Delta Oriented Programming

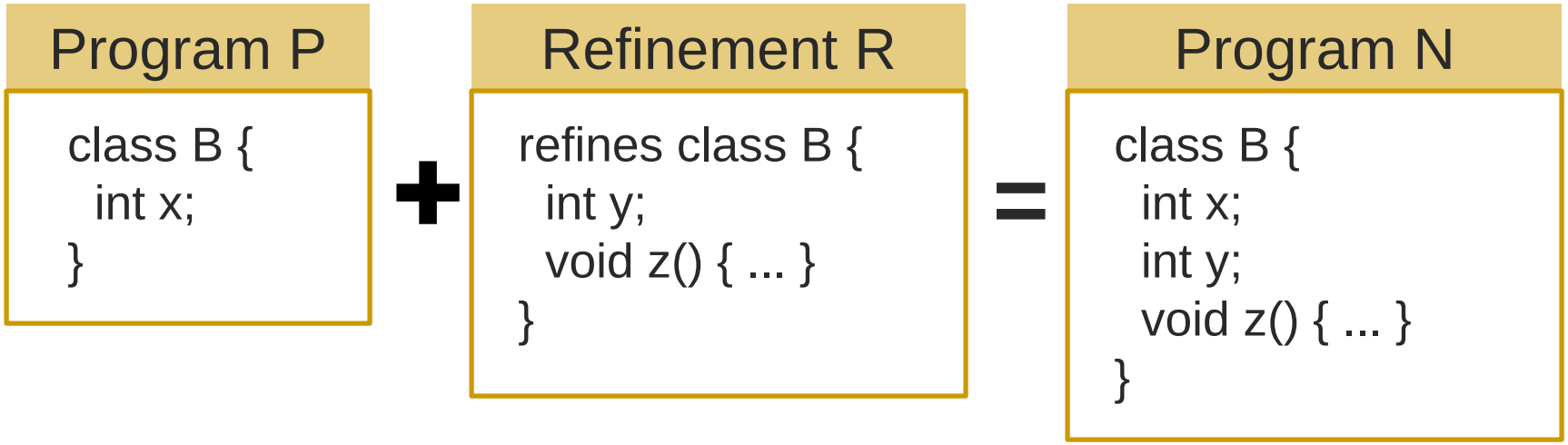
Eduardo Figueiredo

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

Feature Oriented Programming

- Feature Oriented Programming (FOP) relies on stepwise refinements
- In FOP, a software product line (SPL) is composed of constants and refinements
  - Refinements are called feature modules
- Feature modules allow
  - Adding fields and methods
  - Overriding existing methods

Example of Refinement (FOP)



- Feature modules cannot remove code from an implementation

Delta Oriented Programming

- Delta Oriented Programming (DOP) is based on the concept of program deltas
- In DOP, an SPL is composed of core modules and delta modules
  - Core module: implements a valid product of the product line
  - Delta module: specifies changes to the core module

Delta Modules

- Changes in DOP can be addition, removal, or modification
  - They can apply to classes and interfaces
- For instance, a delta module allows
  - Adding classes, methods, and fields
  - Removing classes, methods, and fields
  - Changing super-classes, constructors, etc.
  - Renaming methods and fields

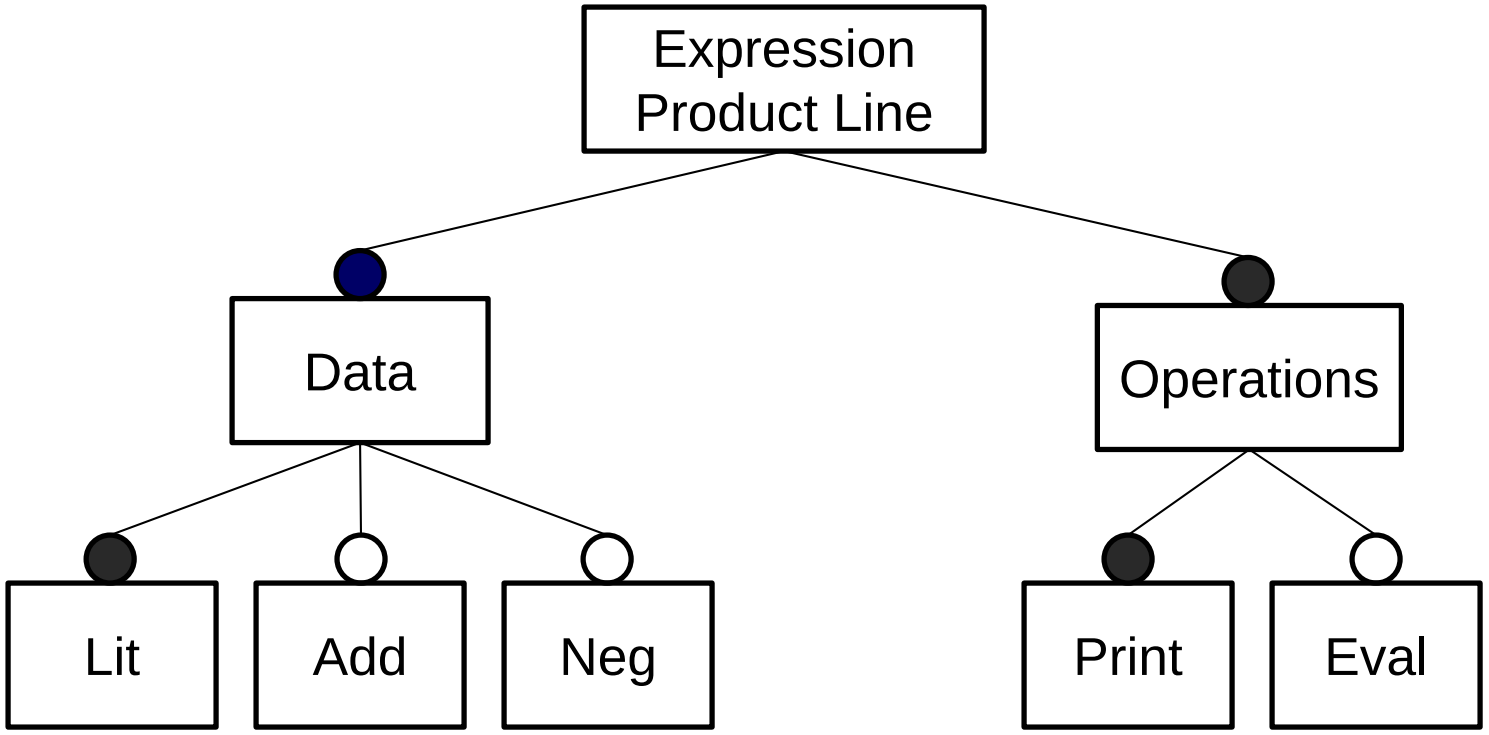
Example: Expression SPL (EPL)

- Let's define the expression product line
- Grammar for expressions

```
Exp ::= Lit | Add | Neg
Lit ::= <non-negative integers>
Add ::= Exp "+" Exp
Neg ::= "-" Exp
```

- Two operations can be performed on the expressions: printing and evaluation

EPL Feature Model



(Simple) Core Module

- The product represented by the core module must be valid
  - Include all mandatory features
  - Include at least the minimal set of required alternative features
- In the EPL example
  - Mandatory features (core): Print and Lit
  - Optional features: Add, Neg, and Eval

Core Module in DeltaJ

```
core Print, Lit {
  interface Exp { void print(); }

  class Lit implements Exp {
    int value;
    Lit(int n) { value = n; }
    void print() {
      System.out.print(value);
    }
  }
}
```

Core Module in DeltaJ

Feature names

```
core Print, Lit {
  interface Exp { void print(); }

  class Lit implements Exp {
    int value;
    Lit(int n) { value = n; }
    void print() {
      System.out.print(value);
    }
  }
}
```

Core Module in DeltaJ

Java classes and interfaces

```
core Print, Lit {
  interface Exp { void print(); }

  class Lit implements Exp {
    int value;
    Lit(int n) { value = n; }
    void print() {
      System.out.print(value);
    }
  }
}
```

Structure of Delta Modules

```
delta <name> [after <delta names>]
     when <application condition> {
   removes <class or interface name>
   adds class <name>
     <standard Java class>
   adds interface <name>
     <standard Java interface>
   modifies interface <name> {
     <remove, add, rename method header> }
   modifies class <name> {
     <remove, add, rename field>
     <remove, add, rename method> }
}
```

Delta Module DEval in DeltaJ

```
delta DEval when Eval {
  modifies interface Exp {
    adds int eval();
  }

  modifies class Lit {
    adds int eval() {
      return value;
    }
  }
}
```

Delta Module DEval in DeltaJ

```
delta DEval when Eval {
  modifies interface Exp {
    adds int eval();
  }

  modifies class Lit {
    adds int eval() {
      return value;
    }
  }
}
```

It modifies the **Exp** interface by adding the **eval()** method signature

Delta Module DEval in DeltaJ

```
delta DEval when Eval {
  modifies interface Exp {
    adds int eval();
  }

  modifies class Lit {
    adds int eval() {
      return value;
    }
  }
}
```

It modifies the **Lit** class by adding the **eval()** method

Delta Module DAdd in DeltaJ

```
delta DAdd when Add {
  adds class Add implements Exp {
    Exp expr1; Exp expr2;
    Add(Exp a, Exp b) {
      expr1=a; expr2=b;
    }
  }
}
```

- A condition is attached to every delta module

Delta Module DAdd in DeltaJ

```
delta DAdd when Add {
  adds class Add implements Exp {
    Exp expr1; Exp expr2;
    Add(Exp a, Exp b) {
      expr1=a; expr2=b;
    }
  }
}
```

The **when** clause specifies that the delta module is applied if the feature **Add** is present

Delta Module DAddEval

```
delta DAddEval after DAdd
    when Add && Eval {
  modifies class Add {
    adds int eval() {
      return expr1.eval()
        + expr2.eval();
    }
  }
}
```

Delta Module DAddEval

```
delta DAddEval after DAdd
    when Add && Eval {
  modifies class Add {
    adds int eval() {
      return expr1.eval()
        + expr2.eval();
    }
  }
}
```

This **when** clause specifies that the delta module is applied if both **Add** and Eval features are present

Delta Module DAddEval

```
delta DAddEval after DAdd
    when Add && Eval {
  modifies class Add {
    adds int eval() {
      return expr1.eval()
        + expr2.eval();
    }
  }
}
```

The **after** clause specifies that this delta module has to be applied after **DAdd**

Delta Module DAddPrint

```
delta DAddPrint after DAdd
     when Add && Print {
   modifies class Add {
      adds void print() {
         expr1.print();
         System.out.print(" + ");
         expr2.print();
      }
   }
}
```

Product: Lit, Add, Print, Eval

```
interface Exp {
  void print(); int eval();
}

class Lit implements Exp {
  int value;
  Lit(int n) { value=n; }
  void print() {
    System.out.print(value);
  }
  int eval() {
    return value;
  }
}
```

```
class Add implements Exp {
  Exp expr1; Exp expr2;
  Add(Exp a, Exp b) {
    expr1=a; expr2=b;
  }
  void print() {
    expr1.print();
    System.out.print(" + ");
    expr2.print();
  }
  int eval() {
    return expr1.eval()
      + expr2.eval();
  }
}
```

Bibliography

- I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella. **Delta-Oriented Programming of Software Product Lines**. Proc. of International Conference on Software Product Lines (SPLC), 2010.
- J. Koscielny *et al*. **DeltaJ 1.5: Delta-oriented Programming for Java 1.5**. Proc. of the Int'l Conf. on Principles and Practices of Progr. on the Java platform (PPPJ), 2014.
- http://deltaj.sourceforge.net/