules. For example, the second lines of the List. 2 (b)–(e) are feature declarations. They declare that those three classes and a reviser belong to the `MediaType` feature. Note that each class declaration is separated into an independent file.

An abstract feature may represent a group made by *is-a* relationships; a sub feature module of that abstract module defines a feature belonging to that group. Here, the `Photo` feature module is a sub-feature of `MediaType`, which is specified in the `extends` clause in List. 3 (a). `Photo` reuses the model-view-controller relation defined in `MediaType`.

After compilation of each feature, developers *select* feature modules needed for a product. Only the selected feature modules are linked together and included in the product. Which features are selected is given at link time. Note that they cannot select abstract features. If an abstract feature module like `MediaType` must be included in a product, the developers must select a sub-feature of that abstract feature.

To implement feature modules, FeatureGluonJ provides three kinds of class-extension mechanisms: subclasses, virtual classes, and revisers. The difference of those mechanisms is the range of effects. The first one is a normal subclass in Java and affects in the narrowest range. The extended behavior takes effect only when that subclass is explicitly instantiated.

The next class extension mechanism is virtual class overriding [23,11]. Virtual classes enable to reuse a family of classes that refer to each other through their fields or new expressions. All classes in a feature module are virtualized in FeatureGluonJ; a reference to a virtual class is late-bound. A sub feature module can implement a virtual class extending a virtual class in its super-feature. It *overrides* the virtual class in the super-feature with the new class, *i.e.*, class

---

<sup>8</sup> We cannot describe all of the semantics of our language in detail due to the space limitation. Knowledge on virtual classes and other languages with inheritance for feature module will help to read this section.

---

```

abstract feature MediaType {
    // MediaType has an empty body.
}
.....
                                (a) MediaType.feature
.....
package mobilemedia.controller;
feature MediaType;
import javax.microedition.lcdui.*;
import mobilemedia.ui.*;

public abstract class MediaController {
    protected boolean handleCommand(Command command) {
        if (command == OPEN) {
            open(getSelected());
            return true;
        } else if (...) { ... }
    }

    protected void open(String s) {
        MediaListScreen scr = new MediaViewScreenScreen(s);
        scr.setCommandListener(this);
        Display.setCurrent(scr);
    }
}
.....
                                (b) MediaController.java
.....
package mobilemedia.ui;
feature MediaType;
import javax.microedition.lcdui.*;

public abstract class MediaViewScreen extends Canvas {
    protected void initScreen() {
        this.add(new Command("Close"));
    }
}
.....
                                (c) MediaViewScreen.java
.....
package mobilemedia.ui;
feature MediaType;
import javax.microedition.lcdui.*;

public class MediaListScreen extends List {
    // forward command to controller if an item is selected
}
.....
                                (d) MediaListScreen.java
.....
package mobilemedia.main;
feature MediaType;
import mobilemedia.ui.MediaListScreen;
import mobilemedia.controller.MediaController;

class MediaTypeInitializer revises Application {
    private MediaListScreen screen;
    private MediaController controller;
    public void startApp() {
        controller = new MediaController();
        screen = new MediaListScreen(controller);
        super.startApp();
    }
}
.....
                                (e) MediaTypeInitializer.java
.....
package mobilemedia.main;

public class Application {
    public static void main() {
        Application app = new Application();
        app.startApp();
    }

    public void startApp() { // initializing this MobileMedia application }
}
.....
                                (f) Application.java

```

---

**List. 2.** The MediaType feature module and the Application class, which has program entry point

---

```

feature Photo extends MediaType {} (a) Photo.feature
.....
feature Photo;
package mobilemedia.ui;
import javax.microedition.lcdui.*;

public class PhotoViewScreen overrides MediaViewScreen {
    public PhotoViewScreen(String s) {
        // : load selected image
    }
    protected void paint(Graphics g) {
        // : draw the selected photo on this screen.
    }
}}

```

(b) PhotoViewScreen.java

---

**List. 3.** The Photo feature in FeatureGluonJ

references to the overridden class is replaced with one to the new class. Virtual class overriding is effective only within the enclosing feature module, which includes its super-feature module executed as a part of the sub-feature. It does not affect new expressions in the siblings of the sub-feature.

The syntax of virtual classes in FeatureGluonJ is different from other languages. To override a virtual class, developers must give a unique name to the new virtual class instead of the same name as the overridden class.<sup>9</sup> List. 3 (b) shows the `PhotoViewScreen` class that overrides `MediaViewScreen` of `MediaType`. An overridden class is specified by an `overrides` clause, placed in the position of an `extends` clause. Another difference in syntax is that virtual classes cannot be syntactically nested, as separated into each class to a single file.

We adopt lightweight family polymorphism [27] to make the semantics and the type system simple by avoiding dependent types. A feature module cannot be instantiated dynamically. It can be regarded as a singleton object instantiated when it is selected at link time.

The third mechanism is a reviser [10]. A reviser can extend any class in a product; the extended behavior affects globally.<sup>10</sup> A reviser plays a similar role to the one of aspect in AspectJ; its code overrides classes appearing in any other feature module. The class-like mechanism with a keyword `revises` in List. 2 (e) is a reviser. The reviser has the `startApp()` method, which replaces the `startApp()` method in the class specified in its `revises` clause, *i.e.*, the `Application` class in List. 2 (f). Whenever the `startApp()` method is called on an `Application` object, the reviser's `startApp()` method is first executed. By calling `super.startApp()`, the replaced method is executed. A reviser can also add new fields to an existing class. The reviser in List. 2 (e) adds the two fields, `screen` and `controller`, to the `Application` class.

Revisers in a feature module are also virtualized. A feature module derives revisers as well as virtual classes from its super-feature to reuse structure made by revisers and classes defined there. The `Photo` feature module in List. 3 does not contain any classes and revisers except the `PhotoViewScreen` class, but it also

<sup>9</sup> Programmers can give the same name by implementing them in a different package.

<sup>10</sup> GluonJ does not support global modification defined in [3].



---

```

feature Music extends MediaType {} (a) Music.feature
.....
feature Music;
package mobilemedia.ui;
import javax.microedition.lcdui.*;

public class PlayMediaScreen overrides MediaViewScreen {
    public PlayMediaScreen(String s) {
        // : load selected image
    }
    protected void paint(Graphics g) {
        // : draw music player
    }
}

```

(b) PlayMediaScreen.java

---

**List. 4.** The Music feature module in FeatureGluonJ

derives virtual classes such as `MediaController` and the `MediaTypelInitializer` reviser from `MediaType` (Fig. 2). A reviser will be executed only if a feature enclosing or deriving that reviser is selected. If `Photo` is selected, `MediaTypelInitializer` derived by the `Photo` is executed in the context of `Photo`. Within this `MediaTypelInitializer`, new `MediaListScreen(...)` will create an object of the class derived by `Photo`. The expression `new MediaViewScreen()` in List. 2 (b) on that object will instantiate `PhotoViewScreen`. The `MediaTypelInitializer` reviser might be derived by siblings of `Photo`. Suppose the `Music` feature module in List. 4 is implemented in the same way as the `Photo`. If both `Photo` and `Music` are selected, two *copies* of `MediaTypelInitializer` will be executed in the `startApp()` method but in different contexts.

These class extension mechanisms provided by `FeatureGluonJ` are an abstraction of the factory method pattern. For virtual-class overriding, each selected feature has its own factory. It receives a name of virtual class and returns an object of the class overriding the given class. Every `new` expression can be considered as a factory method call. Each virtual class has a reference to such factories. When a factory creates an object, it assigns itself to the object. A factory used in a reviser is given by the linker when its feature module is selected. This factory is one used for virtual classes in the feature module containing or deriving that reviser.

A reviser, on the other hand, can be emulated by a factory shared among all the classes in a product. If a class given to a `new` expression is not a virtual class, the global factory will create an object. This global factory is also used inside of a factory for each feature. Note that it is unrealistic to manually implement factory methods for every class. Moreover, a factory method pattern degrades type safety.

### 3.2 Derivatives in FeatureGluonJ

`FeatureGluonJ` provides two other constructs for referring to virtual classes in other modules. One is `import feature` declarations. To make coupling of other features explicit, developers have to declare the features required in a derivative

---

```

feature CopyPhoto {
  import feature c: Copy;
  import feature f: Photo;
}

```

(a) CopyPhoto.feature

---

```

package mobilemedia.copy;
feature CopyPhoto;
import mobilemedia.ui.*;

class AddCopyToPhoto revises f::PhotoViewScreen {
  protected void initScreen() {
    super.initScreen();
    this.addCommand(new c::CopyCommand());
  }
}

```

(b) AddCopyToPhoto.java

---

**List. 5.** The derivative between Copy and Photo rewritten from List. 1

by `import feature` declarations. These declarations just open the visibility scope to virtual classes of imported features. Note that when a feature module with `import feature` is selected, the imported features are also selected. Since an abstract feature module is never selected, it cannot be imported.

An `import feature` declaration is described in the body of a feature declaration. List. 5 (a) contains two `import feature` declarations. An identifier after a colon indicates a feature module used in this module. The left one before the colon is an alias to the imported feature module. Then a *feature-qualified access* is available as a reference to a virtual class of the imported feature module. The access is represented by a `::` operator. The left of `::` must be an alias declared in the feature module and the right of `::` is the name of a virtual class in the feature module expressed by the alias. For example, List. 5 implements a derivative straightforwardly rewritten from List. 1. In the `AddCopyToPhoto`, `p::PhotoViewScreen` refers to the `PhotoViewScreen` class in the `Photo` feature since `p` is an alias of `Photo`. The reviser extends `PhotoViewScreen` and adds a command for copying a medium, which is now represented by the `CopyCommand` class in the `Copy` feature module shown in List. 6 (c) and (d).

The reason `FeatureGluonJ` enforces programmers to use feature-qualified access is that multiple feature modules may contain virtual classes with the same name if they extend the same module. For example, both of `Photo` and `Music` contains the `MediaController` class derived from `MediaType`, which are distinguished by aliases.

### 3.3 Generic Feature Modules

We found that if features are implemented by a feature module with an appropriate interface, most derivatives can be implemented by a special feature module that takes the name of required sub-features as parameters. `FeatureGluonJ` provides a *generic feature module*, which is a reusable feature module to implement derivatives among features extending common feature modules. The `Copy` feature and the `SMS` feature, which is not shown but implemented in the same way in List. 6, are sub feature modules of `MediaOperation` in List. 6 (a) and (b). Now the



---

```

feature MediaOperationMediaType {
    abstract import feature o: MediaOperation;
    abstract import feature t: MediaType;
}
.....................................................................
(a) MediaOperationMediaType.java
package mobilemedia.mediaop;
feature MediaOperationMediaType;
import mobilemedia.ui.*;

class AddCommandToMediaType revises t::MediaViewScreen {
    protected void initForm() {
        super.initForm();
        this.addCommand(new o::MediaOpCommand());
    }
}
(b) AddCommandToMediaType.java

```

---

**List. 7.** A generic derivative implementing common part of derivatives among `MediaOperation` and `MediaType`

---

```

feature CopyPhoto extends MediaTypeFileOp {
    import feature o: Copy;
    import feature t: Photo;
}

```

---

**List. 8.** Another derivative for `Copy` and `Photo` extending the generic derivative

A feature module can extend a generic feature module and bound aliases declared in its super-feature to concrete feature modules. If a feature module imports another feature module with the same alias as one used in its super-feature, the alias is bound to that feature module also in the super-feature. Suppose another implementation of the `CopyPhoto` feature module, which is implemented by in List. 8 by extending the generic feature module in List. 7. It assigns `o` and `t` to `Copy` and `Photo` respectively.

### 3.4 A Composition Language for Trivial Feature Modules

If each interacting feature is properly implemented, a derivative may not contain its own reviser nor class; we call such derivative is trivial. The derivatives among `MediaType` and `MediaOperation` including `CopyPhoto` in List. 8 are trivial. This is because operations such as copying a medium or sending it by SMS are reduced to operations against streams of bytes.

FeatureGluonJ also provides a composition language to define such trivial derivatives implicitly. It includes a construct `defines forevery`. If our linker interprets a feature module with `defines forevery`, it defines sub-features of this derivative automatically at linking time. `defines forevery` receives one or more aliases to feature modules. If the given alias is declared in an `abstract import feature`, it represents a set of its sub-features that are selected for the linker. The linker will define and select sub feature modules for every combination from each given set. Let  $a_1, a_2, \dots, a_n$  be aliases given to the `defines forevery` and  $S_i = \{f | f \in Sub(a_i) \cap f \text{ is selected}\}$  where  $Sub(a)$  is a function returning the

---

```

feature MediaOperationMediaType defines forevery(o, t) {
  abstract import feature o: MediaOperation;
  abstract import feature t: MediaType;
}

```

(a) MediaTypeFileOp.java

```

.....
package mobilemedia.mediaop;
feature MediaOperationMediaType;
import mobilemedia.ui.*;

```

```

class AddCommandToMediaType revises t::MediaViewScreen {
  protected void initForm() {
    super.initForm();
    this.addCommand(new o::MediaOpCommand());
  }
}

```

(b) AddCommandToMediaType.java

---

**List. 9.** Our final version of derivatives among `MediaOperation` and `MediaType` by defines forevery

set of the sub-features that might be bound to  $a$ . A sub-derivative is created for each element of  $S_1 \times S_2 \times \dots \times S_n$ . If  $a$  is an alias of a concrete feature,  $Sub(a)$  returns the set containing the concrete feature only.

List. 9 shows derivatives for sub-features of `MediaOperation` and `MediaType` including derivative between `Copy` and `Photo`. The `defines forevery` clause allows programmers to omit concrete feature modules such as one in List. 8. Even when developers add a new feature for a new media type, they would not implement new derivatives if this generic derivative is applicable for the new feature. Otherwise, programmers would implement extra behavior for the specific combinations of feature modules as a new derivative.

### 3.5 Discussion

We discuss on the limitations of our language. Unfortunately, all derivatives do not become trivial after refactoring. Some derivatives are essential, which must be implemented manually. We can find essential derivatives in the expression product line [20]. Derivatives among a feature for an operator and feature for evaluating expressions is unique to each combination of features. If the feature has redundant parts, `FeatureGluonJ` allows to reuse it with a generic-feature module.

Although inheritance allow us to implement generic derivatives, it may cause extra effort to implement an SPL. We introduced the common super class between `PhotoViewScreen` and `PlayMediaScreen`. As shown in List. 1 (a) and (b), the original derivative uses different methods to add their commands to the menus; in `Photo`, it is the constructor of `PhotoViewScreen`, but in `Music`, it is the `initForm()` method. We add the common super class `MediaViewScreen` and its `initScreen()` method in List. 2 (c) to unify those methods among both features. We also defines `Copy` and `SMS` by extending `MediaOperationCommand` to make the derivatives trivial. The implementations of these feature modules are in a sense composition aware.

Our observation is that whether or not we should implement each feature considering composition is a design decision. The `MediaType` group and the `MediaOperation` group are extension points; in other words, a new feature will be added to these groups in the future. The cost of making these features composable is much lower than a large number of derivatives.

## 4 Related Work

Most of feature-oriented approaches such as AHEAD Tool Suite [5] are based on the idea that a feature is represented as a layer, and a product is linearly stacked layers [7]. FeatureGluonJ and other language with inheritance of feature modules allows to reuse a feature modules including a derivative multiple times in different contexts. ClassBox/J [8] can emulate virtual-classes by refinement and a scope mechanism to control the effects of refinements. Aspectual Mixin Layers [2] provide refinements for modifying advices and pointcuts of aspects defined in other feature modules. However, those languages also do not support to execute such refinements or aspects in different contexts to reuse them.

Delta-oriented programming [28] is a language paradigm where a product is composed of delta modules unique to it. A delta module can modify existing classes as a reviser can. A delta module has a conditional expression specifying when it is applied. Although a derivative is represented by a delta module applied when several features are selected together, delta-oriented programming does not provide mechanisms to reduce the number of delta modules for combinations among groups. We believe that it cannot solve the optional feature problem in this paper.

Caesar [24], CaesarJ [4] are Java-based language supporting both virtual classes and advices from AspectJ. Object Teams [14] is also a language based on Java and provides teams consisting of virtual classes. A virtual class in Object Teams is called a role class and programmers can extend another class so that it plays the role, *i.e.*, behaves as the role class defines. *Callin* binding in Object Teams allows to transfer method call on the extended class, to the role class. In those languages, virtual classes with advices or role classes are also derived from the super-feature like virtual revisers in FeatureGluonJ.

The difference from those languages is language support for generic derivatives. Object Teams provides a dependent team, which behaves polymorphically depending on a given instance of a team. The origin of the dependent team is a dependent class [13]. Dependent revisers could be as expressive as our languages. However current specification of Object Teams [15] does not allow teams dependent to multiple teams. Dependent teams hence cannot be used for derivatives among groups. Those languages may allow to demonstrate our design principle by first class objects of features, but it requires boilerplate code for each product.

In annotation based approaches for SPLs, code regions implementing a feature is annotated with syntactical blocks, `#ifdef` and `#endif`, or a color in CIDE [19]. An interaction is indicated by an intersection of these regions [6]. Although

this representation of interactions is more intuitive than a derivative, reusability of code for the interactions is not clear.

## 5 Conclusion

In this paper, we have shown how derivatives among feature groups are implemented efficiently in FeatureGluonJ. Firstly, designing feature modules hierarchically makes features modular and important for implementing derivatives. FeatureGluonJ facilitates to implement generic derivatives among feature groups represented by the inheritance. Such derivatives are written by using super-features as interfaces.

Our future work includes formal definition of semantics of FeatureGluonJ. FeatureGluonJ is based on GluonJ and light-weight family polymorphism which have formal definitions to prove they are mostly modular and type safe, respectively. The definition of our language will be valuable to show that it derives those properties. Real evaluation of our language requires more SPLs described in FeatureGluonJ.

## References

1. Aksit, M., Rensink, A., Staijen, T.: A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In: AOSD, pp. 39–50. ACM, New York (2009)
2. Apel, S., Leich, T., Saake, G.: Aspect refinement and bounding quantification in incremental designs. In: APSEC, pp. 796–804. IEEE Computer Society, Los Alamitos (2005)
3. Apel, S., Lengauer, C., Möller, B., Kästner, C.: An algebraic foundation for automatic feature-based program synthesis. *Sci. Comput. Program.* 75(11), 1022–1047 (2010)
4. Aracic, I., Gasiunas, V., Awasthi, P., Ostermann, K.: An overview of CaesarJ. In: Rashid, A., Aksit, M. (eds.) *Transactions on Aspect-Oriented Software Development I*. LNCS, vol. 3880, pp. 135–173. Springer, Heidelberg (2006)
5. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. *IEEE Transactions on Software Engineering* 30(6), 355–371 (2004)
6. Batory, D., Höfner, P., Kim, J.: Feature interactions, products, and composition. In: GPCE, pp. 13–22. ACM, New York (2011)
7. Batory, D., O’Malley, S.: The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.* 1(4), 355–398 (1992)
8. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: controlling the scope of change in java. In: OOPSLA, pp. 177–189. ACM, New York (2005)
9. Bowen, T.F., Dworack, F.S., Chow, C.H., Griffith, N., Herman, G.E., Lin, Y.-J.: The feature interaction problem in telecommunications systems. In: SETSS, pp. 59–62 (1989)
10. Chiba, S., Igarashi, A., Zakirov, S.: Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In: OOPSLA, pp. 539–554. ACM (2010)

11. Bateni, M.: Family polymorphism. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 303–326. Springer, Heidelberg (2001)
12. Figueiredo, E., Cacho, N., Sant’Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F.C., Dantas, F.: Evolving software product lines with aspects: an empirical study on design stability. In: ICSE, pp. 261–270. ACM, New York (2008)
13. Gasiunas, V., Mezini, M., Ostermann, K.: Dependent classes. In: OOPSLA, pp. 133–152. ACM, New York (2007)
14. Herrmann, S.: Object teams: Improving modularity for crosscutting collaborations. In: Aksit, M., Awasthi, P., Unland, R. (eds.) NODE 2002. LNCS, vol. 2591, pp. 248–264. Springer, Heidelberg (2003)
15. Herrmann, S., Hundt, C., Mosconi, M.: OT/J language definition v1.3 (May 2011), <http://www.objectteams.org/def/1.3/s9.html#s9.3>
16. Hundt, C., Mehner, K., Preiffer, C., Sokenou, D.: Improving alignment of cross-cutting features with code in product line engineering. *Journal of Object Technology* 6(9), 417–436 (2007)
17. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SE-90-TR-021, Carnegie Mellon University (1990)
18. Kästner, C., Apel, S., Batory, D.: A case study implementing features using AspectJ. In: SPLC, pp. 223–232. IEEE Computer Society, Los Alamitos (2007)
19. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: ICSE, pp. 311–320. ACM, New York (2008)
20. Kästner, C., Apel, S., Ostermann, K.: The road to feature modularity? In: SPLC, vol. 2, pp. 1–5. ACM, New York (2011)
21. Kästner, C., Apel, S., Rahman, S.S.u., Rosenmüller, M., Batory, D., Saake, G.: On the impact of the optional feature problem: analysis and case studies. In: SPLC, pp. 181–190. Carnegie Mellon University (2009)
22. Liu, J., Batory, D., Lengauer, C.: Feature oriented refactoring of legacy applications. In: ICSE, pp. 112–121. ACM, New York (2006)
23. Madsen, O.L., Moller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: OOPSLA, pp. 397–406. ACM, New York (1989)
24. Mezini, M., Ostermann, K.: Conquering aspects with caesar. In: AOSD, pp. 90–99. ACM, New York (2003)
25. Mezini, M., Ostermann, K.: Variability management with feature-oriented programming and aspects. In: FSE, pp. 127–136. ACM, New York (2004)
26. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 419–443. Springer, Heidelberg (1997)
27. Saito, C., Igarashi, A., Viroli, M.: Lightweight family polymorphism. *Journal of Functional Programming* 18, 285–331 (2008)
28. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)
29. Schulze, S., Apel, S., Kästner, C.: Code clones in feature-oriented software product lines. In: GPCE, pp. 103–112. ACM, New York (2010)
30. Young, T., Murphy, G.: Using AspectJ to build a product line for mobile devices. AOSD Demo (2005)