Mapping Features to Aspects

The Road from Crosscutting to Product Lines

(Work in Progress)

Roberto E. Lopez-Herrejon
Computing Laboratory
Oxford University

Motivation – Features

- Feature
  - Informally: A characteristic or prominent part of a product
  - In SEng: An increment in program functionality

- Features are the basis of software product lines
  - Family of similar products
  - Members are distinguished by the set of features they have
  - Feature reuse, reduce time to market, customization

Motivation – Aspects

- Aspects are a sophisticated technology to modularize crosscutting concerns
  - Involve several classes and interfaces

- AspectJ is the most popular AOP language
  - Typical applications: tracing, debugging, …

Big Picture

- How can aspects help implementing product lines?

- Previous work
  - Graph Product Line – Lopez-Herrejon 2002
  - Middleware software – Coyler et al. 2004
  - Evaluation of AOP to PL – Muthig et al. 2004

- Limitations
  - Small scale in loc and features
  - Do not address composition issues

Our Approach

- Functional composition (TOSEM04)
  - Key factor for product line generation
  - Promotes feature reuse

- AspectJ composition model (PEPM06)
  - Not simple functional composition
  - Advice applies globally

- Emulate functional composition in AspectJ
  - Careful use of advice and pointcuts
  - Case study – AHEAD tool suite

Feature Oriented Programming (FOP)

- Has been used in the synthesis of large scale product line programs

- AHEAD tool suite
  - Implementation of FOP
  - Uses a Java language extension called Jak
  - Is in itself a product line
  - Supports definition of extensible DSLs

AHEAD – Composition

- Base programs are constants

```
i          // program with feature i
j          // program with feature j
```

- Program extensions / refinements are functions

```
k • x      // adds feature k to program x
m • x      // adds feature m to program x
```

- Program designs are expressions

```
k • i      // program with features i and k
m •k •j    // program with features m,k,j
```

- Product line is the set of valid expressions

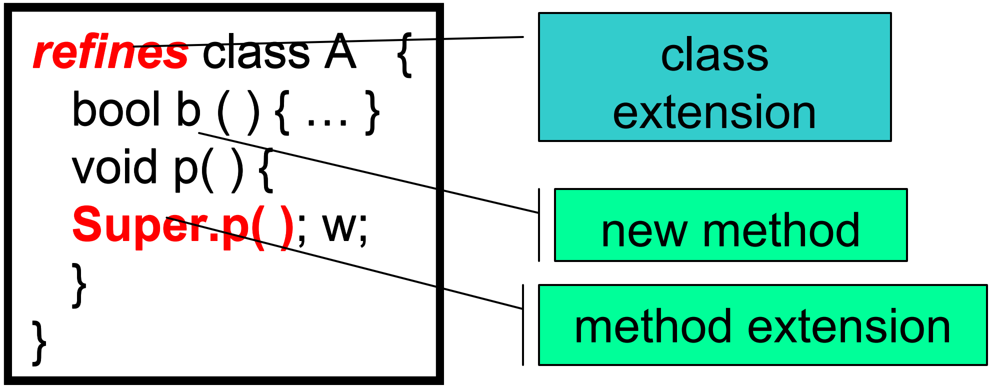
AHEAD – Composition

C

```
class A   {
  double m;
   void p(  ) { s; t; }
}
```

```
class B {
  int x;
  String g( ) {…}
}
```

R



```
refines class A   {
   bool b ( ) { … }
   void p( ) {
   Super.p( ); w;
   }
}
```

class extension

new method

method extension

```
class D {
  int v;
  double k( ) { … }
}
```

R • C

```
class A   {
 double m;
 void p( ) { s; t; w; }
 bool b() { … }
}
```

Similarities and Differences

- FOP and AspectJ ECOOP05
  - New fields
  - New methods and constructors
  - Method and constructor extensions
  - Aspects cannot add new classes
- Composition model PEPM06
  - FOP is functional
  - AspectJ is something else …

Product Lines Example

```
class Point {
   int x;
   void setX(int v) { x = v; }
}
```

```
aspect AddY {
  int Point.y;
  void Point.setY(int v) { y = v; }
}
```

```
aspect AddPrint {
 after (Point p) : execution( * Point.set*(..))  { print (“Hi”);  }
}
```

```
aspect AddColor {
   int Point.color = 0;
   int Point.setColor(int c) { color = c; }
}
```

How many products can be created?

Assuming Functional Composition

```
class Point {
   int x; void setX(int v) { x = v; print(“Hi”); }
   int y; void setY(int v) { y = v; print(“Hi”); }
   int color = 0; void setColor(int c) { color = c; print(“Hi”); }
}
```

AddPrint • AddColor • AddY • Point

We can generate 3 different products

In Reality …

```
class Point {
   int x; void setX(int v) { x = v; print(“Hi”); }
   int y; void setY(int v) { y = v; print(“Hi”); }
   int color = 0; void setColor(int c) { color = c; print(“Hi”); }
}
```

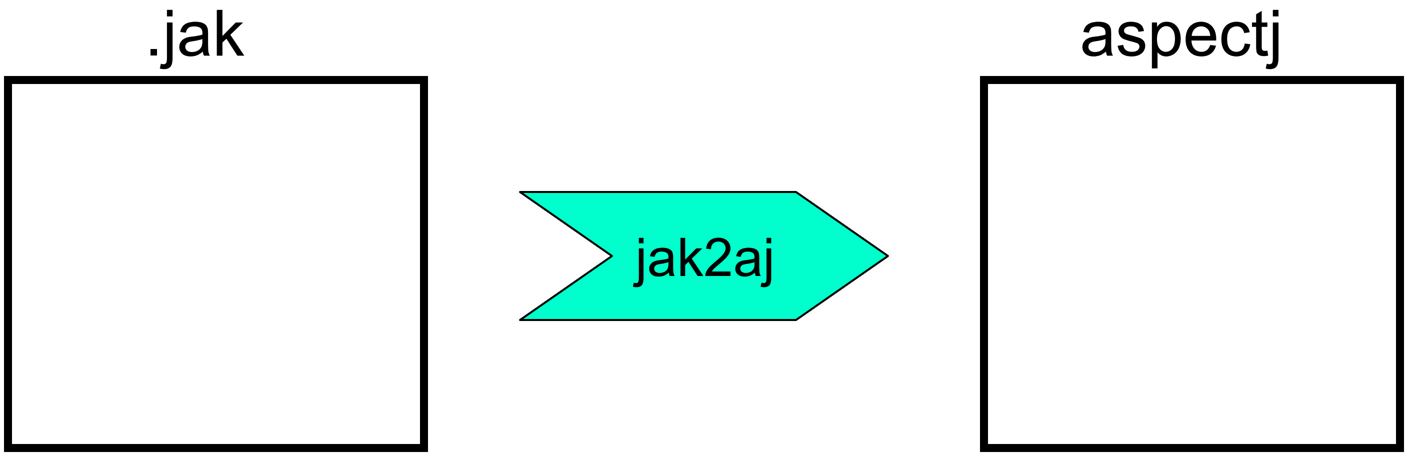
AddPrint ◊ AddColor ◊ AddY ◊ Point

AspectJ needs 3 versions of AddPrint

Emulating Functional Composition For AHEAD Features

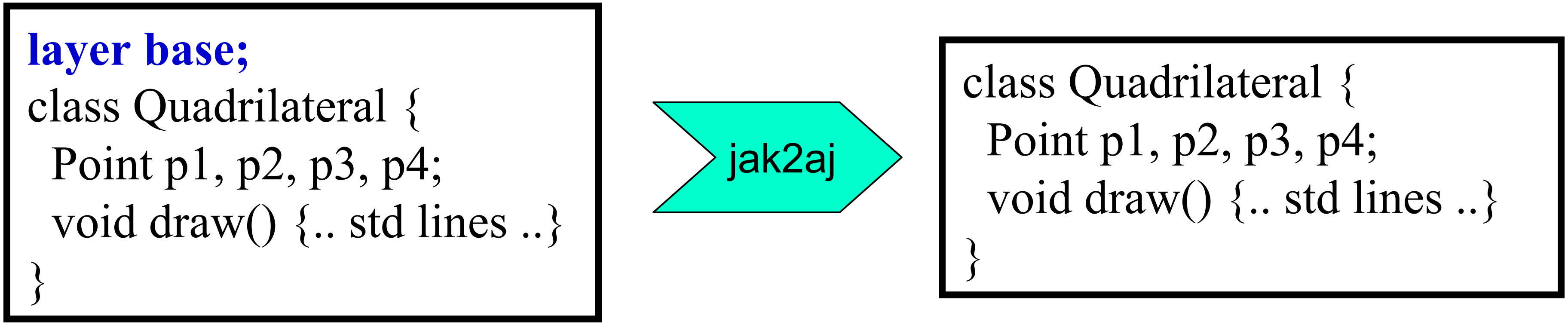
Translating Jak to AspectJ

- AHEAD features
  - Add new fields and methods
  - Extend methods
  - Impose composition order



Translation of Base Code

- Standard classes are mapped without any changes



```
layer base;
class Quadrilateral {
  Point p1, p2, p3, p4;
  void draw() {.. std lines ..}
}
```

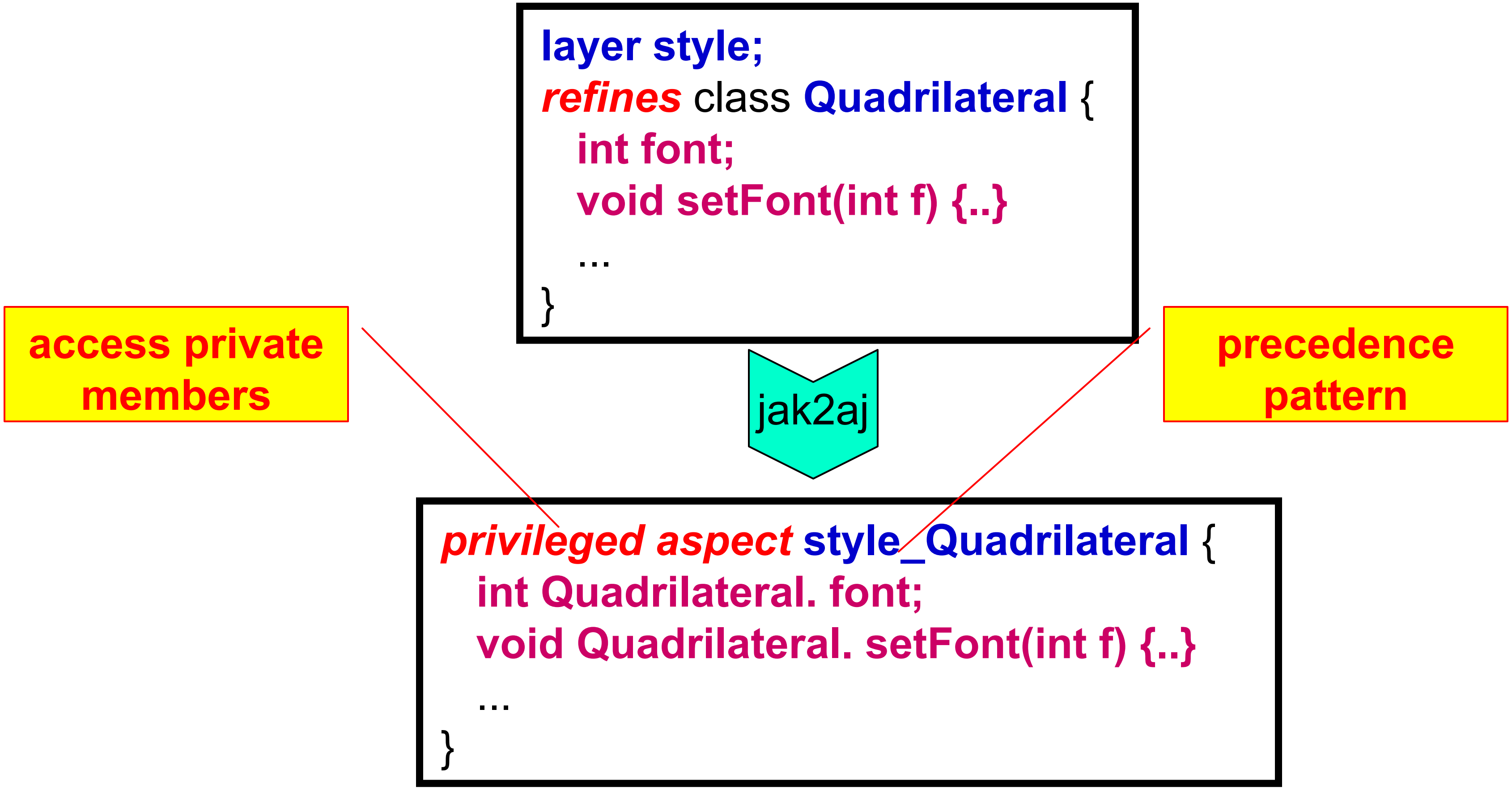
jak2aj

```
class Quadrilateral {
  Point p1, p2, p3, p4;
  void draw() {.. std lines ..}
}
```

- Standard interfaces are translated similarly

New Fields and Methods

- Translated to field and method introductions



```
layer style;
refines class Quadrilateral {
   int font;
   void setFont(int f) {..}
   ...
}
```

jak2aj

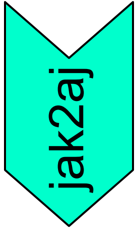
```
privileged aspect style_Quadrilateral {
   int Quadrilateral. font;
   void Quadrilateral. setFont(int f) {..}
   ...
}
```

access private members

precedence pattern

Method Extensions – General Case

```
layer color;
refines class Rectangle   {
 void draw() {
  repaint();
   Super.draw( );
 }
}
```



```
privileged aspect color_Rectangle{
  void around(Rectangle obj$Rectangle) :
   call(void *.draw()) && target(Rectangle) &&
   target(obj$Rectangle)  {
       obj$Rectangle.repaint();
      proceed(obj$Rectangle);
 }
```

- A method
  - extends the method of previous features
  - references extended method
- Translation
  - **around** advice
  - **proceed** calls replace **Super** calls
  - class members referenced through **target** object

Why all these many cases?

- Four different cases for method extension
- In AspectJ
  - Asymmetrical approach to overriding
    - Precedence determines overriding relations in new aspects, but does not allow overriding of base code
  - No notion of method extension
    - mimicked with around advice
- In AHEAD
  - Overloaded meaning of Super
    - Standard inheritance overriding and use of *super*
    - Method extensions

In Retrospective …

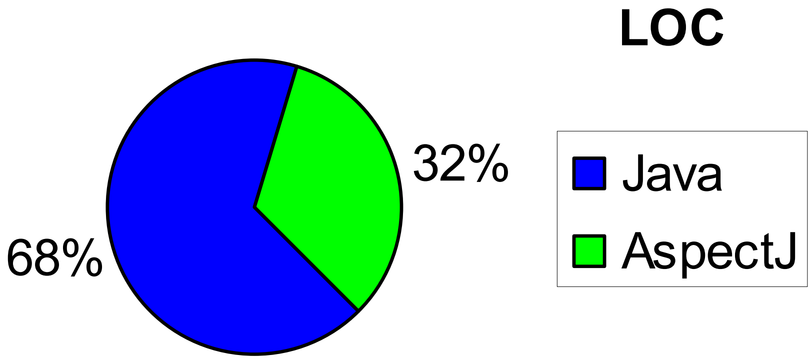
- How functional composition was achieved?
  - Disciplined use of subset of AspectJ

AHEAD	AspectJ
Add fields and methods	→ Introductions
Extend methods	→ Around advice → Join points of a single type → Method calls (target, args)
Impose composition order	→ Precedence clauses

AHEAD Product Line Statistics

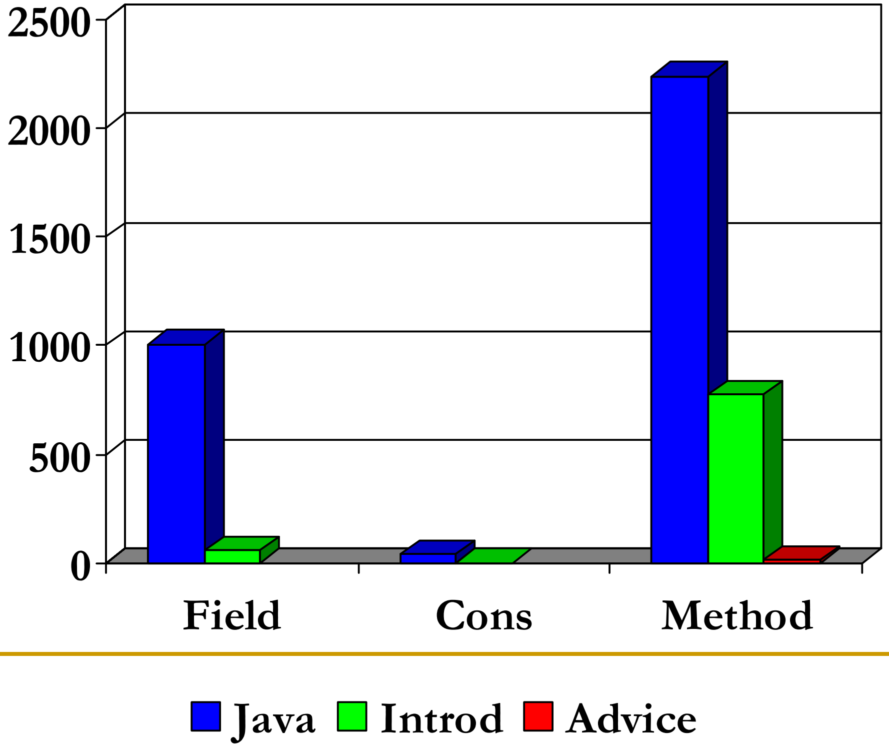
Tools: 5 Num Features: 48 LOC: 205K+

	Java	AspectJ
NumFiles	524	503
LOC	38300	18427



AHEAD Product Line Statistics

	Java	Introd	Advice
Fields	1006	58	0
Constructors	40	0	0
Methods	2238	774	16



Conclusions

- Aspects can be used to implement product lines
  - Significant size 200K+ LOC
- Conditions
  - Emulate functional composition
  - Using modest subset of AspectJ
  - Careful use of precedence and advice

Current Work

- Complete AHEAD tool translation and statistics

- AHEAD is based on an algebraic composition model
  - Program transformations are the central mathematical concept
  - We have developed a basis of an algebraic structural model that unifies aspects and features *PEPM 06*

- Open questions …
  - Can other AOP capability be added to this model?
  - Can functional composition be implemented on full AspectJ?

References

- Roberto E. Lopez-Herrejon and Don Batory. *Mapping Features to Aspects: An Experience Report*. In preparation.

- Roberto E. Lopez-Herrejon, Don Batory, and Christian Lengauer. *A disciplined approach to aspect composition*. PEPM, 2006.

- Roberto E. Lopez-Herrejon, Don Batory, and William Cook. *Evaluating support for features in advanced modularization technologies.* ECOOP 2005.

- D. Batory, J.N. Sarvela, and A. Rauschmayer. *Scaling Step-Wise Refinement*. IEEE Transactions on Software Engineering (IEEE TSE), June 2004.