Feature Oriented Product Line Engineering

KV Product Line Engineering (343.354)

Dr. Roberto Lopez-Herrejon
Dr. Rick Rabiser

JOHANNES KEPLER UNIVERSITY LINZ
ISSE

6.4.2016

Feature Oriented Software Development

- Feature Oriented Software Development (FOSD)
  - A technology that studies feature modularity and its use in program synthesis
  - Key contribution
    - Mathematical models of features and their composition
    - Deals with non-code artifacts
- Program synthesis
  - Automatic program generation from specifications
- Developed at the University of Texas at Austin by Don Batory's research group

Current Focus of FOSD

- Mainly implementation level
  - Programming language extensions and support
  - Feature Oriented Programming (FOP)

- More recently
  - Analysis and modeling
  - Testing

Outline

- Expression Problem
  - Motivation
- Expressions Product Line (EPL)
  - Description
  - Features
- Foundations of Feature Oriented Programming

The Expression Problem

Buckle up !

The Expression Problem

- In the context of Programming Languages
  - Focus: extending mutually recursive types
  - More than a decade of research
- Our twist
  - View it as a canonical example of a product line
- Reading
  - The Expresion Problem Revisited – Torgersen

Problem Statement
Torgersen's version

- Assume you have a program that has two data types which implement an operation
  - Operation print that displays the expressions
  - Data types from the grammar

Exp	:= Lit | Add
Lit	:= non negative numbers
Add	:= Exp + Exp

In Plain Java

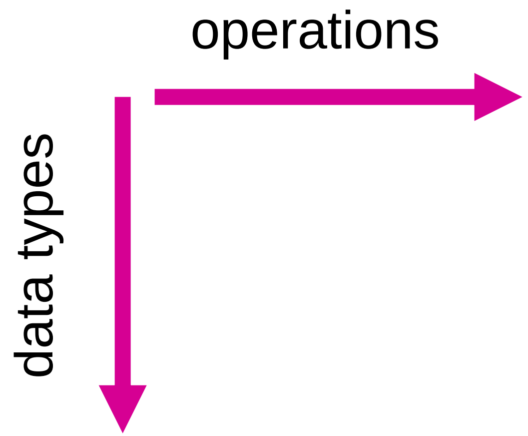
```
interface Exp {
  void print();
}

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

class Add implements Exp {
  public Exp left, right;
  Add(Exp l, Exp r) { left = l; right = r; }
  public void print() { left.print(); System.out.print('+'); right.print(); }
}
```

```
Exp e = new Add(new Lit(2),new Lit(3));
 e.print();
```

2+3

The Task

- Extend this program to
  - Add a new data type
  - Add a new operation



- The challenge – do it in both dimensions
  - Extend data types → support all operations
  - Extend operations → add ops to all the data types

In Torgersen's Example

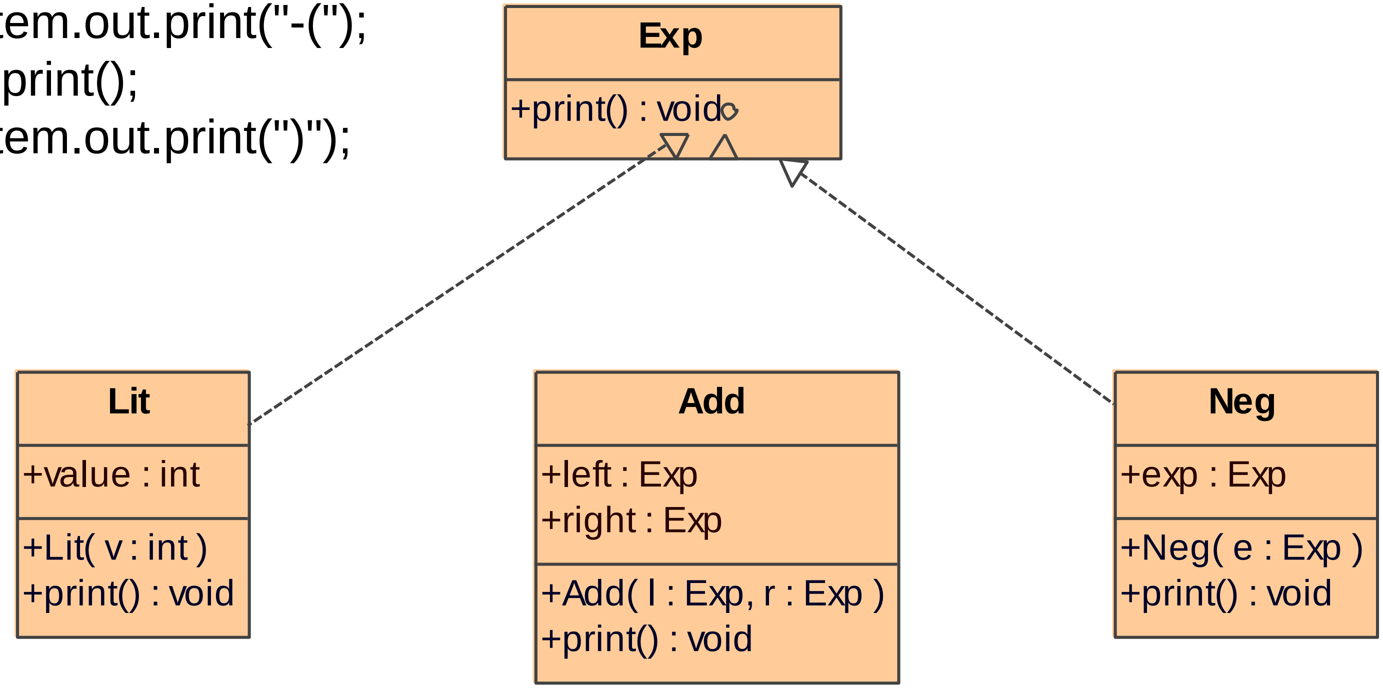
- Adding new operation
  - Eval that evaluates an expression and returns its value
  - Ex. Exp e = new Add(new Lit(2),new Lit(3));
    e.eval(); → 5
- Adding new data type

Exp	:= Lit | Add | Neg
Lit	:= non negative numbers
Add	:= Exp + Exp
Neg	:= - Exp

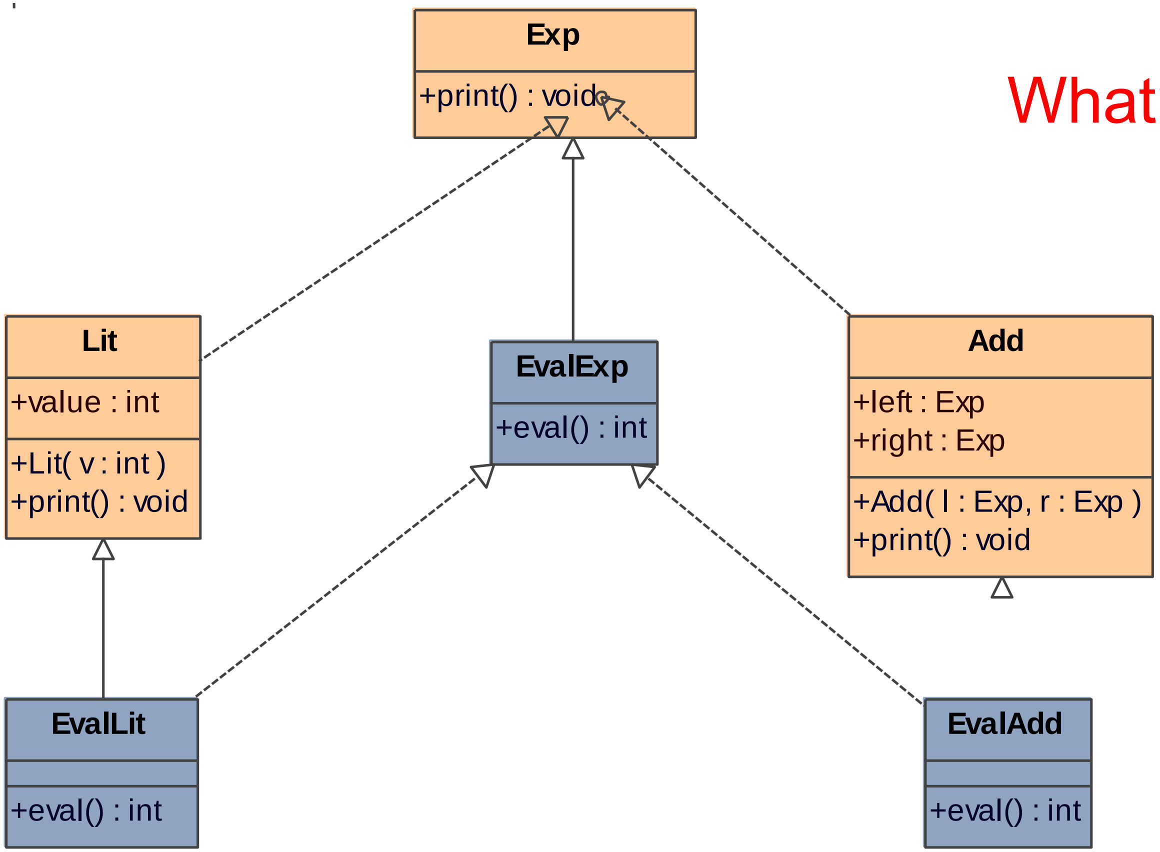
Extending The Data Types

```
class Neg implements Exp {
  public Exp exp;
  Neg(Exp e) { exp=e; }
    public void print() {
            System.out.print("-(");
            exp.print();
            System.out.print(")");
    }
}
```

Easy Part



Extending Operations
Naïve Approach



What's the problem?

The problem is …

- In class Add fields left are right don’t have method eval

```
class EvalAdd extends Add implements EvalExp {
    public int eval() { return left.eval() + right.eval(); }
}
```

program does not compile!!!

Expressions Product Line (EPL)

Relating Expression Problem
to Product Lines

Expressions Product Line (EPL)

- Develop a product line of arithmetic expressions
  - Members have different combinations of data types and operations

Torgersen's

	Print	Eval
Exp Lit Add		
Neg		

EPL

	Print	Eval
Exp Lit		
Add		
Neg		

What are the features?

- Two-dimensional matrix
  - Operations
  - Data types
- A feature
  - Implements an operation on a data type
  - Corresponds to a cell
- Notational convention
  - Feature name is data type initial and operation initial

	Print	Eval
Lit*	lp	le
Add	ap	ae
Neg	np	ne

* Exp is included

Some Members of EPL

- Ex. valid programs
  - Program with lp
  - Program with lp and ap
  - Program with lp, ap, and np

- Ex. invalid programs
  - Program with le – why?
  - Program with lp, ap, le – why?

An Extra Twist – Test Class

- Test class
  - Defines a field for each data type
  - Creates an object for each field
  - Runs selected operations (print and/or eval) on each data type field

- For example
  - The Test class with all data types and operations selected should be *equivalent* to the one next on slide
    - Equivalent means when executed they should do the same

An Extra Twist – Test Class

Example with all features selected

```
class Test {
  Lit ltree;
  Add atree;
  Neg ntree;
  Test() {
    ltree = new Lit(3);
    atree = new Add(ltree, ltree);
    ntree = new Neg(ltree);
  }
  void run() {
    ltree.print();
    atree.print();
    ntree.print();
    System.out.println(ltree.eval());
    System.out.println(atree.eval());
    System.out.println(ntree.eval());
  }
}
```

data types fields

data types object creation

print operation

eval operation

Feature Oriented Programming

Foundations

Recall

- Insight
  - A feature typically involves multiple classes and/or interfaces
- The expression problem
  - Combines data types and operations in an extensible way

Some History

- FOP has its origins in GenVoca
  - A methodology for creating application families and architecturally extensible software

- GenVoca key ideas
  - Feature extensions
  - Collaborations

Feature Extensions

- Feature extensions
  - Adds new functionality to an existing entity or feature
- Characteristics
  - Reusable – can be used to extend other entities
  - Interchangeable – with other feature extensions

Collaborations

- Collaboration
  - Set of objects and a protocol that determines how objects interact
- Role
  - Part of an object that enforces the protocol in a collaboration
- Collaborations goal
  - Provide a more flexible modularity unit to improve reuse in multiple configurations or compositions for the development of different programs (support SPL)

Feature Extensions (1)

```
class Lit {
    int value;
    Lit (int v) { value = v; }
    void print( ) { out.print(value);  }
    int eval( ) { return value; }
}
```

=

```
class Litc {
    int value;
    Litc (int v) { value = v; }
}
```

```
class Litp extends Litc {
  void print( ) { out.print(value); }
}
```

```
class Lite extends Litp{
    int eval( ) { return value; }
}
```

```
class Lit extends Lite   {  }
```

Lit has same functionality on both definitions

Feature Extensions (2)

```
class Litc {
    int value;
    Lit (int v) { value = v; }
}
```

```
class Litp extends Litc {
  void print( ) { out.print(value); }
}
```

```
class Lite extends Litp{
    int eval( ) { return value; }
}
```

```
class Lit extends Lite   {  }
```

≡

```
class Litc {
    int value;
    Lit (int v) { value = v; }
}
```

```
class Lite extends Litc{
    int eval( ) { return value; }
}
```

```
class Litp extends Lite {
  void print( ) { out.print(value); }
}
```

```
class Lit extends Litp   {  }
```

Insights

- A fresh look at inheritance as
  - Operation that increments or refines the functionality of a class
- Order is important
  - For inheritance to work properly
  - Ex. class Litc extends Litp { … } ✘
    - Doesn’t work because Litp needs the field in Litc

FOP Feature Extensions

Standard Inheritance | **Feature Extensions**

```
class Litc {
   int value;
   Lit (int v) { value = v; }
}
```

```
class Lit {
   int value;
   Lit (int v) { value = v; }
}
```

```
class Litp extends Litc {
  void print( ) { out.print(value); }
}
```

=

```
refines class Lit {
  void print( ) { out.print(value); }
}
```

```
class Lite extends Litp{
   int eval( ) { return value; }
}
```

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

```
class Lit extends Lite   {  }
```

refines – keyword for class extensions

Key Insight of FOP

- Classes and their refinements can be algebraically expressed as functions

- Two kinds
  - Values – standard code (constant functions)
  - Functions – class refinements

Values and Functions

value →

```
class Lit {
    int value;
    Lit (int v) { value = v; }
}
```

function →

```
refines class Lit {
  void print( ) { out.print(value); }
}
```

Function Composition

$\mathbf{Lit_c}$

```
class Lit {
    int value;
    Lit (int v) { value = v; }
}
```

$\mathbf{Lit_p}$

```
refines class Lit {
  void print( ) { out.print(value); }
}
```

$\mathbf{Lit_e}$

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

composer

```
class Lit {
    int value;
    Lit (int v) { value = v; }
    void print( ) { out.print(value);  }
    int eval( ) { return value; }
}
```

Composition Expression

$Lit_e\ (Lit_p\ (\ Lit_c\ (\ )\ )\ )$

$Lit_e \bullet Lit_p \bullet Lit_c$ ← **shorthand**

Composition Expressions and Product Lines

- If we use Lit_e, Lit_p, and Lit_c features we can compose 4 different programs
  - Lit_c → core
  - $Lit_p \bullet Lit_c$ → core, print
  - $Lit_e \bullet Lit_c$ → core, eval
  - $Lit_e \bullet Lit_p \bullet Lit_c$ → core, print, eval

Insights

- Product line
  - Set of programs created from all valid composition expressions

- A composition expression is valid if it does not violate any constraints of the features it composes

- A constraint is a problem domain requirement of features in a product line design
  - For example, let f and g be features
    - feature f must be composed before feature g
    - feature inclusion – if f is included g is included
    - feature exclusion – if f is included g is excluded
    - feature optionality – f may or may not be included
    - mandatory features – f must be included always

Invalid Composition Example

- Invalid composition
  - $Lit_e \bullet Lit_c \bullet Lit_p$
- Why?
  - Lit_p depends on Lit_c
- What about?
  - $Lit_e \bullet Lit_p$ ✘

Lit_c

```
class Lit {
    int value;
    Lit (int v) { value = v; }
}
```

Lit_p

```
refines class Lit {
  void print( ) { out.print(value); }
}
```

Lit_e

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

Collaborations

- Typically features consists of more than one class
- Example a Test class
- Question
  - How Lit and Test relate?
- Answer
  - If a program has an operation it must have a test for it

$Test_c$

```
class Test {
  Lit t;
  Test() { t = new Lit(3); }
}
```

$Test_p$

```
refines class Test {
   void testPrint() { t.print(); }
}
```

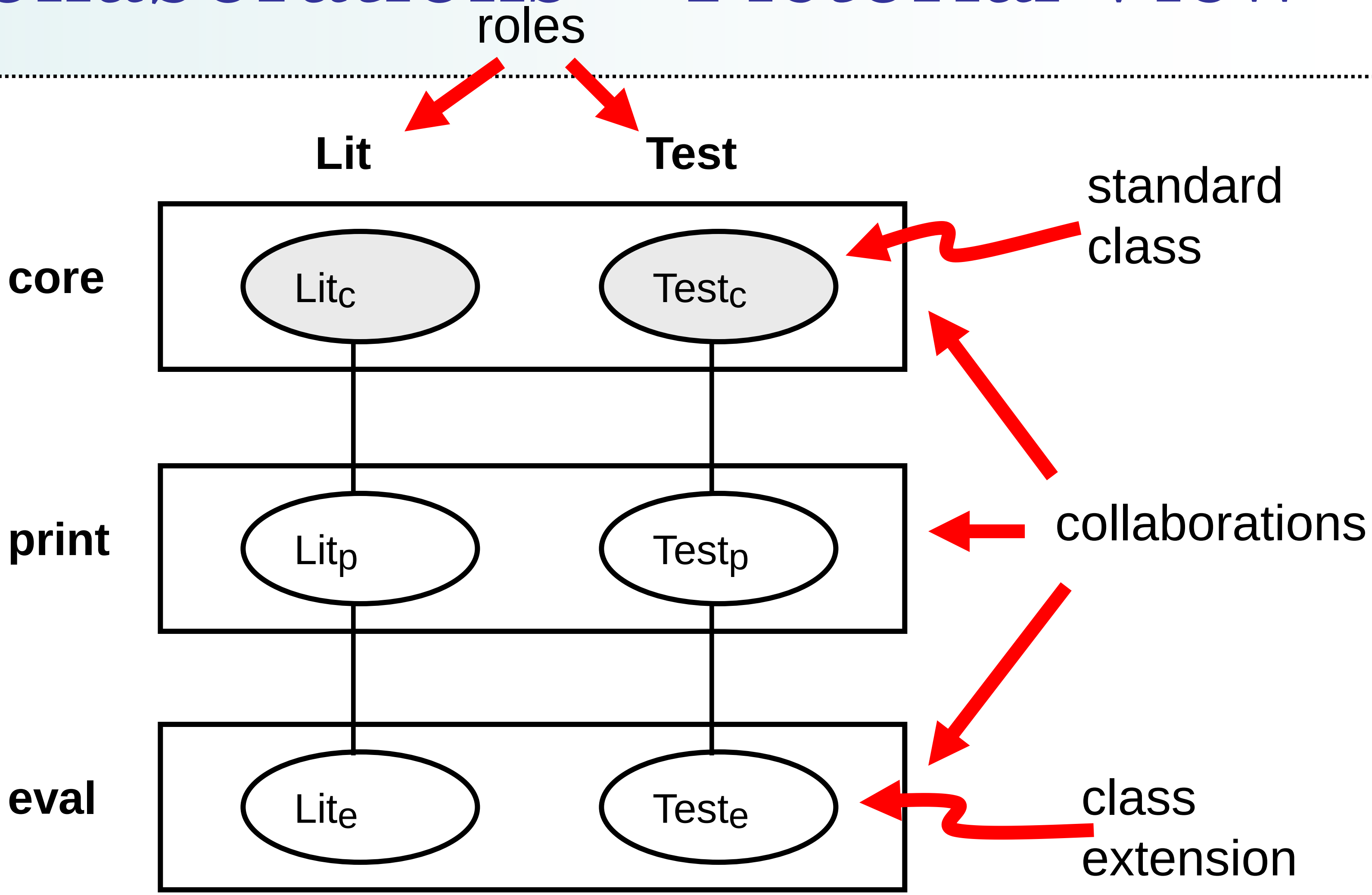
$Test_e$

```
refines class Test {
 void testEval()  { out.print(t.eval()); }
}
```

Lit and Test Collaboration

- Lit class
  - Lit_c
  - $Lit_p \bullet Lit_c$
  - $Lit_e \bullet Lit_c$
  - $Lit_e \bullet Lit_p \bullet Lit_c$
- Test
  - $Test_c$
  - $Test_p \bullet Test_c$
  - $Test_e \bullet Test_c$
  - $Test_e \bullet Test_p \bullet Test_c$

Collaborations – Pictorial View



AHEAD

- AHEAD stands for
  - Algebraic Hierarchical Equations for Application Design
- Generalizes GenVoca
  - Features are hierarchical modules
  - Features can include non-code artifacts

Features as Hierarchical Modules

- AHEAD Model
  - Set of features that can either be values or functions

- Example – our product line of Lit and Test

  LitTest = { core, print, eval } where

  core = { Lit_c, $Test_c$ }

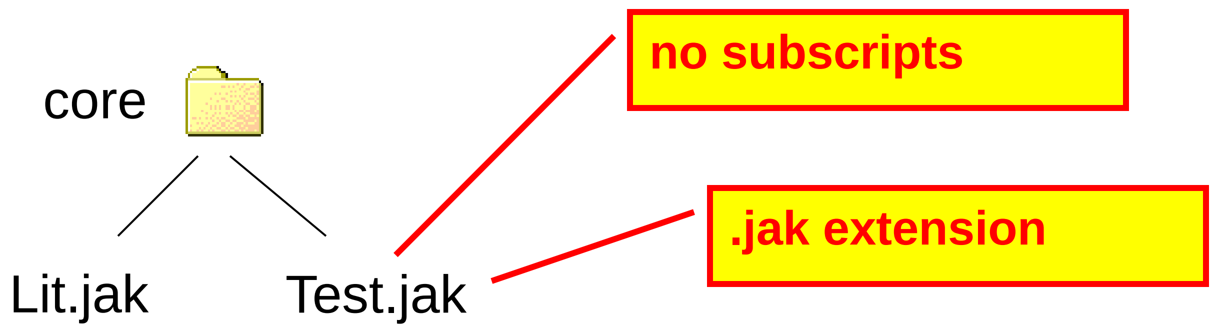
  print = { Lit_p, $Test_p$ }

  eval = { Lit_e, $Test_e$ }

subscripts for distinction only

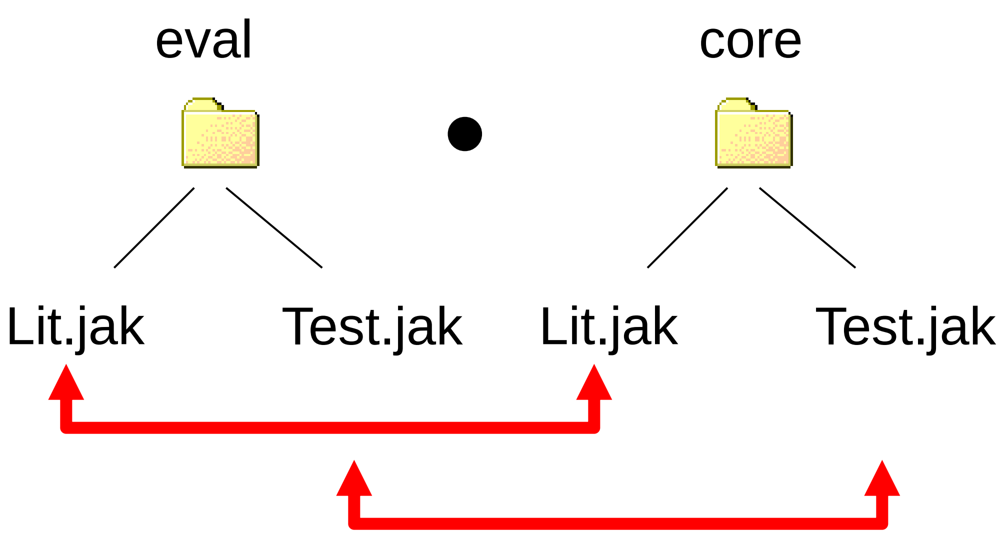
Feature Module Implementation

- Feature modules can have nested modules
- Implementation with file directories
  - A feature is a directory
  - Nested features are subdirectories
- Example: feature core



Feature Module Composition

- Composition
  - Two modules are composed if they have the same names and the same nesting levels
- Ex. CE = eval • core **= {Lit$_e$ • Lit$_c$, Test$_e$ • Test$_c$ }**



core = { Lit$_c$, Test$_c$ }

eval = { Lit$_e$, Test$_e$ }

Law of Composition

- Let X and Y be feature modules defined as:

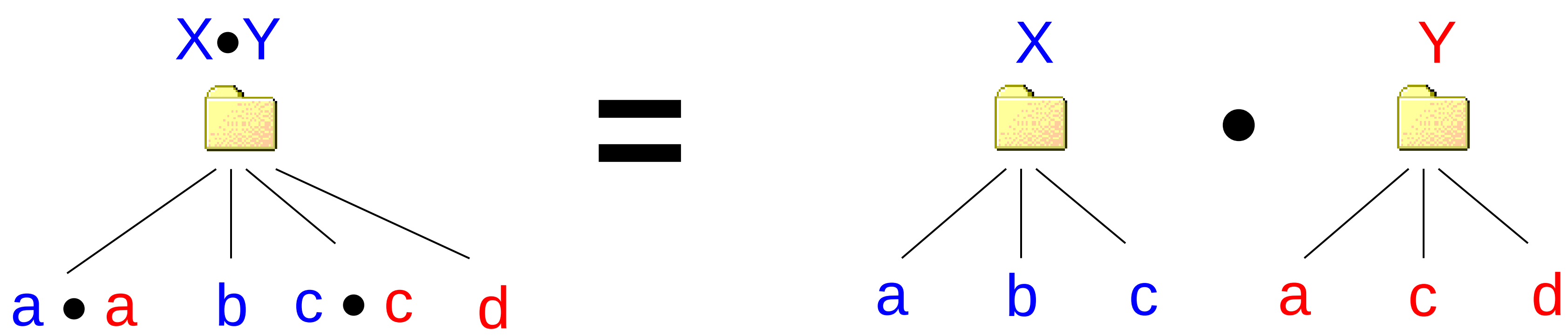
$$X = \{\ a_x\ ,\ b_x\ ,\ c_x\ \}$$
$$Y = \{\ a_y\ ,\ c_y\ ,\ d_y\ \}$$

  where a, b, c, and d are nested features whose subscripts indicate the feature module they belong to. The composition of X and Y, denoted as X•Y, is:

$$X \bullet Y = \{\ a_x\ ,\ b_x\ ,\ c_x\ \} \bullet \{\ a_y\ ,\ c_y\ ,\ d_y\ \}$$
$$= \{\ a_x \bullet a_y\ ,\ b_x\ ,\ c_x \bullet c_y\ ,\ d_y\ \}$$

- Nested features are composed by names – ignoring subscripts
- The features whose names do not have a match, like b_x or d_y, are simply copied

Law of Composition



$$Y = \{ a_y, \quad c_y, \quad d_y \}$$

$$X = \{ a_x, \quad b_x, c_x \quad \}$$

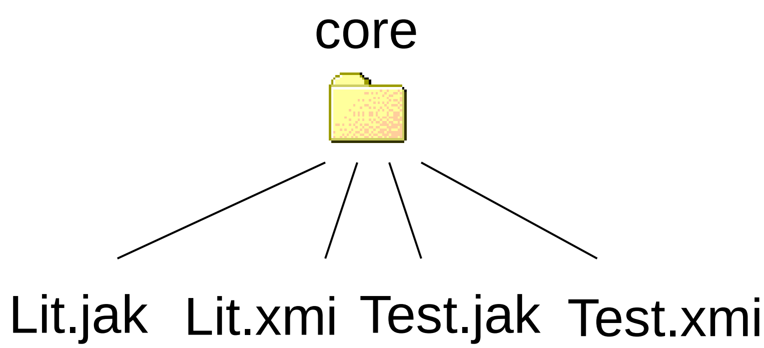
$$X \bullet Y = \{ a_x \bullet a_y, b_x, c_x \bullet c_y, d_y \}$$

Features With Non-Code Artifacts

- Usually programs have other representations besides code
- Examples
  - UML diagrams, XML-based files, grammars, requirements documentations, etc
- Insight
  - Feature modules should be able to contain non-code artifacts and compose them

Non-Code Artifact Example

- Consider UML diagrams representation
  - XML Metadata Interchange (XMI)
- Ex. feature core



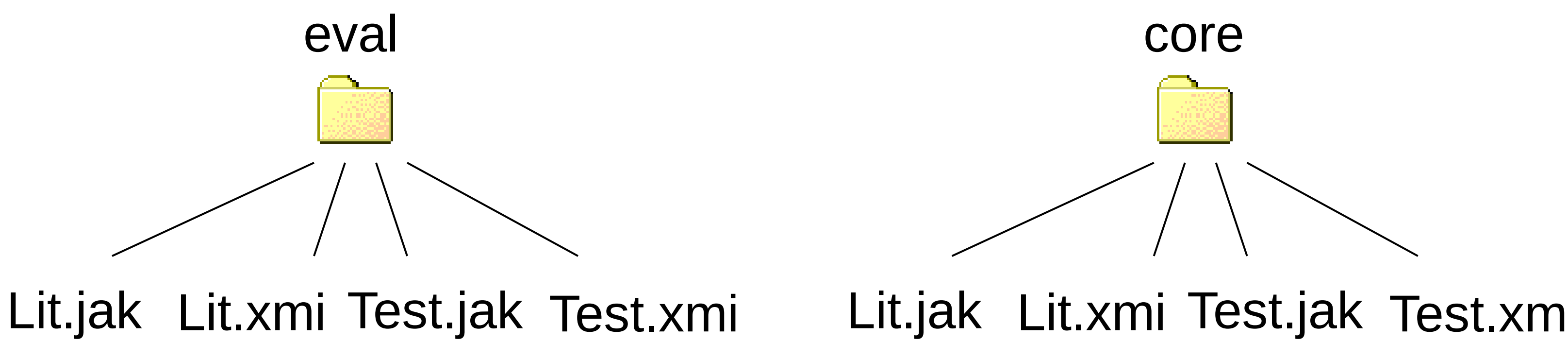
Principle of Uniformity

- Principle of Uniformity
  - Impose a structure on artifacts that resembles the value and function notions of object-based source code

- Implications
  - Treat standard non-code artifact as values
  - Extend artifact definitions to accommodate for functions
  - Provide artifact-specific composition tools

Feature Composition
With Non-Code Artifacts

- As before but now consider file extension for name matching



$$\begin{aligned}\text{eval}\bullet\text{core} &= \{\ \text{Lit}_e.\text{jak},\ \text{Lit}_e.\text{xmi},\ \text{Test}_e.\text{jak},\ \text{Test}_e.\text{xmi}\ \}\ \bullet \\ &\quad\ \{\ \text{Lit}_c.\text{jak},\ \text{Lit}_c.\text{xmi},\ \text{Test}_c.\text{jak},\ \text{Test}_c.\text{xmi}\ \} \\ &= \{\ \text{Lit}_e.\text{jak} \bullet \text{Lit}_c.\text{jak},\ \text{Lit}_e.\text{xmi} \bullet \text{Lit}_c.\text{xmi}, \\ &\quad\ \ \text{Test}_e.\text{jak} \bullet \text{Test}_c.\text{jak},\ \text{Test}_e.\text{xmi} \bullet \text{Test}_c.\text{xmi}\ \}\end{aligned}$$

Tool Support – Several Alternatives

- Original implementation
  - AHEAD Tool Suite (ATS)
  - Limitation: syntax support for Java 1.4 only
- FeatureHouse
  - Does not make extensions to languages
  - Standard Java, C++, and other language
- FeatureIDE
  - Eclipse plug-in for AHEAD

AHEAD Tool Suite (ATS)

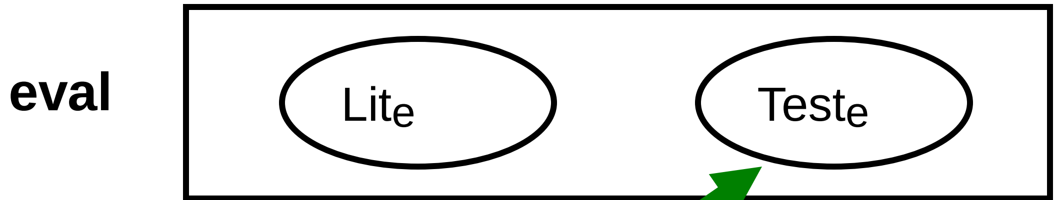
- An implementation of FOP
- We cover to help you understand related literature mostly based on this tool
- Uses a Java-extended language call *Jak*
- It is in itself a product line
- Currently supports composition of XML files, grammar files, equation files

Jak Language

- Jak makes several extensions to Java language

- Key ones
  - Refinement – class refinement
  - Feature containment – feature name in classes
  - *Super* construct – for method extensions
  - Constructor extension

Feature Refinement and Containment




```
layer eval;
refines class Test {
  void testEval( )  { System.out.println(t.eval()); }
}
```

Super Construct

- Indicate method extensions
  - add new functionality to a method in a class extension

- Syntax

  Super(paramtypes).method (actparms);

- Example
  - Remember Test class and its refinements
  - Task. Define a method *run* that is extended to call operation print and *eval* on the test object t

Super Construct Example

Test obj = new Test();
obj.run();

core

```
class Test {
        Lit t; Test() {  t = new Lit(3); }
        void run() {  …  }
}
```

print

```
refines class Test {
        void run() {  Super().run(); t.print(); }
}
```

eval

```
refines class Test {
        void run() { Super().run(); System.out.println(t.eval()); }
}
```

$\text{Test} = \text{Test}_e \bullet \text{Test}_p \bullet \text{Test}_c$

control flow

Constructor Extension

- Counterpart of method extension
- Example

class extension

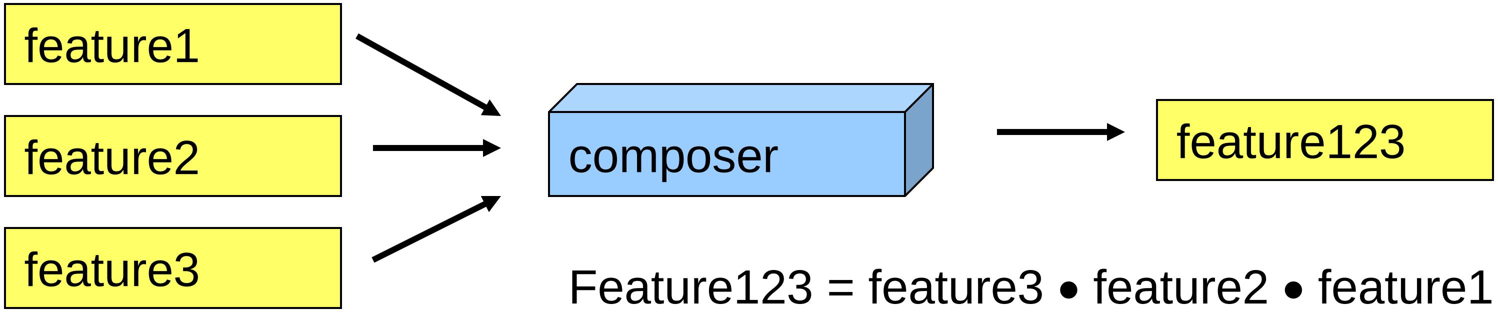
```
layer print;
refines class Test {
   Test t2;
   refines Test() { t2 = new Test(); }
}
```

construction extension

Composing Features

- A program is specified by an equation
  - composition of features in a given order

- Main tool of AHEAD is composer
  - composes the specifications of features



Feature123 = feature3 • feature2 • feature1

Other Topics of AHEAD

- Design rules
  - Mechanism to validate the semantics of feature compositions
- Other tools
  - XML composition – XAK tool
  - Reform – pretty printer
  - guidsl – automatically creates GUI from DSL spec

FeatureHouse

- Based on native Java syntax
  - For simplicity, we will use it for our exercise

- Main differences with AHEAD Tool Suite
  - No use of refines keyword, refinements based on composition equation
  - No use of layer keyword
  - Instead of Super keyword uses special method original

- Execution

  java –jar <FH jar file> -- expression <equationfile>

Further Reading

- Yannis Smaragdakis, Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Transactions on Software Engineering and Methodology (TOSEM). Volume 11 , Issue 2 (April 2002)
- S. Trujillo, D. Batory, and O. Diaz. "Feature Refactoring a Multi-Representation Program into a Product Line", Generative Programming and Component Engineering (GPCE), October 2006.
- Don Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite, Proc. Summer School on Generative and Transformation Techniques in Software Engineering, Braga, Portugal, R. Laemmel, J. Saraiva, and J. Visser, ed., Vol 4143, Lecture Notes in Computer Science, Springer-Verlag, 2006.
- D. Batory, J.N. Sarvela, and A. Rauschmayer. "Scaling Step-Wise Refinement", IEEE Transactions on Software Engineering (IEEE TSE), June 2004.
- D. Batory, S. O’Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions of Software Engineering and Methodology (TOSEM). Vol1. No. 4. October 1992
- J. Blair, D. Batory. A Comparison of Generative Approaches: XVCL and GenVoca. Technical Report. Computer Sciences Department. University of Texas at Austin.

Upcoming Schedule

- On 13.04
  - Aspect-Oriented PLE + Exercise 2.2 (RL)
- Assignment 2. Part I.
  - Due April 20th, 12:00 pm.

Questions?

- Now or later to rick.rabiser@jku.at | roberto.lopez@jku.at