Implementing Feature Interactions with Generic Feature Modules

Fuminobu Takeyama
Tokyo Institute of Technology

Shigeru Chiba
The University of Tokyo / JST, CREST
A family of similar software products

MobileMedia SPL [T. Young, et al., AOSD 2005 demo]
- A multimedia management application SPL
- e.g. Only music for character-display devices

Developed by selecting features

or-features: select at least one
How can we implement features?

- Conditional compilation
  - removes unnecessary feature
- Multiple features shares the same file
  - limits development for each feature
  - No encapsulation, etc...

```java
class Application {
  #ifdef PHOTO
    PhotoListScreen photoListScreen;
    PhotoController photoController;
  #endif
  #ifdef MUSIC
    MusicListScreen musicListScreen;
    MusicController musicController;
  #endif

  public void startApp {
    #ifdef PHOTO
      photoListScreen = new PhotoListScreen();
      photoController = new PhotoController();
    #endif
    #ifdef MUSIC
      musicListScreen = new MusicListScreen();
      musicController = new MusicController();
    #endif
  }

  public static void main(String[] args) {
    Application app = new Application();
    app.startApp();
  }
}
```
Features implementation in AOP

- Many languages allows separating features
  - AspectJ, GluonJ, AHEAD’s Jak, (Feature House), ...

- The PhotoInit reviser
  - add fields to Application destructively
  - overrides startApp method executes it instead of the original

```java
class PhotoInit revises Application {
    PhotoListScreen screen;
    PhotoController cont;

    public void startApp() {
        screen = new PhotoListScreen();
        cont = new PhotoController();
        super.startApp();
    }
}
```

```java
class Application {
    ... screen;
    ... cont;

    public void startApp() {
        screen = ...;
        cont = ...;
        ... other initializations ...
    }
}
```
Feature interaction

- Additional behavior for combination of features
  - needed when they are used together in a product
  - cannot be obtained by naively combining the implementations of features

- Show “SMS” on the photo viewer
  - if Photo and SMS features are used
Derivatives

- Implement feature interactions separately
  - A special feature used only when interacting features used

- Classes and aspects for interacting features should not contain code for the interaction
  - Necessary only when all of them are used

[J. Liu et al., ICSE 2006]
The optional feature problem

Exponential explosion?

- $2^n - n - 1$
  - if all combination of features interacts
- 53 interactions in 38 features [C. Kästner, et al. SPLC 2009]
  - An AspectJ implementation of Berkley DB has

6 derivatives between

- {Photo, Music, Video}, {Copy, SMS}

Derivatives must be prepared

- to build products just by selecting features without writing code

<table>
<thead>
<tr>
<th></th>
<th>Copy</th>
<th>SMS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Photo</td>
<td>PhotoCopy</td>
<td>PhotoSMS</td>
<td>Photo...</td>
</tr>
<tr>
<td>Music</td>
<td>MusicCopy</td>
<td>MusicSMS</td>
<td>Music...</td>
</tr>
<tr>
<td>Video</td>
<td>VideoCopy</td>
<td>VideoSMS</td>
<td>Video...</td>
</tr>
<tr>
<td>...</td>
<td>...Copy</td>
<td>...SMS...</td>
<td>...</td>
</tr>
</tbody>
</table>
Derivatives are often redundant

- Found in MobileMedia
  - Only minor difference in class names.
  - So they cannot be merged into one derivative
  - Since derivatives are implemented like normal features

```java
class AddSMSToPhoto extends PhotoViewScreen {
    void initForm() {
        t.addCommand(new SMSCommand());
        super.startApp();
    }
}
```

```java
class AddCopyToMusic extends MusicPlayerScreen {
    void initForm() {
        t.addCommand(new CopyCommand());
        super.startApp();
    }
}
```
Solution: Generic derivatives

- Write a template and generate derivatives for every combination
  - Template parameters are *features*
  - Automate enumeration of all the combination

- F::C—feature F’s class C

```
Photo

PhotoViewScreen

mt = Photo → mt::X = Photo::PhotoViewScreen

Music

MusicPlayerScreen

mt = Music → mt::X = Music::MusicPlayerScreen
```

```
Derivative(mt, mo)

modifies mt::X
new mo::Y
```
FeatureGluonJ: a new FOP language

FOP: Feature-oriented programming language

- Interface among feature modules
  - Specifies the classes that a feature module must contain
  - For "type-safe" templates when using a feature as a template parameter

- Feature module
  - Contains classes and revisers (aspect)
  - Can inherit from an abstract feature module, which works as an interface.
    - Also found in CaesarJ and ObjectTeams

{Photo, Music, Video} is a MediaType
Inheritance for feature modules

A sub feature module can

- Use the classes and revisers in the super module
- Rename a class in the super module
- Add new classes, methods, and fields.
- Modify a class in the super module by revisers (or aspects)

- Like virtual classes with lightweight semantics

```
new MediaViewScreen ➔ PhotoViewScreen
new MediaViewScreen ➔ MusicPlayerScreen
```

```
MediaType
MediaController MediaViewScreen
new MediaViewScreen

Music
MediaController MediaViewScreen MusicPlayerScreen
```

```
Photo
MediaController MediaViewScreen
new MediaViewScreen ➔ PhotoViewScreen
```
Generic derivatives in FeatureGluonJ

- Implement by a template feature module
  - Parameters are feature modules.

- for every
  - Automatically enumerate every combination of feature modules and instantiate the template

```java
feature MediaTypeFileOp defines for every (mt, mo) {
  abstract import feature mt: MediaType;
  abstract import feature mo: MediaOp;
}

class AddCmdToView revises mt::MediaViewScreen {
  void initMenu() {
    addCommand(new mo::MediaOpCommand());
    super.initMenu();
  }
}
```

- always provided by mt::MediaViewScreen

Photo::PhotoViewScreen
Music::MusicPlayerScreen
Incremental implementation of derivatives

- Can manually instantiate a template (if needed)
  - for customizing the generated feature modules.
Related work

Languages with virtual aspects + virtual classes
- CaesarJ, Object Teams
- designed for reusable collaboration of classes and aspects
  - across product lines
  - require to combine classes (glue code) for each product
- Object Teams supports dependent team (collaboration)
  but it can depends on only a team

Annotation based approach
- #ifdef–#endif, CIDE
- How reduce redundancy of code for interaction?
  - especially between or-features
Conclusion

FeatureGluonJ

- A generic feature module as a template for derivatives
- Automatic template instantiation
- The template instantiation is “type-safe”
  - by introducing a feature interface

Future work

- Case study and evaluation
  - few appropriate SPLs (e.g. non OOP)
- Algebra model, formal definition of semantics
- Feature local variables, method, …