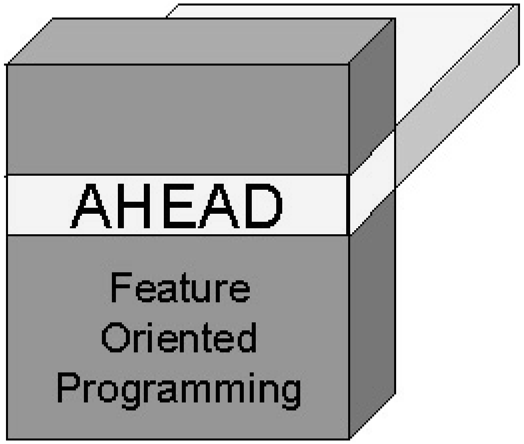
Feature Oriented Programming for Product-Lines

Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu
www.cs.utexas.edu/users/dsb/

February 2005



Presented at:
Summer School on Generative and Transformational Techniques in Software Engineering,
July 4-8, Braga, Portugal

Feature Oriented Programming for Product-Lines

Don Batory
Department of Computer Sciences
University of Texas at Austin
batory@cs.utexas.edu
www.cs.utexas.edu/users/dsb/

Introduction

- A **product-line** is a family of similar systems
  - Chrysler mini-vans, Motorola radios, software
- Motivation: economics
  - amortize cost of building variants of program
  - design for family of systems
- Key idea of product-lines
  - members of product-line are differentiated by features
  - **feature** is product characteristic that customers feel is important in describing and distinguishing members within a family
  - **feature** is increment in product functionality

Introduction

- **Feature Oriented Programming (FOP)** is the study of feature modularity in product-lines
  - features are first-class entities in design
  - often implemented by crosscuts
- History of applications
  - 1986 database systems
  - 1989 network protocols
  - 1993 data structures
  - 1994 avionics
  - 1997 extensible Java compilers
  - 1998 radio ergonomics
  - 2000 program verification tools
  - 2002 ExCIS fire support simulator
  - 2003 AHEAD tool suite
  - 2004 robotics controllers

Very Rich Technical Area...

- Integrates many different areas
  - compilers
  - grammars
  - artificial intelligence
  - databases
  - algebra
  - programming languages
  - compositional programming & reasoning
  - OO software design
  - software engineering
  - aspect-oriented programming
  - others...

Tutorial Overview

- Part I
  - The FOP Paradigm
  - The Theory
  - AHEAD Tool Suite
- Part II
  - Aspect Composition
  - Verification and Design Rule Checking
  - Multi-Dimensional Models

buckle up!

The FOP Paradigm

a general approach to program development and product-line synthesis

Motivation

- Software products are:
  - increasing in complexity
  - increasing in costs to develop and maintain
  - decreasing in ability to understand
- Basic goal of SE is to manage and control complexity
  - structured programming to
  - object oriented programming to
  - component-based programming to...
  - **today's design techniques are too low-level, exposing too much detail to make application's design, construction and modification simple**

progressively increasing abstractions

- Something is missing...
  - future design techniques generalize today's techniques
  - tutorial to expose a bigger universe

Keys to the Future

- New paradigms will likely embrace:
  - **Generative Programming (GP)**
    - want software development to be automated
  - **Domain-Specific Languages (DSLs)**
    - not Java & C#, but high-level notations
  - **Automatic Programming (AP)**
    - declarative specs → efficient programs
- Need simultaneous advance in all three fronts to make a significant change

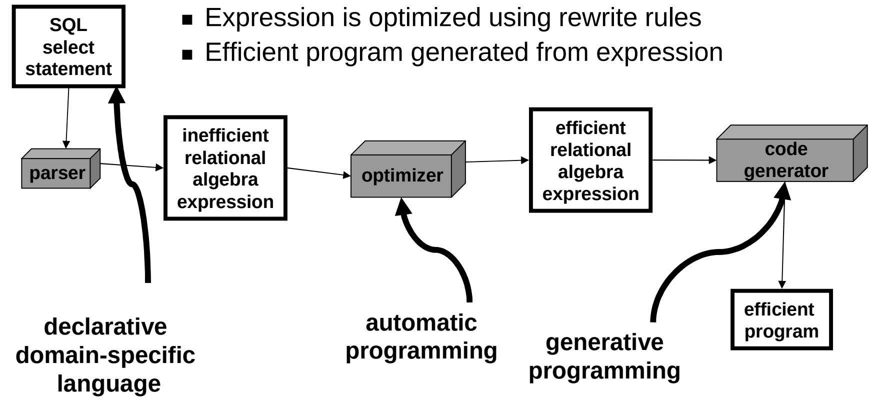
Not Wishful Thinking...

- Example of this futuristic paradigm realized over 25 years ago
  - around time that AI researchers gave up on automatic programming

Relational Query Optimization

Relational Query Optimization

- Declarative query is mapped to an expression
- Each expression represents a unique program
- Expression is optimized using rewrite rules
- Efficient program generated from expression



Keys to Success

- Automated development of query evaluation programs
  - hard-to-write, hard-to-optimize, hard-to-maintain
  - revolutionized and simplified database usage
- Created an **algebra-based science** to specify and optimize query evaluation programs
- Identified fundamental operations of this domain
  - relational algebra
- Represented program designs as **expressions**
  - compositions of relational operations
- Define algebraic identities among operations to optimize equations
- Compositionality is hallmark of great engineering models

Looking Back and Ahead

- Query optimization (and concurrency control) helped bring DBMSs out of the stone age
- Holy Grail Software Engineering:

Repeat this success in other domains

- Not obvious how to do so...
- It can be done! Subject of this tutorial…
  - series of **simple** ideas that generalize notions of **modularity** and lay groundwork for practical **compositional programming** and an **algebra-based science** for software design

A Basis for a Science of Software Design

What motivates FOP and how is it formalized?

Today's View of Software

- Today's models of software are too low level
  - expose classes, methods, objects as focal point of discourse in software design and implementation
  - difficult (impossible) to
    - reason about construction of applications from components
    - produce software automatically from high-level specifications (distance is too great)
- We need a more abstract way to specify systems

A Thought Experiment...

- Look at how people describe programs now...
  - don't say which DLLs are used...
- Instead, say what **features** a program offers its clients

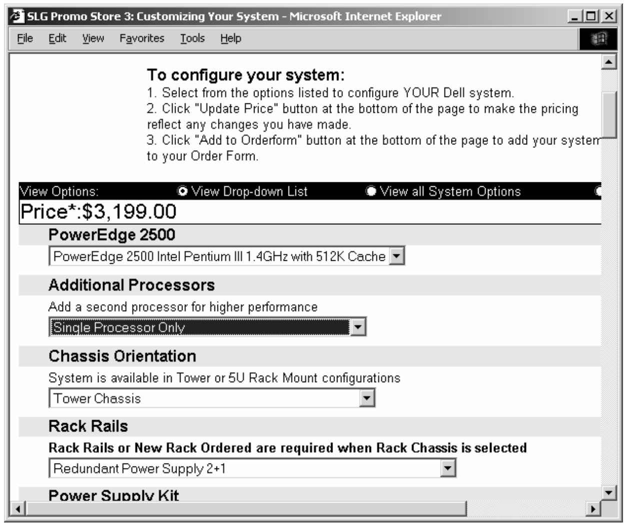
  Program1 = feature_X + feature_Y + feature_Z

  Program2 = feature_X + feature_Q + feature_R

  - why? because features align better with requirements
- We should specify systems as **compositions of features**
  - nobody does this for software (now)
  - done in **lots** of other areas

Dell Web Site

declarative DSL to select features of desired system



Chinese Menu – Declarative DSL

HOME COOKING COUNTRY STYLE

There are five basic styles of Szechuan cooking:

KUNG-PAO

Hot pepper sauce, fresh meat marinated with five spicy sauce, stir with peanuts and diced scallions.

Choose from:

Shrimp	10.95
Chicken	8.95
Beef	9.50
Assorted	10.95

KAN-SHAO

Szechuan style. Sliced water chestnuts and heart of scallions, all gently simmered in spicy hot and sour sauce.

Choose from:

Chicken	8.95
Shrimp	10.95
Lobster	17.95

SIZZLING

A variety of vegetables dramatically served on sizzling hot plate.

Choose from:

Chicken	8.95
Shrimp (Kan-shao Sauce)	10.95
Beef and Scallops	10.50
Assorted	10.95

YU-HSIANG

Garlic style. Shredded fresh meat with ginger and garlic, minced water chestnuts and tree ears in a spicy garlic sauce.

Choose from:

Shrimp	10.95
Pork	8.95
Beef	9.50
Chicken	8.95
Scallops	10.95
Eggplant	8.50
Assorted	10.95

HUNAN STYLE

Fresh broccoli with jalapeno black bean sauce.

Choose from:

Shrimp	10.95
Chicken	8.95
Beef	9.50
Scallops	10.95
Assorted	10.95

APPETIZERS

Steamed Spring Roll	1.75
Vegetarian Egg Roll	1.50
Egg Roll	1.25
Fried Chicken Wings (6)	3.00
Fried Calamari	5.95
B. B. Q. Ribs	5.25
Beef (Chicken) Kabobs (4)	5.25
Fried Won Ton (6)	1.50
Crisp Honey Walnuts	4.95
Butterfly Shrimp (4)	5.25
Crab Puffs (6)	3.50
Shrimp Cake	4.95
Steamed or Fried (Pork or Vegetable) Dumplings (6)	4.95
Pu Pu Tray Combination Platter for Two	8.95
Chicken Lettuce Wrap (Seafood $6.95)	5.95
Spicy Wonton	4.50
Super Crispy Veggies	4.95

SOUP

Egg Drop Soup	1.50
Hot & Sour Soup	1.75
Miso Soup	1.75
Wonton Soup	1.75
Shrimp with Lemon Grass Soup for Two	6.25
Taiwanese Seafood Hot & Sour Soup for Two	6.25
Shredded Chicken with Dry Mushroom Soup for Two	4.95
Dragon Phoenix Soup for Two	6.25
Vegetables and Bean Curd Soup for Two	4.95

©dsbatory2005

Methodology for Construction

- What methodology builds systems by progressively adding details?
- **Step-Wise Refinement**
  - Dijkstra, Wirth early 1970s
  - abandoned in early 1980s as it didn't scale...
  - had to compose hundreds or thousands of transforms (rewrites) to produce admittedly small programs
  - recent work shows how SWR scales
    – scale individual transform to a **feature**
    – composing a few refinements yields an entire system

©dsbatory2005

Terminology Disclaimer

- We use OO meaning of term "refinement"
  - elaboration of an entity (entities) that introduces a new service, feature, or relationship
- In algebraic communities
  - "refinement" means add detail, but **no** new capability e.g., implement an interface
  - our use of 'refinement' is 'extension' in algebraic communities
  - "step wise development"
- Henceforth follow the algebraic community terminology...

©dsbatory2005

What is a Feature?

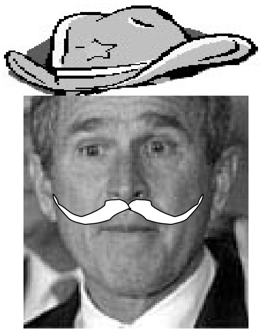
- **Feature**
  - an elaboration or augmentation of an entity(s) that introduces a new service, capability, or relationship
  - increment in functionality
- Characteristics
  - abstract, mathematical concept
  - reusable
  - interchangeable
  - (largely) defined independently of each other
- Illustrate in next few slides

©dsbatory2005

Tutorial on Features (Extensions)

©dsbatory2005

Features are Interchangable



©dsbatory2005

Features are Interchangable

©dsbatory2005

Features are Interchangable

©dsbatory2005

Features are Interchangable

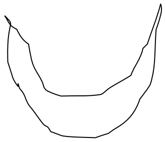
Features are Reusable

Features are Functions!

PersonPhoto beanie(PersonPhoto x)

PersonPhoto uncleSam(PersonPhoto x)

PersonPhoto mustache(PersonPhoto x)



PersonPhoto lincolnBeard(PersonPhoto x)

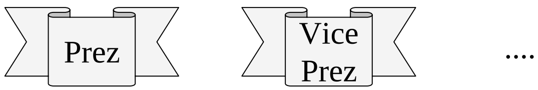
Composing Features

- Feature composition = function composition

 = lincolnBeard(uncleSam())

Large Scale Features

- Called **Collaborations (1992)**
  - simultaneously modify multiple objects/entities
  - extension of single entity is called **role**
  - recognize as crosscuts in software
- Example: Positions in US Government
  - each defines a role



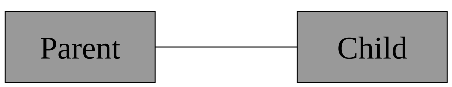
Composing Collaborations

- At election-time, collaboration remains constant, but objects that are extended are different

Example of dynamic composition of collaborations

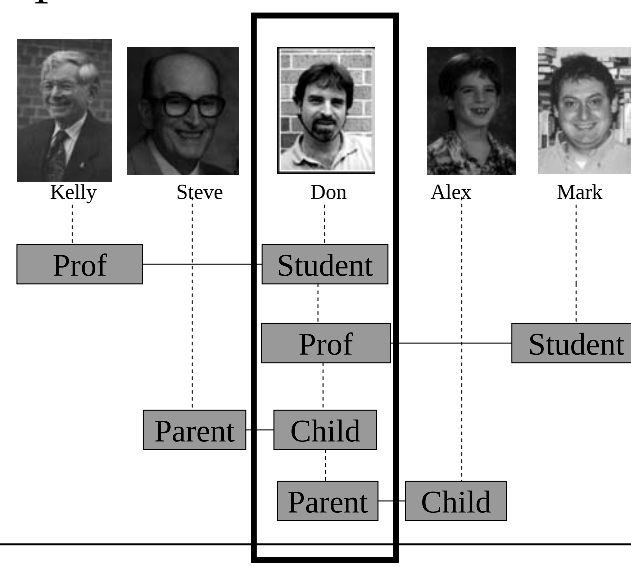
Other Collaborations

- Parent-Child collaboration



- Professor-Student collaboration

Example



Same Holds for Software!

Highly complex entities and relationships in software can be synthesized by composing generic & reusable features

Feature Oriented Programming

- **Feature Oriented Programming (FOP)** is study of feature modularity and programming models for product-lines
  - a powerful form of FOP based on step-wise development
  - advocates complex programs constructed from simple programs by incrementally adding features
- How are features and their compositions modeled?

Part I: The Theory

GenVoca and AHEAD

A Clue…

- Consider any Java class C
  - member could be a data field or method
  - class C below has 4 members **`m1—m4`**

```
class C {
   member m1;
   member m2;
   member m3;
   member m4;
}
```

Have You Ever Noticed…

- Contents of C can be distributed across an inheritance hierarchy?

```
class C {
   member m1;
   member m2;
   member m3;
   member m4;
}
```

=

```
class C1 {
   member m1;
}

class C23 extends C1 {
   member m2;
   member m3;
}

class C4 extends C23 {
   member m4;
}

class C extends C4 {}
```

©dsbatory2005

Another Example...

- C23 decomposed further as:

```
class C23 extends C1 {
   member m2;
   member m3;
}
```

=

```
class C2 extends C1 {
   member m2;
}

class C3 extends C2
   member m3;
}

class C23 extends C3 {}
```

©dsbatory2005

Observe…

- Significance: class definition need not be monolithic, but can be built by incrementally composing reusable pieces via inheritance
- Nothing special about the placement of members **m1…m4** in this hierarchy except...
  - **no-forward references**: member can be introduced as long as all members it references are defined
  - requirement for compilation, step-wise development

©dsbatory2005

Look Familiar?? Remember Algebra?

- Consider sets and union operation (∪)
  - commutative
    almost like inheritance...

```
C1 = { m1 }
C2 = { m2 }
C3 = { m3 }
C4 = { m4 }

C = C1 ∪ C2 ∪ C3 ∪ C4
  = { m1, m2, m3, m4 }
```

- Vector addition (+)
  - is commutative
    almost like inheritance

```
C1 = (m1,0,0,0)
C2 = (0,m2,0,0)
C3 = (0,0,m3,0)
C4 = (0,0,0,m4)

C = C1 + C2 + C3 + C4

  = ( m1, m2, m3, m4 )
```

©dsbatory2005

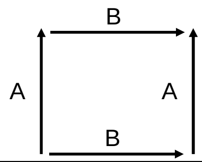
A Closer Analogy

- Vector **join** (→)
- Vector join lays vectors end-to-end to define a **path**
- Not commutative! – Order of composition matters!

```
C1 = (m1,0,0,0)
C2 = (0,m2,0,0)
C3 = (0,0,m3,0)
C4 = (0,0,0,m4)
```

C1 → C2 → C3 → C4 ≠ C4 → C3 → C2 → C1

A → B ≠ B → A



path followed by
A → B is different
than B → A;
end point is the same

Operation We Want...

- Is not quite inheritance...
  - want to add new methods, new fields, and extend existing methods like inheritance
  - also want constructors to be inherited and extended as well, (inheritance doesn't provide this)

```
class C1 {            class C2 {            class C12 {
   constructor#1   ●     constructor#2   =     constructor#1
}                     }                        constructor#2
                                            }
```

The operation ● we want is called **class extension**

Syntax of Class Extension

- Suppose program P has single class B

```
class B { int x; }
```

- A extension R adds y, z()

```
extends class B {
   int y;
   void z(){...}
}
```

- Composition of R with P defines a new program N:

```
class B {
int x;
int y;
void z(){...}
}
```


Algebraic Formulation

- Base programs are **constants**

```
// constant P
class B { int x; }
```

- Extensions are **functions**

```
// function R
extends class B {
   int y;
   void z(){...}
}
```

- Composition is an **expression** or **equation**

```
N = R( P )
  = R ● P
```

- yields:

```
class B {
   int x;
   int y;
   void z(){...}
}
```

Treat programs as values

Another Example

```
class C { member m1; }            // constant C1

extends class C { member m2; }   // function C2
extends class C { member m3; }   // function C3
extends class C { member m4; }   // function C4
```

- Composition is an **expression** or **equation**

```
C   = C4( C3( C2( C1 ) ) )

    = C4 • C3 • C2 • C1
```

Note:
both notations
are equivalent

Method Extension ala Inheritance

```
result =   method_extension      •   base_method

        void foo() {                  void foo() {
   =       /* before stuff */    •       /* do something */
           super.foo();               }
           /* after stuff */
        }

        void foo() {
           /* before stuff */
   =       /* do something */
           /* after stuff  */
        }
```

(or an equivalent encoding)

Connecting the Dots...

- Scalability
  - effects of extension not limited to a single class
  - **collaborations** encapsulate extensions of multiple classes **as well as adding new classes**
    - adding new classes that can be extended is **critical**

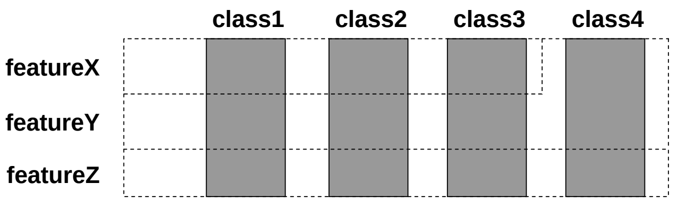
Connecting the Dots...

- A **collaboration** has meaning when it implements a feature
  - ever add a new feature to an existing OO program?
  - several classes must be extended as well as adding new classes
  - crosscuts

Program Synthesis Paradigm

Note: each feature crosscuts multiple classes

Program P = **featureZ ● featureY ● featureX**



By composing features, packages of fully-formed classes are synthesized

Contributors to this view…

- Many researchers have variants of this idea:
  - refinements - Dijkstra, Wirth 68
  - layers - Dijkstra 68, Batory 84
  - product-line architectures - Kang 90, Gomaa 92…
  - collaborations - Reenskaug 92, Lieberherr 95, Mezini 03
  - program verification - Boerger 96
  - aspects - Kiczales 97, et al.
  - concerns - Ossher-Harrison-Tarr 99

Connecting the Dots...

- You can always decompose software in this manner
  - trick is that your extensions be reusable
  - that's the connection with features, product-lines
  - features are reusable – so too must be their implementations

Design is the Key

  - software that is not designed to be reusable, composable, etc. with other software won't be – this is co-design or designing to a standard
  - **Architectural Mismatch** (ICSE 1995)
- **Product-line design** – feature implementations are designed with compositionality, reusability in mind

GenVoca (1988,1992)

- Equates constants, functions with features
- Constants:
  - f – base program with feature f
  - h – base program with feature h
- Functions
  - i ● x – adds feature i to program x
  - j ● x – adds feature j to program x

- A **domain model** or **product-line model** or **GenVoca model M**
  - set of constants (base programs)
  - functions (program extensions)

M = { f, h, ... i, j, ... }

Function Composition

- Multi-featured applications are equations

app1 = i • f	- application with features f and i
app2 = j • h	- application with features h and j
app3 = i • j • f	- your turn...

Given a GenVoca model, we can create a family of applications by composing features

©dsbatory2005

Expression Optimization

- Constants, functions represent both feature and its implementation
  - different functions with different implementations of the *same* feature

```
k1 • x  // adds k with implementation #1 to x
k2 • x  // adds k with implementation #2 to x
```

- When application requires feature **k**, it is a matter of optimization to determine the best implementation of **k**
  - counterpart of relational optimization
  - more complicated rewrites possible too…
- See: Batory, Chen, Robertson, and Wang, **Design Wizards and Visual Programming Environments for GenVoca Generators**, *IEEE Transactions on Software Engineering*, May 2000, 441-452.

©dsbatory2005

Generalization of Relational Algebra

- Keys to success of Relational Optimizers
  - expression representations of program designs
  - rewrite expressions using algebraic identities
- **Here's the generalization**:
  - domain model is an **algebra** for a domain or product-line
    - is set of operations (constants, functions) that represent stereo-typical building blocks of programs/members
    - compositions define space of programs that can be synthesized
  - given an algebra:
    - there will always be algebraic identities among operations
    - these identities can be used to optimize expression representations of programs, just like relational optimizers

©dsbatory2005

Composition Constraints

- GenVoca constants, functions seem untyped...
- **Design Rules** are domain-specific constraints that govern legal compositions
  - ex: it is common that the selection of one feature may enable or disable the selection of other features
- Lecture on **Verification and Design Rule Checking**
- Where we were in the year 2000...

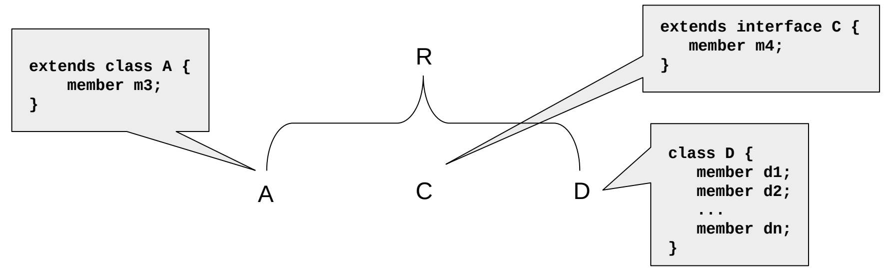
©dsbatory2005

AHEAD:
The Next Generation

Algebraic Hierarchical Equations for Application Design

Feature Encapsulation

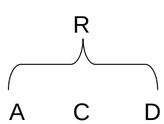
- A feature encapsulates multiple extensions, classes
    - ex: extension R extends class A, interface C, and adds class D



How to Implement?

- Group related files into a directory

Pictorial Representation



Directory Representation

R /
A
C
D

Algebraic Representation

R = { A, C, D }

read as "R encapsulates A, C, and D"

Composition

- Consider constant P and extension R:

$$P = \{ A_P, B_P, C_P \}$$
$$R = \{ A_R, C_R, D_R \}$$

- What is R • P ?

Composition

- Align units by name:

$$
\begin{aligned}
P &= \{\ A_P,\ B_P,\ C_P\ \} \\
R &= \{\ A_R,\ C_R,\ D_R\ \} \\
R \bullet P &= \{\ A_R \bullet A_P,\ B_P,\ C_R \bullet C_P,\ D_R\ \}
\end{aligned}
$$

- Compose units with same name (ignoring subscripts)
- Copy units that aren't extended

- Do the obvious thing...

©dsbatory2005

Law of Composition

$$
\begin{aligned}
R \bullet P &= \{\ A_R,\ C_R,\ D_R\ \} \bullet \{\ A_P,\ B_P,\ C_P\ \} \\
&= \{\ A_R \bullet A_P,\ B_P,\ C_R \bullet C_P,\ D_R\ \}
\end{aligned}
$$

- Fundamental algebraic rewrite of FOP

- Says how composition distributes over encapsulation

- Do you recognize this law?

©dsbatory2005

Inheritance

"class representation"

class P {
  member A_P;
  member B_P;
  member C_P;
}

class R extends P {
  member A_R;
  member C_R;
  member D_R;
}

class R•P extends R {}

"algebraic representation"

$$
\begin{aligned}
P &= \{\ A_P,\ B_P,\ C_P\ \} \\
R &= \{\ A_R,\ C_R,\ D_R\ \} \\
R \bullet P &= \{\ \boxed{A_R \bullet A_P},\ B_P,\ \boxed{C_R \bullet C_P},\ D_R\ \}
\end{aligned}
$$

©dsbatory2005

Composition Corollaries

- f1, f2 are functions
- c1, c2 are constants

f1 ● f2 = f12 – composite function

c1 ● c2 = c1 – c1 overrides c2

c1 ● f1 = c1 – c1 overrides f1

- See examples of these ideas later

©dsbatory2005

Scaling Program Generation

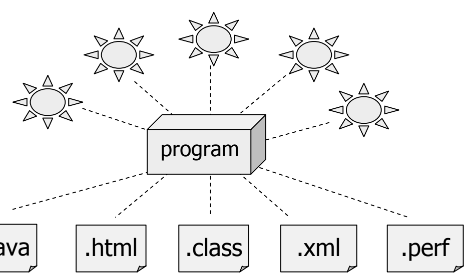
- Generating code for an individual program is OK, but not sufficient
- Today's systems are **not individual programs**, but groups of collaborating programs
  - client-server systems, tool suites (IDEs)
- Further, **systems are not solely defined by code**
  - architects routinely use many knowledge representations
  - formal models, UML models, makefiles, documents, ...

Question

- How does step-wise development scale to the synthesis of multiple programs and multiple-program representations?
- Challenge is not possibility
  - lots of ad hoc ways
  - challenge is to define way that treats all representations – code and non-code – uniformily

Insight #1: Platonic Forms and Languages

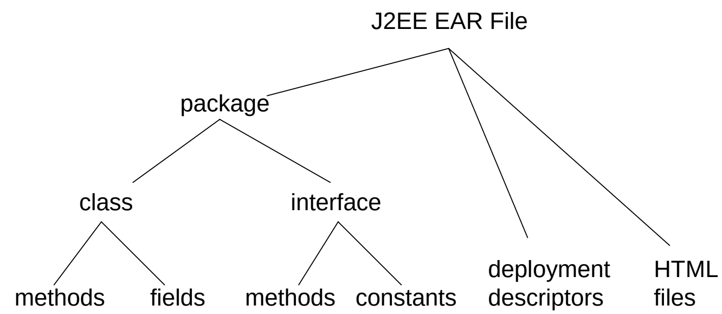
- Each program representation captures different information in different languages



- We want to encapsulate all these representations

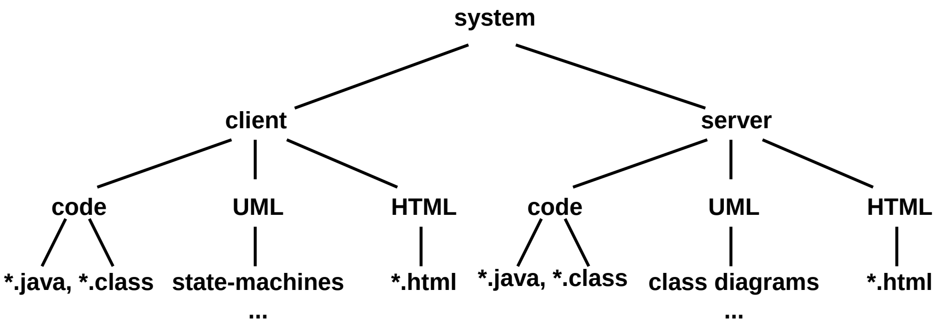
Insight #2: Generalize Modularity

- A **module** is a containment hierarchy of related artifacts



- Generalize module hierarchies to arbitrary depth, contents

Modular Encapsulation of Multiple Programs

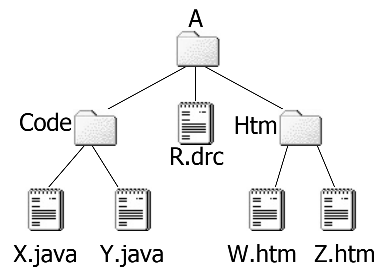


Modules encapsulate all needed representations of a system

Simple Representation

- Module hierarchies = nested sets

directory



algebraic

A = { Code, R.drc, Htm }

Code = { X.java, Y.java }

Htm = { W.htm, Z.htm }

Insight #3: Generalize Features

- When a program is extended, any or all of its representations may be updated
- Ex: Add a new feature F to program P changes:
  - code (to implement F)
  - documentation (to document F)
  - makefiles (to build F)
  - formal properties (to characterize F)
  - performance properties (to profile F)
  - …
- This is a crosscut

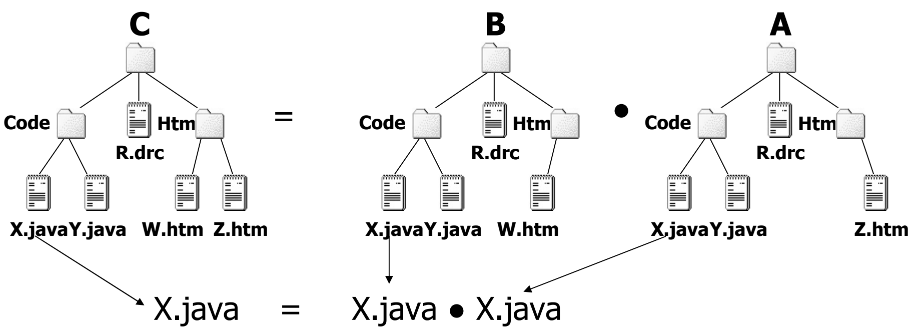
#3: Generalize Features

- Containment hierarchy is a “constant”
- Feature is a “function” that maps (transforms) containment hierarchies

- adds new nodes (e.g., new .java, .html files)
- extends existing nodes

Simple Implementation

- Feature composition = directory composition
  - produces directory isomorphic to inputs



Simple Theory

- Result computed algebraically by **recursively** expanding and applying the law of composition

C = B ● A

= { $Code_B$, $R.drc_B$, Htm_B } ● { $Code_A$, $R.drc_A$, Htm_A }

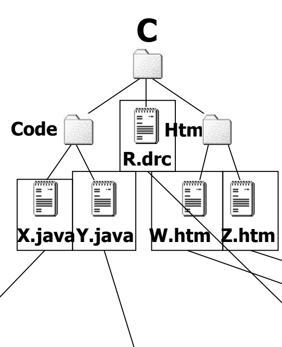
= { $Code_B$ ● $Code_A$, $R.drc_B$ ● $R.drc_A$, Htm_B ● Htm_A }

= { { $X.java_B$, $Y.java_B$ } ● { $X.java_A$, $Y.java_A$ }, $R.drc_B$ ● $R.drc_A$, { $W.htm_B$ } ● { $Z.htm_A$ } }

= { { $X.java_B$ ● $X.java_A$, $Y.java_B$ ● $Y.java_A$ }, $R.drc_B$ ● $R.drc_A$, { $W.htm_B$, $Z.htm_A$ } }

Note!

- Each expression defines an artifact to be produced



C = { { $\boxed{X.java_B ● X.java_A}$, $\boxed{Y.java_B ● Y.java_A}$ }, $\boxed{R.drc_B ● R.drc_A}$, { $\boxed{W.htm_B}$, $\boxed{Z.htm_A}$ } }

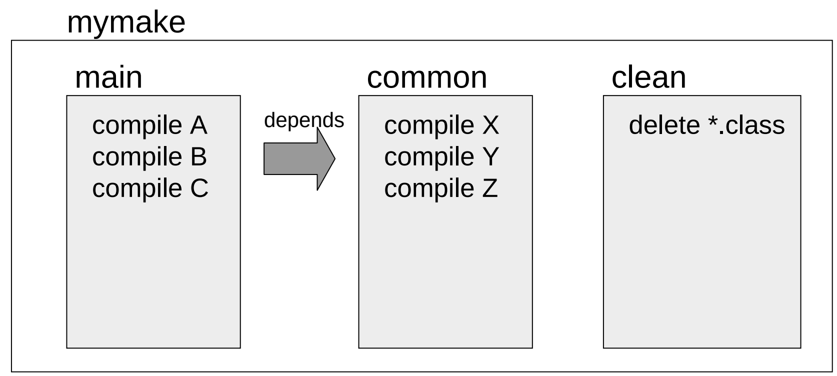
Polymorphism...

- Composition operation ● is **polymorphic**
  - composition law defines how sets are composed
  - different implementation of ● for each representation
    - ● for code
    - another ● for html files, etc.
- But what does extending a non-code artifact mean?
  - what general principle guides extension?

Example: Makefiles

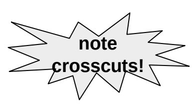
- Instructions to build parts of a system
  - it is a language for synthesizing programs
- When we synthesize code for a system,
  we also have to synthesize a makefile for it
- Sounds good, but...
  - what is a extension of a makefile?????

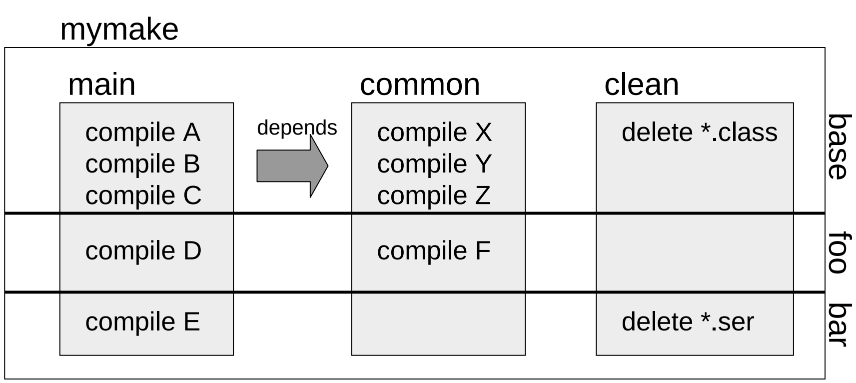
Makefile



command line> make main

Makefile Extensions





Question: what is a general paradigm for extending
non-code artifact types?

Makefiles Have a Class Structure!

```
<project myMake>                          →  class myMake {
   <target main depends="common">         →     void main {
      <compile A>                                { ...
      <compile B>
      <compile C>
   </target>                                    }
   <target common>                        →     void common {
      <compile X>                                ...
      <compile Y>
      <compile Z>
   </target>                                    }
   ...                                          ...
</project>                                   }
```

Makefile Extension is Code Extension

```
<project myMake>
   <target main depends="common">
      <compile A/>
      <compile B/>
      <compile C/>
      <compile D>
   </target>
   <target common>
      <compile X/>
      <compile Y/>
      <compile Z/>
      <compile Q>
   </target>
   ...
</project>
```

new instructions added after existing instructions

correspondence generalizes to makefile properties such as data members, etc.

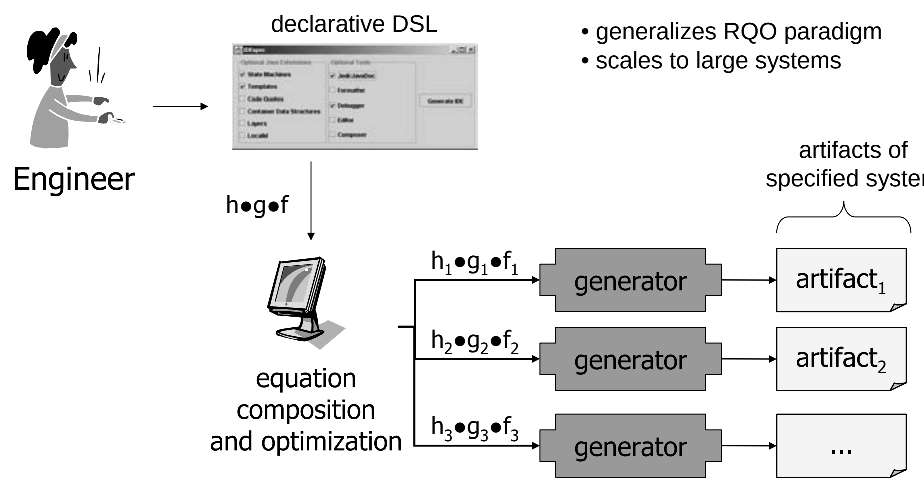
Insight #4: Principle of Uniformity

- **Principle of Uniformity**
  - create analog in OO representation:
    treat all artifacts equally, as objects or classes
  - extend non-code representations same as code representations
- That is, you can extend any artifact
  - understand it as an object, collection of objects, or classes
- We are creating a theory of information structure based on features
  - it works for code and other representations

Big Picture

- Most artifacts today (HTML, XML, etc.) have **or can have** a hierarchical structure
- But there is no extension relationship among artifacts!
  - what's missing are extension operations for artifacts
- Need tools to extend instances of each artifact type
  - MS Word?
  - given such tools, scale step-wise extension scales without bounds...
- Encapsulate changes/additions to **all representations of a system**
  - so all artifacts (code, makefiles, etc.) are updated consistently
- Compositions yield consistent representations of a system
  - exactly what we want
  - simple, elegant theory behind simple implementation

Product Member Synthesis Overview



- generalizes RQO paradigm
- scales to large systems

Recommended Readings

- Batory and O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components". ACM *Transactions on Software Engineering and Methodology*, 1(4):355-398, October 1992.
- Batory, Sarvela, Rauschmayer, "Scaling Step-Wise Extension", *IEEE Transactions on Software Engineering, June 2004.*
- Batory, Johnson, MacDonald, and von Heeder, "Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study", *ACM Transactions on Software Engineering and Methodology*, Vol. 11#2, April 2002, 191-214.
- Batory, Chen, Robertson, and Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE Transactions on Software Engineering*, May 2000, 441-452.
- Batory, Singhal, Thomas, and Sirkin. Scalable Software Libraries. *ACM SIGSOFT 1993*, December 1993.
- Batory, Concepts for a Database System Compiler, *ACM PODS 1988*.
- Baxter, "Design Maintenance Systems", *CACM*, April 1992.
- Czarnecki and Eisenecker, *Generative Programming – Methods, Tools and Applications*, Addison-Wesley 2000.

Recommended Readings

- Czarnecki, Bednasch, Unger, and Eisenecker, "Generative Programming for Embedded Software: An Industrial Experience Report", *Generative Programming and Component Engineering 2002.*
- Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.
- Ernst, "*Higher-Order Hierarchies", ECOOP 2003.*
- Garlan, Allen, and Ockerbloom, "Architectural Mismatch or Why it is hard to build Systems out of existing parts", *ICSE 1995*.
- Flatt, Krishnamurthi, and Felleisen. "Classes and Mixins". *ACM Principles of Programming Languages*, San Diego, California, 1998, 171-183.
- Harrison and Ossher. "Subject-Oriented Programming (A Critique of Pure Objects)", *OOPSLA 1993*, 411-427.
- Kang, et al., "Feature Oriented Domain Analysis Feasibility Study", SEI 1990.
- Kang, et al. "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures", *Annals of Software Engineering 1998*, 143-168.
- Kiczales, et al. "Aspect-Oriented Programming", *ECOOP 97*, 220-242.
- Kiczales, et al. "An Overview of AspectJ". *ECOOP 2001.*

Recommended Readings

- Lieberherr, *Adaptive Object-Oriented Software*, PWS publishing, 1995.
- Mezini and Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development", OOPSLA 1998, 97-116.
- Mezini and Ostermann, "Conquering Aspects with Caesar", AOSD 2003.
- Mezini and Ostermann, "Variability Management with Feature-Oriented Programming and Aspects", SIGSOFT 2004.
- McDirmid, Flatt, and Hsieh, "Jiazzi: new-Age Components for Old-Fashioned Java", OOPSLA 2001.
- Ossher and Tarr. "Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software." CACM October 2001.
- Ossher and Tarr, "Multi-dimensional separation of concerns and the Hyperspace approach." In Software Architectures and Component Technology (M. Aksit, ed.), 293-323, Kluwer, 2002
- Reenskaug, et al., "OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems", Journal of Object-Oriented Programming, 5(6): October 1992, 27-41.
- Simonyi, "The Death of Computer Languages, the Birth of Intentional Programming", NATO Science Committee Conference, 1995.

Recommended Readings

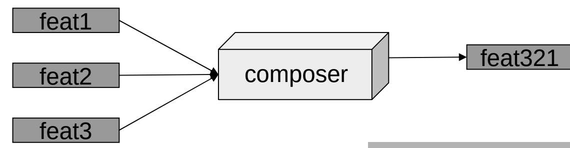
- Smaragdakis and Batory, "Implementing Layered Designs with Mixin Layers". 12th European Conference on Object-Oriented Programming, ECOOP, July 1998.
- Smaragdakis and Batory, "Scoping Constructs for Program Generators". Generative and Component-Based Software Engineering (GCSE), September 1999.
- Smaragdakis and Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Extensions and Collaboration-Based Designs ", *ACM Transactions on Software Engineering and Methodology*, Vol.11#2, April 2002, 215-255.
- Tarr, et al., "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *ICSE 1999*.
- Van Hilst and Notkin, "Using Role Components to Implement Collaboration-Based Designs", *OOPSLA 1996*, 359-369.

AHEAD Tool Suite

kick the tires...

Composer Tool

- Key tool in **AHEAD Tool Suite (ATS)** is **`composer`**
- **`composer`** expands AHEAD equation to yield target system



`feat321 = feat3 • feat2 • feat1`

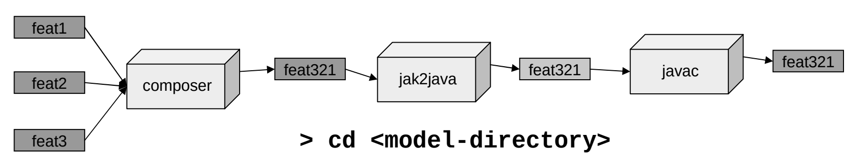
```
> composer –target=feat321 feat1 feat2 feat3
```


Jak Files

- Program in extended-Java files
  - Jak(arta) files
- Java + feature declarations, etc.
  - Jak is an extensible language
- AHEAD is bootstrapped
  - Most AHEAD tools are written in Jak

Other Tools...

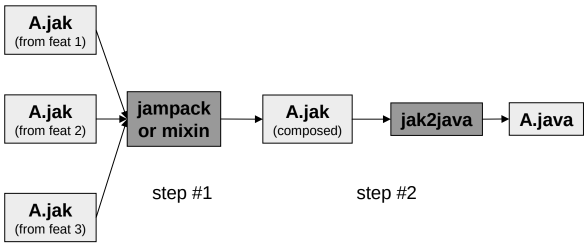
- Besides **`composer`**
  - **`jak2java`** – translates Jak files to Java files
  - **`javac`** – javac compiler
  - **`reform`** – Jak or Java file formatter/pretty-printer
  - others...




```
> cd <model-directory>
> composer –target=...
> reform *.jak
> jak2java *.jak
> reform *.java
> javac *.java
```

Jak-File Composition Tools

- **`composer`** invokes Jak-specific tools to compose Jak files
  - two tools now: **`jampack`** and **`mixin`**
  - **`jak2java`** translates Jak to Java



jampack

- Flattens “inheritance” hierarchies
  - takes expression as input, produces single file as output
  - basically macro expansion with a twist...

```
class top {
   int a;
   void foo() {...}
}
```

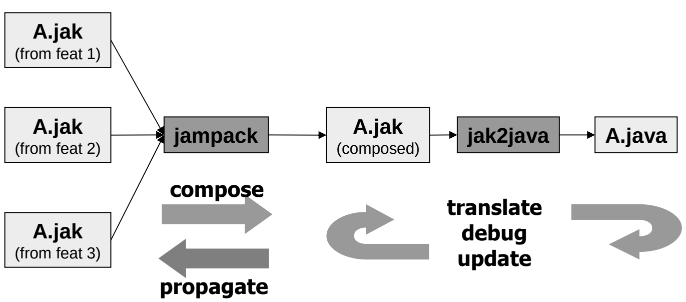
●

```
refines class top {
   int b;
   int bar() {...}
}
```

```
class top {
   int a;
   void foo() {...}
   int b;
   int bar() {...}
}
```

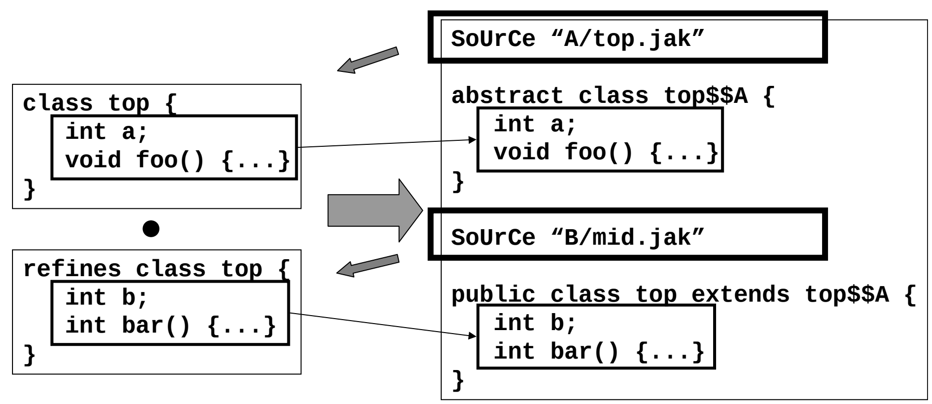

jampack

- **`jampack`** may not be composition tool of choice
  - look at typical debugging cycle
  - problem: manual propagation of changes
  - reason: **`jampack`** doesn’t preserve feature boundaries



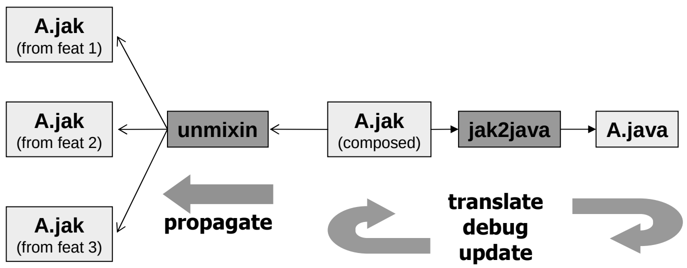
mixin

- Encodes class, extensions as inheritance hierarchy



unmixin

- Edit, debug composed A.jak files
- **`unmixin`** propagates changes from composed file to original feature files automatically



Composable Representations

- Current list...
  - *.jak – extended Java files (Jakarta)
    - class
    - interface
    - state machine (ex: embedded DSL)
  - *. equation – equation files
  - *. b – grammar files
  - *. drc – design rule files
  - others...

AHEAD tools are written in extended Java.

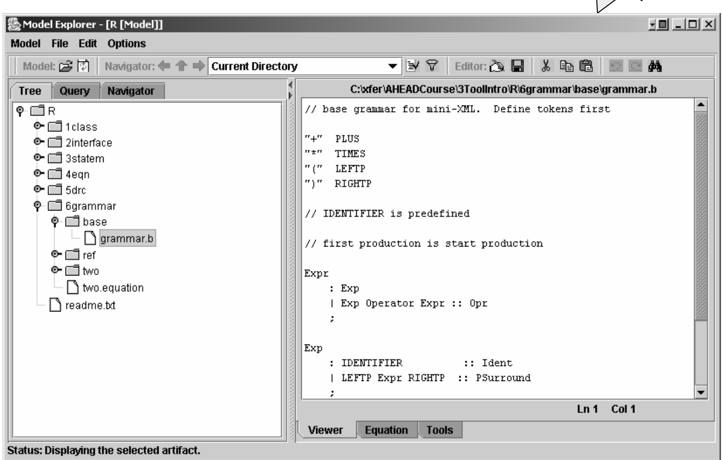
AHEAD has been bootstrapped so that its tools have been written using AHEAD tools.

See Lecture on Origami

Demo...

see files, compositions

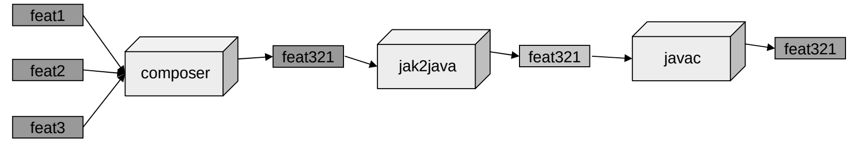
model tree view



file view

Cultural Enrichment

- Note algebraic underpinning...

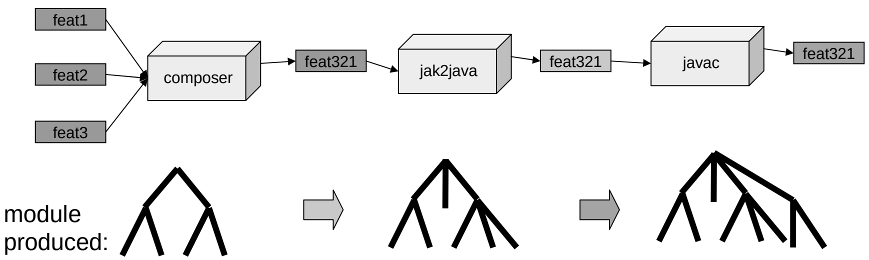


`P = javac( jak2java( feat3 • feat2 • feat1 ))`

- Same algebraic paradigm as AHEAD
  - progressively elaborating a containment hierarchy
  - can optimize expression (not this one...)
  - can generate a makefile from it...

Cultural Enrichment

- To see connection, watch how containment hierarchy is formed...
  - adding new artifacts is example of module extension



- Big picture: lots of operators on AHEAD modules
  - seems that lots of optimizations are possible too... (current work)

A Simple Example

to illustrate concepts, tools

Domain of Graph Applications

- Simple way to express family of related applications is as a grammar
  - different members distinguished by different sets of features

{ undirected | directed } graph { depth-first | breadth-first } search { cycle checking | vertex numbering | connected regions | ... }

choose one — **choose one** — **choose at least one**

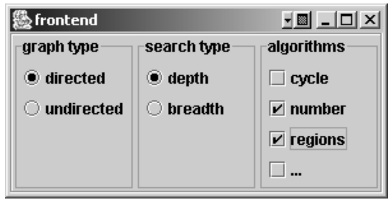
Example Family Members

{ **undirected** | directed } graph { depth-first | **breadth-first** } search { cycle checking | **vertex numbering** | connected regions | ... }

{ undirected | **directed** } graph { **depth-first** | breadth-first } search { **cycle checking** | vertex numbering | **connected regions** | ... }

It is Easy to...

- Imagine a GUI tool that allows you to specify any possible combination
  - declarative language
  - tool generates an explanation of your specification
  - and identifies errors (and suggests corrections) when combinations of features are not possible



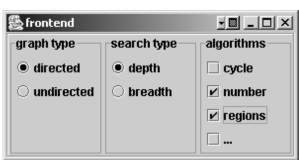
See lecture on
Design Rule Checking

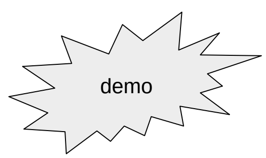
That's Easy...

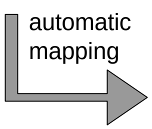
- So too is creating the underlying FOP model:

```
GPL = { directed        -- directed graphs         } constants
        undirected      -- undirected graphs

        bfs             -- breadth first search    } functions
        dfs             -- depth first search

        cycle           -- cycle checking
        number          -- vertex numbering
        regions         -- connected regions
        ...
      }
```


Constructing Applications







graph_app = region ● vertex ● dfs ● directed
= vertex ● region ● dfs ● directed

Further Reading

- Batory, "A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite*", January 2003.*
- Batory, Sarvela, Rauschmayer, "Scaling Step-Wise Refinement", *IEEE Transactions on Software Engineering*, June 2004.
- Batory, Cardone, and Smaragdakis, "Object-Oriented Frameworks and Product-Lines". *1st Software Product-Line Conference*, Denver, Colorado, August 1999.
- Ernst, "*Higher-Order Hierarchies", ECOOP 2003*.
- Holland, "Specifying Reusable Components Using Contracts", *ECOOP 1992*, 287-308.
- Lee, Siek, and Lumsdaine, "The Generic Graph Component Library", *OOPSLA 1999*.
- Lopez-Herrejon and Batory, "A Standard Problem for Evaluating Product-Line Methodologies", *Third International Conference on Generative and Component-Based Software Engineering (GCSE 2001), September 9-13, 2001 Messe Erfurt, Erfurt, Germany.*
- Smaragdakis and Batory, "Implementing Layered Designs with Mixin Layers", *ECOOP 1998*.
- Smaragdakis and Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs", *ACM Transactions on Software Engineering and Methodology*, March 2002.

AHEAD Coding Examples
Class and Class Extension Specifications

base/myclass.jak

```
import initial.stuff;

class myclass {
    int baseVariable;

    // original method is empty
    void baseMethod() {}
}
```

ref/myclass.jak

```
import more.stuff;

refines class myclass  {

    // introduce new variable
    int refVariable = 0;

    // introduce new method
    int refMethod() {
        return refVariable;
    }

    void baseMethod() {
        // extension of baseMethod
        // an "execution" around advice in AOP
        int before_stuff = 1;
        Super().baseMethod();  // AOP "proceed"
        int after_stuff = 2;
    }
}
```

baseRef.equation

```
base
ref
```


JamPack Composition of Classes in baseRef.equation

baseRef/myclass.jak

```
layer baseRef;

import initial.stuff;
import more.stuff;

class myclass {
    int baseVariable;

    // introduce new variable
    int refVariable = 0;

    // original method is empty
    final void baseMethod$$base() {}

    void baseMethod() {
        // extension of baseMethod
        // an "execution" around advice in AOP
        int before_stuff = 1;
        baseMethod$$base(); // AOP "proceed"
        int after_stuff = 2;
    }

    // introduce new method
    int refMethod() {
        return refVariable;
    }
}
```

union of imports

original method

call to original method

Mixin Composition of Classes in baseRef.equation

baseRef/myclass.jak

```
layer baseRef;

import initial.stuff;                    <-- union of imports
import more.stuff;

SoUrCe RooT base "../base/myclass.jak";

abstract class myclass$$base {           <-- base class
    int baseVariable;

    // original method is empty
    void baseMethod() {}
}

SoUrCe  ref "../ref/myclass.jak";

class myclass extends  myclass$$base {   <-- class extension or refinement

    // introduce new variable
    int refVariable = 0;

    // introduce new method
    int refMethod() {
        return refVariable;
    }

    void baseMethod() {
        // extension of baseMethod
        // an "execution" around advice in AOP
        int before_stuff = 1;
        Super().baseMethod(); // AOP "proceed"
        int after_stuff = 2;
    }
}
```


AHEAD Coding Examples
State Machine and State Machine Extension Specifications

base/mysm.jak

```
import something.*;

State_machine mysm {

      Delivery_parameters( Evnt e );

      // start, stop states implicity defineded
      States midpoint;

      Transition begin: start -> midpoint
            condition e != null
            do {
                  commonaction( e );
            }

      Transition end: midpoint -> stop
            condition e != null
            do {
                  commonaction( e );
            }

      void commonaction( Evnt e ) { /* something */
      }
}
```

ref/mysm.jak

```
import evenmore.*;

refines State_machine mysm {

      // add new transition
      Transition loop : midpoint -> midpoint
            condition e == null
            do {}
}
```

baseRef.equation

```
base
ref
```

JamPack Composition of State Machines in baseRef.equation

baseRef/myclass.jak

```
layer baseRef;

import something.*;
import evenmore.*;

State_machine mysm {

      Delivery_parameters( Evnt e );

      // start, stop states implicity defineded
      States midpoint;

      Transition begin: start -> midpoint
            condition e != null
            do {
                  commonaction( e );
            }

      Transition end: midpoint -> stop
            condition e != null
            do {
                  commonaction( e );
            }

      // add new transition
      Transition loop : midpoint -> midpoint
            condition e == null
            do {}

      void commonaction( Evnt e ) { /* ... */
      }
}
```

union of imports

new transition

Mixin Composition of State Machines in baseRef.equation

baseRef/myclass.jak

```
layer baseRef;

import something.*;
import evenmore.*;

SoUrCe RooT base "../base/mysm.jak";

abstract State_machine mysm$$base {

      Delivery_parameters( Evnt e );

      // start, stop states implicitly defined
      States midpoint;

      Transition begin: start -> midpoint
            condition e != null
            do {
                  commonaction( e );
            }

      Transition end: midpoint -> stop
            condition e != null
            do {
                  commonaction( e );
            }

      void commonaction( Evnt e ) { /* ... */
      }
}

SoUrCe  ref "../ref/mysm.jak";

State_machine mysm extends  mysm$$base {

      // add new transition
      Transition loop : midpoint -> midpoint
            condition e == null
            do {}
}
```

union of imports

base class

machine extension or refinement

AHEAD Coding Examples
Design Rules, Design Rule Extensions, and Composition

base/rules.drc

```
constant layer;

// attributes
extern flowleft Int scale;
extern flowright Bool A;

// preconditions
requires flowleft 4 <= scale;

// postconditions
provides flowright !A;
```

ref/rules.drc

```
layer ref;

// attributes
extern flowleft Int scale;
extern flowright Bool B;

// preconditions
requires flowleft scale <= 4;

// postconditions
provides flowright B;
```

composition of above two files →

baseRef/rules.drc

```
constant layer baseRef;

// externally defined attributes

extern flowright Bool A;
extern flowright Bool B;
extern flowleft Int scale;


provides flowright !A and B;
requires flowleft scale == 4;
```


AHEAD Coding Examples
Grammars, Grammar Extensions, and Composition

base/grammar.b

```
// base grammar for mini-calculator
// IDENTIFIER is predefined
// Tokens here

"+"  PLUS

// first production is start production

Expr
      : IDENTIFIER
      | IDENTIFIER Operator Expr :: Opr
      ;

Operator
      : PLUS   :: Plus
      ;
```

ref/grammar.b

```
// adds minus operator
// add new token

"-" MINUS

// import previously defined left-hand side

require Operator;

// add new production

Operator
    : MINUS         :: Minus
    ;
```

composition of above two files →

baseref/grammar.b

```
"-"         MINUS
"+"         PLUS

Expr
      : IDENTIFIER
      | IDENTIFIER Operator Expr :: Opr
      ;

Operator
      : MINUS  :: Minus
      | PLUS   :: Plus
      ;
```

AHEAD Coding Examples
Equations, Equation Extensions, and Composition

base/eq.equation

```
# base equation
# = e . d . c  (listed in left-2-right order)

c
d
e
```

ref/eq.equation

```
# equation extension
# super references base equation

a
b
super
f
g
```

composition of above two files

baseRef/eq.equation

```
# Generated
a
b
c
d
e
f
g
```

Aspect Composition

Current Research...

Introduction

- Core of FOP is:
  - step-wise development (SWD)
  - inheritance-like extension of programs
- AspectJ (AOP in general) seems to provide these capabilities and then some
  - e.g. many more kinds of join-points
- FOP and AOP are duals
  - NOT generalizations of each other
  - they are instances of more general model
  - lecture sketches beginnings of this model

Overview

- Step-wise development with AspectJ is hard
- Illustrate example
- Model of aspect composition using AspectJ
- Present alternative model to support SWD
  - without sacrificing power of AspectJ

An Example

of incremental development

assumes minimal knowledge of AspectJ

Incremental Development Example

- Step 1: Point defines 1-dimensional point

```
class Point1 {
   int x;
   void setX(int v) { x = v; }
}
```


Step 2: Add Y Coordinate and Method

```
class Point1 {
   int x;
   void setX(int v) { x = v; }
}
```

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

```
ajc Point.java TwoD.java
```

```
class Point2 {
   int x;
   void setX(int v) { x = v; }
   int y;
   void setY(int v) { y = v; }
}
```


Step 3: Count # of Coordinate Changes

```
class Point2 {
   int x;
   void setX(int v) { x = v; }
   int y;
   void setY(int v) { y = v; }
}
```

```
aspect Counter {
   int Point.counter = 0;
   after (Point p) : execution( * Point.set*(..))
       && target(p) { p.counter++; }
}
```

```
class Point3 {
   int counter = 0;
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
}
```


Step 4: Add Color Information

```
class Point3 {
   int counter = 0;
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
}
```

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

```
class Point4 {
   int counter = 0;
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
   int color = 0;
   int setColor(int c) { color = c; }
}
```

Surprise!

- AspectJ produces something different!

```
ajc Point.java TwoD.java Counter.java Color.java
```

```
class Point'4 {
   int counter = 0;
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
   int color;
   int setColor(int c) { color = c; counter++; }
}
```

Extra code! Counter aspect applies to all files in ALL steps!

Paradox of Using Aspects

- Building software incrementally:
  - manually
  - automatically using AspectJ
  - may yield different results!
- Redefine Counter could avoid this problem:

```
aspect Counter {
   int Point.counter = 0;
   after (Point p) : execution( * Point.setX(..) )
       && execution( * Point.setY(..) )
       && target(p) { p.counter++; }
}
```


Well...

- It would solve *this* problem, but *not* others
- Ex: if we used the updated Counter, but wanted to build program below, we couldn't do it
  - would need to update Counter again

```
class Point {
   int counter = 0;
   int x;
   void setX(int v) { x = v; counter++; }
   int y;
   void setY(int v) { y = v; counter++; }
   int color;
   int setColor(int c) { color = c; counter++; }
}
```

This code would be missing

The Big Picture

- Premise of **Component-Based Software Engineering (CBSE)** is step-wise development
  - progressively build programs by composing components one at a time
  - reuse components "as is"
- We want to reuse aspect modules "as is"
  - difficult to do
- Core problem:
  - aspect composition does not distinguish development stages

How We Will Proceed

- Create a model of how AspectJ composes aspects to discover source of problem
- Present an alternative model of composition that:
  - retains power of AspectJ
  - support incremental development
  - simplifies reasoning with aspects
- Full treatment in:
  - "Taming Aspect Composition: A Functional Approach" by R. Lopez-Herrejon and D. Batory, May 2005

A Model of Introduction

Introduction Addition (+)

Model of Introduction

- Introduction is a function that maps an input program to an augmented output program

$$\mathbf{Point_2 = TwoD(\ Point_1\ )}$$

- Appealing to intuition, rewrite above as summation:

$$\mathbf{Point_2 = TwoD\ +\ Point_1}$$

Introduction Addition

Introduction Addition

- **Program fragment** is set of methods, variables of 1+ classes
- \+ adds program fragments

A + P1 = P2

{ aspect I1 { A.a; B.b; } , class C { .. } } + { class A { .. } , class B { .. } } = { class A { .. a .. } , class B { .. b .. } , class C { .. } }

Properties of Introduction Addition

- + is set union of program fragments
- **Identity** – denoted by 0
  - 0 is the empty program fragment
  - if X is a program fragment

$$X = X + 0 = 0 + X$$

- **Commutative** – order in which program fragments are added does not matter
- **Associative**: (A + B) + C = A + (B + C)

Properties of Introduction Addition

- **Substitution** (from associativity)
  - TwoD is a composite Introduction

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

→ `TwoD = y + setY`

- can substitute to produce equivalent defn of $Point_2$

$$\mathbf{Point_2} = \mathbf{TwoD} + \mathbf{Point_1}$$
$$= \mathbf{y} + \mathbf{setY} + \mathbf{Point_1}$$

A Model of Advice

Advice Weaving (*)

Advice

```
aspect Log {
   pointcut logP() : execution(* Point.set*(..));
   after() : logP()
      { System.out.println("set called"); }
}
```

- Advice code (in *italics* above) can be regarded as implicit method declaration and call
- Separate concerns by
  - make advice body an explicit method
  - name each advice

Pure Advice – Rewrite Log Aspect

```
aspect Log {
   static void Point.setCalled()
      { System.out.println("set called"); }

   LogP is after(): execution(* Point.set*(..))
      --> Point.setCalled();
}
```

introduction

pure advice

- Not standard AspectJ syntax
- Called **Pure Advice** – separates implicit introduction from advice

Model of Aspects

```
aspect Log {
   static void Point.setCalled()
      { System.out.println("set called");

   LogP is after(): execution(* Point.set*(..))
      --> Point.setCalled();
}
```

- Model as 2-D vector
  - 1st entry is pure advice (advice part)
  - 2nd entry is introduction (introduction part)

`Log = [ LogP, setCalled ]`

Another Example

```
aspect Counter {
   int Point.counter = 0;
   static void Point.IncCtr(Point p)
      { p.counter++; }

   CounterP is after (Point p) :
       execution( * Point.set(..)) && target(p)
       --> Point.CounterA(p);
}
```

- Modeled by vector:

`Counter = [ CounterP, counter + IncCtr ]`

Advice Weaving

- Application of pure advice is operation *

Advice Weaving

- Let **a** pure advice and **P** be a program
- **a*P** = program resulting from advice **a** woven into **P**

Advice Weaving

- **a2** and **a1** are pure advice
- **a2*a1*P** means apply **a1** first to **P**, then **a2**
- Defines precedence ordering of advice

Properties of Advice Weaving

- **Identity** – denoted by 1
  - 1 is the null advice – a pointcut that captures no joinpoints
  - if **P** is a program and **a** is a pure advice:

```
P   = 1*P
a*P = 1*a*P = a*1*P
```

- **Non-commutative** – order in which weaving occurs matters
  - commutative only when join point sets are disjoint

Properties of Advice Weaving

- **Right-Associative**:
  - **a2*a1*P** means apply **a1** first to **P**, then apply **a2**
- **Distributive**: Advice weaving distributes over introduction addition

```
P’  = a*P
    = a*(A + B + C)
    = a*A + m*B + m*C
```


Aspect Composition: Vector Model

- **Composition**:
  - aspect A1 = [a1, i1]
  - aspect A2 = [a2, i2]
  - ◊ is AspectJ composition operation
- ◊ akin to vector addition**:**

```
A2 ◊ A1   = [a2, i2] ◊ [a1, i1]
          = [a2*a1, i2+i1]
```

Aspect Composition

- Let program P = [1, p]

```
A2 ◊ A1 ◊ P    = [a2, i2] ◊ [a1, i1] ◊ [1, p]

               = [ a2*a1*1, i2+i1+p ]

               = [ a2*a1, i2+i1+p ]
```

- What is the resulting program?

Aspect Composition

- Is "**length"** of vector V

```
|V| = |[a,i]| = a*i
```

- So: `|A2 ◊ A1 ◊ P| = a2*a1*( i2+i1+p )`

$$|A_n \lozenge A_{n-1} \lozenge \ldots A_1 \lozenge P| = (a_n * a_{n-1} * \ldots * a_1) * (i_n + i_{n-1} + \ldots + i_1 + p)$$

- Consistent with observable AspectJ semantics

Incremental Development & AspectJ

- Problem seen in expansion

```
|A2 ◊ A1 ◊ P|  =  a2*a1*( i2+i1+p )
               =  a2*a1*i2 + a2*a1*i1
                    + a2*a1*p
```

- *AspectJ programmer needs to know if any advice applied in earlier steps affects the code added by the current or later steps*

$$a_{j-1} * a_{j-2} * \ldots * a_1 * i_j$$

A Simple Fix...

A Functional Model of Composition

- Treat aspects as functions
- Aspect composition is function composition

$$\boxed{A(P) = A \bullet P = a*(i + p)}$$

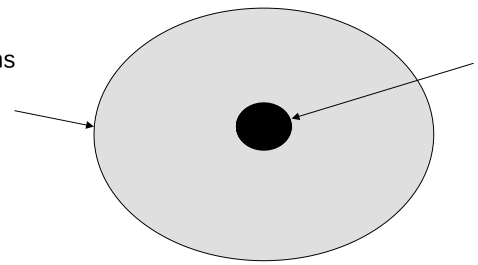
$$A_2 \bullet A_1 \bullet P \quad = a_2*(i_2 + a_1*(i_1 + p))$$
$$= a_2*i_2 + a_2*a_1*i_1 + a_2*a_1*p$$

- The terms we don't want (a_1*i_2) are gone!

©dsbatory2005

Comparison of Composition Models

- Functional Model has more power than AspectJ
  - **provided that aspects are reused as is**



©dsbatory2005

Proof

- Every AspectJ composition can be expressed as a Functional composition

```
|A2 ◊ A1 ◊ P|  =  a2*a1*( i2+i1+p )
```

```
[a2,0] • [a1,0] • [1,i2] • [1,i1] • [1, p]
         = a2*a1*( i2+i1+p )
```

©dsbatory2005

Proof Continued

- Translating arbitrary Functional Model expression into AspectJ composition is not possible by reusing aspects "as is"
  - can do it if you modify the aspects...

$$A_2 \bullet A_1 \bullet P \quad = a_2*(i_2 + a_1*(i_1 + p))$$
$$= a_2*i_2 + a_2*a_1*i_1 + a_2*a_1*p$$

- Reason: Vector Model does not distinguish different development stages

©dsbatory2005

Implication – Recall Point Example

- Can add 3rd dimension to Point, **`ThreeD`**
- Can build 3 different programs
  - program that counts executions of setX and setY

    `Color ● ThreeD ● Counter ● TwoD ● Point`
  - program that counts execution of setX, setY, setZ

    `Color ● Counter ● ThreeD ● TwoD ● Point`
  - program that counts all set methods

    `Counter ● Color ● ThreeD ● TwoD ● Point`

Using AspectJ we would need 3 different versions of Counter

©dsbatory2005

Features are Increments in Program Functionality

- Aspects are features, and vice versa
  - they are the same

KEY

- It's their composition-design models that differ!

KEY

- FOP is based on **step-wise development**
  - distinguish different stages of program development
- AOP (AspectJ) uses a different methodology
  - does not distinguish different stages of development
- FOP and AOP are not directly comparable
  - but they are instances of a more general model
  - preserves power of AspectJ
  - preserves power of step-wise development

©dsbatory2005

Current Work

Integrated Model

special cases

FOP AOP

- Working with **abc** group (Oxford, England) and University of Passau
  - to integrate models
- Stay tuned...

©dsbatory2005

Recommended Readings

- Aspect Bench Compiler. http://www.aspectbench.org
- Aspect Development Tools. http://www.eclipse.org/ajdt
- AspectJ. Programming Guide. http://aspectj.org/ doc/proguide
- Concern Manipulation Environment (CME) http://www.eclipse.org/cme/
- R.E. Filman, T. Elrad, S. Clarke, M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004
- Gregor Kiczales and Mira Mezini. "Aspect-Oriented Programming and Modular Reasoning". *ICSE 2005*.
- R. Lopez-Herrejon and D. Batory, "Improving Incremental Development in AspectJ using Bounded Quantification", *SPLAT 2005*.
- R. Lopez-Herrejon and D. Batory, "Taming Aspect Composition: A Functional Approach", May 2005
- R. Lopez-Herrejon, D. Batory, and W. Cook, "Evaluating Support for Features in Advanced Modularization Technologies", *ECOOP 2005*.
- G. Murphy, A. Lai, R.J. Walker, M.P. Robillard, "Separating Features in Source Code: An Exploratory Study". *ICSE 2001*.
- H. Rajan and K. Sullivan, "Classpects: Unifying Aspect- and Object-Oriented Language Design", ICSE 2005.

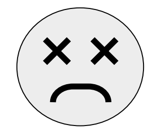
©dsbatory2005

Design Rule Checking

how to verify compositions automatically

Introduction

- Fundamental problem: not all compositions of features are correct
  - but code can still be generated!
  - and maybe code will still compile!
  - and maybe code will run for a while!
  - impossible for users to figure out what went wrong!



Introduction

- Must verify compositions automatically
  - not all features are compatible
  - selection of a feature may enable others, disable others
- **Design rules** are domain-specific constraints that identify illegal compositions
- **Design Rule Checking (DRC)** is process of applying design rules automatically
- Presentation overview:
  - review fundamental relationships of models, grammars, feature diagrams, and propositional formulas
  - tool support

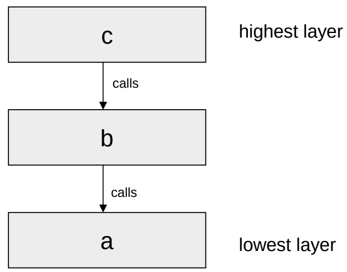
AHEAD Models and Grammars

AHEAD Model $\xleftrightarrow{?}$ Attribute Grammar

Layered Designs 1992

- GenVoca originated from layered designs
- Layers are common form of program extensions

k = c ● b ● a



Typing GenVoca Layers

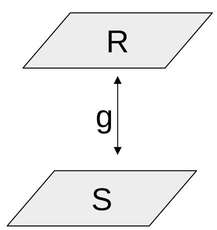
- Layers exported and imported standardized interfaces
  - interfaces == **virtual machines (VM)**
  - "legos"
- Virtual Machines used as types
  - suppose S and R are virtual machines

M = { y:S , z:S , w:S ,

g(x:S):R , h(x:S):R , i(x:R):R }

Types and Realms

- g(x:S):R means feature g:
  - exports virtual machine R
  - imports layer x that implements virtual machine S
  - x is a parameter of "type" S



g is a layer that maps between VMs R and S

- **Realm** is a set of units that implement the same virtual machine

S = { y, z, w }

R = { g(x:S), h(x:S), i(x:R) }

Product-Lines and Grammars

- Model = ∪ set of realms
- Defines a grammar whose sentences are applications

```
S = { y, z, w }                    S ::= y   | z   |  w   ;
R = { g(x:S), h(x:S), i(x:R) }  R ::= g S | h S |  i R ;
```

set of all sentences is a **language** or **product-line**

Symmetry

- Just as recursion is fundamental to grammars; symmetric layers are fundamental to GenVoca
  - export and import same virtual machine
  - composable in virtually arbitrary orders
  - composition order affects semantics, performance
- Symmetric layer of realm **W** has parameter of type **W**

```
W = {  m(x:W),  n(x:W), p  }
ex:  m(n(p)),  n(m(p)),  m(m(p)),  n(n(p)),...
```


A Symmetric Layer...

- Augments or enriches existing abstractions
  - relational DBMS – add transposition, data cube ops
  - relational interface still the same, except it has been enriched
    - think of extending a class with a subclass
    - same idea, except on a **system** level
  - enormous number of such features....
- Happens in **ALL** domains...

Example

- What are the standard operations of a container?
  - call this layer “base”
- All other operations are “optional”
  - encapsulate in separate layer that extends interface of base
  - these layers are “symmetric”
  - map container abstraction to augmented container abstraction

Perspective...

- Assign types to constants, functions...
  - so that all our equations are “typed”
  - catches type errors!

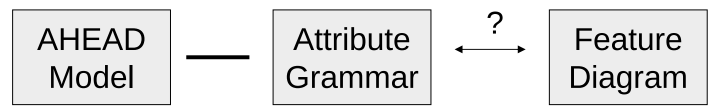
```
S = {  y,  z,  w  }

R = { g(x:S), h(x:S), i(x:R) }
```

- Syntax checking in this grammar guarantees type correctness of expressions
  - is this enough?

No!!

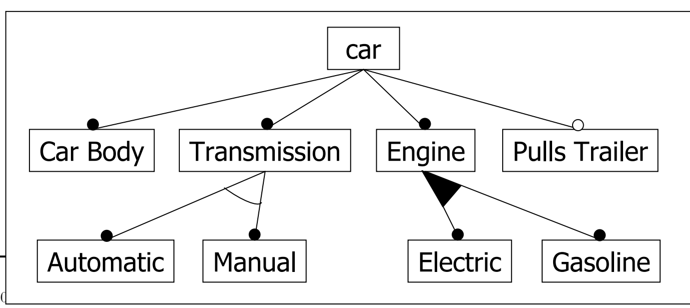
- Syntax checking is not enough!
  - matching input/output signatures insufficient!
  - just because your Java program is syntactically correct doesn't mean that it is semantically correct
- DRC uses same techniques used by compilers!
  - **use attribute grammars to define constraints**
  - AHEAD model is an **grammar**
  - design rules are **grammar attributes, predicates**

Feature Diagrams and Grammars



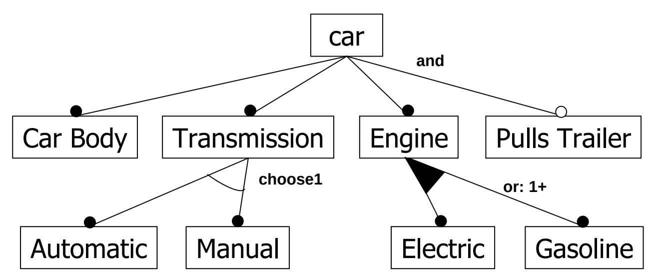
Feature Diagrams

- **Feature diagrams** are standard product-line notations
  - declarative way to specify products by selecting features
- FDs are trees:
  - leaves are primitive features
  - internal nodes are compound features
  - parent-child are containment relationships



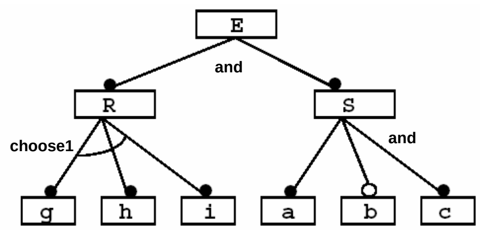
Feature Diagrams

- Mandatory – features that are required
- Optional – features that are optional
- And – all subfeatures (children) are selected
- Alternative – only 1 subfeature can be selected
- Or – 1+ or 0+ subfeatures can be selected



Example

- What is a legal product specification?
  - E is ?
  - R is ?
  - S is ?

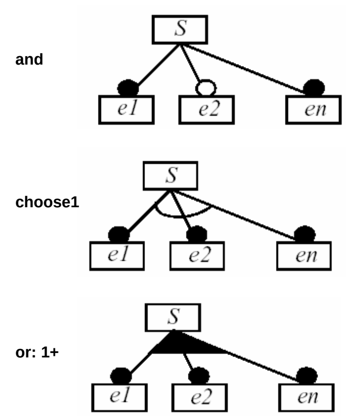


- Sound familiar?
  - de Jonge and Visser (2002):
  - FDs are graphical representations of grammars
  - “GenVoca Grammars” 1992

Mapping of FDs to Grammars

Diagram

Grammar



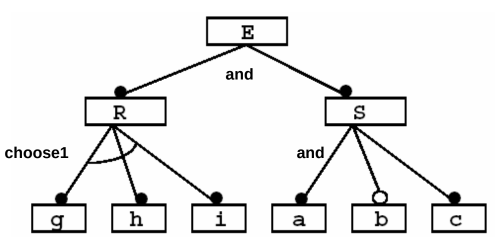
S ::= e1 [e2] en ;

S ::= e1 | e2 | en ;

… S+ …

S ::= e1 | e2 | en ;

Example: Convert FD to Grammar



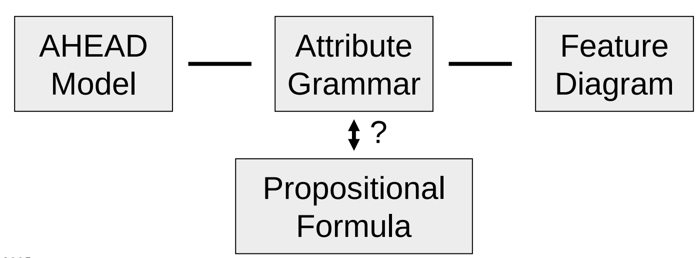
E ::= R S ;

R ::= g | h | i ;

S ::= a [b] c ;

- Application defined by FD = sentence of grammar E
- Adding attributes allows further constraints to be expressed
- Again back to attribute grammar foundation

Grammars and Propositional Formulas



Propositional Formula

- Set of boolean variables and propositional logic predicate that constrains values of these variables
- Standard $\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow$ operations
- Nonstandard:
  - $choose_1(e_1 \ldots e_k)$ – exactly one e_i is true
  - $choose_{n:m}(e_1 \ldots e_k)$ – at least n, at most m
  - anything else...

Insight

- A grammar is a compact representation of a propositional formula
- Variable is:
  - a token
  - name of a non-terminal
  - name of a pattern
- How many variables in the production below?

```
R :  a  b     :: P1
  |  c [R1]   :: P2
  ;
```


Mapping Productions to Formulas

- Given production R : P1 | ... | Pn ;
- R can be referenced in two ways:

Pattern	Predicate
... R+ ... (choose 1 or more)	$P1 \vee P2 \vee \ldots \vee Pn$
... R ... (choose 1)	choose1(P1,P2, ..., Pn)

Mapping Patterns to Formulas

- T1 T2 ... Tn :: P

  formula: $P \Leftrightarrow T1$ ^ $P \Leftrightarrow T2$ ^ ... ^ $P \Leftrightarrow Tn$

- T1 [T2] ... Tn :: Q

  formula: $Q \Leftrightarrow T1$ ^ $T2 \Rightarrow Q$ ^ ... ^ $Q \Leftrightarrow Tn$

Example: Grammars to Formulas

- Convert each production, pattern to formula
- Take conjunction of all formulas
- Conjoin root=true (root is root of grammar)

```
E ::= R S ;

R ::= g | h | i ;

S ::= a [ b ] c ;
```

grammar

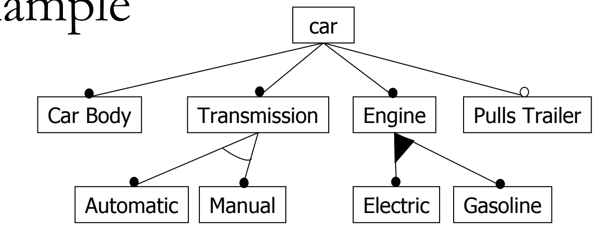
$$E \Leftrightarrow R \wedge E \Leftrightarrow S \wedge$$
$$R \Leftrightarrow \text{choose1}(g, h, i) \wedge$$
$$S \Leftrightarrow a \wedge b \Rightarrow S \wedge S \Leftrightarrow c \wedge$$
$$E = \text{true}$$

propositional formula

A sentence of E satisfies the propositional formula
and vice versa

©dsbatory2005

Another Example



$Car \Leftrightarrow CB \wedge Car \Leftrightarrow Tr \wedge Car \Leftrightarrow Eng \wedge Pt \Rightarrow Car \wedge$

$Tr \Leftrightarrow \text{choose1}(Auto,Man) \wedge$

$Eng \Leftrightarrow (Ele \vee Gas) \wedge$

$Car = \text{true}$

©dsbatory2005

Summarizing...

- We can map any AHEAD model or FD to a propositional formula
  - a sentence of grammar = assignment to variables that satisfy the formula
  - but what about constraints?
- **Any** additional, **arbitrary** propositional formulas conjoined onto grammar formula
  - Ex: if features i and b are incompatible, we would conjoin the formula

$$i \vee b \Rightarrow \neg (b \wedge i)$$

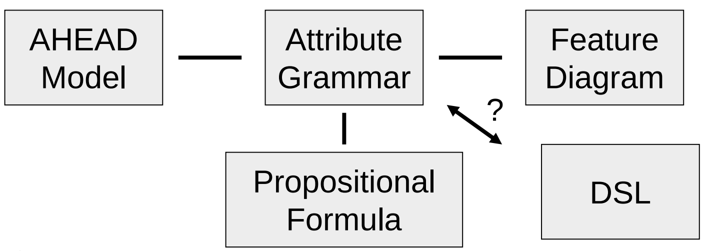
©dsbatory2005

In Summary

- An AHEAD Model is a propositional formula!
  - primitive features are variables
  - compound features are variables
  - arbitrary set of propositional constraints supported
  - can be mapped to attribute grammars
- Grammar:
  - specifies ordering constraints on features
  - ordering very important for AHEAD
- Additional propositional constraints:
  - weed out incompatible features

©dsbatory2005

Declarative Domain-Specific Languages



Declarative Languages

- Features enable declarative program specifications
  - that’s what feature diagrams are for!
  - counterpart of SQL
- Want a declarative GUI DSL that acts like a syntax-directed editor
  - user selects desired features
  - tool precludes specifying incorrect programs
- **`guidsl`** tool...

An Example

- Recall GPL from Tools Lecture

```
Gpl = {  DIRECTED      -- directed graphs       } constants
         UNDIRECTED    -- undirected graphs

         BFS           -- breadth first search  ⎫
         DFS           -- depth first search    ⎪
                                                ⎬ functions
         CYCLE         -- cycle checking        ⎪
         NUMBER        -- vertex numbering      ⎪
         REGIONS       -- connected regions     ⎭
         ...
      }
```


GPL Grammar

```
Gpl : Alg+ [Src] Wgt Gtp :: MainGpl ;

Gtp : DIRECTED | UNDIRECTED ;

Wgt : WEIGHTED | UNWEIGHTED ;

Src : DFS | BFS ;

Alg : NUMBER | CONNECTED |
    | [TRANSPOSE] STRONGC :: StronglyC
    | CYCLE | MSTPRIM | MSTKRUSKAL | SHORTEST ;
```

Additional Constraints

- Straight from Graph Algorithm Text

Algorithm	Required Graph Type	Required Weight	Required Search
Vertex Numbering	Any	Any	BFS, DFS
Connected Components	UNDIRECTED	Any	BFS, DFS
Strongly Connected Components	DIRECTED	Any	DFS
Cycle Checking	Any	Any	DFS
Minimum Spanning Tree	UNDIRECTED	WEIGHTED	None
Single-Source Shortest Path	DIRECTED	WEIGHTED	None

Encode as Additional Predicates

```
NUMBER implies Gtp and Src;
CONNECTED implies UNDIRECTED and Src;
STRONGC implies DIRECTED and DFS;
CYCLE implies Gtp and DFS;
MSTKRUSKAL or MSTPRIM implies
      UNDIRECTED and WEIGHTED;
SHORTEST implies DIRECTED and WEIGHTED;
MSTKRUSKAL or MSTPRIM implies
      not( MSTKRUSKAL and MSTPRIM );
```


guidsl Specification

```
Gpl : Alg+ [Src] Wgt Gtp :: MainGpl ;
Gtp : DIRECTED | UNDIRECTED ;
Wgt : WEIGHTED | UNWEIGHTED ;
Src : DFS | BFS ;
Alg : NUMBER | CONNECTED
    | [TRANSPOSE] STRONGC :: StronglyC
    | CYCLE | MSTPRIM | MSTKRUSKAL
    | SHORTEST ;
%%
NUMBER implies Gtp and Src;
CONNECTED implies UNDIRECTED and Src;
STRONGC implies DIRECTED and DFS;
CYCLE implies Gtp and DFS;
MSTKRUSKAL or MSTPRIM implies
       UNDIRECTED and WEIGHTED;
SHORTEST implies DIRECTED and WEIGHTED;
MSTKRUSKAL or MSTPRIM implies
       not( MSTKRUSKAL and MSTPRIM );
```

grammar

constraints

Demo

- Propagation of constraints involves
  - classic results from Artificial Intelligence
  - Logic Truth Maintenance System
  - improves quality of teaching material
- Help to debug model using SAT solver
  - **Satisfiability (SAT) Solver** tries to find assignment to boolean values to make propositional formula true

Generated DSL for GPL Spec

Demo!

Key Papers

- Batory, “Feature Models, Grammars, and Propositional Formulas”, SPLC 2005
- Benavides, et al. “Automated Reasoning on Feature Models”, CAISE 2005
- Generalize predicates to include numerical constraints
  - count number of products that satisfy constraints
  - select product that maximizes/minimizes criteria (performance)
  - restrict models based on feature requirements, criteria
  - standard constraint solvers
- Next-generation FD tools based on these ideas

Experience with DRC Tools

- Have worked well...
- Predicates are simple
- Use off the shelf constraint solvers
- Reason: architects think in terms of features
  - if predicates were really complicated
    - architects couldn’t design
    - people couldn’t program
    - because it would be too difficult
- We are making explicit what is implicit now...

Recommended Readings

- Batory and O'Malley. “The Design and Implementation of Hierarchical Software Systems with Reusable Components”. *ACM TOSEM*, October 1992.
- Batory and Geraci. “Composition Validation and Subjectivity in GenVoca Generators”, *IEEE Transactions on Software Engineering* (special issue on Software Reuse), February 1997, 67-82.
- D. Benavides, P. Trinidad, and A. Ruiz-Cortes, “Automated Reasoning on Feature Models”, *Conference on Advanced Information Systems Engineering (CAISE)*, July 2005.
- Beuche, Papajewski, and Schroeoder-Preikschat, “Variability Management with Feature Models”, *Science of Computer Programming*, Volume 53, Issue 3, Pages 333-352, December 2004.
- Czarnecki and Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000
- Czarnecki, Helson, Eisenecker, “Staged Confiruation Using Feature Models”, Software Product-Line Conference 2004.
- K.D. Forbus and J. de Kleer, *Building Problem Solvers*, MIT Press 1993.
- M. de Jong and J. Visser, “Grammars as Feature Diagrams”, 2002. `http://www.cwi.nl/events/2002/GP2002/papers/dejonge.pdf`
- S. Neema, J. Sztipanovits, and G. Karsai, “Constraint-Based Design Space Exploration and Model Synthesis”, *EMSOFT 2003*, LNCS 2855, p. 290-305.
- Perry, “The Logic of Propagation in the Inscape Environment”, ACM SIGSOFT 1989.

Multi-Dimensional Models

Synthesis of Tool Suites

©dsbatory2005

Multi-Dimensional Models (MDMs)

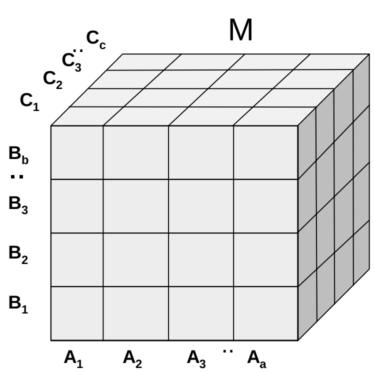
- Are a fundamental design technique in FOP
- Given model $F = \{ F_1, F_2, \ldots F_n \}$
- Let program $G = F_8 + F_4 + F_2 + F_1$
  - where + denotes composition operator •
  - we'll see shortly why the change in notation is useful
- Can write G as:

$$G = \Sigma_{i \in (8,4,2,1)} F_i$$

©dsbatory2005

N-Dimensional Models

- Use *n* FOP models called **dimension models**
  - to specify features or indices along a dimension
- A 3-D model M with A, B, C as dimension models
  - $A = \{A_1, \ldots A_a\}$
  - $B = \{B_1, \ldots B_b\}$
  - $C = \{C_1, \ldots C_c\}$
- M has a*b*c entries
  - M_{ijk} implements (A_i, B_j, C_k)



©dsbatory2005

N-Dimensional Models

- A program is now specified by *n* equations
  - 1 per dimension
- Program P in product-line of M has 3 equations:

$$P = A_6 + A_3 + A_1 \qquad = \Sigma_{i \in (6,3,1)} A_i$$

$$P = B_7 + B_4 + B_3 + B_2 \qquad = \Sigma_{j \in (7,4,3,2)} B_j$$

$$P = C_9 + C_1 \qquad = \Sigma_{k \in (9,1)} C_k$$

©dsbatory2005

Summing (Aggregating) Dimensions

- The 3-eqn specification of P is translated into an M equation by summing M along each dimension

$$P = \sum_{i \in (6,3,1)} \sum_{j \in (7,4,3,2)} \sum_{k \in (9,1)} M_{i,j,k}$$

A indices B indices C indices

- Order in which dimensions are summed does not matter
  - commutativity property of MDMs
  - provided that dimensions are orthogonal

Significance of MDMs: Scalability!

- Complexity of program is # of features
- Given *n* dimensions with *d* feature per dimension
  - program complexity is $O(d^n)$
  - using MDM model O(dn)
  - ex: program P specified by 3*4*2 features of M or only 3 + 4 + 2 dimensional features!
- **FOP program specifications are exponentially shorter when using MDMs**

Academic Legacy

- “Extensibility Problem” or “Expression Problem” (EP)
  - classical problem in Programming Languages
  - see papers by: Cook, Reynolds, Wadler, Torgensen
- Multi-Dimensional Separation of Concerns (MDSoC)
  - Tarr, Ossher IBM
- MDM is an algebraic formulation of MDSoC and EP
  - first present a micro example (15 line programs)
  - then a large example (30K line programs)
    - synthesis of the AHEAD Tool Suite

A Micro Example

- Model L defines a set of programs that implement an elementary linked list

```
L = {  sglIns,      // bare-bones singly-linked list with
                    // insert operation
       addDel,      // adds deletion operation to sglIns
       dblIns,      // extends sglIns to doubly-linked list
       dblDel       // extends addDel to deletion on
                    // doubly-linked list
    }
```

Enumerated Product-Line

- Set of all legal equations (designs) for L

slist w. ins
- `sglIns`

dlist w. ins
- `dblIns + sglIns`

slist w. ins & del
- `addDel + sglIns`

dlist w. ins & del
- `dblDel + dblIns + addDel + sglIns =`

  `dblDel + addDel + dblIns + sglIns`

Why are last two expressions equal?

Ans: orthogonal

Incorrect Compositions

- **`dblIns + addDel + sglIns`**
  - insert method works on a doubly-linked list
  - delete method works on a singly-linked list
- **`dblDel + addDel + sglIns`**
  - insert method works on singly-linked list
  - delete method works on a doubly-linked list

resulting programs have design errors, are inconsistent

Common Problem in FOP

- If list structure is extended (single–to–double)
  - all operations must be consistently updated
  - ex: both insert and delete must work on same structure
- Equivalently, if a new method is added, then it should work for that structure and not some other structure
  - insert can't work on singly-linked list, delete on doubly-linked list
- **Consistent Refinement Problem**
- Representative of a large class of problems in FOP
  - models define features that are not truly independent
  - features must be applied in groups lock-step (all-or-nothing)
  - when this occurs, recognize groups implement "higher-level" features
- MDMs abstract this complexity….

Orthogonal Dimensional Models

- Create operation model **`Ops`**

linked list with insert operation

adds delete operation to list

`Ops = { insert, delete }`

- Model says *nothing* about list structure
  - could be single-linked, double-linked, keyed, non-keyed…
  - only 2 legal equations

```
w_ins         = insert
w_ins_and_del = delete + insert
```

Orthogonal Dimensional Models

- Create structure model **`Struct`**

singly-linked list

extend to doubly-linked list

```
Struct = { singleLink, doubleLink }
```

- Model says *nothing* about list operations
  - could have insert, deletion, update, ….
  - only 2 legal equations

```
single = singleLink
double = doubleLink + singleLink
```


Given These Two Models

- A list program is completely defined by 2 equations
  - P = doubly-linked list with ins and del operations

P = delete + insert // equation #1 uses Ops Model

P = doubleLink + singleLink // equation #2 uses Struct Model

- These equations must be equal
  - because they represent the **same** program
  - how to show their equivalence?

MDMs

- Define relationship between Ops & Struct models by a matrix
- Rows represent units of the Ops model (insert, delete)
- Columns are units of the Struct model (singleLink, doubleLink)
- Entries are features of L

	doubleLink	singleLink
insert	dblIns	sglIns
delete	dblDel	addDel

MDM Matrix for L

Sum (Aggregate) MDM Matrix by Rows

- Ops equation P = delete + insert
- Sum corresponding entries in each column

	doubleLink	singleLink
delete + insert	dblDel + dblIns	addDel + sglIns

extends to doubly-linked

single-linked w. ins and del

Now Sum by Columns

- Struct equation P = doubleLink + singleLink
- Sum corresponding entries in each column
  - yields 1x1 matrix whose contents is first of the two equations that defines P (doubly-linked list structure with insert and delete methods)

	doubleLink + singleLink
delete + insert	dblDel + dblIns + addDel + sglIns

Again, But Sum Columns First

- Struct equation P = doubleLink + singleLink
- Means sum corresponding entries in each column

	doubleLink + singleLink
insert	dblIns + sglIns
delete	dblDel + addDel

doubly-linked w. ins operation

extends by adding del operation

Now Sum Rows

- Ops equation P = delete + insert
- Sum corresponding entries in each column
  - yields second of the two equations that defines doubly-linked list structure with insert and delete methods

	doubleLink + singleLink
delete + insert	dblDel + addDel+ dblIns + sglIns

Perspective

- By abstracting model L as a pair of orthogonal dimensional models and specifying a program as a pair of equations, we generate only the legal equations of L

slist w. ins	sgl
dlist w. ins	dbl + sgl
slist w. ins & del	sgldel + sgl
dlist w. ins & del	dbldel + dbl + sgldel + sgl = dbldel + sgldel + dbl + sgl

A Macro Example

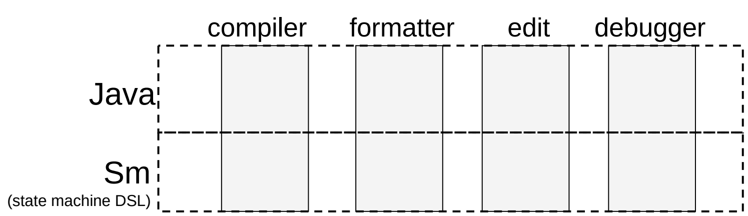
Synthesizing the AHEAD Tool Suite

Perspective

- So far, our models customize **individual programs**
  - set of all such programs is a **product-line**
- **Tool Suite** is an integrated set of programs, each with different capabilities
  - MS Office (Excel, Word, Access, ...)
- Question: Do features scale to tool suites?
  - product-line of tool suites
  - Ans: Yes!

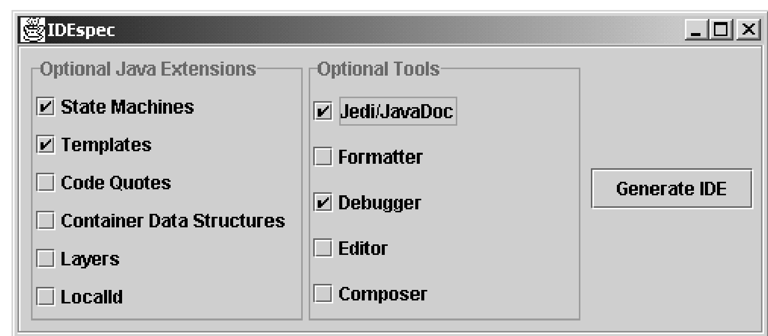
IDEs: A Tool Suite

- Integrated Development Environment (IDE)
  - suite of tools to write, debug, document programs
  - AHEAD variant: Java language extensibility



In principle, features scale!!!

The Problem – Declarative IDE



From this declarative DSL specification, how do we generate AHEAD tools?

Define Dimensional Model #1

- AHEAD Model of Java Language Dialects

$$\texttt{J = \{ } \underbrace{\texttt{Java,}}_{\text{constant}} \; \underbrace{\texttt{Sm, Tmpl, Ds, ...}}_{\text{functions (optional features)}} \texttt{ \}}$$

- Dialects of Java specified by equation

```
Jak =  Tmpl + Sm + Java    // java + state mach
                           // + templates
…
```


Define Orthogonal Model #2

- Tools can be specified by a different, orthogonal model

$$\texttt{IDE = \{ } \underbrace{\texttt{Parse,}}_{\text{constant}} \; \underbrace{\texttt{ToJava, Harvest, Doclet, ...}}_{\text{functions (optional features)}} \texttt{ \}}$$

- Different tools have different equations

```
jak2java =  ToJava + Parse

jedi     =  Doclet + Harvest + Parse
...
```


Tool Specification

- Defined by a pair of equations
  - one equation defines the tool in terms of its language features
  - other equation defines the tool in terms of its tool features
  - ex: **`jedi`** (i.e., **`javadoc`**) for the Jak dialect of Java

```
jedi = Tmpl + Sm + Java              // using J Model

jedi = Doclet + Harvest + Parse      // using IDE Model
```

- Synthesize **`jedi`** from these specs by defining and summing matrix that relates the J and IDE models

MDM Matrix for `jedi`

- Rows are language features
- Columns are tool features
- Entries are modules that implement a language feature for a tool feature
- Shows relationship between IDE and J models

	Doclet	Harvest	Parse
Java	JDoclet	JHarvest	JParse
Sm	SDoclet	SHarvest	SParse
Tmpl	TDoclet	THarvest	TParse

MDM Matrix for jedi

MDM Matrix

- Composition of these modules yields **`jedi`**
- Synthesize **`jedi`** equation by summing matrix according to its dimensional equations

	Doclet	Harvest	Parse
Java	JDoclet	JHarvest	JParse
Sm	SDoclet	SHarvest	SParse
Tmpl	TDoclet	THarvest	TParse

MDM Matrix for jedi

©dsbatory2005

Sum the Matrix!

- IDE equation **`jedi`** = **`Doclet`** + **`Harvest`** + **`Parse`**
- Tells us the column summation order

	Doclet		Harvest		Parse
Java	JDoclet	+	JHarvest	+	JParse
Sm	SDoclet	+	SHarvest	+	SParse
Tmpl	TDoclet	+	THarvest	+	TParse

©dsbatory2005

Sum Rows

- J equation **`jedi`** = **`Tmpl`** + **`Sm`** + **`Java`**
- Tells us the row summation order

	Doclet		Harvest		Parse
Java	(JDoclet	+	JHarvest	+	JParse)
			+		
Sm	(SDoclet	+	SHarvest	+	SParse)
			+		
Tmpl	(TDoclet	+	THarvest	+	TParse)

An Equation for jedi

©dsbatory2005

Application Produced by Aggregation

- Result:

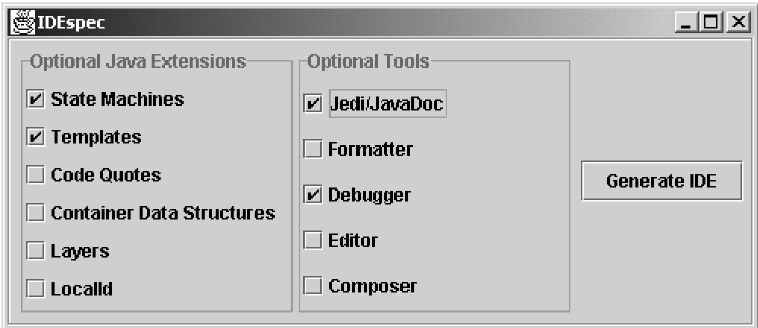
```
jedi = ( TDoclet + THarvest + TParse ) +
       ( SDoclet + SHarvest + SParse ) +
       ( JDoclet + JHarvest + JParse )
```

Using MDM we can synthesize an equation for a language-dialect specific tool

©dsbatory2005

Using MDMs to Generate

- Tool Suites...



MDM Matrix

- That relates J and IDE models
- Rows are language features
- Columns are tool features

	Parse	ToJava	Harvest	Doclet	Signat
Java	JParse	J2Java	JHarvest	JDoclet	JSig
Sm	SParse	S2Java	SHarvest	SDoclet	SSig
Tmpl	TParse	T2Java	THarvest	TDoclet	TSig
Ds	DParse	D2Java	DHarvest	DDoclet	DSig

To Synthesize IDE Tools

- Remove unneeded rows and columns
  - directly from IDE GUI
  - example: jedi, jak2java for Java + Sm + Tmpl

	Parse	ToJava	Harvest	Doclet
Java	JParse	J2Java	JHarvest	JDoclet
Sm	SParse	S2Java	SHarvest	SDoclet
Tmpl	TParse	T2Java	THarvest	TDoclet

MDM Matrix for IDE Tools

- Sum rows
- Note the semantics of the result...

	Parse	**ToJava**	**Harvest**	**Doclet**
Java **Sm** **Tmpl**	JParse + SParse + TParse	J2Java + S2Java + T2Java	JHarvest + SHarvest + THarvest	JDoclet + SDoclet + TDoclet

Yields Equation For Each Tool Feature!

Parse = TParse + SParse + JParse

ToJava = T2Java + S2Java + J2Java

Harvest = THarvest + SHarvest + JHarvest

...

	Parse	ToJava	Harvest	Doclet
Java	JParse +	J2Java +	JHarvest +	JDoclet +
Sm	SParse +	S2Java +	SHarvest +	SDoclet +
Tmpl	TParse	T2Java	THarvest	TDoclet

Resulting Row

- Is AHEAD model for IDE product-line!

```
IDE = { Parse, ToJava, Harvest, Doclet, ... }

        Parse   = ...
        ToJava  = ...
        Harvest = ...
```

- And we know equations for each tool!

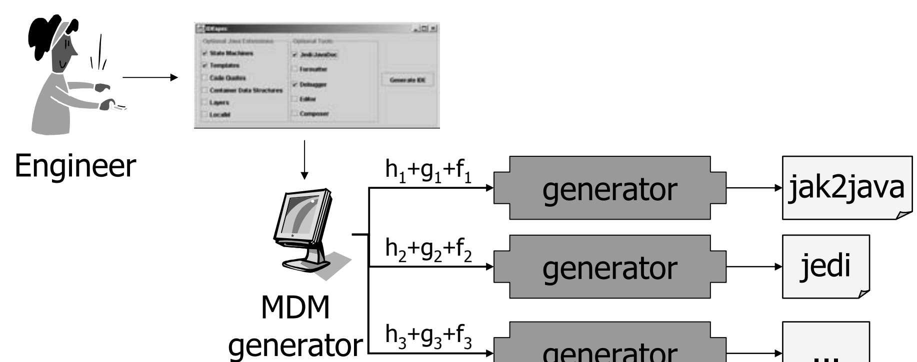
```
jak2java  = ToJava + Parse

jedi      = Doclet + Harvest + Parse
...
```


IDE Generator is Simple

- For each selected tool, evaluate its eqn

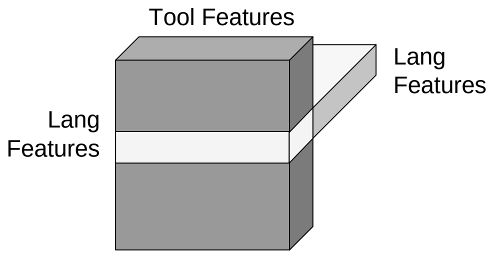
And generate the code for each tool automatically!

Generator of IDE Tool Suite



Bootstrapping AHEAD

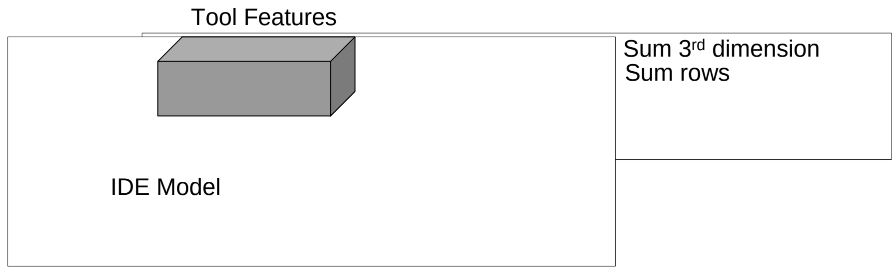
- We used 3-Dimensional (8x6x8) MDM Matrix to generate 5 tools of the AHEAD Tool Suite



3rd dimension captures language feature interactions

Bootstrapping AHEAD

- Sum matrix to produce IDE model, from which we can generate tool equations



Results of AHEAD Bootstrap

- 90 distinct features
- Typical tool contains 20-30 features
  - most tools share 10 features
- Generated Java for each tool is ~34K LOC
- Generating well close to 200K from simple, AHEAD declarative specifications
  - exactly what we want
- Making designs for multiple tools to conform to a matrix
  - controlling the complexity of tool suites

Relationship to AOP

- Allows you to add “advice” to existing programs
  - ex: before, after methods
  - ex: advising method calls

$$\text{Eqn} = C_{after2} + C_{after1} + C + B_{after2} + B_{after1} + B + A_{after2} + A_{after1} + A$$

- Summing rows of MDM matrix looks identical!

jedi = (TDoclet + THarvest + TParse) +
(SDoclet + SHarvest + SParse) +
(JDoclet + JHarvest + JParse)

MDM Advising Architectural Specs!

- Representing program designs as expressions is enormously powerful
  - ideal for generators
- **Algebraic representations scale!!**
  - micro example ~150 LOC, AHEAD example ~150K LOC
  - **3 orders of magnitude**
  - **ideas of MDM apply to all levels of abstraction equally**
  - **algebraic representations scale to *much* larger systems**

Final Words

- As researchers in AOP, MDSoC scale their ideas to tool suites...
- They'll encounter MDM...

Recommended Reading

- Batory, Lopez-Herrejon, Martin, "Generating Product-Lines of Product Families", Automated Software Engineering 2002. Updated version submitted for journal publication.
- Batory, Liu, Sarvela, "Refinements and Multi-Dimensional Separation of Concerns", ACM Sigsoft 2003.
- Cook, W.R. "Object-Oriented Programming versus Abstract Data Types". Workshop on Foundations of Object-Oriented Languages, Lecture Notes in Computer Science, Vol. 173. Spring-Verlag, (1990) 151-178
- W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", *OOPSLA 1993*, 411-427.
- Ossher and Tarr, "Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software." CACM 44(10): 43-50, *October 2001*.
- Reynolds, J.C. "User-defined types and procedural data as complementary approaches to data abstraction". Reprinted in C.A. Gunter and J.C.Mitchell, Theoretical Aspects of Object-Oriented Programming, MIT Press,1994.
- Tarr, Ossher, Harrison, and Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *ICSE 1999*.
- Torgensen, M., "The Expresion Problem Revisited. "Four new solutions using generics", ECOOP 2004.
- Wadler, P. "The expression problem". Posted on the Java Genericity mailing list (1998)

Recap

Summary of Tutorial...

©dsbatory2005

FOP and Product-Lines

- Design individual program → think classes
- Design product-line (program family) → think features
  - members are distinguished by their features
- FOP is study of feature modularity
  - raises features to first-class, quantum increments of design
  - features implemented by "cross-cuts"
  - close to OO framework designs
  - aspects are complimentary
- AHEAD is example of FOP
  - step-wise development
  - builds complex systems by adding features incrementally

©dsbatory2005

Bigger Picture of Software Engineering

- Future of Software Engineering is in automation
- Most successful example of automated software engineering is relational query optimization
  - declarative specification → efficient program
  - relational algebra
  - program (of family of equivalent programs) is expression
- AHEAD product-line models are generalizations
  - declarative feature specifications → program
  - domain models are algebras
  - program is an expression (equation)
  - code and non-code artifacts treated uniformly
  - synthesize consistent representations of all program artifacts
  - equational representations scale, simple, practical

©dsbatory2005

Other Results...

- Domain-Specific Equation Optimization
  - IEEE Transactions on Software Engineering May 2000 (IEEE TSE)
- Domain-Independent Equation Optimization
  - 2004 Generative Programming and Component Engineering (GPCE)
- Feature Interactions and Software Derivatives
  - 2005 International Conference on Feature Interactions (ICFI)
- Generative Programming Design Methodologies
  - to appear
- Byte Code Composition
  - to appear

©dsbatory2005

Thank you!

Questions?

For more information, papers, and AHEAD tools, visit our web site:

http://www.cs.utexas.edu/users/schwartz/

©dsbatory2005