DCC / ICEx / UFMG

Feature Oriented Programming with AHEAD

Eduardo Figueiredo

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

Agenda

- Feature Oriented Programming (FOP)
  - By Don Batory
- AHEAD by Examples
- FeatureIDE

Feature Oriented Programming (FOP)

Definition of Feature

- A feature is an increment in program development or functionality
- Features can be classified
  - Mandatory features
  - Optional features
  - Or features
  - Alternative features (XOR)

Program Configuration

- Programs are described or differentiated by features
  - Entry-level versions have a minimal set of features
  - Deluxe versions include most features
- The ability to add and remove features implies that features are modularized

Feature Oriented Programming

- FOP is the study of programming models that support feature modularity
  - It is a general paradigm for programming software product lines.
- FOP is based on step-wise refinement

Step-wise Refinement

- Step-wise refinement advocates that complex programs can be constructed from simple programs
  - By incrementally adding details
- In FOP, simple incremental units are called features
  - SPL is the complex program

Example of Refinement

- Let's consider two different implementations of a class C

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

equivalent

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

A Bit of History

- FOSD arose out of layer-based designs and levels of abstraction in database systems
- A program was a stack of layers
  - Each layer added functionality to previously composed layers
  - Different compositions of layers produced different programs

From Layers to Features

- Later, the idea of layers was generalized to features
- Why using features?
  - A program is composed of several artifacts, such as models and code
  - A feature unifies all artifacts in a single modular unit

FOP and AOP

- Some ideas behind FOP are similar to Aspect-Oriented Programming (AOP)
  - Both cases aim to improve separation of concerns
  - Their goal is to make programs easy to extend and maintain
  - Both FOP and POA can be used to implement a software product line

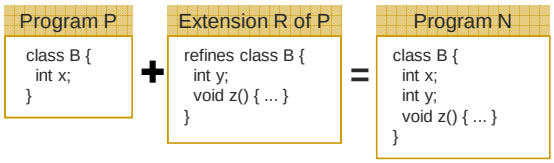
GenVoca

GenVoca

- GenVoca is a compositional paradigm for defining programs of a SPL
  - Combination of the names Genesis and Avoca
- GenVoca is based on the stepwise development of programs
  - Programs are constants
  - Refinements are functions

Example of Refinement

- Let's consider a program P
  - P has only a class B



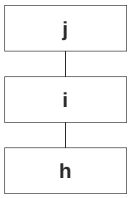
Constants and Refinements

- We can use algebras
- In the previous example
  - The program P is a constant (base)
  - The refinement R is a function
  - The program N is a the resulting expression
- N = R(P) equivalent to N = R • P

GenVoca is a FOP Notation

- 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))

Examples of Notation



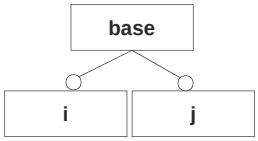
j (i (h))

j • i • h

Definições

- GenVoca é uma notação para definir produtos de uma linha de produtos
  - Cada expressão define um produto
  - Exemplo: p1 = j • i • 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 • denota composição

Expressões
- i significa o programa base com a característica i
- j significa o programa base com a característica j
- i • 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 • j • 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 *mixin layers*
  - Solução adotada em AHEAD

AHEAD by Example

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, Iadd, Idiv, Isub, 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

BigI e BigD são mutuamente exclusivos (XOR)

```
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

Base + Operações

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

Inteiro

```
refines class calc {
   void add() {
      e0 = e0.add(e1);
      e1 = e2;
   }
}
```
(e) Iadd/calc.jak and Dadd/calc.jak

Inteiro

```
refines class calc {
   void divide() {
      e0 = e0.divide( e1 );
      e1 = e2;
   }
}
```
(d) Idiv/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 (XOR)

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() {                                             2
      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;
    }
}

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

Característica **Iadd**

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;
   }
```

Jampack

Mixin

```
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 = zer

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

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

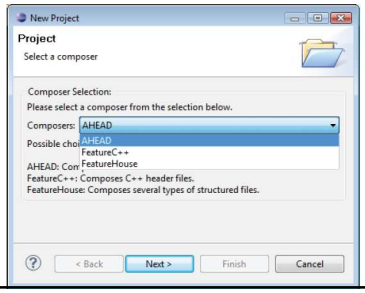
AHEAD with FeatureIDE

FeatureIDE

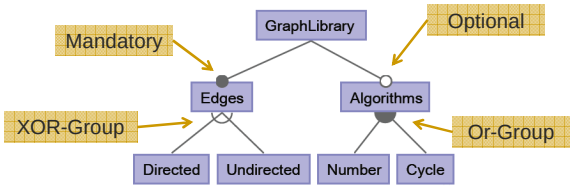
- Eclipse extension for FOP
- It supports several composition languages
  - **AHEAD**
  - AspectJ
  - FeatureC++
  - FeatureHouse, etc.

Project Wizard

- Requires selecting a composition language

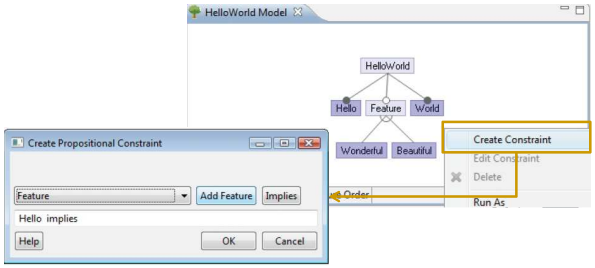


Feature Model Editor



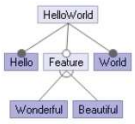
Constraints

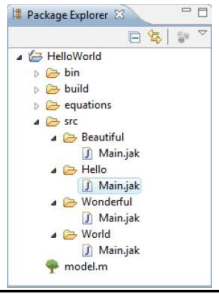
- To create cross-tree constraints



Model to Code

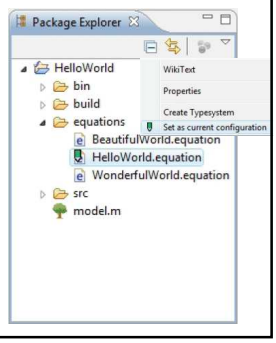
- Source code contains a folder for each concrete feature
  - Including files to compose





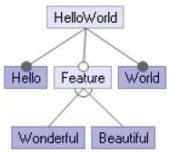
Default Product

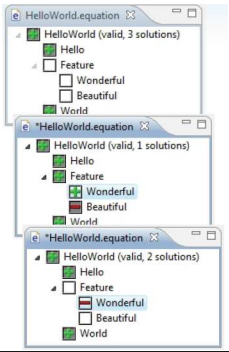
- The equations folder has configurations of products from the feature model
  - Each equation represents a product
  - Current product is marked



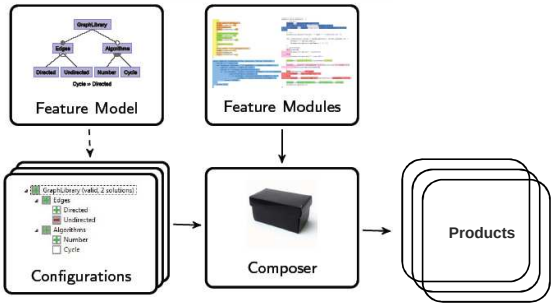
Product Configuration

- Manual selection
- Automatic selection
- Consistency check





The FeatureIDE Process



Bibliography

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