Feature Oriented Programming for Product-Lines

Don Batory
Department of Computer Sciences
University of Texas at Austin

Introduction

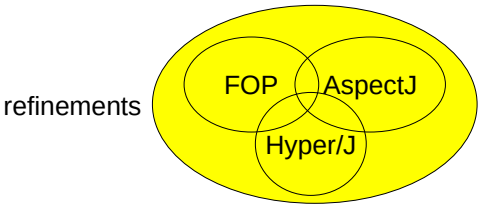
- A **product-line** is a family of similar systems
  - very common
  - Chrysler mini-vans, Motorola radios, and software
- Motivation: economics
  - amortize cost of building variants of program
  - design for family
- Key idea of product-lines
  - members of product-line are characterized by features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

Introduction

- **Feature Oriented Programming (FOP)** is the study of feature modularity in product-lines
  - features are first-class entities in design
  - implemented by "cross-cuts"
- Lots of success
  - 1986 database systems
  - 1989 network protocols
  - 1993 data structures
  - 1994 avionics
  - 1997 extensible Java compilers
  - 1998 radio ergonomics
  - 2000 program verification tools
  - 2003 AHEAD tool suite

FOP Cross-Cuts

- Utilize small fraction of power of AspectJ
  - successful with simple techniques
  - other ways, has more power



Conjecture: all part of an encompassing paradigm based on refinements

- Close to Hyper/J in **Multi-Dimensional Separation of Concerns (MDSOC)**
  - differences too

Tutorial Overview

- Part I
  - The FOP Paradigm
  - The Theory
  - Intro to AHEAD Tools
- Part II
  - Relationship to AspectJ and HyperJ
  - Design Rule Checking
  - Origami and Product-Families

The FOP Paradigm

feature-orientation is general paradigm for program, product-line synthesis

Motivation

- Software Engineering is in a perpetual crisis; 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...

  progressively increasing abstractions
  - **today's design techniques too low-level, exposing far too much detail to make application's design, construction and modification simple**
- Crisis is statement that SE technologies are failing
  - future design techniques will be different, but upwards-compatible way
  - 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 before crisis will noticeably diminish

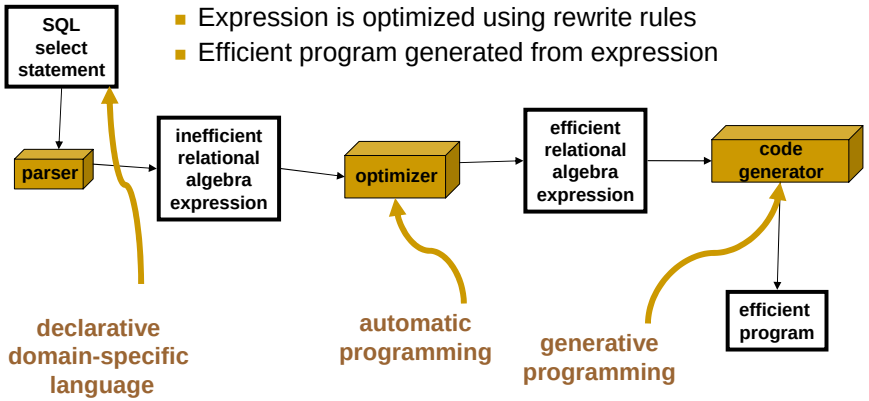
Not Wishful Thinking...

- Example of this futuristic paradigm realized over 20 years ago
  - around time that people 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 a **science** to specify and optimize query evaluation programs
- Identified fundamental operators of this domain
  - relational algebra
- Represented programs as **expressions (equations)**
  - compositions of relational operators
- Define algebraic identities among operators 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 have been developed over last 20 years that generalize basic notions of **modularity** and lay groundwork for practical **compositional programming** and a **mathematical science** for software design

A Basis for a Science for 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

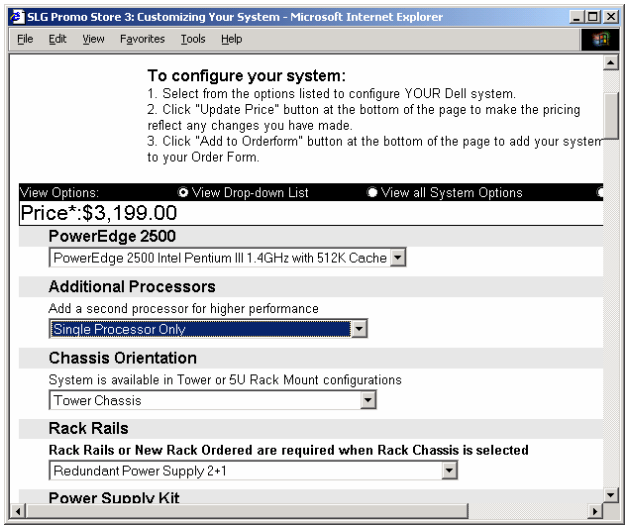
  Program1 = feature_X + feature_Y + feature_Z

  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 or constraints on features for desired system



A-V Recording Technologies

- 1940's recording technologies…
  - expensive, "gotta-get-it-right-the-1st-time", hard to change
- Today, audio recordings made in sound studios
  - "mixin" different (but simple) sound tracks to create rich, sophisticated recordings
  - sound tracks are **features (a.k.a. "layers")**
- Same for video images (e.g., Titanic)
- Layer or feature composition simplifies construction of complex artifacts from simple artifacts, controls cost, and improves product

Methodology for Construction

- What methodology builds systems by adding features one at a time?
- **Step-Wise Refinement**
  - Dijkstra, Wirth early 1970s
  - abandoned in early 1980s as it didn't scale...
  - had to compose hundreds or thousands of refinements to produce admittedly small programs
  - recent work shows how SWR scales
    – scale individual refinements to a **feature refinement**
    – so that composing a few of them yields an entire system

What is a Feature Refinement?

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

Tutorial on Feature Refinements

Refinements are Interchangeable

Refinements are Interchangeable

Refinements are Interchangeable

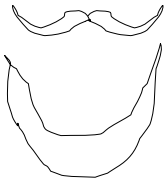
Refinements are Interchangeable

Refinements are Reusable

Refinements are Functions

PersonPhoto beanie(PersonPhoto x)

PersonPhoto uncleSam(PersonPhoto x)



PersonPhoto mustache(PersonPhoto x)

PersonPhoto lincolnBeard(PersonPhoto x)

Refinement Compositions

- Refinement composition = function composition

= lincolnBeard(uncleSam(

))

Large Scale Refinements

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

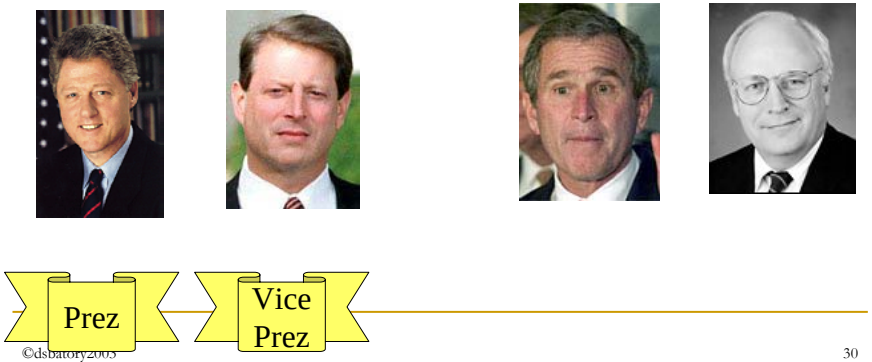
....

Composing Refinements

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

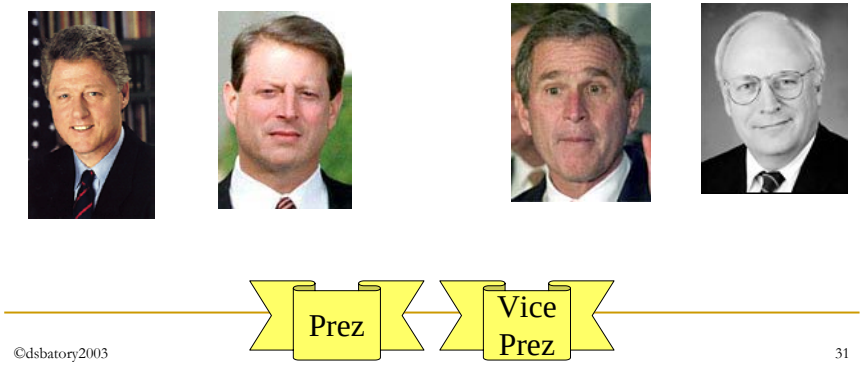
Composing Refinements

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



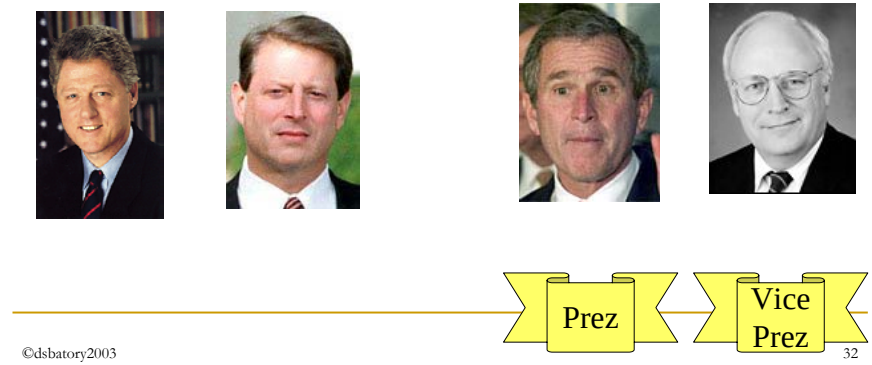
Composing Refinements

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



Composing Refinements

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



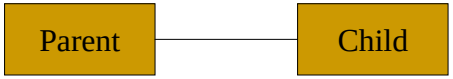
Composing Refinements

Example of dynamic composition of collaborations

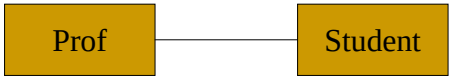


Other Collaborations

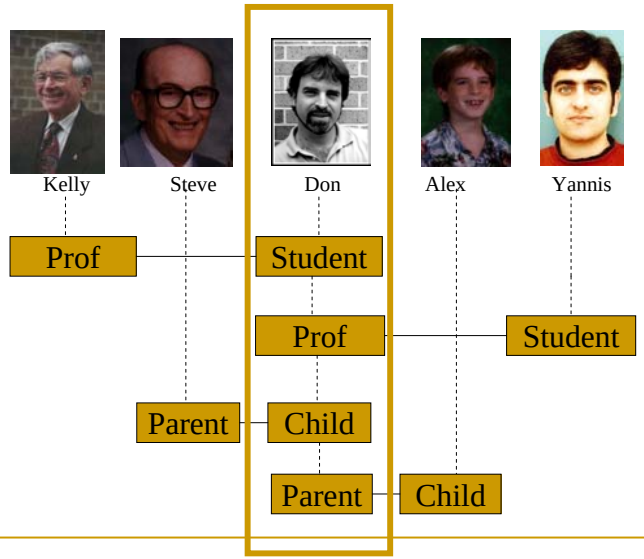
- Parent-Child collaboration



- Professor-Student collaboration



Example



Same Holds for Software!

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

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 refinement
  - advocates complex programs constructed from simple programs by incrementally adding features
- But what are feature refinements and how are their compositions modeled?

The Theory

Models of FOP based on
Step-Wise Refinement

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…

- You can distribute the contents of C 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 {}
```

Another Example...

- C23 could be rewritten or 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 {}
```


Observe…

- 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 refinement

Look Familiar?? Remember Algebra?

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

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

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


Look Familiar?? Remember Algebra?

- Consider sets and union operator (∪)
  - 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 )
```

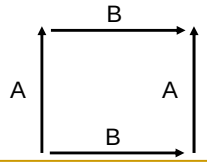
A Closer Analogy

- Vector vector **join** →
- Vector join lays vectors end-to-end to define a path
- 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

Operator We Want...

- Is not quite inheritance...
  - we want to add new methods, new fields, and override or extend existing methods as inheritance does
  - also want constructors to be "inherited" and extended as well, and inheritance doesn't provide this

```
class C1 {
   constructor#1
}
```

●

```
class C2 {
   constructor#2
}
```

=

```
class C12 {
   constructor#1
   constructor#2
}
```

The operator ● that we want is called **class refinement**

Syntax of Class Refinement

- Suppose program P has single class B

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

- A refinement R adds y,z()

```
refines 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; }
```

- Refinements are **functions**

```
// function R

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

Algebraic Formulation

- Base programs are **constants**

```
// constant P

class B { int x; }
```

- Refinements are **functions**

```
// function R

refines 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(){...}
}
```


Another Example

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

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

- Composition is an expression or equation

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

    = C4 • C3 • C2 • C1
```

Note: we use both notations for function composition interchangeably

Refinement of Members

- Data members aren't refined
  - actually they can be: replace type with subtype

    "OutputStream x;" replaced by "FileOutputStream x;"
  - existing tools don't support this concept, but could...
- Methods are refinable (just like using inheritance)
  - calling "super" method to achieve effects of:
    - before-method
    - after-method
    - around-method

Method Refinement ala Inheritance

```
result =   method_refinement    •   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 refinement are not limited to single class
  - **large-scale refinements** encapsulate refinements of multiple classes **as well as adding new classes**
    - such a refinement would augment existing classes but also add classes that could be subsequently refined
    - ability to add new classes that can be refined is **critical**

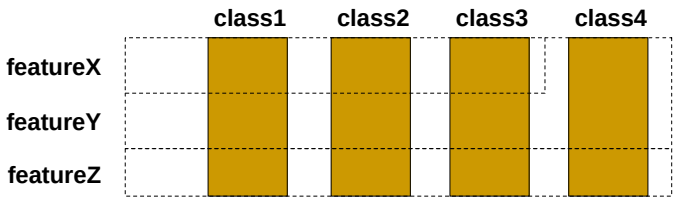
Connecting the Dots...

- Refinements have meaning when they encapsulate the implementation of a feature
  - ever add a new feature to an existing OO program?
  - discover that you have to extend several classes as well as adding new classes
    - "cross-cuts"
  - a **feature refinement** is a large-scale refinement

Program Synthesis Goal

Note: each feature cross-cuts multiple classes

Program P = **featureZ • featureY • featureX**



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

Contributors to this view…

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

Connecting the Dots...

- You can always decompose software in this manner
  - trick is that your refinements 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
  - 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 (Refinements)
  - 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 (refinements)

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 through function composition

Equation Optimization

- Constants, functions represent both feature and its implementation
  - different functions with different implementation 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 equation optimization to determine the best implementation of k
  - counterpart of relational optimization
  - more complicated rewrites....
- Lecture on **Design Wizards (Day Tutorial)**

Generalization of Relational Algebra

- Keys to success of Relational Optimizers
  - equational representations of programs
  - equational rewrites using algebraic identities
- **Here's the generalization**:
  - domain model is an **algebra** for a domain or product-line
    - is set of operators (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 operators
    - these identities can be used to optimize equational representations of programs, just like relational optimizers

Functions vs. Operators

- We equate terms "function" and "operator"
  - mathematicians would object: ideas are not same
  - **operator** is a higher-order function (i.e., a function that maps functions)
    - ex: calculus integration, differentiation operators
- Are refinements functions or operators?
  - Ans: operators
  - programs have command-line inputs, and thus are functions
  - refinements are program-to-program mappings (function-to-function mappings)
  - hence they're operators
- Again, domain models are algebras (sets of operators)

Composition Constraints

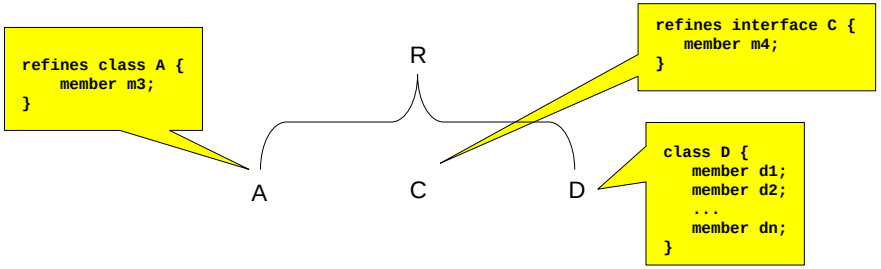
- Although GenVoca constants, functions seem untyped, constraints exist
- **Design Rules** are domain-specific semantic constraints that govern legal compositions
  - ex: it is common that the selection of one feature may enable or disable the selection of other features
- Lecture on **Design Rule Checking (Second Part of Tutorial)**
- Lecture on **Typing Refinements (Day Tutorial)**

AHEAD

- **Algebraic Hierarchical Equations for Application Design (AHEAD)**
  - embodies fundamental generalizations of GenVoca
  - theoretical basis for current tool suite
- Generalizations:
  - law of composition
  - scalability
  - domain-specific languages
  - generalize modularity
  - generalize refinements

Encapsulation of Class Refinements

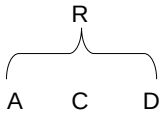
- A feature refinement encapsulates multiple class refinements
  - ex: refinement R extends class A, interface C, and adds class D



How to Implement / Encapsulate?

- A group of related files?
  - Ans: as a directory
  - conceptually as a set or **collective**

Pictorial Representation



Directory Representation

R /
A
C
D

Algebraic Representation

R = { A, C, D }

terminology:
collective is a set of units

What Does Composition Mean?

- Consider constant P and refinement R:

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

- We want to compute R ● P
- Here's how:
  - "spread" P and R out so that units with same names (ignoring subscripts) align

What Does Composition Mean?

- After "alignment" we have:

$$P = \{ \quad A_P, \; B_P, \quad C_P \quad \}$$
$$R = \{ \quad A_R, \quad C_R, \; D_P \; \}$$
$$R \bullet P = \{ \; A_R \bullet A_P, \; B_P, \; C_R \bullet C_P, \; D_P \; \}$$

- Compose units with same name (ignoring subscripts)
- Copy units that aren't refined
- Do obvious thing...

Law of Composition

$$R \bullet P = \{ A_P, C_R, D_P \} \bullet \{ A_P, B_P, C_P \}$$

$$= \{ A_P \bullet A_R, B_P, C_P \bullet C_R, D_P \}$$

- Fundamental algebraic rewrite of FOP
- Do you recognize this law?

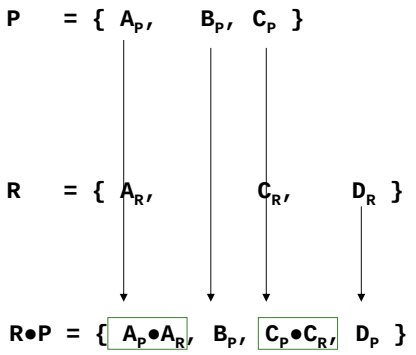
Inheritance...!

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"



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

- We will see examples of these ideas later....

Scaling Program Generation

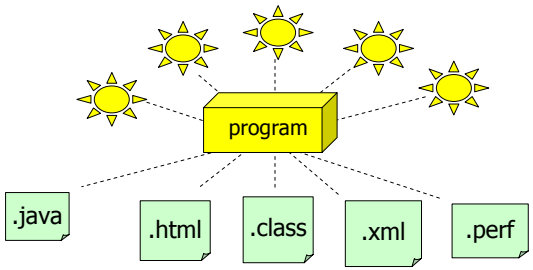
- Generating code for an individual program is OK, but not sufficient
- Today's systems are **not individual programs**, but groups of collaborating programs
  - client-server systems, tool suites (MS Office)
- Further, **systems are not solely defined by code**
  - architects routinely use many knowledge representations to express a system's design
  - process models, UML models, makefiles, documents

Question

- How does step-wise refinement scale to the synthesis of multiple programs and multiple-program representations?
- Challenge is not possibility
  - lots of ad hoc ways to do this
  - challenge is to define an algebraic model of application synthesis that treats all representations – code and non-code – alike

Insight #1: Platonic Forms and Domain-Specific Languages

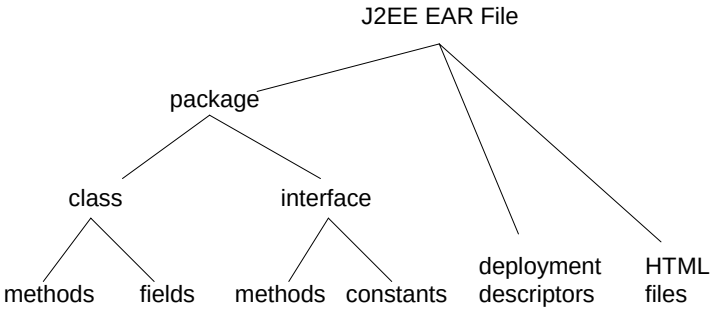
- Each program representation captures different information, and written in a **DSL**



- We want to encapsulate all these representations

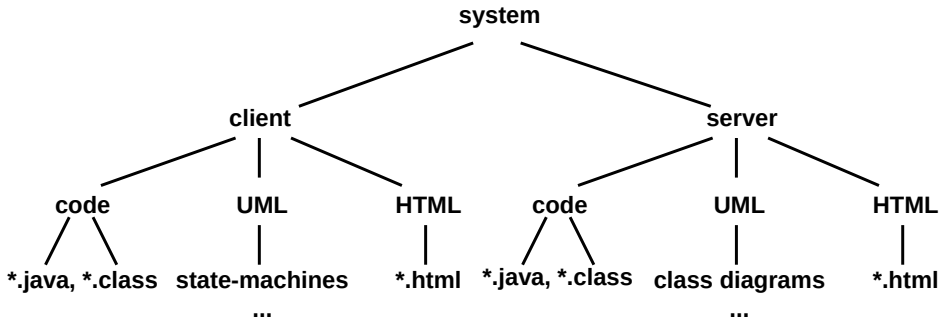
#2: Generalize Modularity

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



- Generalize module hierarchies to arbitrary depth, contents

Another Example

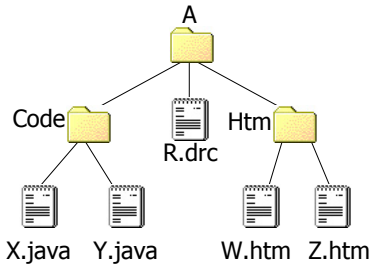


modules encapsulate all needed representations of a system

Simple Representation

- Module hierarchies = nested collectives

directory



algebraic

A = { Code, R.drc, Htm }

Code = { X.java, Y.java }

Htm = { W.htm, Z.htm }

#3 Generalize Refinements

- When a program is refined, 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)
  - ...

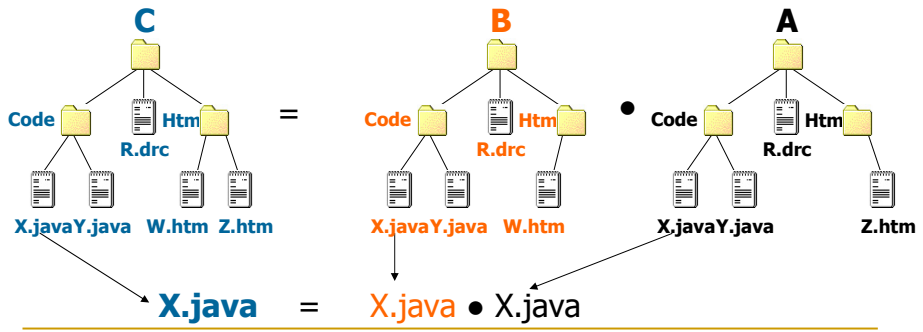
#3: Generalize Refinements

- Containment hierarchy is a "constant"
- Refinement is a function that maps (transforms) containment hierarchies

- can add new nodes (e.g., new .java, .html files)
- can refine 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

$C = B \bullet A$

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

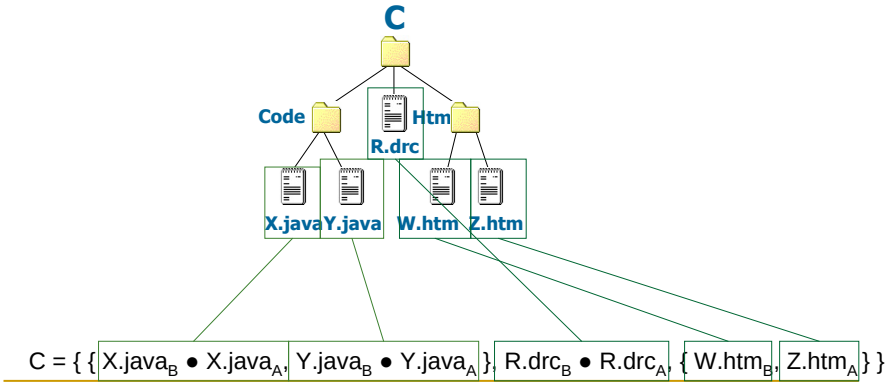
$= \{ Code_B \bullet Code_A, R.drc_B \bullet R.drc_A, Htm_B \bullet Htm_A \}$

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

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

Note!

- Each expression defines a refinement chain for an artifact that is to be produced



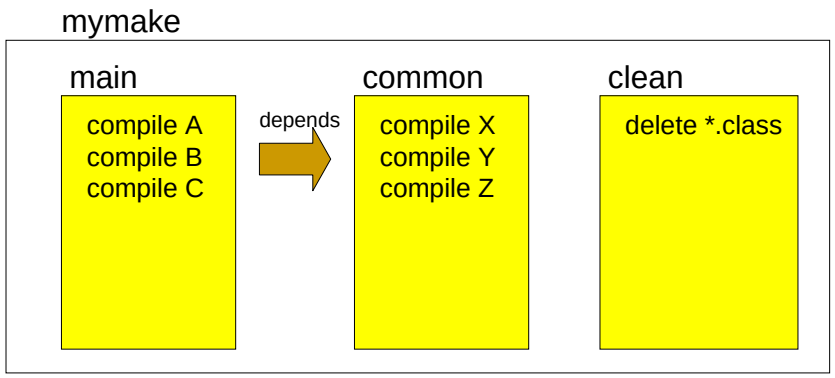
Polymorphism...

- Composition operator ● is **polymorphic**
  - composition law defines how collectives are composed
  - different implementation of the ● operator for each representation
    - operator ● means one thing for code
    - operator ● means something different for html files, etc.
- But what does refinement of non-code artifact mean?
  - what is a general principle that guides refinement?

Example: Makefiles

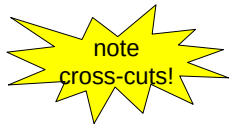
- Instructions to build parts of a system
  - it is a DSL for synthesizing programs
- When we synthesize code for a system, we also have to synthesize a makefile for it
- Sounds good, but...
  - what is a refinement of a makefile?????

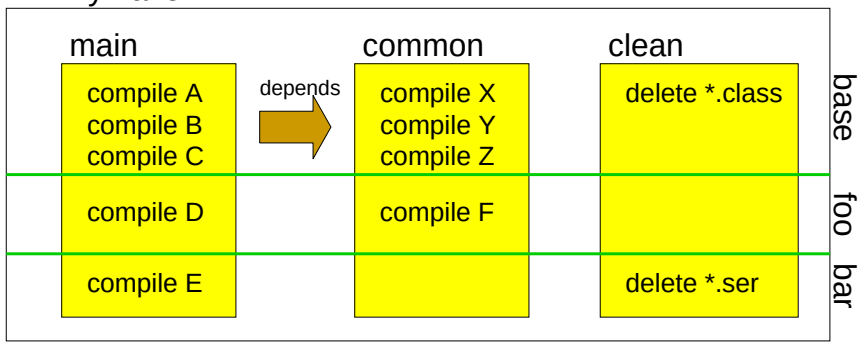
Makefile



command line> make main

Makefile Refinements





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

Makefiles are Classes!

```
<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 Refinement is Code Refinement

```
<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 (e.g., data members), etc.

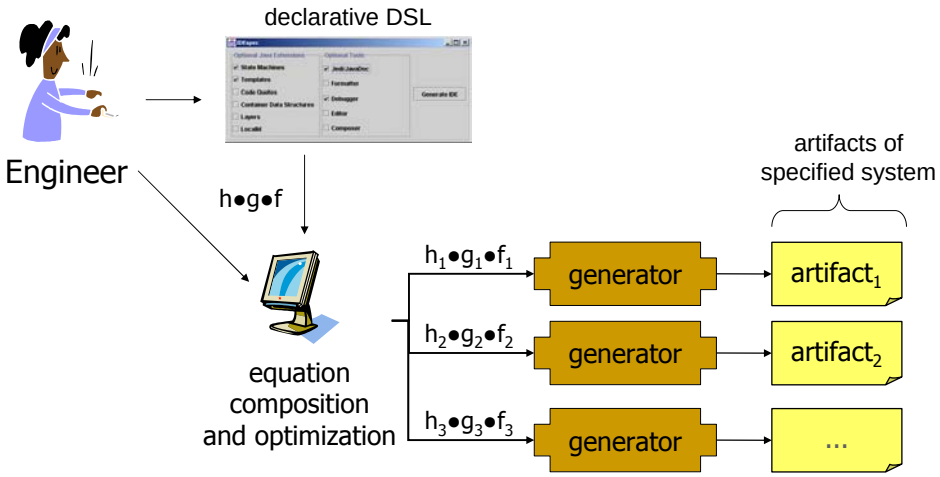
#4:Guiding Principle

- Principle of Artifact Uniformity
  - Create analog in OO representation:
    treat all artifacts equally, as objects or classes
  - refine non-code representations same as code representations
- That is, you can refine any artifact
  - understand it as an object, collection of objects, or classes

Big Picture

- Most artifacts today (HTML, XML, etc.) have **or can have** a hierarchical structure
- There is no refinement relationship among files
  - what's missing are refinement operators for non-code artifacts
- Requires tools to add refinement relationships among certain file types
  - not all (e.g. MS Word)
  - once present, you can scale step-wise refinement 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



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 Refinement", *International Conference on Software Engineering* (ICSE-2003), Portland, Oregon.
- 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, Singhal, Thomas, and Sirkin. Scalable Software Libraries. *ACM SIGSOFT '93 Conference* (Los Angeles), 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.
- Czarnecki, Bednasch, Unger, and Eisenecker, "Generative Programming for Embedded Software: An Industrial Experience Report", *Generative Programming and Component Engineering 2002*.

Recommended Readings

- Dijkstra, A Discipline of Programming. Prentice-Hall, 1976.
- 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.
- Kiczales, et al. "Aspect-Oriented Programming", ECOOP 97, 220-242.
- Kiczales, et al. "An overview of AspectJ". ECOOP 2001.
- Lieberherr, Adaptive Object-Oriented Software, PWS publishing, 1995.
- Mezini and Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development", OOPSLA 1998, 97-116.
- McDirmid, Flatt, and Hsieh, "Jiazzi: new-Age Components for Old-Fashioned Java", OOPSLA 2001.

Recommended Readings

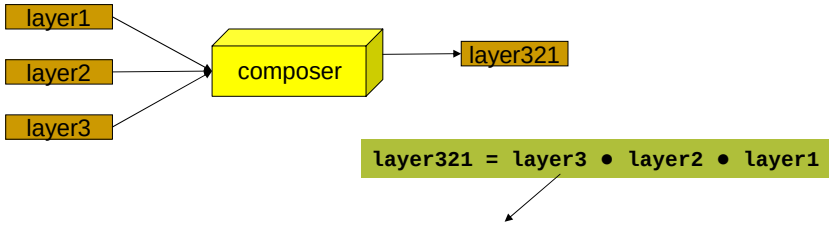
- Ossher and Tarr. "Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software." CACM October 2001.
- Ossher and Tarr, "Multi-dimensional separation of concerns and the Hyperspace approach." In Software Architectures and Component Technology (M. Aksit, ed.), 293-323, Kluwer, 2002
- 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.
- 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 Refinements 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.

An Introduction to AHEAD Tools

kick the tires...

Composer Tool

- Key tool in AHEAD Tool Suite (ATS) is composer
- composer composes layers/features to yield target system

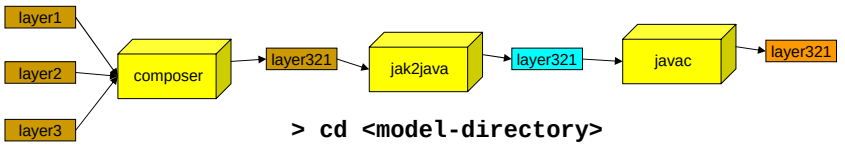


Jak Files

- We program in extended-Java files
  - Jak(arta) files
- Java + refinement declarations, etc.
  - Jak is an extensible language
- Most AHEAD tools are written in Jak files
  - all will be some day

Other Tools...

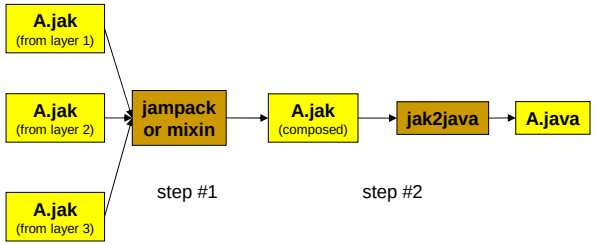
- Besides composer
  - **jak2java** – translates Jak files to Java files
  - **javac** – javac compiler
  - **reform** – Jak or Java file formatter/pretty-printer
  - others...




```
> 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 specifications
  - two tools now: jampack and mixin
  - jak2java translates .jak to .java



jampack

- Flattens refinement hierarchies
  - takes equation of refinement hierarchy (.jak equation) as input, produces single spec as output
  - basically macro expansion with a twist...

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

●

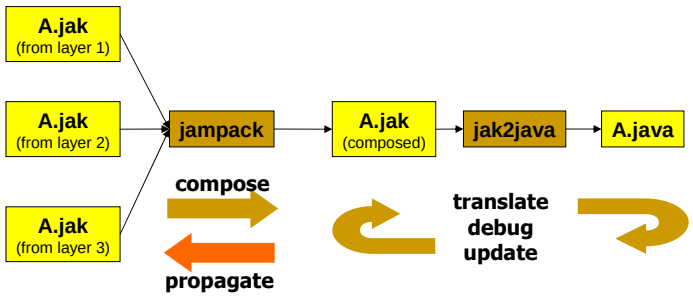
```
refines class top {
   int b;
   int bar() {...}
}
```

⇒

```
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 boundaries of refinements



mixin

- Encodes refinement hierarchy as inheritance hierarchy

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

●

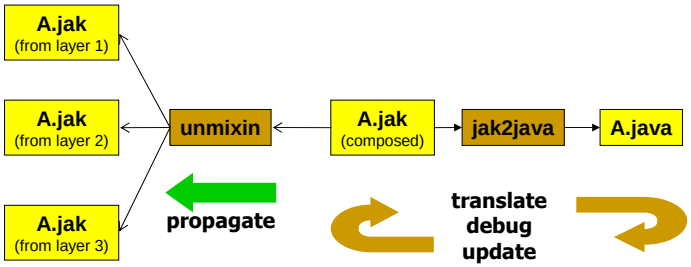
```
refines class top {
   int b;
   int bar() {...}
}
```

⇒

```
SoUrCe "A/top.jak"

abstract class top$$A {
   int a;
   void foo() {...}
}

SoUrCe "B/mid.jak"

public class top extends top$$A {
   int b;
   int bar() {...}
}
```

unmixin

- Edit, debug composed A.jak files
- unmixin propagates changes to original layer 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.

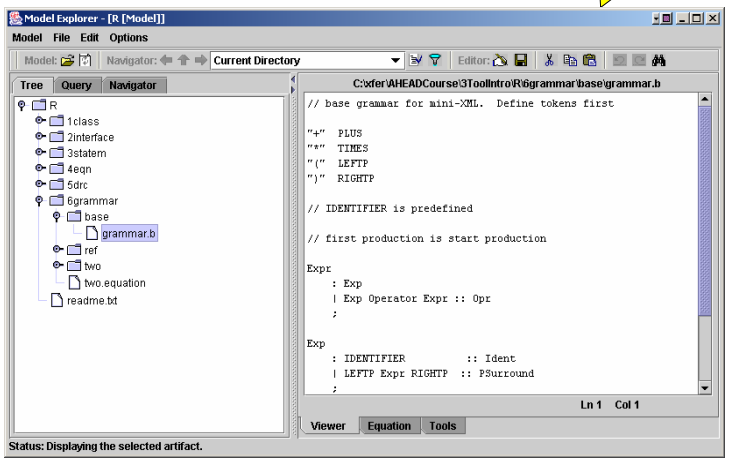
AHEAD has been bootstrapped so that its tools have been written using AHEAD tools.

See Lecture on Origami

Live Demo...

see files, compositions

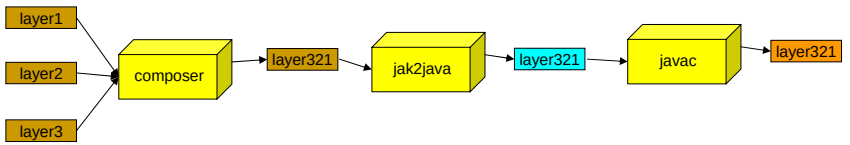
model tree view



file view

Cultural Enrichment

- Note algebraic underpinning...

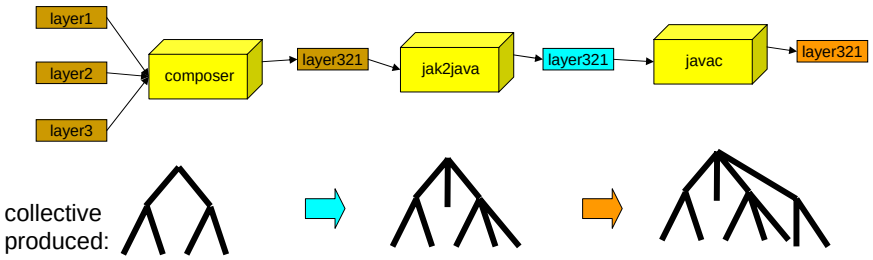



```
P = javac( jak2java( layer3 • layer2 • layer1 )))
```

- Same algebraic paradigm as AHEAD
  - progressively elaborating a containment hierarchy (collective)
  - can optimize equation (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 collective refinement



- Big picture: lots of operators on collectives
  - seems that lots of optimizations are possible too... (future work)

A Simple Example

to illustrate concepts, tools

Domain of Graph Applications

- Simplest 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**

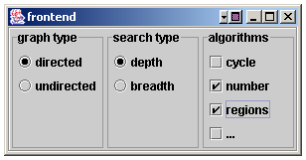
Example Family Members

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

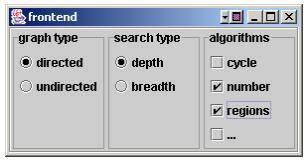


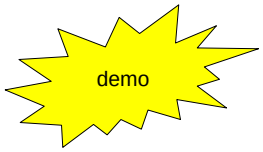
That's Easy...

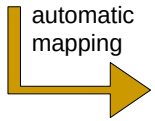
- So too is creating the underlying FOP model:

```
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

- order of function composition dictates order in which applications are refined...

Further Reading

- *Batory,* "A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite"*, January 2003.*
- Batory, Sarvela, Rauschmayer, "Scaling Step-Wise Refinement", *International Conference on Software Engineering* (ICSE-2003), Portland, Oregon.
- Batory, Cardone, and Smaragdakis, "Object-Oriented Frameworks and Product-Lines". *1st Software Product-Line Conference*, Denver, Colorado, August 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, July 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.

Relationship to AspectJ and Hyper/J

brief overview
of a very interesting topic

In Summer 2002

- Intuitive connections between models of AspectJ and Hyper/J
- We built GPL using not our tools but using
  - AspectJ
  - Hyper/J
- Presentation gives brief summary of results

How AHEAD Layers Map to Aspects

- Strong similarities, some differences
- Conjecture: models could eventually merge... (Here's why)

AHEAD Layer	Expressed as AspectJ Aspect
add new variables, methods, constructors	static cross-cuts
refine existing methods	impoverished use of dynamic cross-cuts (trivial predicates)
encapsulate addition of new classes (that are subsequently refined)	add new classes, but not encapsulated in aspects
add embedded DSLs to Java (e.g. state-machines)	?
refine non-code representations	? (Gray's thesis)
algebra, composition optimization	?

Classical Aspect: Monitor Rewrite

- Base class

```
class foo {

   void meth1() {...}

   ...

   void methn() {...}
}
```

- Monitor Refinement

```
refines class foo {

   semaphore S;

   void meth1() {
     S.wait();
     super.meth1();
     S.signal();
   }

   ...// more rewrites
}
```

repeat for each method

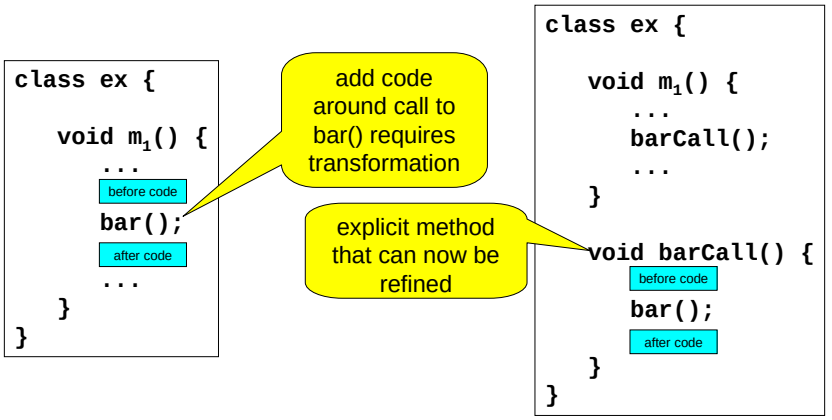
repetitious code – aspects (ala AspectJ) can specify this refinement elegantly

Monitor Rewrite in AspectJ

```
aspect monitor {
  public semaphore foo.S;
  pointcut semaphore_methods(foo f) :
           execution(void foo.meth*()) && target(f);
  void around(foo f) : semaphore_methods(f) {
       f.S.wait();
       proceed();
       f.S.signal();
  }
}
```


Aspects (in AspectJ) Do Much More...

- In AHEAD, join-points correspond to implicit methods



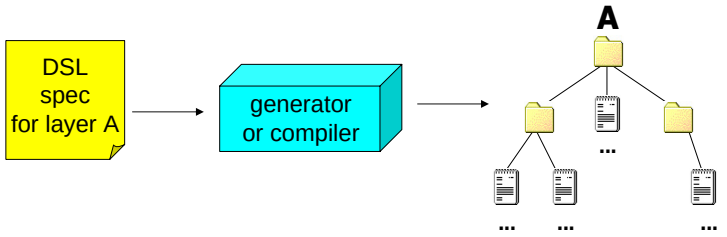
Aspects (in AspectJ) do this rewriting automatically

Join Point Rewrites in AspectJ

```
aspect more {
  pointcut bar_calls(): call(void foo.bar());
  void around() : bar_calls() {
      before code
    proceed();
      after code
  }
}
```


Bigger Picture...

- AHEAD layers are hand-coded
- In principle, no reason why layers can't be generated from higher-level (or DSL) specifications



  - doing this now for some AHEAD compiler-compiler tools
  - ex: translate a BNF specification into a parser (which is a layer)

Bigger Picture...

- Dynamic cross cut (pointcut, advice pairs) is a function that transforms/refines its input program

Base Program	P
Pointcut-Advice	A( )

$$P_{new} = A(P)$$

- Aspects can be viewed as generators of AHEAD layers
  - generates layer A_{gen} in AHEAD
    but this layer is specific to P, and probably no other program

$$P_{new} = A(P)$$
$$= A_{gen}(P)$$

  - achieve the same effect if we are willing to hand-code a layer
  - because aspects generate layers, it's more powerful, practical for these situations

Perspective

- In 20 years, 300+ layers many different domains
  - **in my experience**, 10% have "pointcut-advice" like implementations
    - e.g. same advice inserted at multiple places
  - we used editor, not sophisticated tools to solve these problems.
  - only few cases demanded more...
- In AHEAD tools, each tool has 10-15 distinct methods that are refined, and chains are short

Example AHEAD Layer

TypEscape.jak

```
public refines class TypEscape {
   public void check(int stage, String file) {
      super.check( stage-1, file );
   }
}
```

TlstEscape.jak

```
public refines class TlstEscape {
   public void check(int stage, String file) {
      super.check( stage-1, file );
   }
}
```

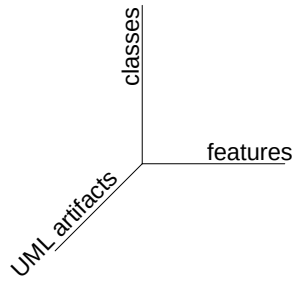
...

Why is use of Quantification Limited?

- Novelty and power of AspectJ is in quantification
- Our use of quantification is same as traditional OO designs
  - we normally give different advice to different join-points (i.e., predefined methods)
  - design methodology for layers is very similar to OO frameworks
  - to add a feature to an OO framework, there are specific, predefined methods that must be extended...
- This is how we have built product-lines for years...

Hyper/J

- ***Multidimensional Separation of Concerns***
  - dimensions consist of sets of concerns
  - space populated by units
- Current units are code-related artifacts:
  - Java code
  - Class files
- Hyperspace
  - All units in a software system
  - Organized in a multi-dimensional matrix based on a concern mapping of the form *Unit -> (dimension, concern)*
  - Dimension: partition of the set of units
- Hyperslice is an encapsulation mechanism
  - Set of units
    unit_description : Dimension.Concern



More Basics

- Hypermodule is composition mechanism
  - set of hyperslices
  - integration/composition relationships
- Different ways to define what to compose
  - merge by name
  - non corresponding merge
  - override by name
  - equate
  - ...

```
hypermodule modulename
 hyperslices:
   Dimension1.Concern1,
   Dimension2.Concern2,
   Dimension3.Concern3,
   …
   DimensionN.ConcernN
relationships:
   mergeByName;
end hypermodule;
```

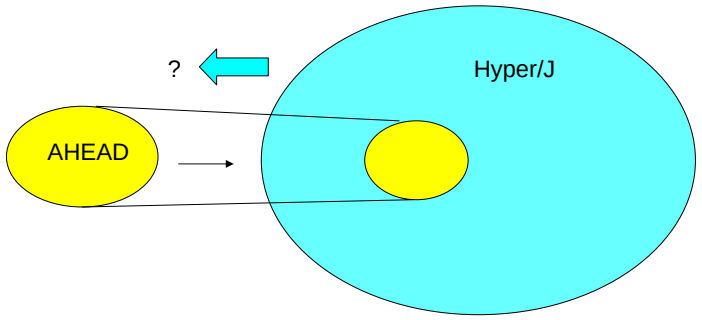

GPL

- Defined 1-dimensional space of features
- Each hyperslice was a GPL feature
- AHEAD equation == hypermodule
- Easy mapping as GPL only used "before" and "after" refinements

```
hypermodule Equation
 hyperslices:
   Dimension1.Directed,
   Dimension1.BFS,
   Dimension1.Number,
   …
   Dimension1.Cycle
relationships:
   mergeByName;
end hypermodule;
```


Open Problem

- We (kinda) know how to map AHEAD code models to Hyper/J
- Reverse mapping not yet fully understood
- Origami (Lecture 5) interesting connection to multi-dimensional models



How AHEAD Layers Map to Hyper/J

- Strong similarities, some differences

AHEAD Layer	Hyperslice
add new variables	✓
add new methods, constructors	✓
refine existing methods	✓
add new classes (that are subsequently refined)	✓
add embedded DSLs to Java (e.g. state-machines)	?
refine non-code representations	See tengger (UML)
algebra, composition optimization	?

To Summarize...

- AspectJ, Hyper/J, and AHEAD all use cross-cuts to implement features
  - there are clear similarities, differences
  - not clear if the differences are important (ultimately)
- AHEAD use of quantification has been minimal to date
  - along the lines of traditional OO designs
  - AHEAD closer to Hyper/J in this respect
- Open question: is lack of quantification typical for product-lines?

(stay tuned...)

Recommended Readings

- Barton, Harrison, and Raghavachari, “Mapping UML Designs to Java”, OOPSLA 2000.
- Barton, Harrison, and Raghavacarhi, “Tengger – Guide to Use”, IBM 2000.
- Harrison and Ossher. “Subject-Oriented Programming (A Critique of Pure Objects)”, OOPSLA 1993, 411-427.
- Gray et al., “Handling Crosscutting Constraints in Domain Specific Modeling”, CACM October 2001.
- Gray, “Aspect-Oriented Domain-Specific Modeling: A Generative Approach using a Metaweaver Framework”, Ph.D. Vanderbilt University, 2002.
- Kiczales, et al. “Aspect-Oriented Programming”, ECOOP 97, 220-242.
- Kiczales, et al. “An overview of AspectJ”. ECOOP 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.

Design Rule Checking

how to validate compositions automatically

Introduction

- Fundamental problem: not all syntactically correct equations are semantically correct
  - 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

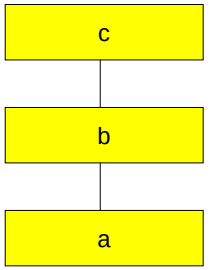
Introduction

- Absolute necessity to validate 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:
  - relationship to layering, grammars
  - DRC algorithms

Layering

- GenVoca originated from layered designs
- Layers are common form of refinements

$k = c \bullet b \bullet a$



Typing Layers (Refinements)

- GenVoca layers exported and imported standardized interfaces
  - "legos"
- Gave input-output types to layers (functions)
  - suppose S and R are interfaces

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

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

Types and Realms

- R: g(S:x) means refinement R:
  - exports interface R
  - imports layer x that implements interface S
  - x is a parameter
- Realm is a set of units that implement the same interface
  - model partitioning

S = { y, z, w }

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

Note

M = S • R

(set union)

Product-Lines and Grammars

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

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

```
S : y | z | w
R : g S | h S | i R
```

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

Symmetry

- Recursion is fundamental to grammars; symmetric layers are fundamental to GenVoca
  - export and import same interface
  - composable in virtually arbitrary orders
  - composition order affects semantics, performance
- A symmetric layer of realm **W** has parameter of type **W**

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

ex: m(n(p)), n(m(p)), m(m(p)), n(n(p)),...

What Does Symmetry Mean?

- Augments or enriches existing abstractions
  - relational DBMS – add transposition, data cube
  - 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 enrichments....
- Experience: **very** common in all domains....

Perspective...

- If models are grammars, what's wrong with normal syntax checking?
- Assign types to constants, functions...

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

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

- Ensure that all equations are type correct...

Syntax Checking Not Sufficient!!

- Recall relationship between grammars & sentences and product-lines & equations
- Type checking corresponds to syntax checking
  - just because your Java program is syntactically correct doesn't mean that it is semantically correct
  - we need MORE than syntax checking!
- Validation of compositions requires testing semantic constraints in addition to syntax checking
  - that's what DRC is all about

Overview

- DRC same as semantic checking performed by compilers
  - solution: use attribute grammars to define constraints
- Same here: AHEAD model is a grammar
  - design rules are **grammar attributes**
  - DRC algorithms propagate attribute values up and down parse (equation) trees and evaluate constraint predicates

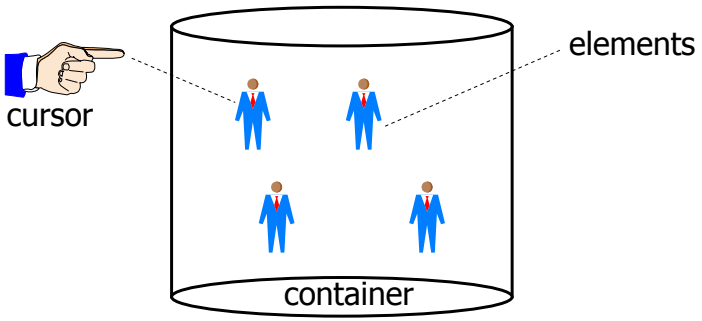
Motivating Example: P3

- Generator of **container data structures (CDS)**
  - see SIGSOFT 1993, 1994 papers on extension to C
  - see IEEE TSE 2000 paper on extension to Java
- Extended Java has embedded domain-specific language (DSL) for CDS
  - declarative specs that treat containers as database relations
  - container implementations are compositions of P3 components

P3 Model

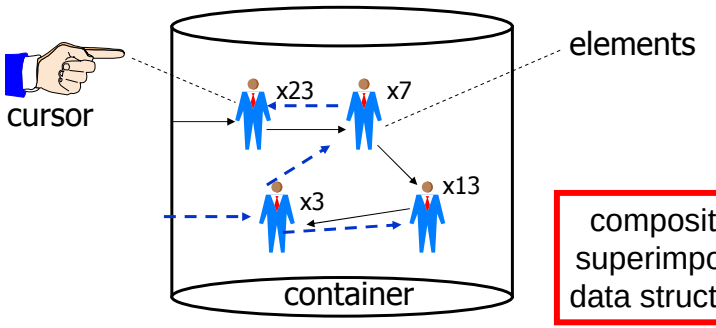
```
ds =  {  bintree( ds:x )       // binary tree
         dlist( ds:x )         // unordered list
         odlist( ds:x )        // ordered list
         avail( ds:x )         // free-list manager
         array( mem:x )        // sequential storage
         malloc( mem:x )       // random storage
         inbetween( ds:x )     // common delete code
         markdelete( ds:x )    // logical delete elements
            ...                // many more ....
      }

mem = {     transient          // in-memory storage
            persistent         // memory-mapped
      }
```


Data Structure Designs as Equations



container_eqn =

Data Structure Designs as Equations



composition superimposes data structures

container_eqn = bintree(odlist(malloc))

Efficient too!

		Dlist	Bstree	Rbtree	Hash
1997 paper	JDK	82.3	N/A	N/A	8.2
	CAL	117.4	19.4	17.3	13.5
	JGL	116.9	N/A	N/A	8.1
	Pizza	99.2	N/A	N/A	8.7
	P3	74.9	13.8	12.8	7.9

See: Batory, Thomas, and Sirkin. **Reengineering a Complex Application Using a Scalable Data Structure Compiler**. *ACM SIGSOFT* '1994.

Need for DRC

- Typical equations reference from 5 – 15 layers
  - earlier examples were simplified
- Too elaborate to validate by inspection
  - even I can't remember them and I wrote these layers!
- Some layers have obscure rules for their use
  - look at an example...

Example Design Rules

- inbetween(ds:x) encapsulates:
  - algorithms shared by all data structures (bintree, dlist, ...)
  - positioning of cursor after element is deleted
- Correct usage requires
  - one copy in eqn with 1+ data structures AND
  - precedes all such data structures in equation

Example P3 Design Rules

- Rules should not be borne by programmers
  - too easy to forget and be misapplied

```
correct   = ... inbetween( ... dlist( bintree(...) ))

incorrect = ... dlist(  ... inbetween( bintree(...) ))
```

Want rules to be tested automatically

Software Architecture Results

- Perry's **Inscape** (1989) is environment for managing evolution of software
  - light semantics: obligations and consistency checking
  - components have pre-, post-conditions, obligations

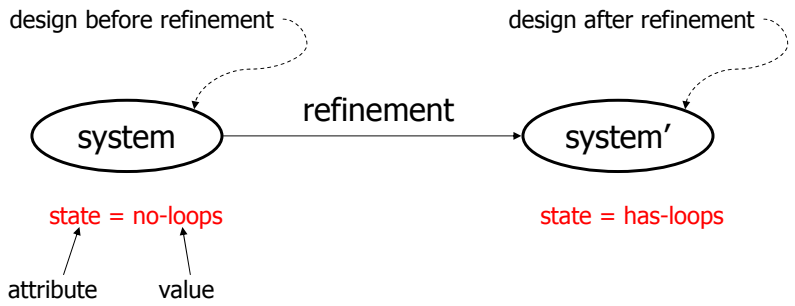
    bank loan example
- Obligations are conditions that must be satisfied by system that uses the component
  - beyond type checking – requires "action-at-a-distance" which are predicates that are nonlocally satisfied
  - propagated to enclosing module where they are eventually satisfied by some postcondition

Inscape (Cont)

- Full-fledged verification not attempted
  - primitive predicates declared (but informally defined)
  - pre-, post-, obligations expressed using primitives
  - practical and powerful form of "shallow" consistency checking using pattern matching and simple deductions

DRC: Adapt Inscape to Layers

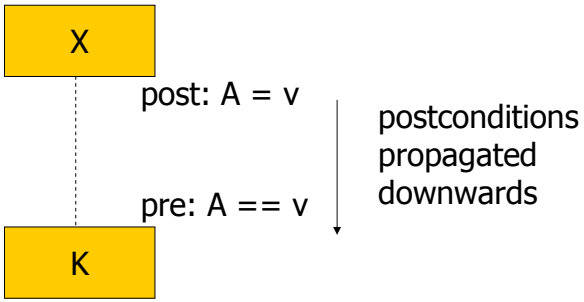
- DRC models state of equation design
  - not states of system execution



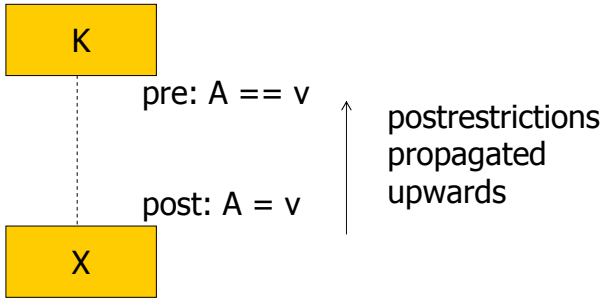
DRC: Adapt Inscape to Layers

- Preconditions and obligations of layer K are satisfied "at-a-distance" by layers either (far) below K or (far) above K
  - constraints typically not satisfied by adjacent layers (c.f. Goguen, Sitaraman)
  - properties exported to "higher" layers not the same as those exported to "lower" layers
  - leads to 2 kinds of design rules

#1: Constraints for Layer Usage



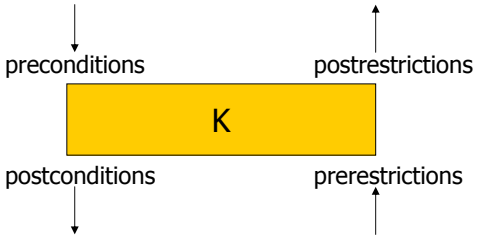
#2: Constraints for Parameter Instantiation



Note: corresponds to Inscape obligations

DRC Basics

- Layers have:



- DRC involves:
  - top-down propagation of postconditions and testing of layer preconditions
  - bottom-up propagation of postrestrictions and testing of parameter prerestrictions
- Basically very simple....

DRC Attributes and Predicates

- 3-value logic: attribute represents property whose value is:
  - asserted
  - negated
  - no information
- Predicates are conjunctions:
  - A ^ B properties A and B are asserted
  - (¬ A) ^ B property A is negated, B asserted

Condition Propagation Operator

- Postconditions, existing conditions specified by simple predicates
- Predicate composition operator ⊕
  - Post ⊕ Existing = conditions after composition
  - (A) ⊕ (⌐A ^ B) = (A ^ B)

Condition Testing

- Layer can be used if precondition P is satisfied
  - E is existing condition
  - test: E ⇒ P
- Example:
  - E = ⌐A ^ B
  - P = ⌐A
  - E ⇒ P is satisfied
  - implemented easily by property lists...

Top-Down DRC

top -- initial conditions for composition S

S

A — **top ⇒ precondition-A**

postcondition-A ⊕ top = top'

B — **top' ⇒ precondition-B**

postcondition-B ⊕ top' = top"

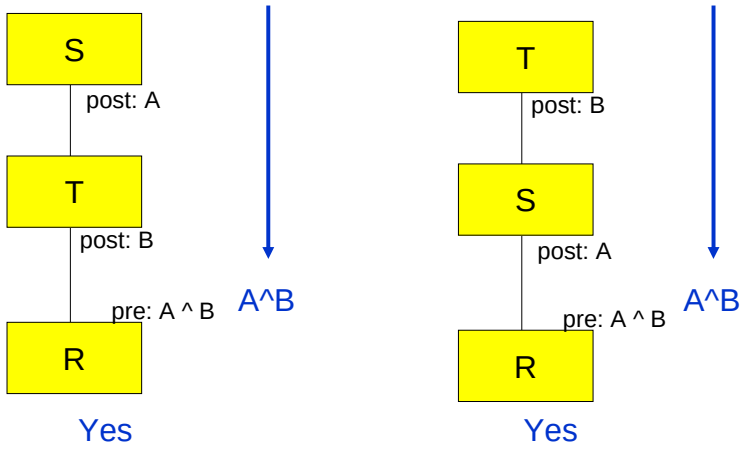
C — **top" ⇒ precondition-B**

postcondition-C ⊕ top" = top'"

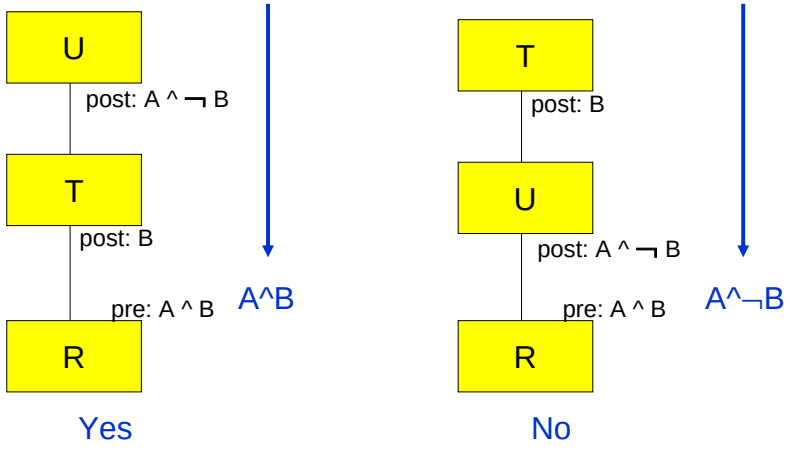
postconditions propagated by ⊕

preconditions tested by ⇒

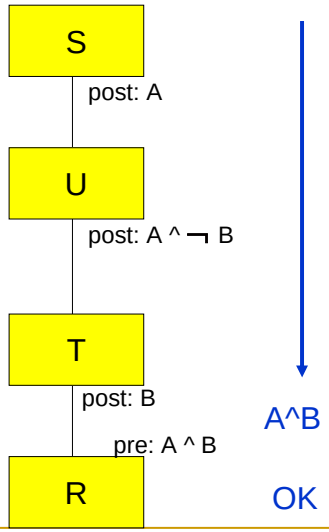
Is Composition Valid?



Is Composition Valid?

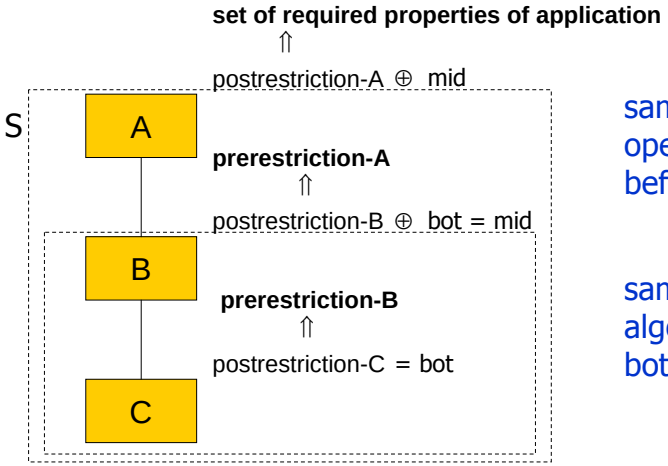


Is Composition Valid?



- Simple recursive algorithm for top-down propagation of conditions and testing preconditions
- Experience: all domains we've seen are like this
- Simple predicates
- Simple inferences
- **Don't need nuclear-powered theorem provers**

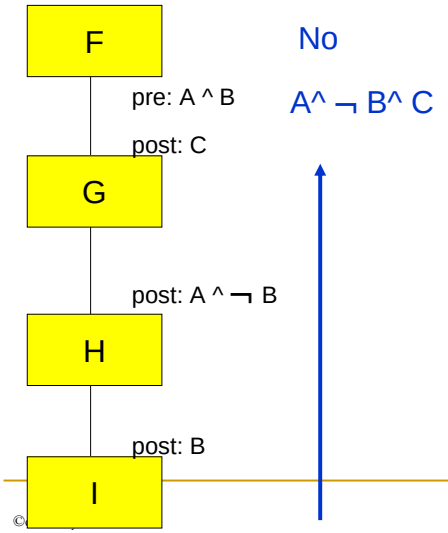
Bottom-Up DRC



same set of operators as before ⊕, ⇒

same algorithm for bottom-up DRC

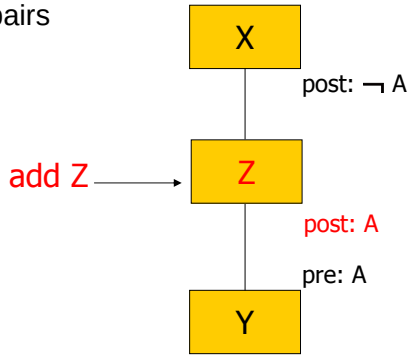
Is Composition Valid?



- Virtually identical to top-down DRC

Suggesting Error Corrections

- Besides detecting errors, DRC algorithms can suggest repairs
  - *precondition ceilings of Inscape*
- Error located in between X and Y
  - helps identify solution
- Similar technique for prerestrictions



Attribute Grammars

- McAllester observed attribute grammars unify realms, attributes, DRC algorithms
  - realms of layers are grammars
  - states of program design modeled by attributes
  - postconditions are inherited attributes (values determined by ancestors above)
  - postrestrictions are synthesized attributes (values determined by descendants below)

Implementation Notes

- Straightforward implementation
- DRC algorithm is efficient: O(mn)
  - m = # of attributes
  - n = # of layers

Domain	#Realms	#Layers	#Attributes
Genesis (databases)	9	52	14
JTS (Java precompilers)	1	10	10
P3 (data structure)	3	50	7

Experience

- Has worked well... (Perry has had similar experiences)
- Predicates are simple
- No need nuclear-powered theorem provers
- Reason: architects think in terms of features/refinements
  - 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, Singhal, Thomas, and Sirkin. “Scalable Software Libraries”. In *Proceedings of the ACM SIGSOFT '93 Conference* (Los Angeles), December 1993.
- Batory, Thomas, and Sirkin. “Reengineering a Complex Application Using a Scalable Data Structure Compiler”. *ACM SIGSOFT '94,* December 1994.
- Batory and Geraci. “Composition Validation and Subjectivity in GenVoca Generators”, *IEEE Transactions on Software Engineering* (special issue on Software Reuse), February 1997, 67-82.
- Batory, Chen, Robertson, and Wang, ”Design Wizards and Visual Programming Environments for GenVoca Generators”, *IEEE Transactions on Software Engineering*, May 2000, 441-452.
- *Batory, “*A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite*”, January 2003.*
- *Goguen, “Reusing and Interconnecting Software Components”, Computer, February 1986.*
- *Sitaraman and Weide, “Component-Based Software using RESOLVE”, ACM Software Engineering Notes, October 1994.*
- Perry. “The Logic of Propagation in the Inscape Environment”, ACM SIGSOFT 1989.

Origami and Product-Families

Scaling AHEAD to Product-Families

Introduction

- **Multi-Dimensional Separation of Concerns (MDSOC)**
  - Tarr, Ossher IBM
  - idea that modularity can be understood through multi-dimensional hyperspaces of units
  - slices of hyperspace are modules (such as refinements)
- Origami is an interesting example of MDSOC
  - first present a micro example
  - then a macro example – how we are synthesizing 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
       dblIns,      // refines sglIns to doubly-linked list
       addDel,      // adds deletion operation to sglIns
       dblDel       // refines addDel to deletion on
                    // doubly-linked list
    }
```


Example Refinements

sglIns =

list.jak

```
class list {
   node  first = null;

   void insert( node n ) {
      n.next = first;
      first = n;
   }
}
```

node.jak

```
class node {
   node  next = null;
}
```

dblIns =

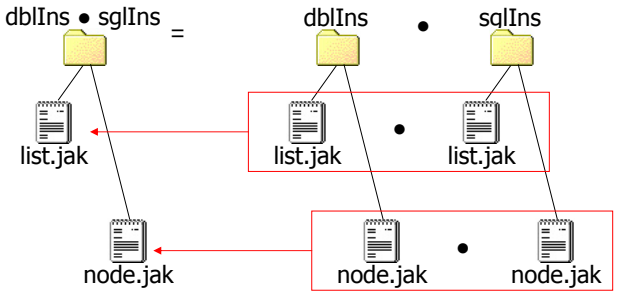
list.jak

```
refines class list {
   node  last = null;

   void insert( node n ) {
      if (last == null)
         last = n;
      if (first != null)
         first.prior = n;
      Super(node).insert(n);
      n.prior = null;
   }
}
```

node.jak

```
refines class node {
   node  prior = null;
}
```

Composition in AHEAD



Composition Results dblIns • sglIns

list.jak

```
class list {
   node first = null;
   node last = null;

   void insert$$sgl( node n ) {
      n.next = first;
      first = n;
   }

   void insert( node n ) {
      if (last == null)
         last = n;
      if (first != null)
         first.prior = n;
      insert$$sgl(n);
      n.prior = null;
   }
}
```

node.jak

```
class node {
   node   next = null;
   node prior = null;
}
```


Remaining Refinements

addDel =

list.jak

```
refines class list {

   void delete( node n ) {
      if (n == first) {
        first = first.next;
      }
      else
        findAndDelete(n);
   }

   void findAndDelete(node n) {
      node prev = first;
      while (prev.next != n)
         prev = prev.next;
      prev.next = n.next;
   }
}
```

adds delete operation to singly-linked list

dblDel =

list.jak

```
refines class list {

   void findAndDelete(node n) {
      if (n.prior != null)
         n.prior.next = n.next;
      if (n.next != null)
         n.next.prior = n.prior;
   }
}
```

refines delete operation to work 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 refinements

Interpreting 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 are inconsistent

Common Problem in FOP

- If data 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 refinement adds a new method, then it should work for that data structure and not some other structure
  - insert can't work on singly-linked list, delete on doubly-linked list
- Key idea of refinement: **preserve consistency**
- Representative of a large class of problems in FOP
  - models define refinements that are not truly independent
  - refinements must be applied in groups lock-step (all-or-nothing)
  - when this occurs, recognize groups implement "higher-level" features

Origami

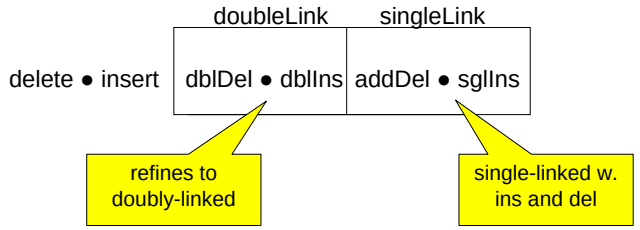
- Create matrix
- Rows represent operations (insert, delete)
- Columns are structure variants (singleLink, doubleLink)
- Entries are refinements of L
- "Higher-level" features are operations and structure variants

	doubleLink	singleLink
insert	dblIns	sglIns
delete	dblDel	addDel

Origami Matrix for L

Fold Origami Matrix (Hence Name)

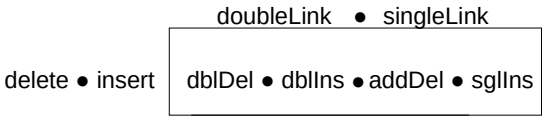
- Fold rows where corresponding entries in each column are composed



- Interpret entries of resulting matrix

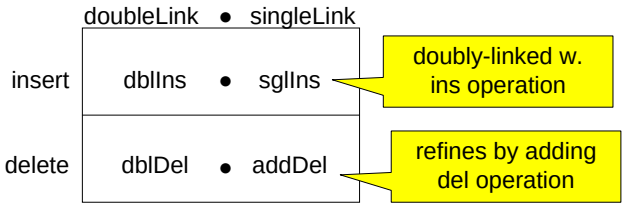
Now Fold Columns

- To produce 1x1 matrix
  - yields first of the two equations that defines doubly-linked list structure with insert and delete methods



Again, But Fold Columns First

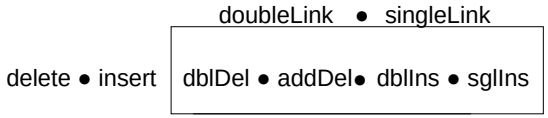
- Fold columns where corresponding entries in each row are composed



- Interpret entries of resulting matrix

Now Fold Rows

- To produce 1x1 matrix
  - yields second of the two equations that defines doubly-linked list structure with insert and delete methods

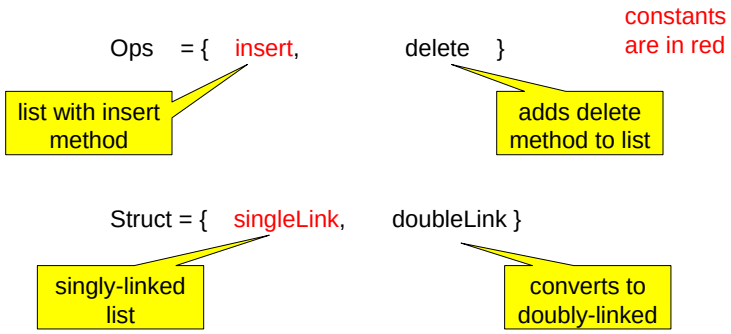


Perspective

- Folding (or not folding) the matrix, we generate only the legal equations of L
  - these are the programs of L:
    - sgl — slist w. ins
    - dbl • sgl — dlist w. ins
    - sgldel • sgl — slist w. ins & del
    - dbldel • dbl • sgldel • sgl = dbldel • sgldel • dbl • sgl — dlist w. ins & del
  - don't generate the illegal equations

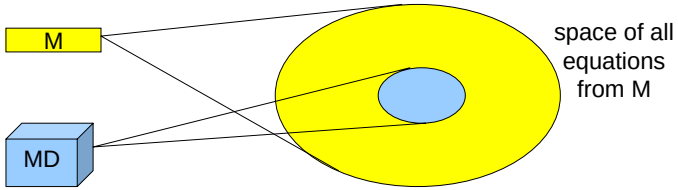
Perspective

- We've defined 2 orthogonal, abstract models
  - both can be used to describe same programs
  - matrix defines the relationship between these models



So What?

- 1D model M generates a vast space of eqns.
  - hard to interpret some of these equations, write design rules
  - most are incorrect



- n-D model MD expresses separation of concerns
  - generate many fewer (but correct!) equations
  - easier to specify, write design rules
  - easier to manage complexity

Building AHEAD Tool Suite

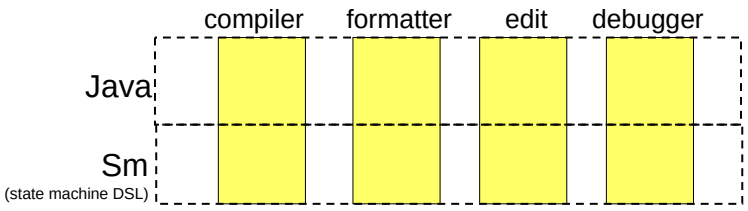
scaling Origami to product-families

Perspective

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

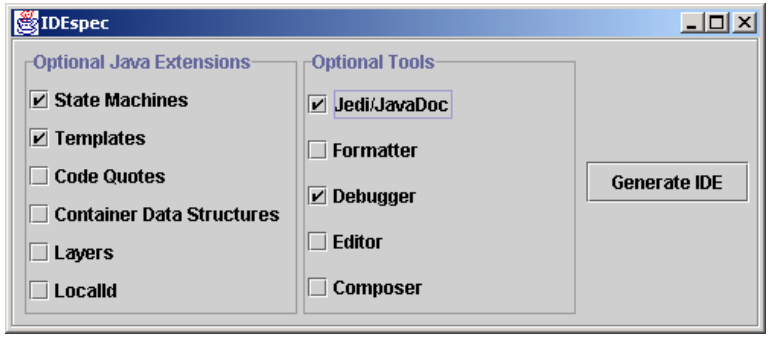
IDEs: A Product-Family of Tools

- Integrated Development Environment
  - 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?

Problem Formulation

- AHEAD Model of Java Language Dialects

constant **functions (optional features)**

J = { Java, Sm, Tmpl, Ds, ... }

Problem Formulation

- AHEAD Model of Java Language Dialects

constant **functions (optional features)**

J = { Java, Sm, Tmpl, Ds, ... }

- Java extended with state machine and templates is (Jak):

Jak = Tmpl • Sm • Java
= Sm • Tmpl • Java

Design Rules: start with Java, compose Sm, Tmpl in either order

IDE Tools...

- IDE tools are specified by a different, orthogonal model

constant functions (optional features)

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

- Different IDE tools have different equations

jak2java = ToJava • Parse

Jedi = Doclet • Harvest • Parse

…

Design Rules: Parse then Harvest then Doclet...

Feature Orthogonality

- Tool, Language features, models are orthogonal
  - implementation of each tool feature is function of language dialect

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

  - define relationships between J and IDE models using Origami
- Key idea: constants and functions can always be decomposed into compositions of more primitive constants and functions

$$C = f_1(f_2(f_3(\ldots f_n (C_0) \ldots)))$$

$$F(x) = f_1'(f_2'(f_3'(\ldots f_n'(x) \ldots)))$$

Origami 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

Origami Matrix for Jedi

Origami Matrix

- We know that compositions of these modules (refinements) yields the Jedi tool
- Question: how to map matrix to an equation?

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

Origami Matrix for Jedi

Fold the Matrix!

- Compose rows, columns in design rule order to yield an equation

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

Fold Harvest with Parse

Application Produced by Folding

- Compose rows, columns in design rule order to yield an equation

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

Fold remaining columns

Application Produced by Folding

- Compose rows, columns in design rule order to yield an equation

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

Fold Rows Java with Sm

Application Produced by Folding

- Compose rows, columns in design rule order to yield an equation

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

Fold remaining Rows

Application Produced by Folding

- Compose rows, columns in design rule order to yield an equation

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

An Equation for Jedi

Application Produced by Folding

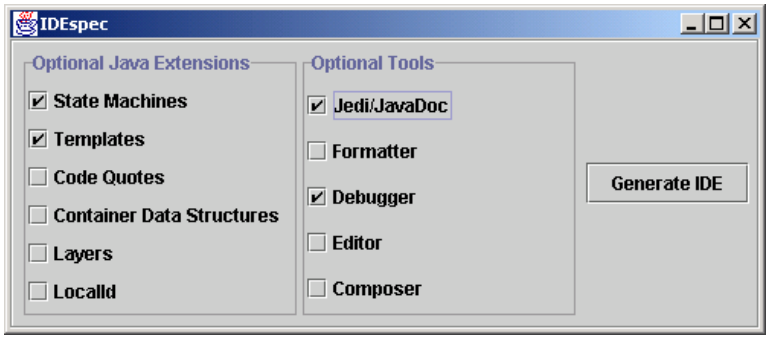
- Compose rows, columns in design rule order to yield an equation

Jedi = (TDoclet • THarvest • TParse) •
(SDoclet • SHarvest • SParse) •
(JDoclet • JHarvest • JParse)

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

Using Origami to Generate

- Product-families...



Origami 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 rows, columns that are not needed for desired tools and language dialects
  - 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

Origami Matrix for IDE Tools

- Now fold rows in design rule order

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

Origami Matrix for IDE Tools

- Now fold rows in design rule order

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

Origami Matrix for IDE Tools

- Now fold rows in design rule order
- Note the semantics of the result...

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

Yields Equations for Tool Features

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!
  - each equation defines a tool feature

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

  - and we know equations for each tool!

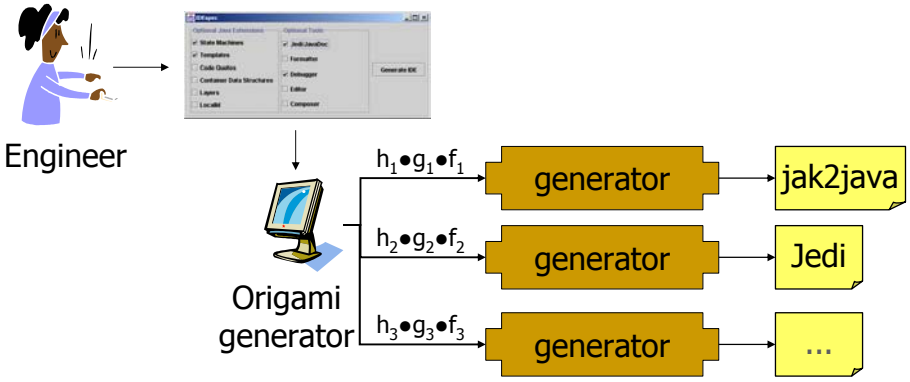
```
jak2java  = ToJava • Parse

Jedi      = Doclet • Harvest • Parse
...
```


IDE Generator is Simple

- For each selected tool, evaluate its eqn

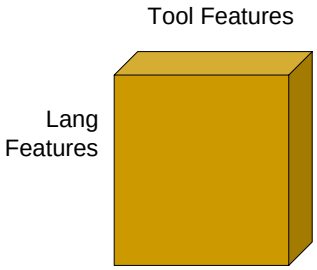
And generate the code for each tool automatically!

Generator of IDE Product-Family



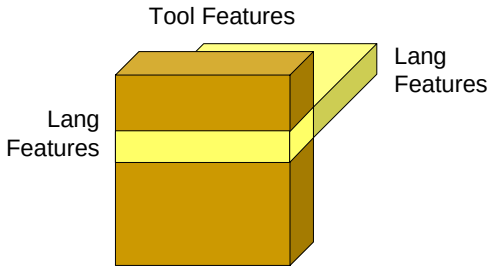
Bootstrapping AHEAD

- We used 3-Dimensional (8x6x8) Origami Matrix to generate 5 tools of the AHEAD tool suite



Bootstrapping AHEAD

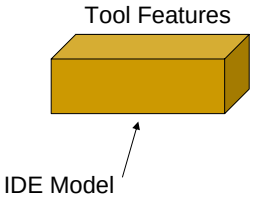
- We used 3-Dimensional (8x6x8) Origami Matrix to generate 5 tools of the AHEAD tool suite



3rd dimension captures language feature interactions

Bootstrapping AHEAD

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



Fold 3rd dimension
Fold rows

Results of AHEAD Bootstrap

- 76 distinct refinements
- Typical tool contains 20-30 refinements
  - most tools share 10 refinements
- Generated Java for each tool is 30K LOC
- Generating well close to 150K from simple, AHEAD declarative specifications
  - exactly what we want
- Making designs for multiple tools to conform to a matrix
  - controlling the complexity of product-families

Relationship to AOP

- Allows you to add "advice" to existing programs
  - typically in form of before, after methods
  - ex: method calls

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

- Folding rows of Origami matrix looks similar!

Jedi = (TDoclet • THarvest • TParse) •
(SDoclet • SHarvest • SParse) •
(JDoclet • JHarvest • JParse)

Insight

- Representing **architectural specifications** of programs as equations
  - Origami allows us to annotate (give advice to) architectural specs
  - specs are **not** Java programs, but equational representations of programs
- Representing software designs as equations is enormously powerful
  - ideal for generators
  - equational representations scale
  - micro example is ~150 LOC, AHEAD example is ~150K LOC
    - **3 orders of magnitude**
    - **ideas of origami apply to all levels of abstraction equally**
    - **no reason to believe that equational representations can't scale to much larger systems**

Insight

- Exploiting the power of refinements that are designed for reuse
  - non-coordinated designs, deal with arbitrary complexity
  - problem isn't how complicated we can make models; challenge is how simple you can make it
  - simple tools
  - Origami folding is a simple shell-script, now ant-script
- Others will encounter Origami as AOP, MSOC scales
- See Origami papers for more details

Recommended Reading

- Batory, "A Tutorial on Feature Oriented Programming", December 2002.
- Batory, Lopez-Herrejon, Martin, "Generating Product-Lines of Product Families", Automated Software Engineering 2002. Updated version submitted for journal publication.
- W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", *OOPSLA 1993*, 411-427.
- Tarr, Ossher, Harrison, and Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *ICSE 1999*.
- Ossher and Tarr, "Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software." CACM 44(10): 43-50, *October 2001*.

Recap

Summary of what we have covered...

FOP and Product-Lines

- Design individual program → think classes
- Design product-line (program family) → think features
  - members are distinguished by their features
- FOP study of feature modularity
  - raises features to first-class entities in design
  - features implemented by "cross-cuts"
  - close to OO framework designs
  - little or no quantification
- AHEAD is example of FOP
  - step-wise refinement
  - builds complex systems by adding features incrementally

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

Thank you!

Questions?

For more information and papers, visit our web site:

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