

## Programação Orientada a Características com AHEAD

Eduardo Figueiredo

<http://www.dcc.ufmg.br/~figueiredo>

## Agenda da Aula

- Programação Orientada a Características (FOP)
  - Segundo Don Batory
- FOP usando AHEAD
- AHEAD com o FeatureIDE

## Programação Orientada a Características (FOP)

## Característica Modular

- Um sistema é descrito através de suas *características*
  - A diferença entre uma versão básica e completa de um software são as características disponíveis
- A implementação de uma característica tende a ser não modular
  - Entretanto, é desejável que ela seja modular para facilitar a plugabilidade

## Definições

- Característica é um incremento funcional no desenvolvimento do software
- Programação orientada a características
  - Paradigma para desenvolvimento de linha de produtos de software
  - Considera características como entidades primárias de projeto e implementação
  - Busca a modularidade de características

## Refinamento Sucessivo

- Um dos alicerces de FOP é o *refinamento sucessivo*
- O conceito de refinamento sucessivo é simples e antigo
  - Programas complexos devem ser desenvolvidos a partir de partes simples
- Partes Simples = Características
- Programas Complexos = LPS

## Exemplos de Decomposições

- Considere as seguintes implementações da classe C

```
class C {  
  int field1;  
  void method2() { ... }  
  void method3() { ... }  
  void method4() { ... }  
}
```

equivalentes

```
class C1 {  
  int field1;  
  void method2() { ... }  
}  
class C2 extends C1 {  
  void method3() { ... }  
}  
class C extends C2 {  
  void method4() { ... }  
}
```

## Histórico de FOP

- Surgiu da proposta de modelagem em camadas para sistemas BD extensíveis
  - Início da década de 90
- Estrutura geral em camadas
  - Um programa é uma pilha de camadas
  - Cada camada adiciona novas funcionalidades às camadas anteriores
  - Diferentes composições das camadas resultam em programas diferentes

## De Camada para Característica

- Com FOP, a ideia de camadas foi generalizada para características
- Por que considerar as características?
  - Um programa tem várias representações (código, modelos, documentação, etc.)
  - Uma característica aparece em todas as representações
  - As representações devem ser consistentes

## FOP vs. POA

- Há várias semelhanças entre Programação Orientada a Aspectos (POA) e FOP
  - Ambos procuram modularizar interesses transversais
  - Ambos buscam facilidade de extensão de programas
  - Ambos podem ser usados no desenvolvimento de linha de produtos

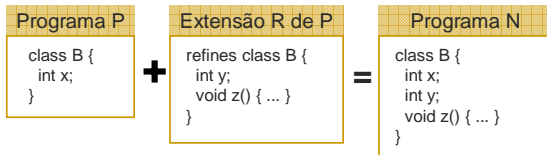
## FOP vs. POA

- O modelo de composição é diferente
- Modelo de Composição FOP
  - Decomposição funcional
  - Refinamento sucessivo
- Modelo de Composição POA
  - Mais "intrusivo"
  - Não requer refinamento sucessivo

Notação GenVoca

## Entendendo o Refines

- Considere o seguinte programa P
  - P possui apenas uma classe B



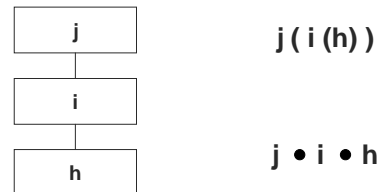
## Valores e Funções

- Podemos expressar refinamentos de programas algebricamente
  - O programa P é um valor (base)
  - O refinamento R é uma função
  - O programa resultante N é uma expressão
- $N = R(P)$  equivalente a  $N = R \bullet P$

## Notação de FOP

- Álgebra elementar pode ser usada para expressar o conceito de composição
  - Uma função adiciona novo código ao programa
  - Uma expressão (formada por funções) representa o programa
- Exemplo: fun1 (fun2 (fun3))

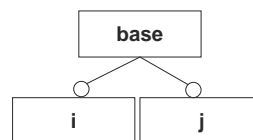
## Exemplos da Notação



## Definições

- GenVoca é uma notação para definir produtos de uma linha de produtos
  - Cada expressão define um produto
  - Exemplo:  $p1 = j \bullet i \bullet h$
- Características são representadas como funções na expressão
- Nem todas as combinações são possíveis
  - O modelo de características define as restrições

## Exemplo



- Um produto da linha é chamado expressão
- O símbolo  $\bullet$  denota composição

### Expressões

- i significa o programa base com a característica i
- j significa o programa base com a característica j
- $i \bullet j$  significa adicionar i ao programa j (acima)

## Refinamento Sucessivo

- GenVoca é voltado para o refinamento sucessivo no desenvolvimento
  - O processo enfatiza a simplicidade e facilidade de compreensão
- Considere o produto  $p1 = i \cdot j \cdot h$ 
  1. O desenvolvimento começa com o refinamento h
  2. Então a característica j é adicionada
  3. Finalmente a característica i é adicionada

## Implementações de GenVoca

- GenVoca é uma notação com várias implementações possíveis
  - Originalmente, características em GenVoca eram implementadas usando diretivas de pre-processamento (`#ifdef`)
- A técnica que surgiu recentemente é chamada *mixín layers*
  - Solução adotada em AHEAD

## AHEAD by Exemplos

## Conceitos de AHEAD

- *Algebraic Hierarchical Equations for Application Design*
- AHEAD fornece fundamentação matemática para as expressões GenVoca e seus relacionamentos

## AHEAD Tool Suite

- A linguagem AHEAD implementa os conceitos da formalização
  - Arquivos em AHEAD tem extensão .JAK
  - Sintaxe semelhante a Java
- Exemplo da Calculadora
  - Linha de produtos para inteiros (BigInteger) e decimais (BigDecimal)
  - Operações matemáticas de adição, divisão e subtração

## Código da Aplicação Base

```
class calc { }  
(a) Base/calc.jak
```

- Características da LPS
  - Base, BigI, BigD, ladd, ldiv, lsub, Dadd, Ddivd, Ddivu, Dsub

## Aplicação Base + Inteiro

```
class calc { }
```

(a) Base/calc.jak

```
import java.math.BigInteger;

refines class calc {
    static BigInteger zero = BigInteger.ZERO;
    BigInteger e0 = zero, e1 = zero, e2 = zero;

    void enter( String val ) {
        e2 = e1;
        e1 = e0;
        e0 = new BigInteger(val);
    }

    void clear() {
        e0 = e1 = e2 = zero;
    }

    String top() { return e0.toString(); }
}
```

(b) BigI/calc.jak

## Aplicação Base + Decimal

```
class calc { }
```

(a) Base/calc.jak

```
import java.math.BigDecimal;

refines class calc {
    static BigDecimal
        zero = new BigDecimal("0");
    BigDecimal e0 = zero, e1 = zero,
        e2 = zero;

    void enter( String val ) {
        e2 = e1;
        e1 = e0;
        e0 = new BigDecimal(val);
    }

    void clear() {
        e0 = e1 = e2 = zero;
    }

    String top() { return e0.toString(); }
}
```

(c) BigD/calc.jak

BigI e BigD são mutuamente exclusivos

## Base + Operações

```
class calc { }
```

(a) Base/calc.jak

```
Inteiro
refines class calc {
    void divide() {
        e0 = e0.divide( e1 );
        e1 = e2;
    }
}
```

(d) Idiv/calc.jak

```
Inteiro Decimal
refines class calc {
    void add() {
        e0 = e0.add(e1);
        e1 = e2;
    }
}
```

(e) Iadd/calc.jak and Dadd/calc.jak

```
Decimal
import java.math.BigDecimal;

refines class calc {
    void divide() {
        e0 = e0.divide( e1,
            BigDecimal.ROUND_DOWN );
        e1 = e2;
    }
}
```

(f) Ddivd/calc.jak

## Base + Operações

```
class calc { }
```

(a) Base/calc.jak

```
Inteiro
refines class calc {
    void divide() {
        e0 = e0.divide( e1 );
        e1 = e2;
    }
}
```

(d) Idiv/calc.jak

Idiv e Ddivd são mutuamente exclusivos

```
Decimal
import java.math.BigDecimal;

refines class calc {
    void divide() {
        e0 = e0.divide( e1,
            BigDecimal.ROUND_DOWN );
        e1 = e2;
    }
}
```

(f) Ddivd/calc.jak

## Instância da LPS

- Uma calculadora (produto da LPS) é definida por uma equação GenVoca
- Alguns produtos possíveis
  - i1 = Iadd • BigI • Base
  - i2 = Isub • Iadd • BigI • Base
  - i3 = Idiv • Iadd • BigI • Base
  - d1 = Dadd • BigD • Base
  - d2 = Ddivd • Dadd • BigD • Base

## Código do Produto i3

```
import java.math.BigInteger; 1
```

```
class calc {
    static BigInteger zero = BigInteger.ZERO;
    BigInteger e0 = zero, e1 = zero, e2 = zero;
```

```
void add() {
    e0 = e0.add(e1);
    e1 = e2;
}

void clear() {
    e0 = e1 = e2 = zero;
}
```

```
void divide() {
    e0 = e0.divide( e1 );
    e1 = e2;
}

void enter( String val ) {
    e2 = e1;
    e1 = e0;
    e0 = new BigInteger(val);
}
```

```
String top() { return e0.toString(); }
}
```

Figure 2. i3/calc.jak

## Compor, Traduzir e Compilar

- Cria um produto (compor)
  - `composer --target=i3 Base BigI Idiv Iadd`
  - A composição cria um arquivo .JAK (`calc.jak`)
- Comando para traduzir para Java
  - `jak2java *.jak` ou
  - `jak2java calc.jak`
- Compilar Java
  - `javac *.java`

## Mixin and Unmixin

## Jampack x Mixin

- AHEAD têm duas formas de compor módulos
  - jampack: resulta em uma única classe
  - mixin: cada refinamento em uma classe
- Usando mixin, cada classe é precedida por uma anotação SoURce
  - Esta anotação indica a característica da qual a classe é derivada

## Exemplo de Produto (Mixin)

```
SoURce Iadd "../Iadd/calc.jak";
abstract class calc$$Iadd extends calc$$BigI {
    // adds BigIntegers
    void add() {
        // adds BigIntegers
        e0 = e0.add( e1 );
        e1 = e2;
    }
}
```

Característica  
**Iadd**

```
SoURce Idiv "../Idiv/calc.jak";
class calc extends calc$$Iadd {
    void divide() {
        e0 = e0.divide( e1 );
        e1 = e2;
    }
}
```

Característica  
**Idiv**

## Exemplo: Jampack x Mixin

```
import java.math.BigInteger;

class calc {
    static BigInteger zero = BigInteger.ZERO;
    BigInteger e0 = zero, e1 = zero, e2 = zero;

    void add() {
        e0 = e0.add(e1);
        e1 = e2;
    }

    void clear() {
        e0 = e1 = e2 = zero;
    }
}

import java.math.BigInteger;

SoURce Root base "../base/calc.jak";
abstract class calc$$base {}

SoURce BigI "../BigI/calc.jak";
abstract class calc$$BigI extends calc$$base {
    static BigInteger zero = BigInteger.ZERO;
    BigInteger e0 = zero, e1 = zero, e2 = zero;

    void add() {
        e0 = e0.add( e1 );
        e1 = e2;
    }
}
```

Mixin

Jampack

## Código Composto por Mixin


- Todas as classes da cadeia de "refinamentos" são abstratas
  - Exceto a última classe
- Um sufixo é adicionada para cada classes com o nome da característica
  - O objetivo é manter nomes únicos

## Por que usar mixin?

- Usar mixin ao invés de jampack permite rastreamento reverso para o código fonte
- Exemplo
  - Após usar o jak2java, o código gerado pode possuir um erro
  - Se todo o código estiver em uma única classe, como saber em qual característica está o erro
  - O rastreamento é possível usando mixin

## Unmixin

- Mecanismo de AHEAD para propagar alterações do código java para o jak
- Caso o código java seja alterado, ao usar o comando unmixin, as alterações são propagadas para o jak original

Slides in English 

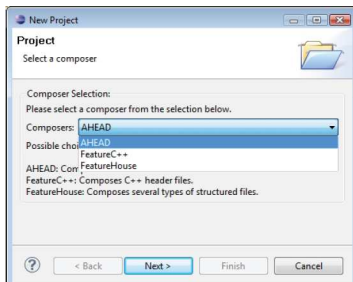
## AHEAD with FeatureIDE

## FeatureIDE

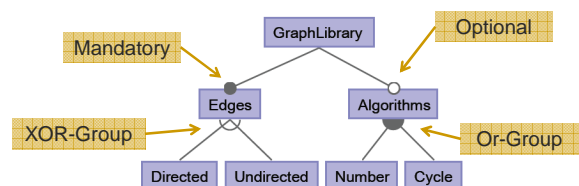
- Eclipse extension for FOP
- It supports several composition languages
  - **AHEAD**
  - AspectJ
  - FeatureC++
  - FeatureHouse, etc.

## Project Wizard

- Requires selecting a composition language

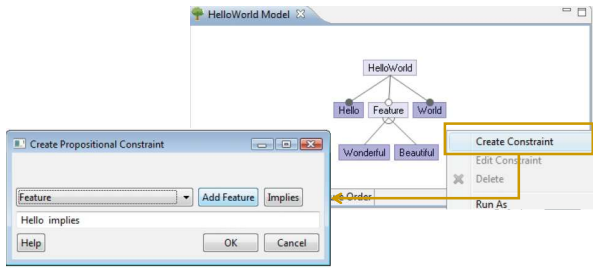


## Feature Model Editor



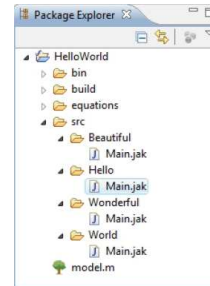
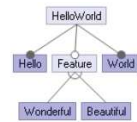
## Constraints

- To create cross-tree constraints



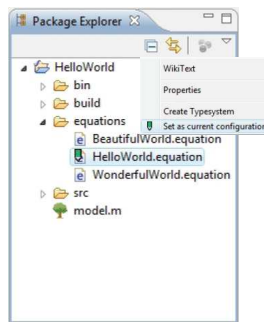
## Model to Code

- Source code contains a folder for each concrete feature
  - Including files to compose



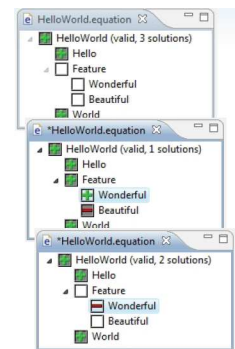
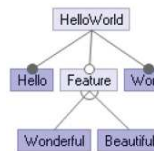
## Default Product

- The equations folder has configurations of products from the feature model
  - Each equation represents a product
  - Current product is marked

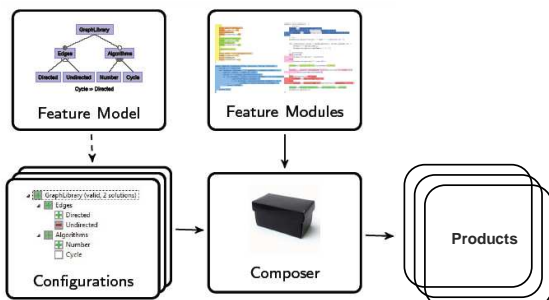


## Product Configuration

- Manual selection
- Automatic selection
- Consistency check



## The FeatureIDE Process



## Bibliografia da Aula

- Don Batory. **A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite**. GTTSE, 2006.