The Finland Tutorials

Don Batory
Department of Computer Sciences
University of Texas at Austin

June 2002

My Background

- Professor in Computer Sciences University of Texas at Austin
- Research:
  - extensible software
  - software product-lines
  - domain-specific languages
  - automated software construction
- Research goals: build customized software faster, cheaper, and better

Overview

- Thesis: simple ideas can streamline the design, construction, and evolution of complex software in an elegant way
  - result: a theory of software design based on generative programming
- Very different way to understand and develop software
  - takes time to appreciate
- Goal: create a scientific theory of software design and implementation – a body of knowledge organized around principles, expressible by mathematics

A Guiding Analogy

- Audio recording techniques then and now
  - 1950's – expensive, "get-it-right-the-1st-time", hard to change
  - today's recordings made in sound studios that "mixin" different (but simple) sound tracks to create rich artifacts
  - same for video images (e.g., Titanic)
  - layering simplifies construction of sophisticated artifacts from simple artifacts, controls cost, reduces complexity, and improves product
- We are building Y2K+ software using 1950's tech.
  - very expensive, hard to change
  - show how to build software a more modern way

Ideas are Applicable

- Small-to-medium systems
  - 10K – 200K LOC
- Special cases are COM, CORBA components
  - 200K+ LOC

Overview

- First 2 lectures summarize work prior to 2000
  - review basic ideas
  - coherent & elegant architectural models
  - composition validation
  - automatic programming
- Last lecture outlines vista AHEAD
  - ideas that have radically altered my understanding of my own work, greatly expanded what is possible
  - tunnel analogy

At Stake...

- Next generation software design and programming technologies

aspects

generative programming (generators)

collaboration-based designs

metamodels

Refinements

layers and hierarchical designs

Gen(esis+A)Voca

declarative programming

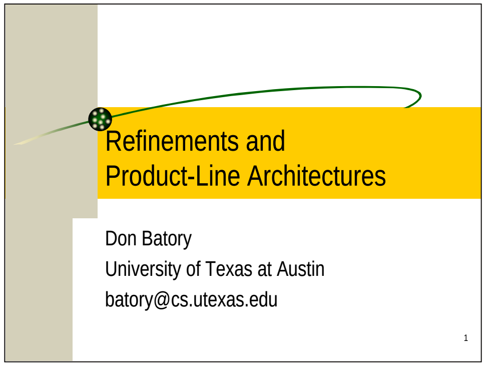
design rules

design wizards

Tutorial Lectures

- #1: Refinements and Product-Line Architectures
- #2: Design Rules and Design Wizards
- #3: Scaling Refinements
- Collection of previous talks
  - 50-minute invited presentations (2000-)
  - 20-minute conference presentations (1998-)
  - some from earlier tutorial (1994-1998)
- Ask questions whenever!!

Lecture 1


Refinements and
Product-Line Architectures
Don Batory
University of Texas at Austin
batory@cs.utexas.edu
1

Lecture 1a: Refinements and Product-Line Architectures

Don Batory
University of Texas at Austin
batory@cs.utexas.edu

This Lecture

- About a new kind of modularity for software
  - ideal for (product-line) architectures, software synthesis
  - introduce ideas through series of short presentations
- Ideas are:
  - simple, easy to understand, easy to recognize
  - deep, hard to understand
  - applicable now...

So What?

- Why do we need a new kind of modularity when we're satisfied now...?
- Ans: you're not satisfied!
  - – add/remove feature from existing application
  - – COM-DCOM-CORBA components aren't universal
    - show example later where COM modularity is opposite of what we want

Historical Perspective

- Software design and programming languages influenced by modularity
  - module encapsulates primitive functionality or service that (ideally) can be reused
- Module granularities became progressively larger
  - small - function
  - medium - class = suites of interrelated functions
  - large - package = suites of interrelated classes

Granularity vs. Reuse

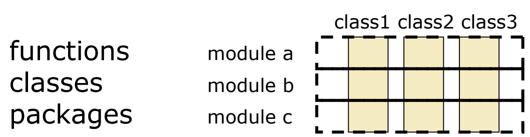
- Benefits of scaled granularity driven by reuse
- More a module is used, more valuable it is
- Biggerstaff 1994 observed:
  - larger the module, more specific its functionality, less likely to be reused
  - scaling modularity seems to defeat the purpose of reuse
  - opposite of what we want

Solution

- Answer is not entire function, class, package

Solution

- Answer is not entire function, class, package
- Lot of independent research today says solution is a module encapsulates fragments of



  - composing modules yields packages of fully-formed classes

Contributors to this view…

- Different researchers have different variants (implementations) of this idea
  - refinements – Dijkstra, Wirth 68, Neighbors 84, Smith 89
  - layers – Dijkstra 68, Batory 84
  - collaborations – Reenskaug 92
  - traversals – Lieberherr 96
  - aspects – Kiczales 97, et al.
  - concerns – Ossher-Harrison-Tarr 99
  - feature-based product-lines – Kang 90, Gomaa 92

Common Idea...

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

Tutorial on Refinements

Refinements are Interchangable

Refinements are Interchangable

Refinements are Interchangable

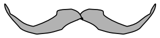
Refinements are Interchangable

Refinements are Reusable

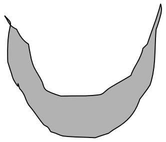
Refinements are Functions

PersonPhoto beanie(PersonPhoto x)

PersonPhoto uncleSam(PersonPhoto x)



PersonPhoto mustache(PersonPhoto x)



PersonPhoto lincolnBeard(PersonPhoto x)

Refinement Compositions

- Refinement composition == function composition

= lincolnBeard(uncleSam())

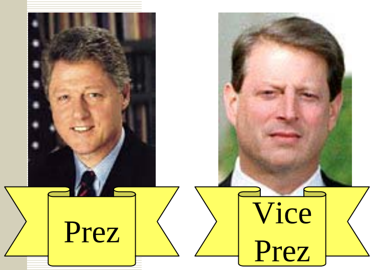
Large Scale Refinements

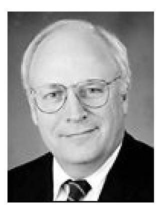
- called Collaborations (1992)
  - simultaneously modify multiple objects/entities
  - refinement of single entity is called role
- Example: Positions in US Government
  - each defines a role

….

Composing Refinements

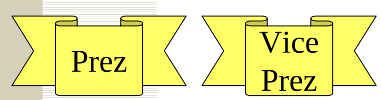
- At election-time, collaboration remains constant, but objects that are refined are different





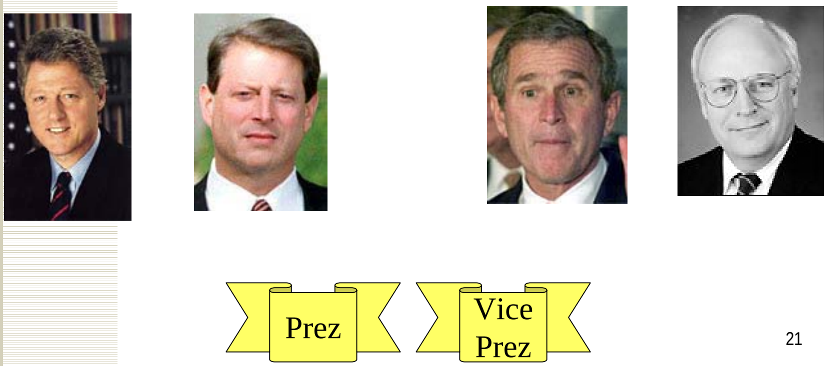
Composing Refinements

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



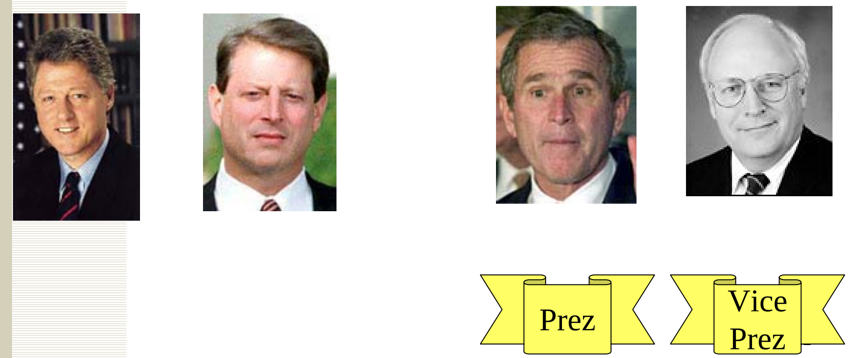
Composing Refinements

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



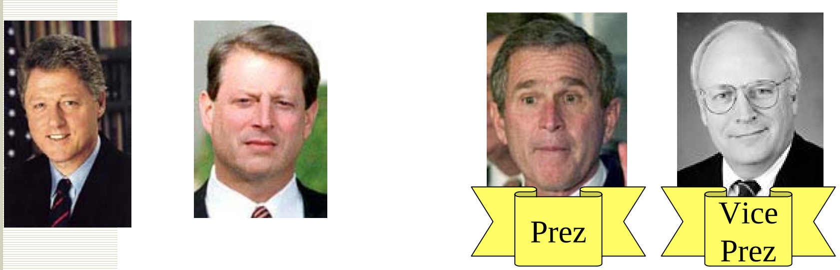
Composing Refinements

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



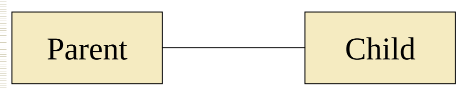
Composing Refinements

Example of dynamic composition of collaborations



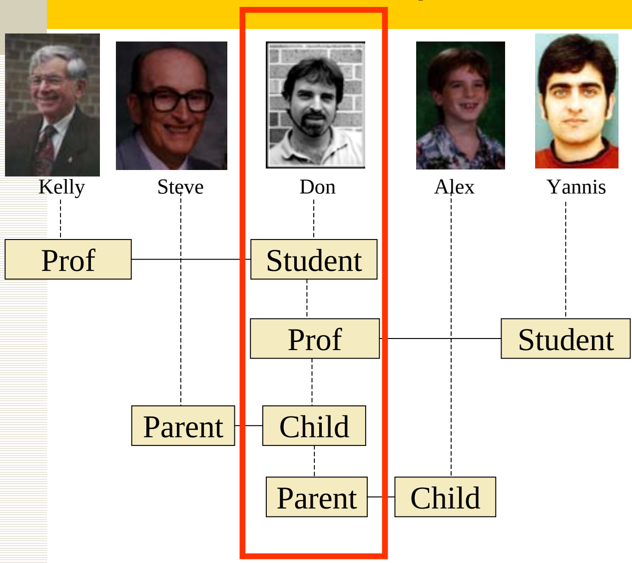
Other Collaborations

- Parent-Child collaboration



- Prof-Student collaboration

Example



Same Holds for Software!

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

Returning to Computer Science

- Refinement – an elaboration or extension of a program that introduces a new service or feature
- Prominent characteristic is “cross cutting”
  - refinement modifies multiple classes of an application simultaneously and consistently
- “Aspect” is the currently popular term for this effect
  - “refinement” was original name
  - does not imply particular implementation (as does “aspects”)

Connecting the Dots…

- Resurrection of age-old design methodology **step-wise refinement**
  - idea of progressively building programs by adding one detail or feature at a time
  - abandoned because it failed to produce programs of significant size
  - reason: use of microscopic refinements required hundreds/thousands of refinements to produce admittedly small programs
- **Step-wise refinement** is fundamental and shouldn’t be abandoned
  - **but it needs to be scaled!**

Novelty of Current Work

- Addresses key limitations:
  - scaling refinements – where single refinement impacts multiple classes
  - composing a few refinements yields entire application
- Consequences:
  - inverse relationship between module size and reusability (which crippled conventional concepts of modules) no longer applies
  - software modularity is a topic of wide-spread interest
  - leads to talk on product-line architectures…

Introduction to Product-Lines

- 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)

Product-Line Architectures

- Problems become evident in PLAs
  - goal: build families of related applications through component compositions…
- With PLAs we want:
  - simple specifications of applications
  - reason about application implementations using components
  - automatically optimize designs given application constraints

Can be done...

- Provided that components encapsulate implementation of individual features that can be shared by multiple applications
  - app1 has features x,y,z
  - app2 has features x,q,r

Focus of discourse is on FEATURES not CODE

Can be done...

- Provided that components encapsulate implementation of individual features that can be shared by multiple applications
  - app1 has features x,y,z
  - app2 has features x,q,r

Focus of discourse is on FEATURES not CODE

- Features align better with requirements
  - more abstract form of modularity
- But refinements are what features are all about...!
- Outline a model of software development based on refinements...

Next Few Slides...

High-level view of application specifications

→ Abstract model for implementing specifications

→ Concrete implementation of this model

→ Relate to Other issues

Example: Domain of Graph Applications

– Simplest way to express family of related applications is as a grammar
  - different members have 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 | ... }

(selected: undirected, breadth-first, vertex numbering)

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

(selected: directed, depth-first, cycle checking, connected regions)

Now its your turn...

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

- Easy to imagine a GUI tool that would allow you to specify any possible combination
  - and generate an explanation of your specification
  - and identify errors (and suggest corrections) when some combination of features is not possible

That's easy... but what's hard?

- Mapping to an abstract model of product-lines
- Basic ideas:
  - programs are values
  - functions map input values (programs) to output values (programs)
  - **GenVoca Model**

Programs as Values

- Constants:
  - f – an application with feature f
  - h – an application with feature h
- Functions (Refinements)
  - i(x) – adds feature i to application x
  - j(x) – adds feature j to application x

Key idea: equating features with refinements (constants, functions)

Function Composition

- 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 set of "building block" constants and functions, we can create a family of applications through function composition

Graph Application Domain

- Constants:

  directed
  undirected

- Functions:

  dfs(x) – depth first search
  bfs(x) – breadth first search
  cycle(x) – cycle checking
  number(x) – vertex numbering
  region(x) – connected regions
  …

Constructing Applications

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

- graph_app = region(vertex(dfs(directed)))
- order of function composition is dictated order in which applications are refined....

Where we are…

High-level view of application specifications

→ Abstract model for implementing specifications

→ Concrete implementation of this model

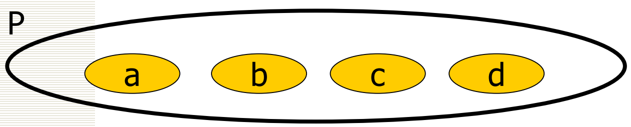
Questions to Answer...

- How do we represent programs as constants?
- How do we represent refinements as functions?
- Note: there are lots of answers.

  Here is the simplest...

Programs are Constants

- Application P is a set (package) of classes

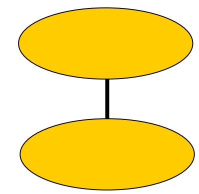


represent P as a class, not as a package!

```
class P {
   class a { ... } // inner classes
   class b { ... }
   class c { ... }
   class d { ... }
}
```


Functions?

- How do we statically refine classes in OO?



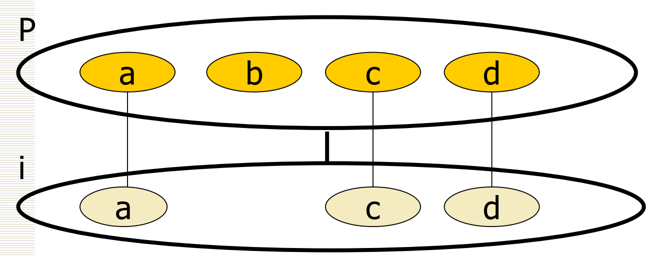
Ans: inheritance

Scaling Refinements...

- When you add a new feature to an existing OO application, what do you notice?
- changes aren't localized
  - many classes are refined
  - "cross cuts"

Functions

- Apply function i() to application P




```
class i <x> extends x {              // mixins =
   class a extends x.a { ... }       // parameterized
   class c extends x.c { ... }       // inheritance
   class d extends x.d { ... }
}
```

Mixin-Layers

- Nest mixins inside mixins – called mixin-layers

```
class i <x> extends x {          outer mixin
   class a extends x.a { ... }
   class c extends x.c { ... }   inner or nested mixin
   class d extends x.d { ... }
}
```

- An elegant way to implement collaborations (refinements)
  - as we will see later, not the only way...
  - there are lots of ways...

Summarizing

- Functions are implemented as mixins
  - take superclass as input and produce subclass as output
- Function composition corresponds to template composition

j(i(h)) ➔ j< i< h > >

Where we are…

High-level view of application specifications

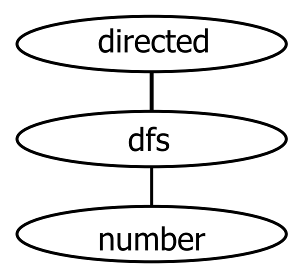
→ Abstract model for implementing specifications

→ Concrete implementation of this model

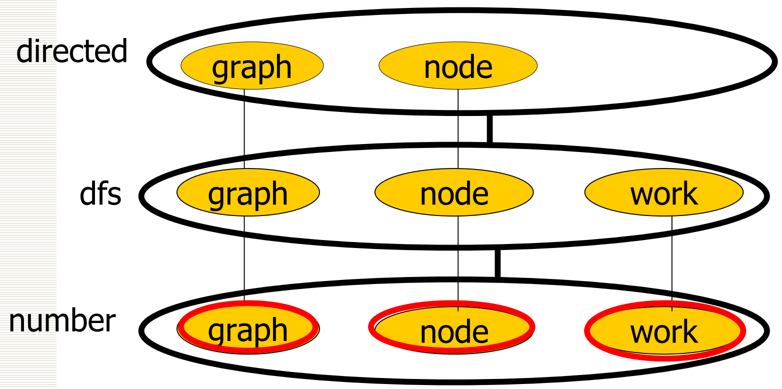
Graph Domain

- Consider application:

app1 = number(dfs(directed))

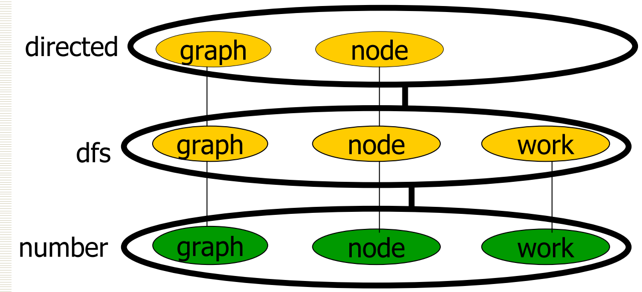


app1



What If app1 Written By hand?

- Wouldn't have the inheritance hierarchy



  – only write bottom-most classes
  – these classes can be automatically generated

Big Picture

- Lots of ways to implement refinements:
  - objects
  - templates
  - metaprograms
  - rule-sets of program transformation systems...
- Lots of success: Product-lines created for
  - database systems (1988)
  - network protocols (1989)
  - data structures (1993)
  - avionics (1994)
  - extensible Java compilers (1997)
  - radio ergonomics (1998)
  - verification tools (2000)

Remaining Topics…

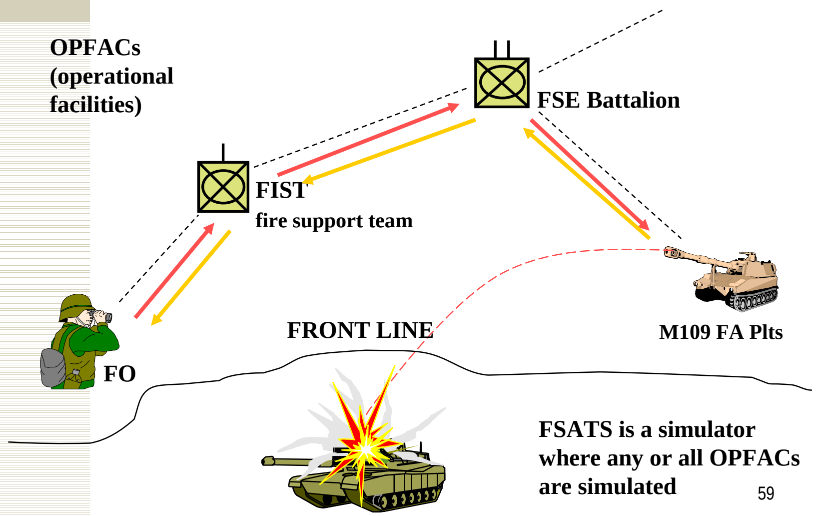
- Nontrivial example of these ideas
- Future areas of research….

A Real Example…

FSATS

- **Fire Support Automated Test System**
  – command-and-control simulator for Army fire support
  – 1st generation system (10 years old)
- Problems common to other applications
  – difficult to understand, maintain, debug
  – new capabilities projected, existing revamped
  – **design fatigue** – don't want to extend current version

Overview of Fire Support



Vanilla Distributed Application

- Set of collaborating objects that work collectively to process mission
- Different types of missions (collaborations):
  – WRFFE artillery
  – WRFFE mortars
  – Adjust-Fire artillery, Adjust-Fire mortars
  – about 20 mission types in all, more are projected
- OPFAC takes different actions per mission type
- Can simultaneously process any number of mission instances (2 WRFFE-mortars, 3 AF-Arts)

Original Implementation

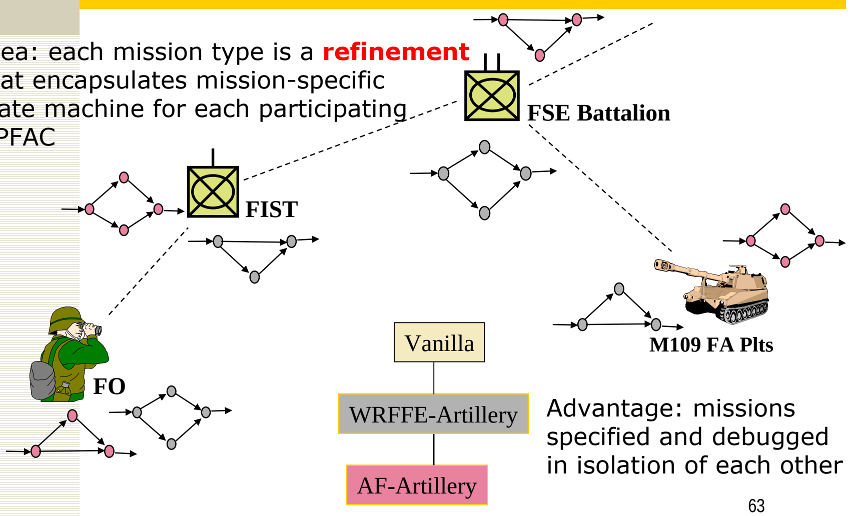
- Was monolithic; each OPFAC is an Ada program that sends and receives tactical messages
- Received message processed by rules:
  - **if (conditions$_1$) do-action$_1$;**
  - **if (conditions$_2$) do-action$_2$;**
  - **if (conditions$_3$) do-action$_3$; . . .**
- Complicated...
  - **conditions are conjunctions of 5-10 primitives**
  - **200-1000+ rules per OPFAC**
  - **hard to see what rules would actually apply to given mission**
  - **difficult to write, understand, debug rules**

Key Goals of Redesign

- Disentangle logic of different mission types
  - implementation and testing of different missions independent of existing missions
- Reduce conceptual distance from logic specification to implementation
  - trace implementation to requirements
- Easy to add new mission types, experiment with different implementations

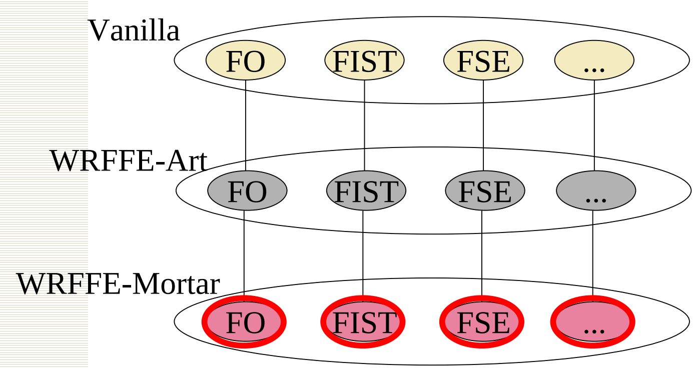
FSATS Prototype

Idea: each mission type is a **refinement** that encapsulates mission-specific state machine for each participating OPFAC



Advantage: missions specified and debugged in isolation of each other

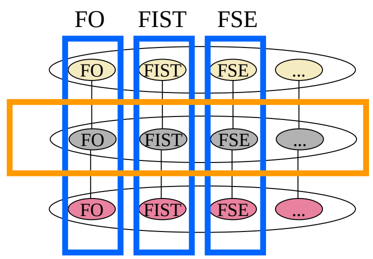
Mixin-Layer Implementation



FSATS = WRFFE-Mortar(WRFFE-Art(Vanilla))

Perspective

- Each vertical inheritance chain defines an OPFAC program
  - CORBA or DCOM component
- Each mission type (an FSATS building block) cuts across OPFAC programs
  - layer or refinement



Concrete Benefits

- Code complexity reduced by factor of 4
- Added feature in 3 days
  would have taken over a month previously
- Regained intellectual control over FSATS design

See "Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study"
Int. Conf. Software Reuse, June 2000

- $2.2M project in 2002 from STRICOM to build next-generation version of FSATS
- **More in later lecture...**

Future Areas of Research

Automatic Programming

Separation of Concerns

Automatic Programming

- Ancient problem of program synthesis
- Goal: translate declarative specifications on program use to efficient implementation
- Largely abandoned in mid-1980s because techniques didn't scale, too complicated
  - See Balzer's paper in Biggerstaff & Perlis Reuse Text
- Still an important problem!!

Automatic Design of Software

– Remember: applications are represented by equations!

– Optimizations arise when there are multiple ways to implement the same feature

- suppose we want an application with features a, b, c
- 3 ways to implement b:

$b_1(\ldots)$, $b_2(\ldots)$, $b_3(\ldots)$

Equation Optimization

- We know one of the following equations best defines our application:

$$\text{App} = \min\{\ \$(\,a(\,b_1(\,c\,)\,)\,),\ \ \$(\,a(\,b_2(\,c\,)\,)\,),\ \ \$(\,a(\,b_3(\,c\,)\,)\,)\ \}$$

Equation Optimization

- Intelligently walk the space of all equations
  - convert each equation into cost function
  - evaluate cost function to assess “efficiency” of design
  - having found “best” design, convert equation into software
  - analogous to relational query optimization
- Refinements “encapsulate” changes to:
  - source code
  - performance models...

Equation Optimization

- See “Design Wizards…” IEEE TSE May 2000
  - automatically designs software for given domain
  - automatically generates this software
- Concrete results:
  - generated code typically faster than hand-written code
  - designs typically as good (sometimes better) than experts
- Exciting area for further research…
  - **more in later lecture...**

Separation of Concerns

- People model applications from different viewpoints:
  - requirements, source code, documentation
  - formal properties, performance properties, ...
  - PLA conference – one group maintains 9 different views of their software (process, class-diagram, …)!!
- All are ***concerns***
  - *different dimensions and representations in which to conceptualize, understand, and build software*

Relevance to Refinements?

- Refinements are very abstract concept
  - need not be limited to expressing changes to source code

    (which is almost all that we look at today)
- When you apply a refinement to an application, you *change* the application's:
  - source code, performance properties, documentation,
  - formal properties, ...
  - "cross cutting effects"

Refinements and Concerns

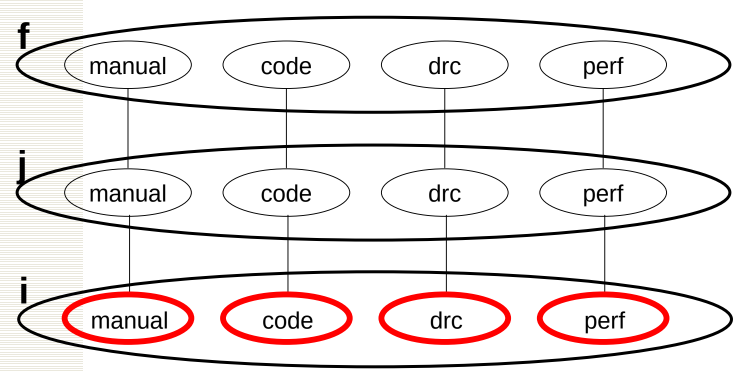
- When we write applications as equations:

  app1 = i(j(f))

- We could be updating multiple representations – concerns – simultaneously and

  Consistently

Visualization of i(j(f))



Our Experience

- We built distinct tools and specifications for refinements:
  - source code
  - formal properties
  - documentations
  - performance properties ...
- Had no model that allowed us to relate all the pieces together into a coherent whole
  - now we do...
  - may not solve all problems, but it gets us up the curve...

Consistency of Refinements

- Maintaining the consistency of different representations/concerns is key
  - but this is a collaboration!!
- Refinements provide a way to simplify this problem to the consistency of concerns on a per-feature basis...
- Saying
  "when modularity grows up...
  we'll be talking about refinements"
- More in later lecture ...

Conclusions

- Years of work has taught me that refinements are fundamental to building blocks of software applications
  - took me years to realize that programs are values…
- Ideas are important
  - raise level of modularity from "code" to "design"
  - raise level of programming to the architectural level
  - allows us to reason about applications in terms of their features (as real architects do)
  - structured way to automate the development of complex, efficient software
  - provides us with a broader view of our universe
  - its simple (but it requires you to think differently)

Kysymyksiä?

Questions?

Lecture 1b: Heritage of Refinements

refinements are not new, but were already part of our software design vocabulary...

Background

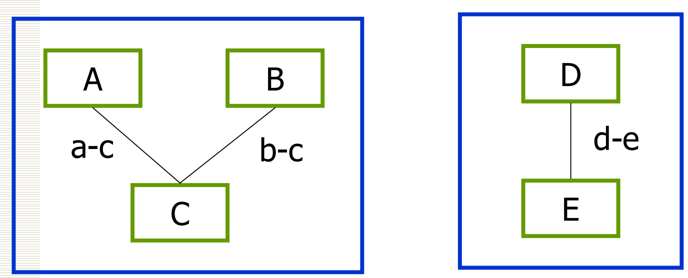
- GenVoca arose circa 1983:
  - legos: idea of components that export and import standardized interfaces taken to logical conclusion
  - outgrowth of **layered designs**
    - each layer adds new functionality
    - or extends existing functionality
- Develop GenVoca ideas from first principles

Hierarchical Software

- Virtual Machines (Dijkstra 1968)
  - design each level of a hierarchical system independently
  - **virtual machine** – operations on level i+1 defined in terms of operations on level i
- Refresh using OO ideas:
  - **OOVM interface** – set of Java interfaces
  - hierarchical design = set of OOVMs, 1 per level

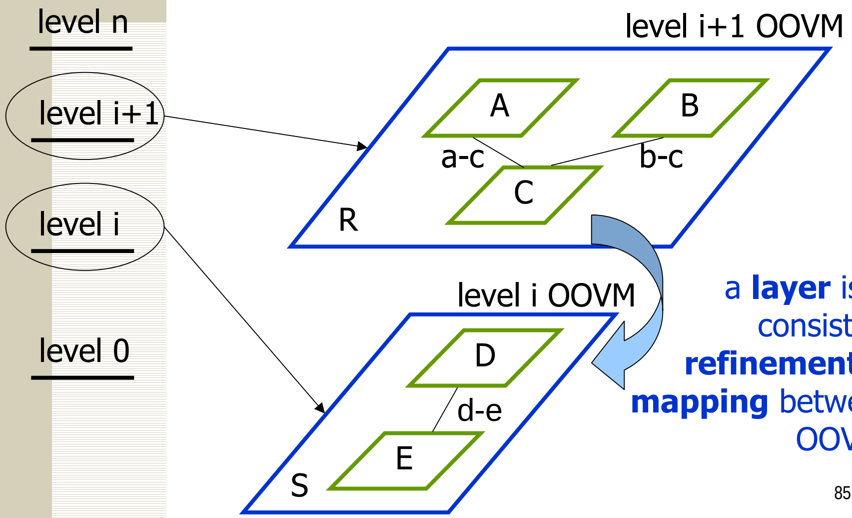
Object Model Notation

- Use E-R like notation (any will do)



object model R   object model S

Hierarchical Designs



a **layer** is a consistent **refinement** or **mapping** between OOVMs

GenVoca – the early years...

- Interface of layer is an OOVM
- Realm is set of all layers that implement same OOVM

S = { y, z, w }

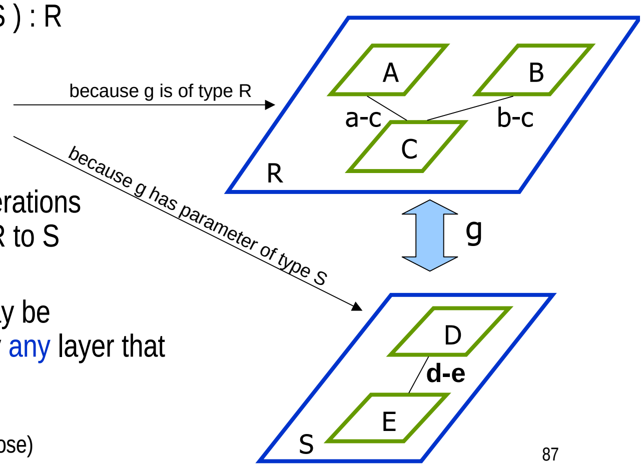
R = { g(x:S), h(x:S), i(x:R) }

- plug-compatible, interoperable, interchangeable
- parameterized layers are functions
- non-parameterized layers are constants

Parameterized Layers

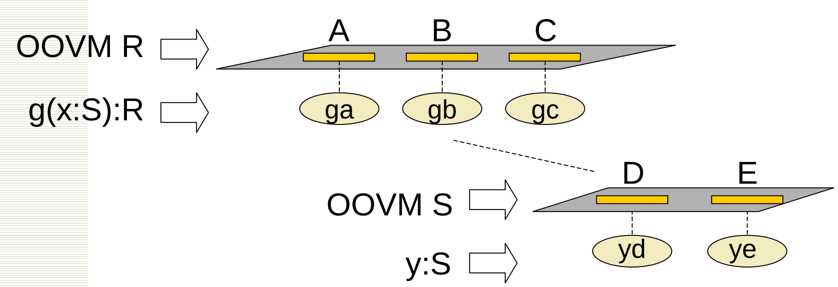
- Consider g(**x**:S) : R
- g exports R;
  g imports S
- g translates operations and objects of R to S
- parameter **x** may be implemented by any layer that implements S

(not quite true, but close)

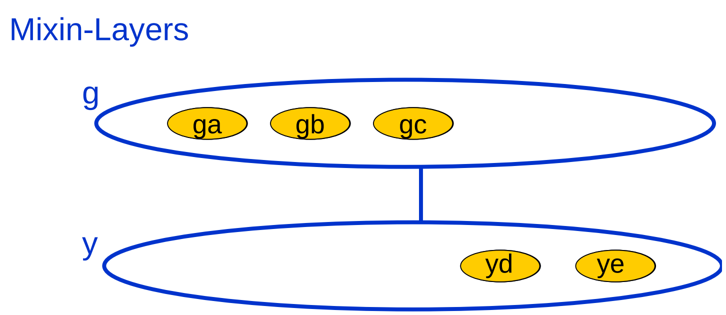


Mixin-Layer Representations

- g(y):R



Mixin-Layer Representations



Applications are Equations

- Equations model abstraction hierarchies
  - type of equation defines interface of resulting application

S = { y, z, w }

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

```
• app1 = y        // implements OOVM S
• app2 = g( w )   // implements OOVM R
• app3 = g( z )   // implements OOVM R
```


Product-Lines and Grammars

- Model = realms and layers
- Realm/Model representation
- Grammar representation:

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**

Symmetric Layers

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

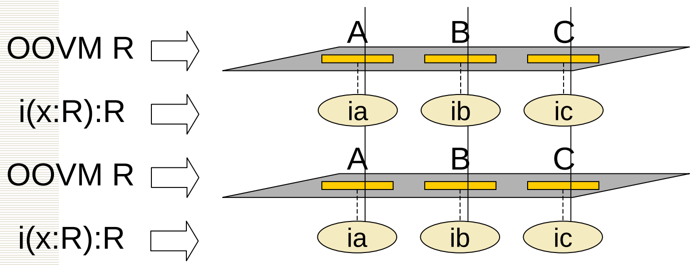
W = { m(x:W), n(x:W), p }

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

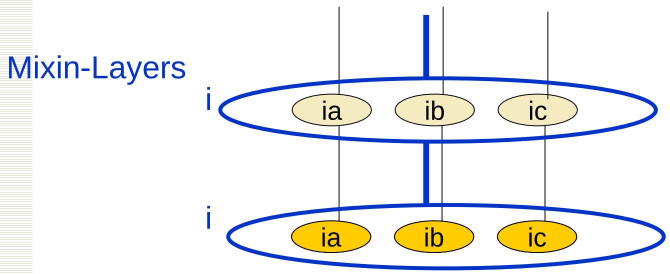
What does Symmetry mean?

- 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
  - seemingly infinite number of such enrichments....
- Experience: **very** common in all domains...
  - should be easy to see…
  - “creeping featurisms”

Mixin-Layer Composition: i(i(x))



Symmetric Layers



Scalability

- Adding a new layer (function, constant) to a realm (model) is equivalent to adding a new rule to a grammar
  - family of applications enlarges exponentially (in the length of the equation)
  - because huge families can be built using relatively few layers (refinements), GenVoca models are **scalable**...

Important Special Cases – COM

- Microsoft's Component Object Model (**COM**)
  - components export and import "standardized" interfaces
  - applications are compositions of COM components
- Differences are vanishing slowly
  - .Net now supports inheritance among COM components
    - » not true "refinement" yet
  - COM components are single (binary) class that exports multiple interfaces
    - – note: not (yet) critical to class of applications we've seen
  - previously not much plug-and-play
    - – only one implementation of interfaces typically
    - – e.g., windows media player
  - **Microsoft's Open Information Model 1998**

Important Special Cases – Aspects

- **Aspects implement refinements**
  - **implement cross-cuts**
- We've implemented the Graph Product Line using **AspectJ**
  - **AspectJ** is flagship tool for **Aspect-Oriented Programming (AOP)**
  - here's how we expressed class refinement
  - note: this is on-going work with Roberto Lopez-Herrejon

Aspects (Cont)

- **AspectJ** has two cross-cut implementations
  - "static" and "dynamic"

```
refines class C {                public aspect aspectName {
   int newVar;          →           int C.newVar;
   void newmeth() {...} →           void C.newMethod() {...}
}                                }
```

refinement   static AspectJ Cross-Cut

Aspects (Cont)

- Refining methods references super

method refinement

```
refines class C {
   void myMethod(int z) {
      // before code
      super.myMethod(z+2);
      // after code
   }
}
```

Dynamic Cross-Cuts of AspectJ

```
public aspect aspectName {

   pointcut override_method(C c, int z):
      target(c) && args(z) &&
      call(void C.myMethod(int));

     void around(C c, int z): override_method(c, z) {
         // before code
         proceed(c, z + 2);  // roughly = to super
         // after code
    }

}
```

target of refinement

how to refine

Aspects (Cont)

- Equation X = A(B(C)) is **AspectJ** call:

  `> ajc C B A` **(order doesn't matter)**

- Composition order not fully defined
  - can linearize order by "dominates" declaration
- Aspects can't add classes that can be subsequently refined...
  - simple work-around

Summary of Special Cases

all (aspects / mixin-layers / COM components / …) are implementations of refinements

not all aspects can be implemented as mixin-layers
not all mixin-layers can be implemented by aspects
not all COM can be implemented by aspects

they are all refinement implementation techniques that have their advantages, disadvantages

And Let's Not Forget...

- Lots of other work and viewpoints on refinements
  - Doug Smith (Kestrel)
  - Jim Neighbor's Draco
    - » program optimizations
  - Ira Baxter's Design Maintenance (CACM'92)
  - ...

Recap Heritage

- Rich (largely forgotten) history of software design related to refinements
  - layers, collaborations are examples of refinements
  - equations model hierarchical systems
  - models of refinements are grammars
  - set of all sentences = language = product-line
  - symmetric layers export and import the same type = recursion in grammars
  - special cases reduce to traditional component models (e.g. COM, CORBA) and nontraditional models (aspects)

Lecture 2: Design Rules and Design Wizards

Don Batory
Department of Computer Sciences
University of Texas at Austin

Three Fundamental Topics

- **Object-Oriented Frameworks and Product-Lines**
  - further insight into power of layers by relating to OO frameworks
- **Composition Validation** – not all eqns are valid
  - impossible for users to debug generated code
  - need automated help to validate compositions
  - **design rules** (composition constraints) are an answer...
- **Automatic Programming** – generation of efficient programs from declarative specs
  - largely abandoned problem now in renaissance
  - equation optimization
  - **design wizards** technology is an answer...

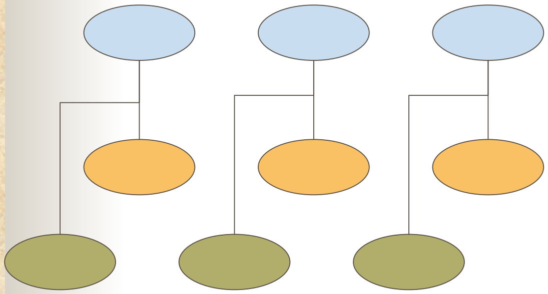
Lecture 2a: Object-Oriented Frameworks and Product-Lines

Cultural Enrichment…

Introduction

- **OO Framework** is a set of abstract classes that encapsulate common algorithms of a family of applications
  - certain methods left unspecified (abstract)
  - a framework is a "code template" – key details are missing
  - **framework instance** provides these details, by supplying concrete class for each abstract class

Frameworks (Continued)



framework with
3 abstract classes

framework instance

another framework
instance

each instance defines another member of an application family

Houston... we have a problem...

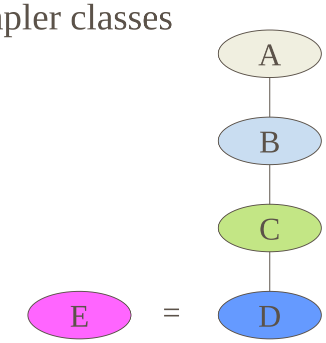
- Delineation between abstract and reusable code from instance-specific code is arbitrary
  - concrete classes of different framework instances can have much in common – e.g., replicate with maintenance problems.
  - abstract classes can have variations – leads to a proliferation of frameworks (with maintenance problems)
- Practical problem:
  IBM's San Francisco Project has seen this happen

Key Problem...

- Product-lines with optional features are not handled well by frameworks
  - over-featuring – a lot of not-entirely general functionality may be in abstract classes
  - replication of code in framework instances
- Our contribution:
  - create a Product-Line of frameworks
  - assemble both abstract and concrete classes of frameworks from primitive and reusable layers
  - eliminate the problem of arbitrary delineation of abstract from concrete

Illustration

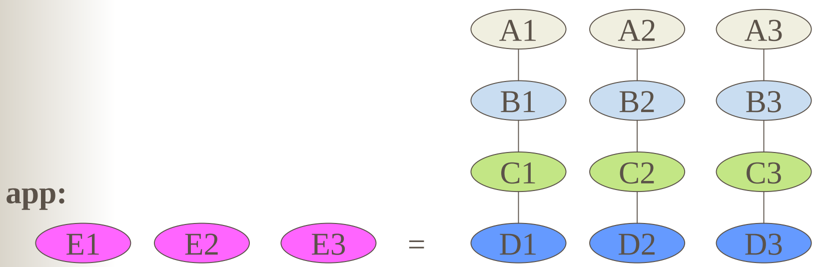
- Recall a fundamental "law" of OO – a class can be decomposed into a linear inheritance chain of simpler classes



always pull a
complex class
apart and express
as compositions of
simpler classes

Collaborations

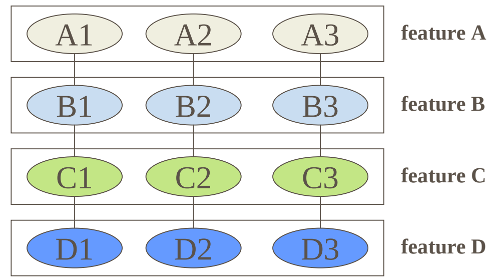
- Scale “law” to multi-class collaborations



pulling classes apart on basis of features that they implement

Collaborations

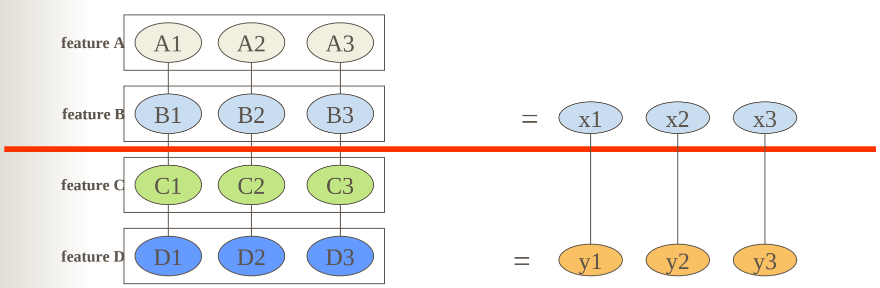
- Each collaboration is a “layer” or “feature”



app = D(C(B(A)))

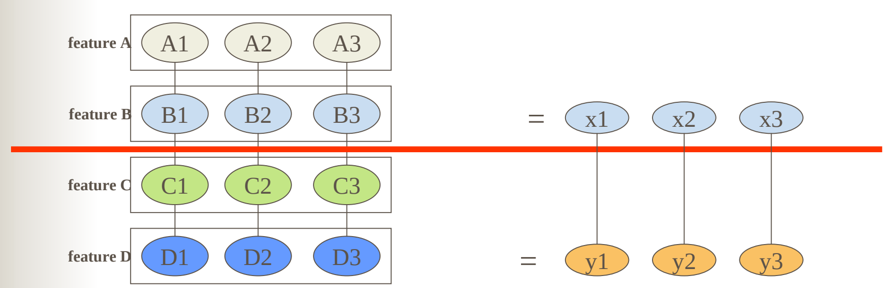
Solution to Framework Problem

- Look how frameworks are interpreted here – abstract above horizonal line, concrete below



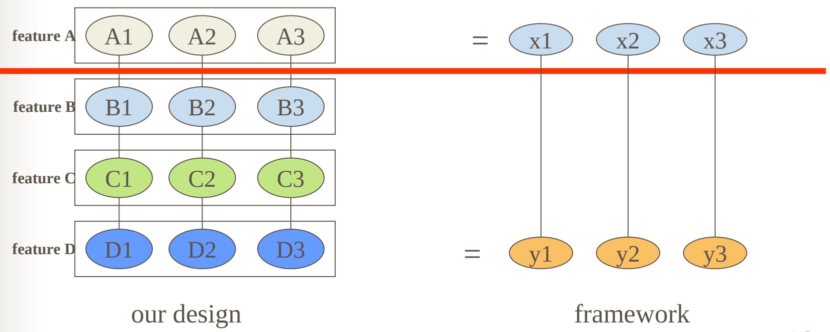
our design framework

Placing Line is arbitrary!

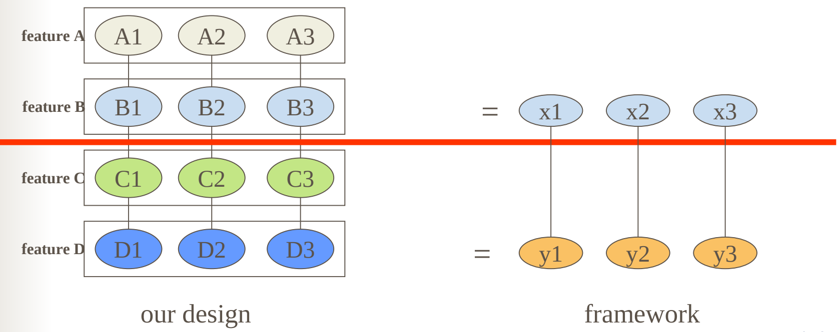


our design framework

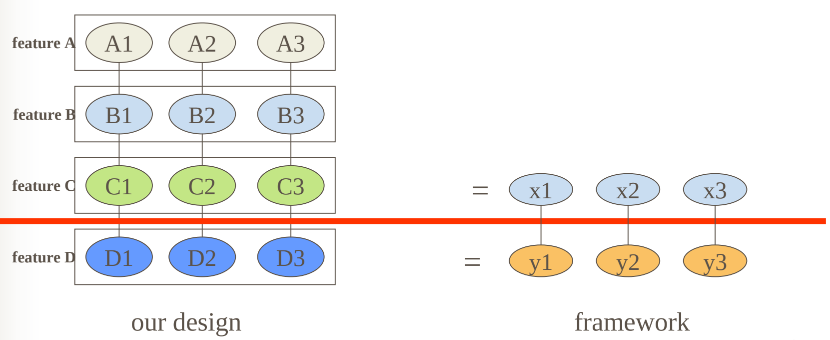
Placing Line is arbitrary!



Placing Line is arbitrary!



Placing Line is arbitrary!



In the Paper…

- Show that collaborations are building blocks of:
  - abstract classes of frameworks
  - concrete classes of framework instances
- Abstract/concrete line always drawn horizontally
  - because framework, instance always implements an integral number of “features”
  - if they weren’t integral, then every framework instance would have the same code (to fill in the part of the feature that was missing)

Example

- Graph Product Line Domain
  - different applications implement different graph traversal algorithms/applications
  - our building blocks:

```
undirected      --   undirected graph
directed        --   directed graph
dft( x )        --   depth-first traversal
bft( x )        --   breadth-first traversal
number( x )     --   vertex numbering
cycle( x )      --   cycle checking
```


Product-Line

- derives from different compositions

```
app1 = number( dft( undirected ) )

app2 = cycle( bft( directed ) )

app3 = cycle( dft( directed ) )

app4 = number( cycle( dft( directed ) ) )

...
```


Frameworks

- A framework is an (inner) expression

```
frame1 = dft( directed )

app4 = number( cycle( frame1 ) )
app1 = number( frame1 )
```

- Framework is expression
- Instances are expressions with same inner expression

Code Replication in Frameworks

- Framework #1:

```
frame1 = dft( directed )
```

- Framework#1 instances

```
inst11 = number( frame1 )
inst12 = cycle( frame1 )
inst13 = number( cycle ( frame1 ) )
```

Framework Proliferation

- Framework #2:

```
frame1 = dft( directed )
frame2 = dft( undirected )
```

  - note: replicated code (dft)

In the Paper...

- We demonstrate freedom to mix-and-match optional features using collaborations
- Building blocks of abstract classes of frameworks as well as the concrete classes of framework instances can be synthesized from primitive and reusable collaborations
- Show corresponding framework – where ever the “line” is drawn – leads to problems outlined earlier

Conclusions

- Frameworks seem ideal for PLA because they encapsulate reusable code in abstract classes
  - fail miserably in common case of optional features
- Reason: frameworks based on inflexible design where relationship between common and application-specific code is fixed
  - using layers provides a more flexible solution

Lecture 2b: Design Rule Checking

how to validate compositions of refinements 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 appear to run for a while!
  - impossible for users to determine 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 automatically applying design rules

But wait!!

- What's wrong with normal type checking?
- Assign types to constants, functions?

S = { y, z, w }

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

- Ensure that all equations are type correct...

Type 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 additionally requires testing semantic constraints
  - that's what DRC is all about

Overview

- DRC is no different than semantic checking performed by compilers
  - not all syntactically correct Java programs are semantically correct…
  - solution: use attribute grammars to define constraints
- Same here: GenVoca model is a grammar
  - **design rules** are grammar attributes
  - DRC algorithms propagate attribute values up and down parse (equation) trees and evaluate constraint predicates

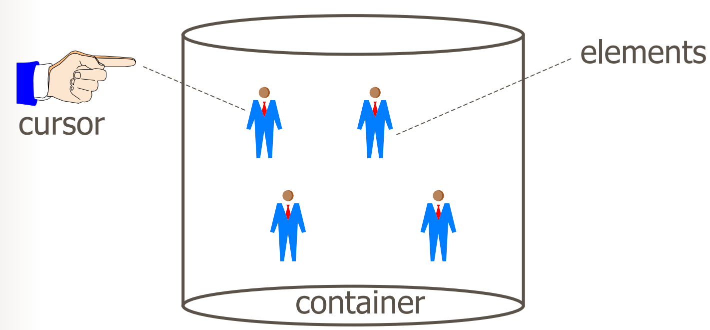
Motivating Example: P3

- Generator of **container data structures (CDS)**
- Extended Java to have embedded **domain-specific language (DSL)** for CDS
  - declarative specs that treat containers as database relations
  - container implementations are composition of P3 components

P3 Model

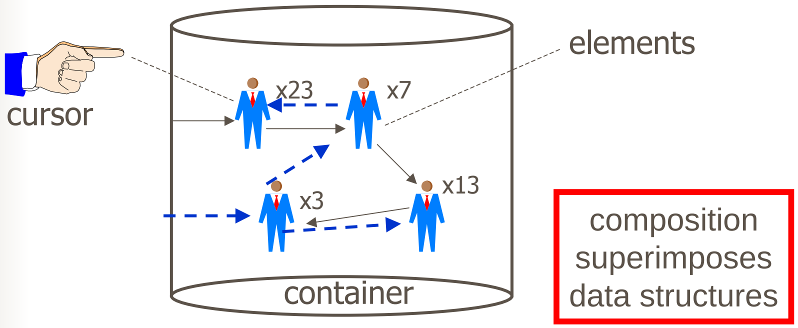
```
ds =   {      bintree( x:ds )        // binary tree
              dlist( x:ds )          // unordered list
              odlist( x:ds )         // ordered list
              avail( x:ds )          // free-list manager
              array( x:mem )         // sequential storage
              malloc(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

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


Data Structures are Equations



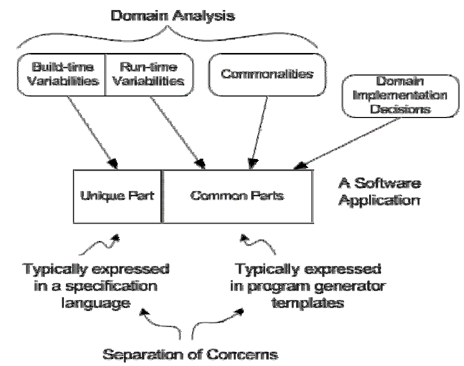
container_eqn =

Data Structures are Equations



container_eqn = bintree(odlist(malloc))

Perspective: Cleaveland's Talk



Construction by Refinement

- Simultaneous refinement of multiple types

		odlist	bintree
element type	data fields	next, prior	left, right
container type	name	first, last	root

P3 Specifications extend Java

- Containers
  - empcont is generated container class of emp instances
  - odlist(age, malloc()) defines its implementation

```
container empcont<emp> using odlist( age, malloc(transient));
```

- Cursors
  - few is a generated cursor class over empcont containers
  - instances retrieve specified container elements

```
cursor few( empcont e ) where dept() = "Computer Science"
                              orderby -age;
```

In Principle...

- Providing declarative, relational database-like specifications for:
  - containers and customized container implementations
  - retrieval (SQL select, update, delete) statements
  - greatly simplifies data structure programming
- And P3 does the hard work:
  - performs query optimization
  - generates efficient code...

P3 (Cont)

- Generates HUGE libraries
  - dwarfs any standard container structure library
  - create useful structures not found in any library
    - with n data structure layers
    - 4 different memory layouts (rand/seq, trans/persist)
    - $2^{(2+n)}$ different structures (ignoring key parameters)
    - >> $2^{(2+n)}$ different structures with key parameters

bintree(bintree(bintree(malloc(transient))))

key A key B key C

Efficient too!

	Dlist	Bstree	Rbtree	Hash
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(x:ds) 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

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

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

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

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

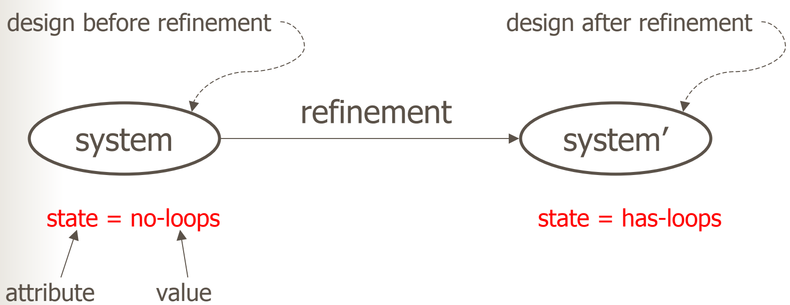
- **Obligations** are conditions that must be satisfied by system that uses the component
  - beyond type checking – requires "action-at-a-distance" – predicates nonlocally satisfied
  - propagated to enclosing module where they are eventually satisfied by some postcondition

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

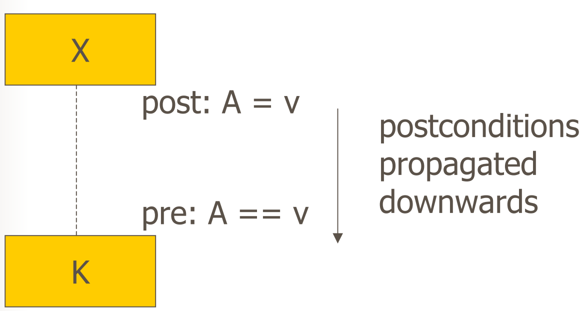


DRC: Adapt Inscape to Layers

- 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, Tracz, Sitaraman)
  - properties exported to “higher” layers not the same as those exported to “lower” layers
  - leads to 2 kinds of design rules

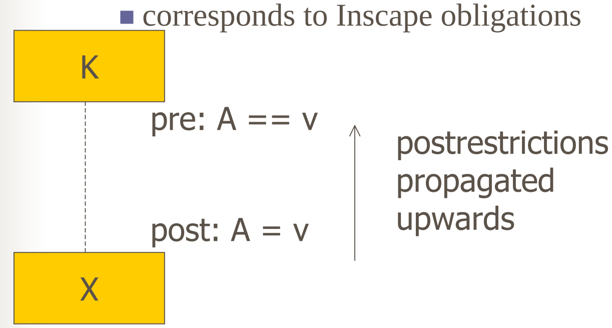
#1: Preconditions

- for layer usage



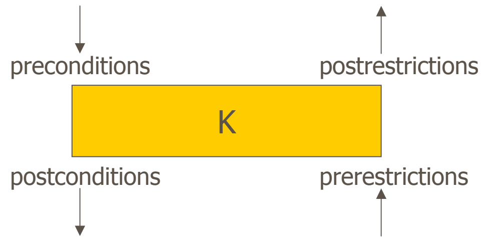
#2: Prerestrictions

- Preconditions for parameter instantiation
  - 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 layer 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

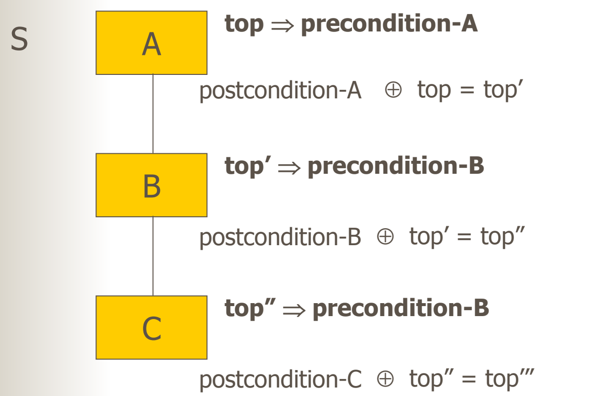
- Postconditions, existing conditions specified by simple predicates
- Predicate composition operator ⊕
  - **Existing** is ¬**A** ^ B
  - **Post** is A
  - **Post** ⊕ **Existing** = conditions after composition
  - (A) ⊕ (¬**A** ^ B) = (A ^ B)

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

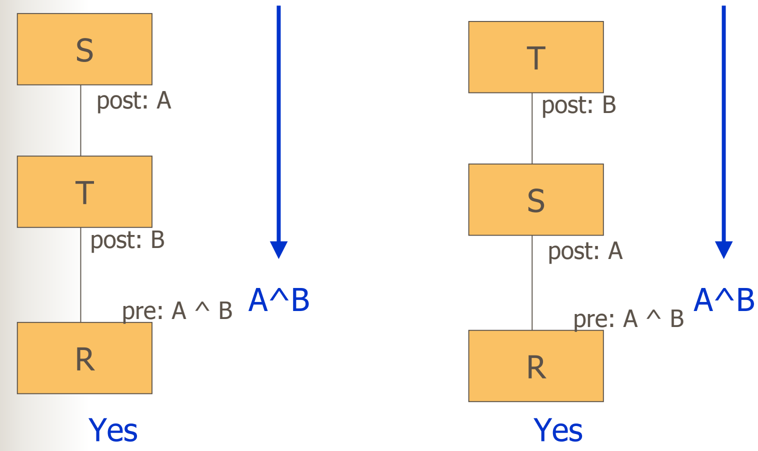


postconditions propagated by ⊕

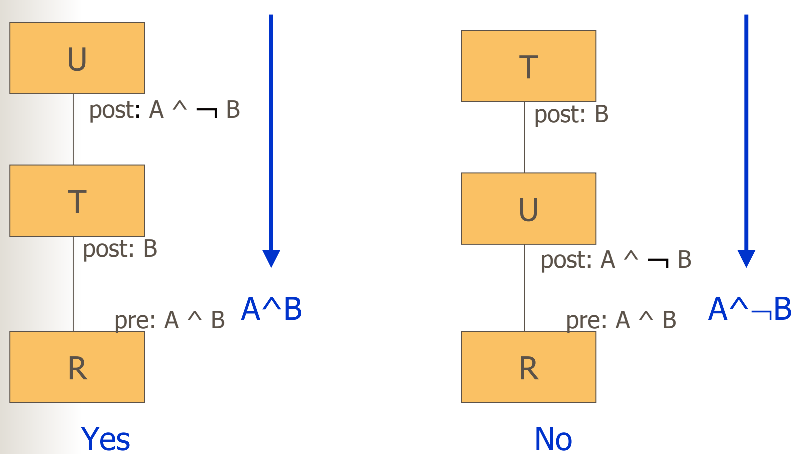
preconditions tested by ⇒

simple recursive algorithm for top-down DRC

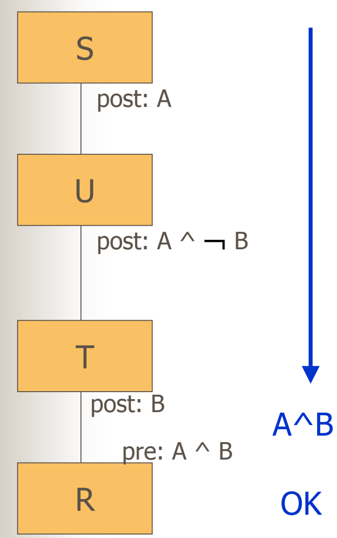
Is Composition Valid?



Is Composition Valid?

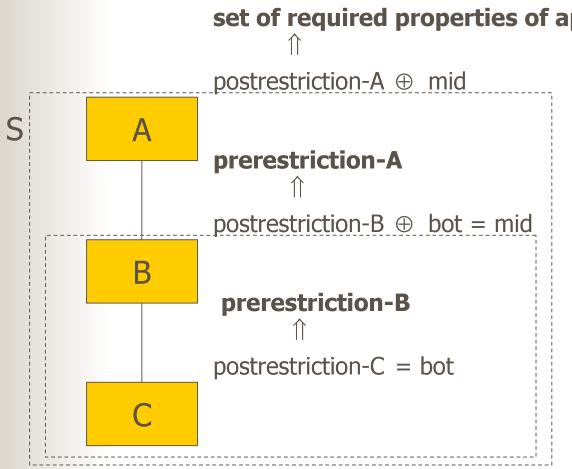


Is Composition Valid?



- 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**

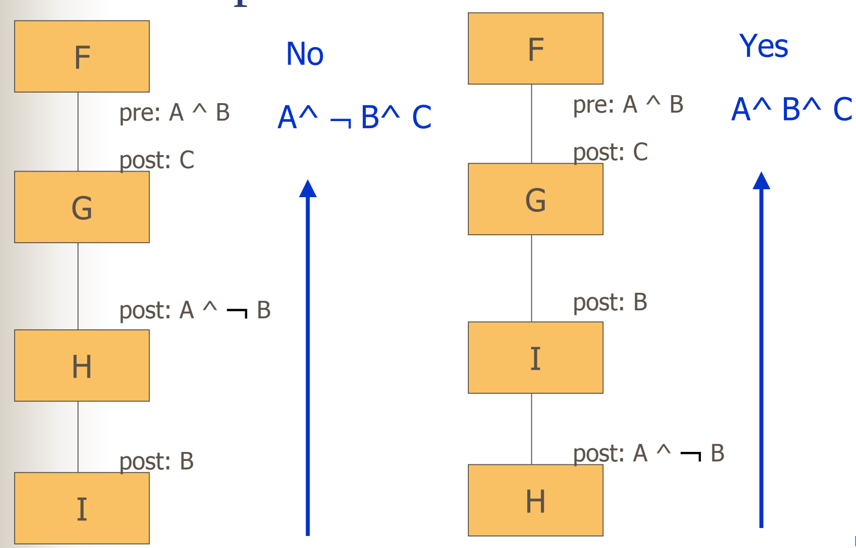
Bottom-Up DRC



same set of operators as before ⊕, ⇒

simple recursive algorithm for bottom-up DRC

Is Composition Valid?



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 – 1500 loc
- DRC algorithm is efficient: O(mn)
  - m = # of attributes
  - n = # of layers

Domain	#Realms	#Layers	#Attributes
Genesis (databases)	9	52	14
FSATS	1	25	41
P3 (data structure)	3	50	7

Design Rule File for P3

```
properties = {
        logical_key        "a logical-key-ordered layer"
        retrieval          "a retrieval layer"
        inbetween          "a layer needed for element deletion"
        mark_delete        "a layer that marks elements deleted"
}

# Here are layer signatures and design rules.

bintree( ds ) : ds {
        assert above { retrieval   logical_key }
        require above { inbetween }
}

array( mem ) : ds {
        require above  { mark_delete }  // mark-delete layer required above array

        assert above    { retrieval }      // assert array is retrieval layer
        assert below    { retrieval }      // to all descendents and ancestors
}
```

signature

design rules

Big Picture – DRC Composition

```
bintree( ds ) : ds  {
        assert above { retrieval  logical_key }
        require above { inbetween }
}
```

DRC for bintree(array(x)):

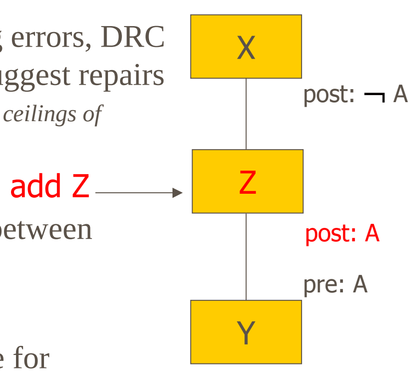
```
bintree-array( mem ) : ds {
        assert above { retrieval  logical_key }
        require above { inbetween,
                        mark_delete }
        assert below { retrieval }
}
```

```
array( mem ) : ds  {
        require above  { mark_delete }

        assert above     { retrieval }
        assert below     { retrieval }
}
```

Composition algorithms specific to DRC representations

Suggesting Error Corrections

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



Example

- Want container that stores elements onto a binary tree whose nodes are stored sequentially in transient memory. 1st try:

```
                mark_delete
first = top2ds( bintree( array( transient ) ) )
              inbetween
                qualification
```

- DRC response:

```
precondition errors:
     an inbetween layer is expected between top2ds and bintree
     a mark_delete layer is expected between top2ds and array
prerestriction error:
     top2ds expects a subsystem with a qualification layer
```

Example (Cont)

- Clumsy fix:

```
second = top2ds( inbetween( bintree( qualify(
          mark-delete( array( transient ) ) ) ) ) )
```

- DRC response

```
precondition error:
    a retrieval layer (bintree) not expected above qualify
```

- Correct equation – swap qualify and bintree

```
third = top2ds( inbetween( qualify( bintree(
         mark-delete( array( transient ) ) ) ) ) )
```


Insights

- DRC directs users to modify eqn to the “nearest” correct eqn in space of all eqns
  - generally is what you want
- Why isn’t DRC a challenging problem in program verification?
  - solution unlikely to be automatable, forget about efficiency
- Inscape work and our own have observed
  - problem is straightforward
  - solution is automatable AND efficient! but WHY?

Reason #1

- #1: Shallow consistency checking goes long way
- Most design errors are shallow
  - conjecture: all errors at layer/refinement composition level are shallow
- Remaining errors must be dealt with by layer (refinement) implementers

Reasons #2, #3

- #2: Modeling states of program design (not execution) vastly reduces number of properties to examine
- #3: GenVoca is a methodology for creating reusable designs as refinements
  - it really works well

The Key

- What makes OO designs so powerful and attractive?
  - Ans: ability to manage and control software complexity
- Standardization is a powerful way of managing and controlling software complexity in product-lines

The Key (Cont)

- Standardization makes problems tractable that would otherwise be very difficult
  - ex: composing COTS components (Garlan's Architectural Mismatch paper)
  - composition is simple in GenVoca
  - standardization seems to limit the ways in which refinements can constrain each other's behavior
    - makes DRC tractable
  - historical perspective... (eigenvectors)

Additional Insights

- Understanding software in terms of implementation-independent refinements:
  - enhances power of DRC
  - DRC tells you whether two refinements (features) can be composed **regardless of how they are implemented**

    ex: bintree(encrypt(...)) may be correct
    ex: encrypt(bintree(...)) is never correct
  - design rules define the compatibility of features
  - if it was harder, architects couldn't design, people couldn't program...

Recap of DRC

- Fundamental problem in architectures is consistency of component compositions
- Simple, automatic, and efficient algorithms for validating consistency of GenVoca equations
  - GenVoca models are grammars
  - design rules are attributes of this grammar
    - express semantic compositional constraints
  - DRC worked well in every domain we've encountered...

Assignment

- Try example problem in back of notes!!

Kysymyksiä?

Questions??

Lecture 2c: Design Wizards

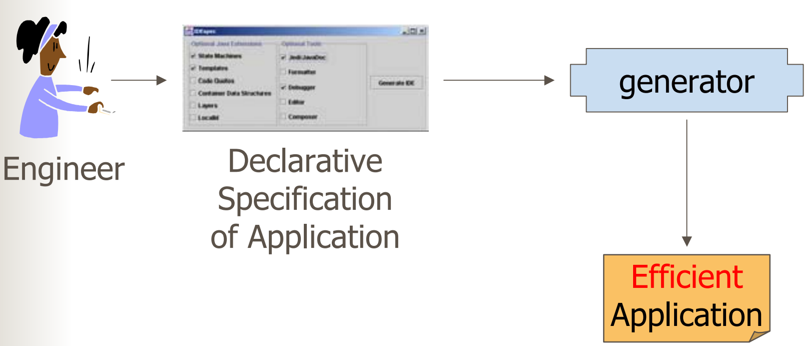
Resurrecting

Automatic

Programming

Automatic Programming

- Holy grail of Software Engineering, Artificial Intelligence



Perspective

- Domain-specific generators like P3 will be common
  - specify application by declaratively listing required features
  - no code to write!
- A user of this technology is confronted with:
  - generator, well-stocked library of layers, features
  - papers, results demonstrate power of approach
  - benchmarks on how much better it is than hand-written code...
- But...

Problems Arise Quickly...

- What to do next...?
  How to solve my problems?
  - need help in selecting features/layers
  - need expert guidance in application design
    - generators don't help us here...
    - also problems inherent in software design anyway

Fundamental Problems

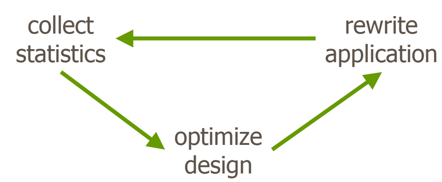
- Designers generally don't have full knowledge of application's use
  - P3 – will know queries (from cursor declarations), but not frequency of execution
  - need to guess at actual workload
- Even if workload is known, can be challenging to infer efficient design
  - example...

Guess the Best Data Structure!

- Easy if workload is simple:
  - access elements that satisfy query: N = = value
- Hard for slightly more complex workloads:
  - 20,000 elements
  - 3000 elements inserted/deleted per period
  - N = = value1 && A = = value2 : 2000 times per period
  - all elements retrieved in S order : 60 times per period
  - what data structure would be best?

Manual Solutions Costly

- Cycle:



  - requires lots of sophisticated programmer support
  - very costly
  - few cycles ever performed

  "if it isn't broke, don't fix it..."

Future Solution: Automation

- Automate steps and close loop
  - program monitors itself
  - program initiates self-evaluation, self-optimization
  - program initiates self-regeneration

self-adaptive software

- **Design Wizard** is tool that performs this optimization

Optimization of Equations

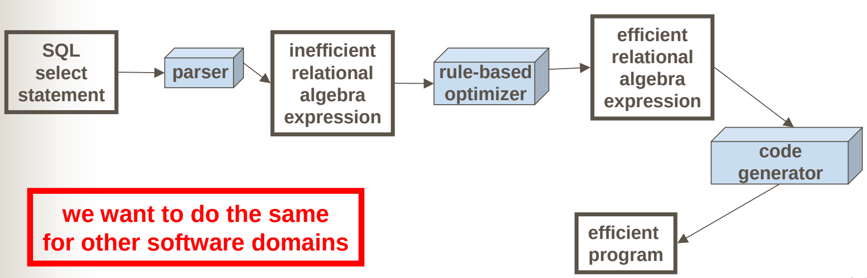
- We express application design and implementation as an equation:

  application = a(b(c))

- How to deduce an efficient equation for a given workload?
  - knowledge typically not present in domain models
  - not same as "design rules"
  - want **rules for optimization**, not **rules for correctness**

Relational Query Optimization

- Classic example of automatic programming:
  - declarative query is mapped to an expression
  - each expression represents a unique program
  - expression is optimized using rewrite rules
  - efficient program generated from expression



we want to do the same for other software domains

Use Same Paradigm In Other Domains!

- **P3 is a case study**
  - space of all equations given by P3 model + design rules
  - must additional information:
    - develop cost model that estimates efficiency of design (equation) for given workload
    - rewrite rules tell us WHEN to use particular layers/features

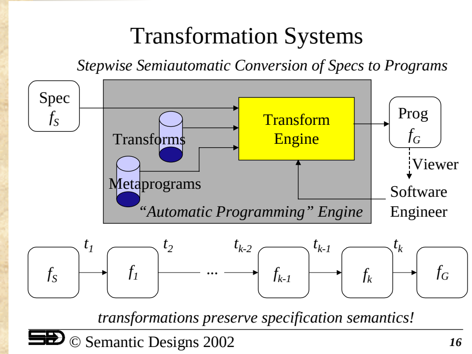
odlist(x) ⇒ bintree(x)

; replace ordered doubly-linked list with bintree

; if both random and ordered key access are needed

- search space for equation that is the cheapest

Perspective: Baxter's Talk



$T_2(T_1(x)) \Rightarrow T_0(x)$

$F_g = T_k(\ldots T_3(T_2(T_1(F_S)) \ldots)$

$F'_g = T_k(\ldots T_3(T_0(F_S)) \ldots)$

My "rewrite rules" are on the above equation

Perspective

- Proposing a theory of software architecture design based on large scale refinements
- If application designs truly are equations, we should be able to optimize them
- If we can optimize equations, we can achieve a level of automatic programming

Upcoming Slides

- Show how automatic programming is possible
- Design Wizard for P3
  - P3 Workload Specifications
  - Cost Model
  - Space of P3 Equations
  - Automatic Optimization of Equations
  - Automatic Critique
  - Conclusions

P3 Workload Specification

- Data structure optimization well-studied
  - relational DB optimization
  - late '70s and early '80s research
- Workload characterized by:
  - type and cardinality of element attributes
  - frequency of each cursor & container operation

P3 Workload Specification

```
cardinality = 10000;

element = {
#       ID      TYPE        CARDINALITY
#----------------------------------------------------------------------------
        name    String      10000;
        age     int         60;
}

workload = {
#       CATEGORY        FREQUENCY
#------------------------------------------------------------------
        insertion                   300;
        deletion                    300;
        ret orderby name            100;
        ret where name == “Don” && age > 20  orderby age     200;
}

Equation = odlist(age, malloc());
```

starting equation

Performance Model

- Given equation E and workload W: how do we compute cost(E,W)?
  - assign a “rank” to evaluate equations
- Ans: create a performance model for each layer
  - foreach layer L, we have performance model L_p
  - given equation

$$E = X(\ Y(\ Z\ )\ )$$

we **compose** its **performance model**

$$E_p = X_p\ (\ Y_p\ (\ Z_p\ ))$$

Big Picture

- Following slides:
  - illustrate traditional approach to performance modeling in databases, data structures
  - different domains have their own approach, techniques for performance modeling which would require their own adaptation to this organization
  - case study to show how to compose performance models in domain of data structures

Performance Model

- Follows classical database research
  - sum of costs of processing each cursor, container operation times frequency of execution

$$Cost(E,W) = I(E) \times InsFreq + D(E) \times DelFreq + \sum_{i \in W} \left(U(E, Field_i) \times UpdFreq_i\right) + \sum_{i \in W} \left(R(E, Ret_i) \times RetFreq_i\right)$$

- now how to compute I(E), D(E), ... ?

Performance Model (Cont)

- Computed per equation E

$$I(E) = \sum_{i \in E} insertionCost(layer_i)$$

$$D(E) = \sum_{i \in E} deletionCost(layer_i)$$

$$U(E, Field_j) = \sum_{i \in E} updateCost(layer_i, Field_j)$$

$$R(E, Ret_j) = Min_{i \in E}(retrieval(layer_i, Ret_j))$$

- What is insertionCost(...) .... per layer?

Aspect Performance Model

- Elementary analysis of each data structure
  - cost equation for each operation
  - c is a layer-specific constant

Layers	insertion	deletion	update	equality retrieval	range retrieval	scan retrieval
dlist	*c*	*c*	*c*	*c*n*	*c*n*	*c*n*
rbtree	*c*log(n)*	*c*log(n)*	*key: c*log(n)* *non-key: c*	*key: c*log(n)* *non-key: c*n*	*key: c*log(n)* *non-key: c*n*	*c*n*
hash	*c*	*c*	*key: c* *non-key: c*	*key: c(n/b)* *non-key: c*n*	*c*n*	*c*n*

- Now, how to find a good equation E??

Space of P3 Equations

- P3 layers characterized by 3 kinds of attributes:
  - properties – classify layers/features
  - signatures – specify realm membership, parameters
  - design rules – composition constraints
- Design Rule File (previously shown) specifies all of this

Design Rule File (again)

```
properties = {
        logical_key        "a logical-key-ordered layer"
        retrieval          "a retrieval layer"
        inbetween          "a layer needed for element deletion"
        mark_delete        "a layer that marks elements deleted"
}

# Here are layer signatures and design rules.

bintree( ds ) : ds {
        assert above { retrieval   logical_key }
        require above { inbetween }
}

array( mem ) : ds  {
        require above  { mark_delete }  // mark-delete layer required above array

        assert above    { retrieval }    // assert array is retrieval layer
        assert below    { retrieval }    // to all descendents and ancestors
}
```

Space of P3 Equations

- Graph G = { V, A }
  - V is set of valid equations that can be composed with given layers
  - A is set of arcs – connects equation **x** with **y** if there is a rewrite rule that transforms **x** into **y**
- So what are the rewrite rules?

Rewrite Rules

- Derived from analysis of personal use
  - we analyzed our own thought patterns to deduce equational rewrite rules for the P3 model
- When rewrite is attempted:
  - resulting equation had to be valid
  - cost of resulting equation was unchanged or lowered
  - if both hold, result is kept
  - greedy search heuristic …

Example Rules

- Some rewrites about element attributes
  - if element attribute A is listed as an order-by key in the workload specification, then try to insert a logical_key layer (e.g., rbtree or ordered-list) with A as its key

    else
  - try to replace the logical_key layer with A as its key with a more efficient logical_key layer
- Note: we use design rule file to identify layers that assert logical_key property

Another Rewrite Rule

- If element attribute `A` is used in an equality retrieval predicate (e.g., `A == 'Don'`) then try to insert a hash_key layer with A as its key

  else
- if there already exists such a layer, try to substitute it with a more efficient hash_key layer

Optimization

- Run to fix-point

```
foreach element attribute A {
  apply each "attribute growth" rewrite for A
}

apply each "non-attribute growth" rewrite
apply each "shrink" rewrite
```

- Guarantees finding a local minimum
- No guarantees for global minimum
  - general problem is NP-hard

P3 Workload Specification

```
cardinality = 10000;

element = {
#        ID        TYPE              CARDINALITY
#---------------------------------------------------------------
         name      String            10000;
         age       int               60;
}

workload = {
#        CATEGORY          FREQUENCY
#-----------------------------------------------------
         insertion                   300;
         deletion                    300;
         ret orderby name            100;
         ret where name == "Don" && age > 20 orderby age    200;
}

Equation = odlist(age, malloc());
```


Critique

```
Original Equation is: odlist(age, malloc( ))
cost = 19593

Equation P3 Wizard recommends is :
    hashcmp(name, hash(name,5000, odlist(name, malloc( ))))
cost = 1606

Projected improvement: 1119%

Reasons why we choose this type equation:

   hashcmp: field name is hashed because it will be faster to
      compare the values of two string fields when they are hashed.

   hash: A hash data structure with hash key name is used because
      11% of the operations involve equality retrieval on name.

   odlist: A doubly linked list ordered by name is used because
      many retrievals will be ordered by name.
```


Analysis

- Original container implementation inefficient
  - store elements on list in age order
- Suggested design:
  - fast access to elements via name using hashing
  - elements stored on list in name order
  - using hashcmp where predicates like name="Don" are replaced with hash_of_name=hash("Don") ^ name="Don" speeds up searches
- Suggested design is not immediately obvious
  - tedious to implement by hand
  - easy for P3 to do it

Big Picture

- Equation synthesis is precursor to self-adaptive software
  - wizards will be critical in “closing” the loop that will help automate certain forms of software maintenance
- Not all users of generators will be domain-experts
  - wizards will help avoid blunders, find better implementations of target systems automatically

Conclusions

- First example of Design Wizard
  - can be generalized to other domains
    - typically uncommon – most domains have only one implementation of a feature, so there’s little to optimize
    - in principle, it always arises when there are multiple implementations of a feature
  - **substantial** improvement over previous work (ex. SETL, AP5, Mitoma’s Optimizer)

Perspective: Baxter’s Talk

- I disagree!
- Counter examples
  - Relational optimizers
  - Data Structure Design Wizard
- Why?
  - possible to find abstraction level for specifications that can be implemented automatically – collaborations/features
  - level at which architects reason

Fully Automatic Programming? NO!

- Problems:
  - Impossible to find abstraction level for specifications that can always be implemented automatically
    (Gödels incompleteness theorem)
  - Unsuitable notation to describe problem
    (who implements the AP engine for “my” problem domain?)
  - Limited control over performance of implementation
    (why does the ∈-test on sets need linear time in the size of the set?)
    (why doesn’t yacc produce COBOL code?)
- Solution:
  - Use highly configurable semi-automatic engine

 15

Conclusions

- Self-adaptive software is important topic
  - adding more automation to generative programming
  - attempt to have software maintain itself
  - we’ve shown relationship of self-adaptive software to generators and equation-rewriting technologies
  - start on a promising line of research

Kysymyksiä?

Questions??

Lecture #3: Scaling Refinements

Don Batory
Dept of Computer Sciences
University of Texas at Austin

Lecture 3a: The AHEAD Model

Don Batory
Dept of Computer Sciences
University of Texas at Austin

Requests from Yesterday…

- Want to see real examples
- Want to see future directions
  - this is how we are building FSATS
  - how we now view the world of software… (significantly altered my understanding of my own work…)
  - first presentation of these ideas outside Austin
- Want to see architectural models
- Want to see tools…

State of Art

- Emphasis on application synthesis using refinements focuses largely on generation of:
  - source code
  - individual programs
  - a GenVoca eqn = source code for single application
- Code synthesis alone inadequate for building complex systems of today and those of tomorrow
  - scale to multiple programs
  - systems are program suites – client-servers, MS Office
  - scale to multiple representations
  - code, makefiles, documentation, performance models,…

Scaling Refinements & Generators

- Challenge is not HOW
  - lots of ad hoc ways to do this
  - challenge do so in principled manner, so that generators are not ad hoc collection of tools and a patch work of techniques
- Generators are technological proof
  - that software in a domain has been simplified to point that its development can be automated
- Don't want complexity to shift from systems that are generated, to generators themselves
  - controlling the complexity of generators, like the systems they produce, is a fundamental problem

This Lecture

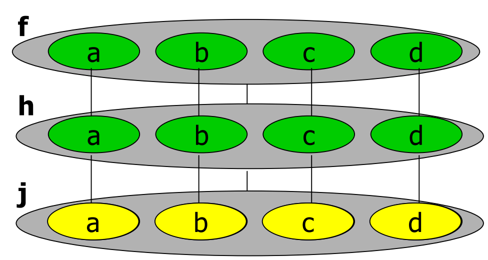
- Presents two fundamental results on refinement scalability and modularity:
  - **AHEAD** – **A**lgebraic **H**ierarchical **E**quations for **A**pplication **D**esign
    - architectural model and tool suite for scaling refinements to multiple representations, programs
  - **AHEAD** tool demonstration
  - Scaling Refinements to Product Families
    - scaling to multiple programs

Preliminaries

core problems that motivate a generalization of GenVoca

#1: Code Representation

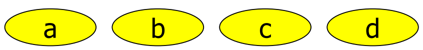
- Engineers, Programmers: this is weird...



Always instantiate bottom-most classes; never intermediaries

#1: Code Representation

- What engineers want is this:

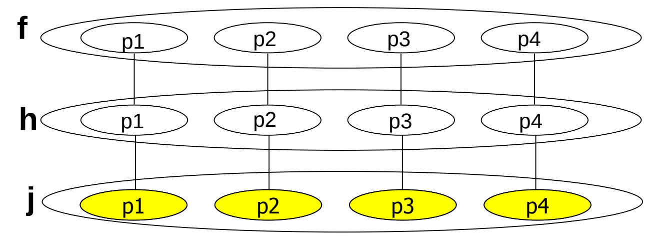


Generate only bottom-most classes; never intermediaries

Flatten refinement hierarchies!

#2: Scale to Refinements to Multiple Programs

- How to express that a single refinement modifies (cross-cuts) multiple programs
  - briefly….



#2: Scale Refinements to Multiple Programs

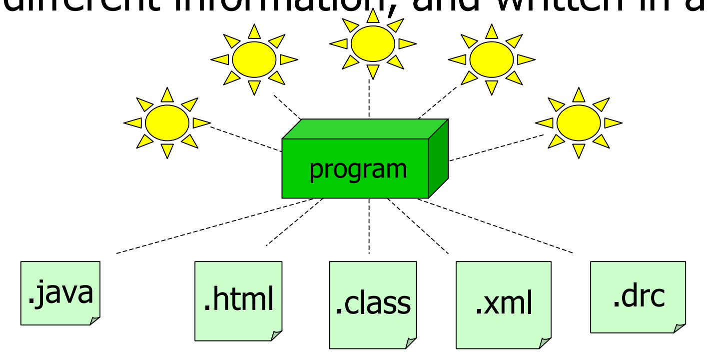
- More complicated than this…
- "Origami" is an extension of GenVoca that solves this problem
- Talk about later if time…
  - AHEAD subsumes Origami

#3: Non-Code Representations

- Architects use multiple models to design systems
  - fact: no single representation is adequate to capture all information about a design
    - can't express everything in Java
  - fact: different documents/artifacts capture different information or concerns
    - manuals, code, makefiles, performance models, etc.
    - each is expressed in its own **DSL** (HTML, XML, Java, DRC…)
- Generate non-code representations... but how?

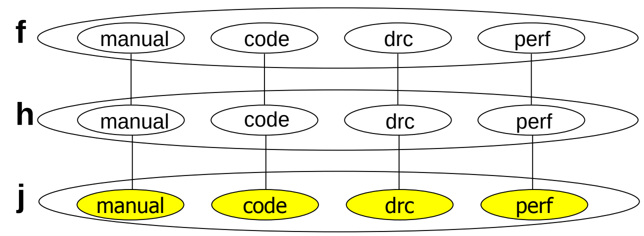
Recall Insight

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



Recall Insight

- When a feature is added to a program, all of its representations may be modified
  - recent Ph.D. by Jeff Gray @ Vanderbilt



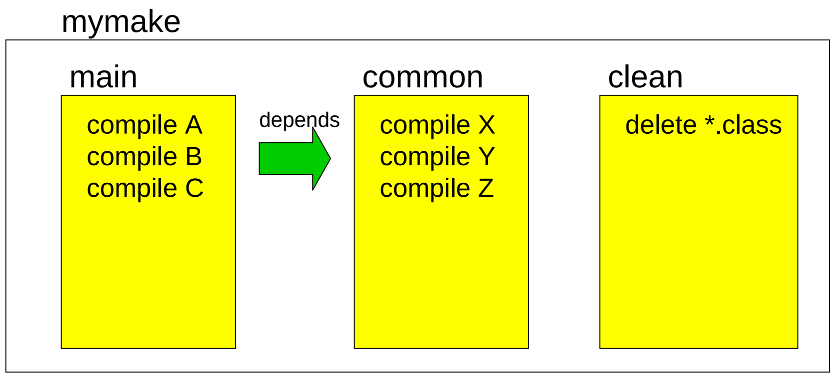
We've done this before...

- Design Wizards
  - from an equation, we compose:
    - design rules (to verify compositions)
    - performance models (to evaluate compositions)
    - code (to generate compositions)
- JTS
  - from an equation, we compose:
    - grammar files (to generate parser)
    - layers (to generate code for preprocessor)
- But how to compose non-code representations?
  - what are principles that can guide us?

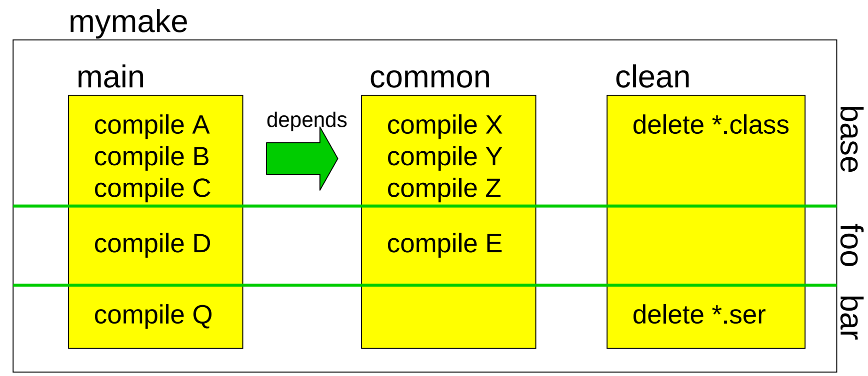
Example: Makefiles

- Instructions to build parts of a system
- 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?
  - how do we compose makefile refinements?

Makefile



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”>     →     static void main
      <compile A>                           { ...
      <compile B>
      <compile C>
   </target>                                }
   <target common>                    →     static void common
      <compile X>                           { ...
      <compile Y>
      <compile Z>
   </task>                                  }
   ...                                      ...
</project>                               }
```


Makefile Refinement is Inheritance!

```
<project myMake>                        <subproject myMake>
   <target main depends=“common”>          <target main>
      <compile A/>                            <super main/>
      <compile B/>                            <compile D/>
      <compile C/>                         </target>
   </target>
   <target common>                         <target common>
      <compile X/>                            <super common/>
      <compile Y/>                            <compile E/>
      <compile Z/>                         </target>
   </target>                               ...
   ...                                  </subproject>
</project>
```

BASE FOO

Foo (Base)

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

added
as result
of composition

note: we're flattening
refinement hierarchies,
like previous slide...

Guiding Principle

- For structuring and refining non-code artifacts
  - create analog in OO representation
  - express refinements in terms of inheritance
    (could be more sophisticated, but OK for first pass)
  - composition flattens inheritance/refinement hierarchies
- Principle of Artifact Uniformity
  - treat all artifacts equally, as objects or classes
  - refine non-code representations same as code representations

Big Picture

- Most artifacts today (HTML, XML, etc.) have **or can have** a class structure and thus are **object-based**
- **Not object-oriented** – there is no inheritance relationship among files
  - what's missing are inheritance (refinement) operators for non-code artifacts
  - should be able to refine any kind of artifact
- Requires tools to add inheritance (refinement) relationships among file types
  - not all (e.g. MS Word)

#3: Unification

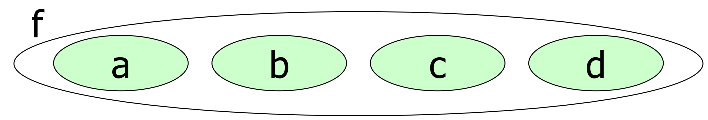
- What is an elegant model that unifies and generalizes these ideas?
  - GenVoca
  - squash refinement chains
  - refine multiple programs (Origami)
  - refine multiple representations
  - Principle of Artifact Uniformity

Core Ideas

AHEAD
Algebraic **H**ierarchical **E**quations
for **A**rtifact **D**esign

Equations

- Every mature science and engineering discipline is driven by equations except software design
  - we can change this...
  - consider GenVoca constants...

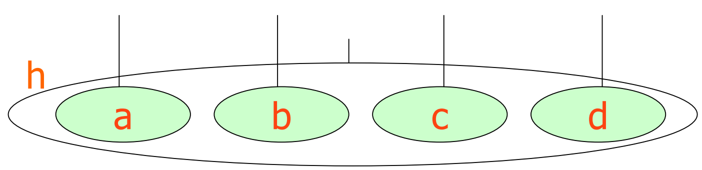


$f = \{ a, b, c, d \}$

constant f is
a set of constants

Equations (Cont)

- GenVoca functions are sets too!



$h = \{ a, b, c, d \}$

function h is
a set of
functions

Equations (Cont)

- Composition is governed by equations!

$$h \circ f = \{ a, b, c, d \} \circ \{ a, b, c, d \}$$

$$= \{ a \circ a,\ b \circ b,\ c \circ c,\ d \circ d \}$$

Note:
shift in
notation
$h(f) = h \circ f$

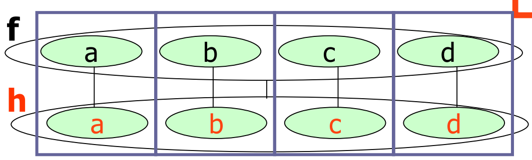
- Pairwise composition by name
  - exactly same rules as mixin-layer/inheritance composition

Equation Semantics

$h \circ f = \{ a, b, c, d \} \circ \{ a, b, c, d \}$

$= \{ a \circ a, b \circ b, c \circ c, d \circ d \}$

Every expression defines an artifact to build.

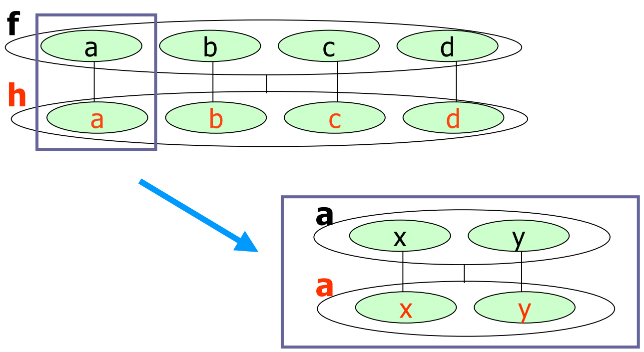


AHEAD Terminology

- Set is a collective of units
- Unit is a:
  - constant
  - function
- Model is another name for a collective

Scalability Through Recursion

- Any constant, function may be a collective



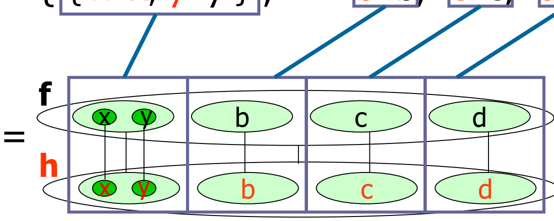
Expressed Mathematically

$h \circ f = \{ a \circ a, b \circ b, c \circ c, d \circ d \}$

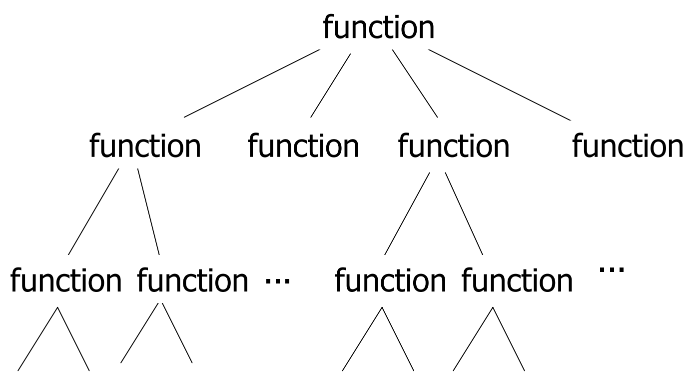
$= \{ \{ x, y \} \circ \{ x, y \}, b \circ b, c \circ c, d \circ d \}$

$= \{ \{ x \circ x, y \circ y \}, b \circ b, c \circ c, d \circ d \}$

=



What Equation Hierarchies Mean



Composing refinements composes all their representations

Scalability

- Treat all levels of abstraction the same
  - yields powerful algebra for application specification
- Nest programs arbitrarily deep
  - sets of programs
    - distributed system (FSATS)
  - sets of sets of programs
    - system of systems
- Nest representations arbitrarily deep
  - code libraries
  - document libraries
  - etc
- All represented by hierarchical equations

Scalability (Cont)

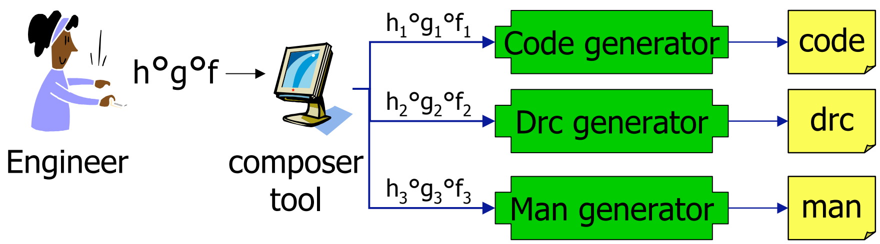
- There are LOTS of other operators, besides °, for collectives and units
- Collective, unit are objects
  - manipulated by a rich set of methods
  - each method is a tool of IDE
- Rich algebra associated with collectives

More Generally

- Expressing mathematically what OO languages do now for refining code
  - GenVoca eqn = code representation of one program
  - AHEAD eqn = multiple representations of multiple programs
- Advance:
  - equations work for all representations
  - equations scale…
  - by imposing uniformity, we control the complexity of generators, and systems they generate

Important – Simplifies Tools!

- **Generator Scalability**
  - don’t have one big generator
  - use simple artifact-specific generators coordinated by a composer that submits equations to them



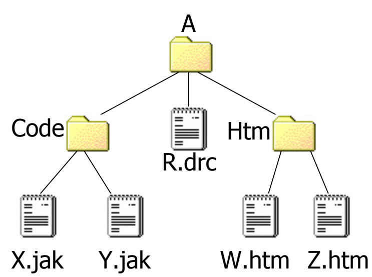
How to Implement AHEAD?

Collective = Directory!

A = { Code, R.drc, Htm }

Code = { X.jak, Y.jak }

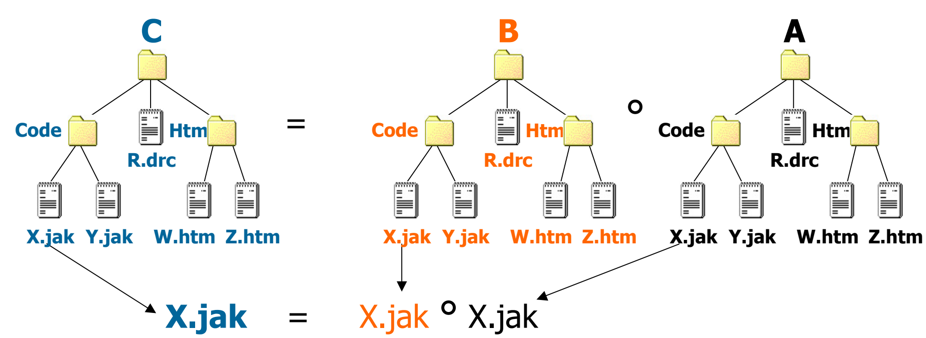
Htm = { W.htm, Z.htm }



A refinement, and all of its representations, is a directory

Composition

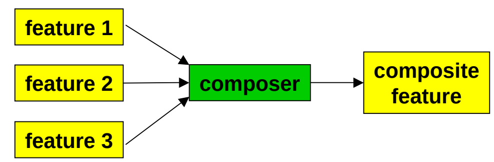
- feature composition = directory composition
  - produces directory isomorphic to inputs



Tools built using JTS

Composer Tool

- Composes Features
  - takes equation as command-line input
  - internally, recursively expands equations
  - creates composite feature directory
  - invokes artifact-specific-composition tools



Artifact Composition Tools

- Most interesting are .jak tools (next slides)
- For non-.jak files:
  - XC – composes .html files
  - VM – composes velocity files to produce ant build.xml makefiles
  - Equation – composing equation files
  - DRC – composing .drc files
- Soon to appear tools:
  - Grammar composing tools (to bootstrap JTS)
  - MSC – composing message sequence charts
  - ...

Code Files are .jak Files

- Constant

```
aspect A;

import java.util.*;

class myClass {
   ...
   int counter;

   int getCounter() {...}

   public myClass() {...}
}
```

- Function

```
aspect B;

import foo.bar;

refines myClass {
   ...
   int counter2;

   int getCounter2() {...}

   void anotherMethod() {...}
}
```

.jak Files have Embedded DSLs

- Constant

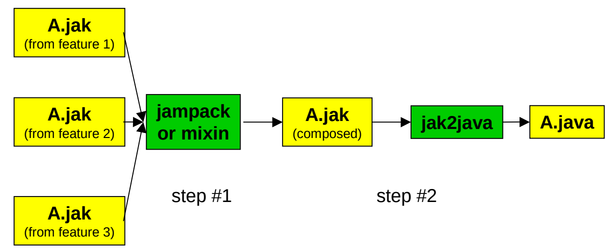
```
aspect A;

import java.util.*;

state_machine example {
   ...
   states s1, s2, s3;

   edge e1: s1 -> s2 ...;

   edge e2: s2 -> s3 ...;

   public example() {...}
}
```

- Function

```
aspect B;

import foo.bar;

refines state_machine example {

   states s4;

   edge e3: s3 -> s4 ...;

   void anotherMethod() {...}
}
```


.jak 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() {...}
}
```

o

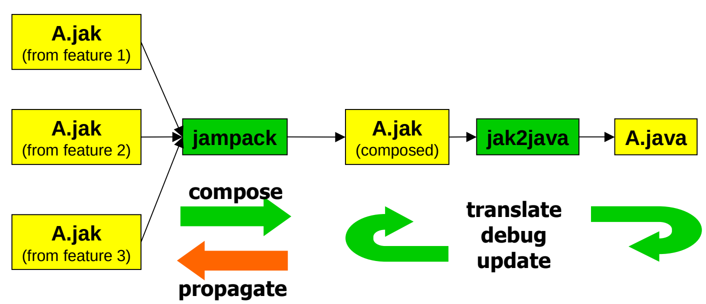
```
refines class top {
   int b;
   int bar() {...}
}
```

→

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


jampack (Cont)

- 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 features



mixin

- Preserves refinement hierarchy as inheritance hierarchy

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

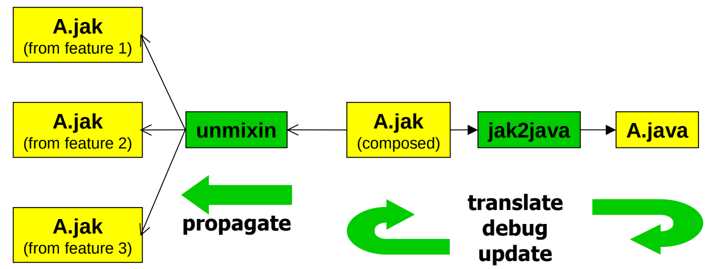
o

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

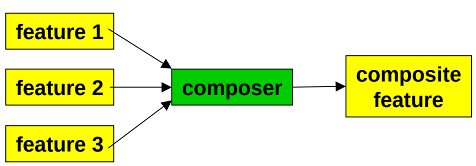
```
SoUrCe "c:\...A\top.jak"

abstract class top$$A {
  int a;
  void foo() {...}
}
SoUrCe "c:\...B\mid.jak"

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

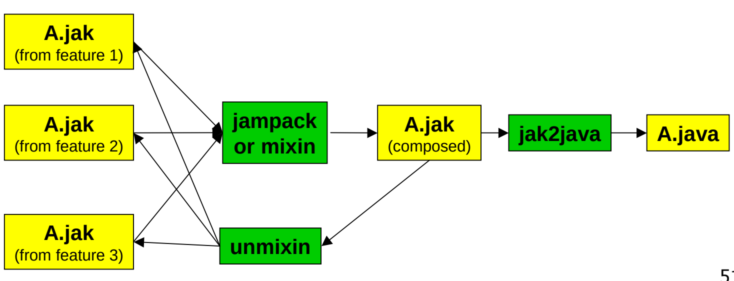

unmixin

- Edit, debug composed A.jak files
- unmixin propagates changes back to constitute feature files automatically



Recap of Code Tools





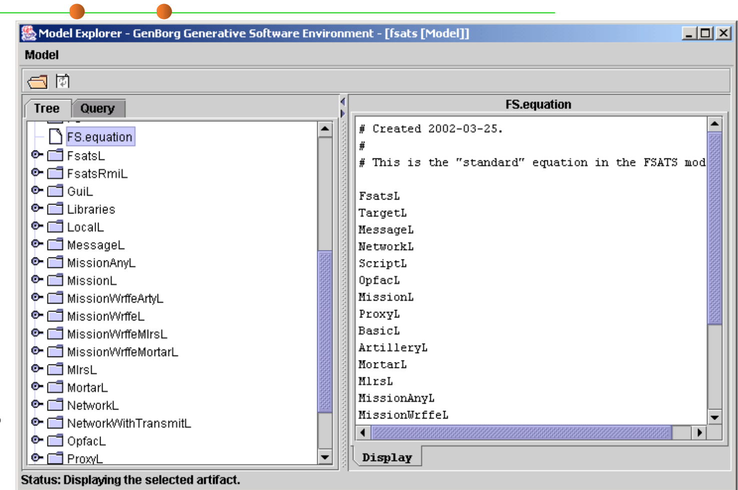
Tool Demo

ModelExplorer

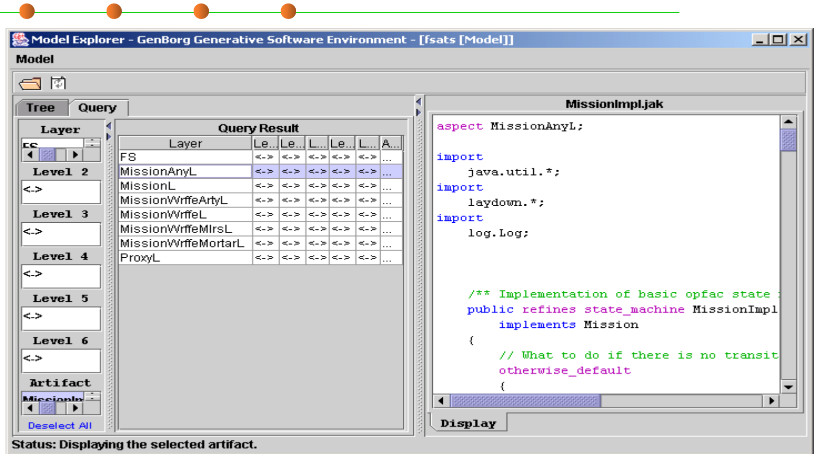
- Enables "exploration" of collective via
  - directory hierarchy ala MS file Explorer
  - relational-like query
    - where hierarchy is stored in a database
    - suitable for querying via XQuery
  - eventually will be able to invoke composer(s)

ModelExplorer

FSATS model has ~30 units most are collectives



ModelExplorer



Composer

- Build using equation file:

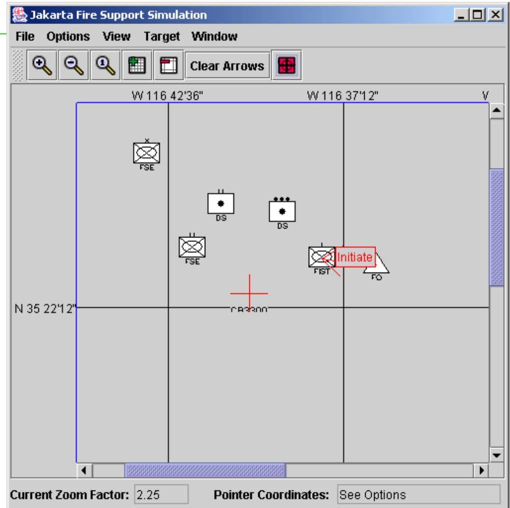
  > composer --equation=FS.equation --logging=info

  FS.equation composes 21 refinements in FSATs model

  generates code, drc files, makefiles + other representations..

- Runs ant makefile to produce FSATS prototype

FSATS Prototype



Lecture 3b: Scaling Refinements To Product-Families

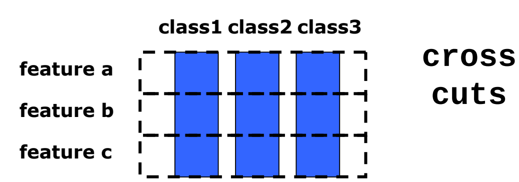
Don Batory
Dept. of Computer Sciences
University of Texas at Austin

Raise Two Questions

- State of the art: GenVoca models customize individual programs
  - set of all such programs is a product-line
- Larger scale: **Product-family** is an integrated suite of programs, each with different capabilities
  - MS Office (Excel, Word, Access, ...)
- Question #1: Do GenVoca refinements scale
  - to product-families?
  - product-line of product-families?

Question #2

- Features (refinements) are building blocks of classes, packages

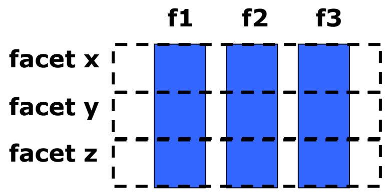


compositions of features yields packages of fully formed classes

•Question #2: What are building blocks of features?

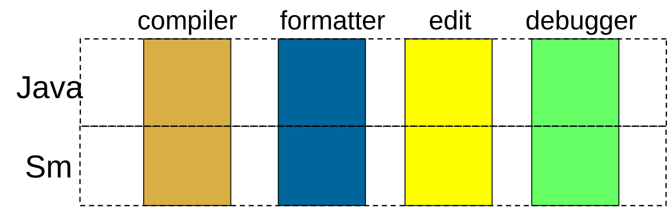
Ans: Facets

- Composition of facets yields sets of fully formed features
- Not figure on last slide turned on its side: facet != classes
- Do facets exist?...



Yes!

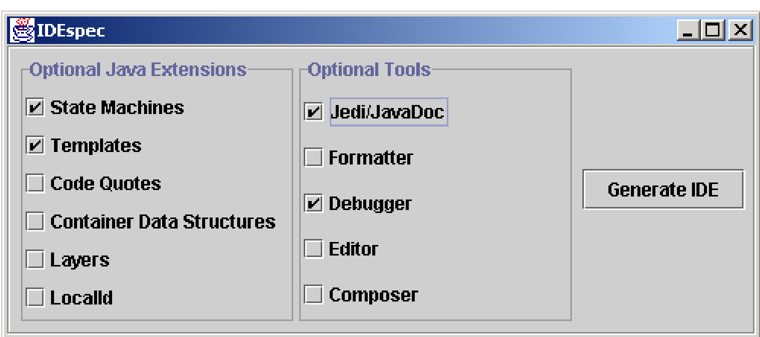
- Integrated Development Environment (IDE)
  - product-family of tools to write, debug, document programs
  - our variant: Java language extensibility



In principle, features scale to multiple programs!

Should be Simple...

- Fill in this form and IDE tools are generated



Surprise! Not That Simple!

- Features are no longer atomic
  - features composed from more elementary features (gluons)
  - gluons are structured and composed in very regular ways giving rise to composite features and facets
- Model of gluons & facets shows that software has an elegant mathematical structure
  - simpler designs
  - powerful models of code generation (product-families)
  - illustrating example: IDE generator

This Talk

- New results on GenVoca refinement modularity, scalability
- Generalization of GenVoca
  - 1st indication of significant generalization of basic model
- Sophisticated example of Multi-Dimensional Separation of Concerns
  - Tarr, Ossher IBM
  - idea that modularity can be understood through multi-dimensional hyperspaces of units
  - slices of hyperspace are modules (such as aspects)

An Example

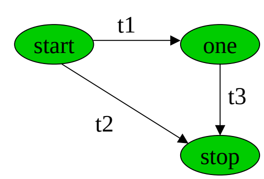
that motivates gluons
and facets

Jakarta Tool Suite (JTS) Overview

- **JTS** is a suite of compiler-compiler tools
  - to create extensible-versions of Java language
  - product-line of Java dialects using GenVoca models
- Current dialect Jak extends Java with state machines and templates
  - but why extend Java????

But Why Extend Java?

- Ans: here's a state machine….



- Do you want to write….

in Pure Java … or

```
class example {
    final static int start = 1000;
    final static int one = 1001;
    final static int stop = 1002; int current_state;
  // getState method
    public String getState() {
      if (current_state == start) return "start";
      if (current_state == one) return "one";
      if (current_state == stop) return "stop";
       System.err.println("unrecognizable state "
       + current_state);
       System.exit(1);
       return /* should never get here */ null;
    }
  // methods for state one
    void one_branches(M m) {
       if ( t3_test(m))
          { t3_action(m); stop_enter(m); return; }
       ; one_otherwise(m);
    }
    void one_enter(M m) { current_state = one; }
    void one_exit(M m) { }
    void one_otherwise(M m) { otherwise_Default(m); }
 // otherwise_Default Method
    void otherwise_Default(M m) { ignore_message(m); }
    public void receive_message(M m) {
       if (current_state == start) {
         start_exit(m); start_branches(m); return; }
       if (current_state == one) {
         one_exit(m); one_branches(m); return; }
       if (current_state == stop) {
          stop_exit(m); stop_branches(m); return; }
       error( -1, m );
    }
```

```
  // methods for state start
    void start_branches(M m) {
       if ( t1_test(m))
          { t1_action(m); one_enter(m); return; }
       if ( t2_test(m))
          { t2_action(m); stop_enter(m); return; }
       ; start_otherwise(m);
    }
    void start_enter(M m) { current_state = start; }
    void start_exit(M m) { }
    void start_otherwise(M m) { otherwise_Default(m); }
  // methods for state stop
    void stop_branches(M m) {
       ; stop_otherwise(m);
    }
    void stop_enter(M m) { current_state = stop; }
    void stop_exit(M m) { }
    void stop_otherwise(M m) { otherwise_Default(m); }
 // methods for edge t1
    void t1_action(M m) { }
    boolean t1_test(M m) { return !booltest(); }
 // methods for edge t2
    void t2_action(M m) { }
    boolean t2_test(M m) { return booltest(); }
 // methods for edge t3
    void t3_action(M m) { }
    boolean t3_test(M m) { return true; }
 //
 boolean booltest() {  }

 example() { current_state = start; }
 }
```


Jak = Java + State Machine DSL

Error exits

```
state_machine example {
   event_delivery receive_message(M m);
   no_transition { error( -1, m ); }
   otherwise_default { ignore_message(m); }
```

State decls

```
   states start, one, stop;
```

Edge decls

```
   edge t1 : start -> one
   conditions !booltest() do { /* t1 action */ }

   edge t2 : start -> stop
   conditions booltest() do { /* t2 action */ }

   edge t3 : one -> stop
   conditions true  do { /* t3 action */ }
```

Constructor, methods

```
   //
   boolean booltest() { ... }
   example() { current_state = start; }
}
```


Jak (Continued)

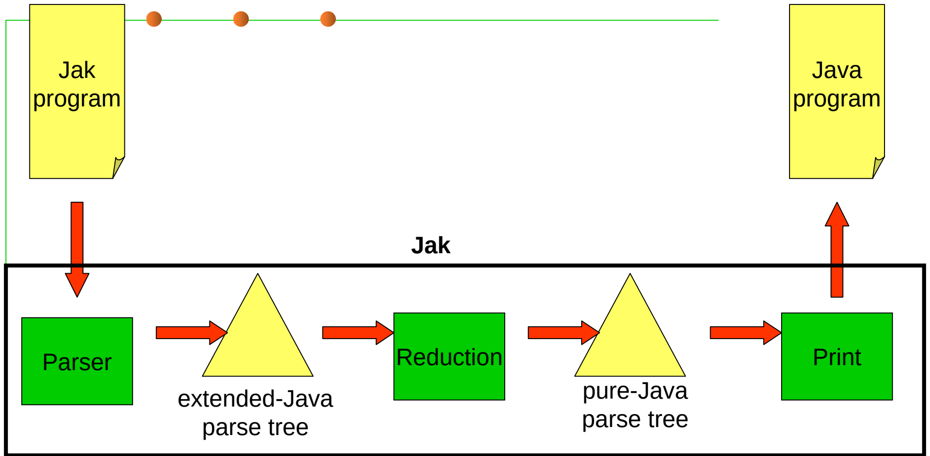
- DSL-extended Java simplifies programming
  - perform analyses (e.g., reachability) impossible to do in pure Java program
  - programs are about ½ the size of pure-Java
  - easier to understand, maintain, extend
- Similar benefits of template-extensions of Java

Conclusion – we want to program in DSL-extended Java languages…

So…

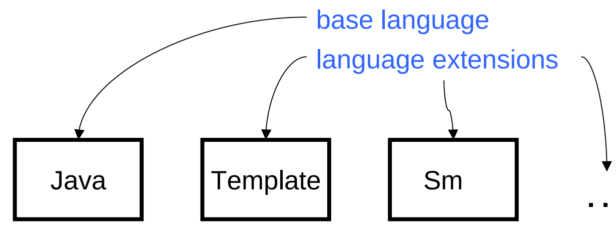
- We need tools (IDEs) for extended Java languages…
- Use JTS to build such tools
- Look at how Jak is built…
  - Jak is a preprocessor
  - translates extended-Java programs to pure-Java programs

Jak is a Preprocessor



JTS Model - Library

- Set of "feature" extensions to the Java language

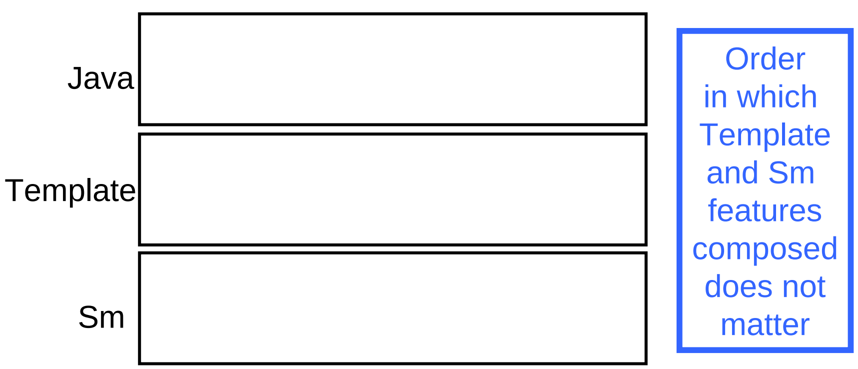


J = { Java, Template, Sm, ... }

- Compose them to produce required dialect
- Example...

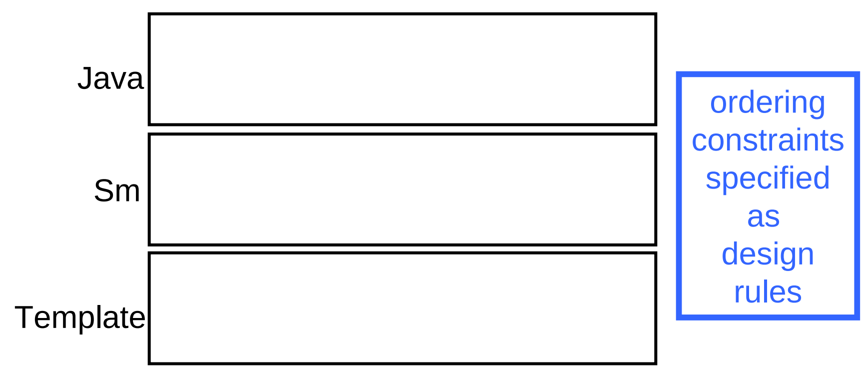
Architecture of Jak

Jak = Sm o Template o Java



Architecture of Jak

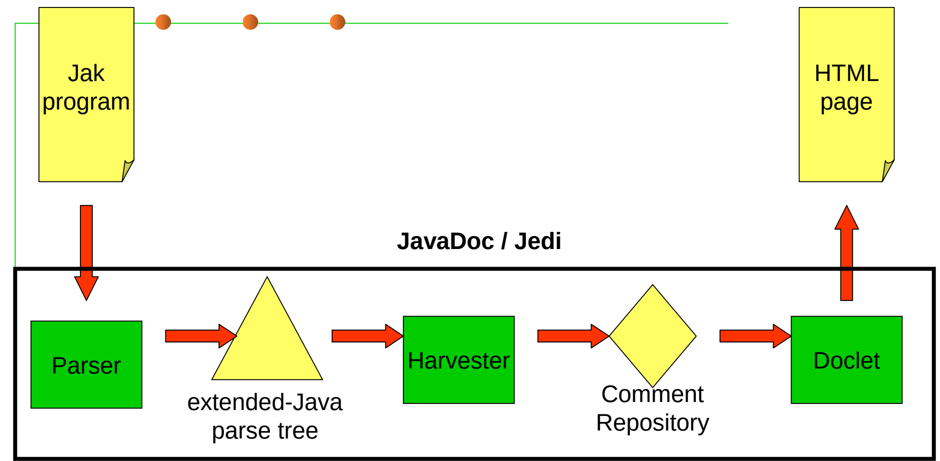
Jak = Template o Sm o Java



IDE Problem

- Today, we are writing extended-Java programs
  - built FSATS using state-machine/template extended Java
- Want JavaDoc-like HTML documents for extended-Java programs
- Can't use JavaDoc directly
  - because it only understands pure Java programs
- Need language-extensible version of JavaDoc
  - Jedi (Java Extensible DocumentatIon)

JavaDoc / Jedi



Jedi Model and Equation

- Has own model
  - elements are 1-1 correspondence with J model

D = { JavaDoc, TmplDoc, SmDoc, ... }

- Jedi defined by equations

Jedi = TmplDoc **o** SmDoc **o** JavaDoc

= SmDoc **o** TmplDoc **o** JavaDoc

Order in which Template and Sm features composed does not matter

IDE Model using Tool Features

```
IDE_Model = { parse, reduce, print,
              harvest, doclet, ... }
```

- Each const, function is feature of IDE tools
- Different equations are different tools
- Design rules govern legal compositions of features

```
Jak  = print o reduce o parse

Jedi = doclet o harvest o parse
...
```

Wait!

- We have different equations for each tool!

Jak = Sm o Template o Java // using language features
= print o reduce o parse // using tool features

Jedi = SmDoc o TmplDoc o JavaDoc // using lang. features
= doclet o harvest o parse // using tool features

- How do we prove their equivalence?

Relating Different GenVoca Models

in search of gluons...

Feature Orthogonality

- Language, tool features are orthogonal
- We can understand modularity of Jak and Jedi in terms of matrices
  - rows are language features
  - columns are tool features
  - entries denote modules that implement a tool feature for a particular language feature

Jedi Matrix

	Doclet	Harvest	Parse
Java	Jdoclet	Jharvest	Jparse
Sm	Sdoclet	Sharvest	Sparse
Tmpl	Tdoclet	Tharvest	Tparse

- Each entry is a module that implements a "feature of a feature"
- Composition of these modules implements Jedi

Gluons and Facets

	Doclet	Harvest	Parse
Java	Jdoclet	Jharvest	Jparse
Sm	Sdoclet	Sharvest	Sparse
Tmpl	Tdoclet	Tharvest	Tparse

matrix entries are called gluons

- Row is language feature, implemented by composition of gluons in that row
- Columns are facets – cross-cut each row

Gluons and Facets

	Doclet	Harvest	Parse
Java	Jdoclet	Jharvest	Jparse
Sm	Sdoclet	Sharvest	Sparse
Tmpl	Tdoclet	Tharvest	Tparse

- Column is a tool feature, implemented by composition of gluons in that column
- Rows are facets – cross-cut each column

Jak Matrix

	Print	Reduce	Parse
Java	Jprint	Jreduce	Jparse
Sm	-	Sreduce	Sparse
Tmpl	-	Treduce	Tparse

- Note absent modules
- Composition of these modules implements Jak

What is a Gluon?

- Ans: Mixin-Layer
  - elementary refinement (layer) that implements a "feature of a feature" or a building-block of a language/tool feature
  - GenVoca constant or function

Why do Gluons Exist?

- Ans: always can decompose composite constant, function into primitives

$$C = F_1(F_2(\ldots F_n(c) \ldots))$$

$$F(x) = F_1'(F_2'(\ldots F_n'(x) \ldots))$$

- Decomposing software is modeled by decomposing equations

Applications with Gluons are Equations

Jdoclet	Jharvest	Jparse
Sdoclet	Sharvest	Sparse
Tdoclet	Tharvest	Tparse

```
Jedi = TDoclet o THarvest o TParser o
       SDoclet o JDoclet o SHarvest o
       JHarvest o SParser o JParser
```

Jprint	Jreduce	Jparse
-	Sreduce	Sparse
-	Treduce	Tparse

```
Jak  = Print o TReduce o SReduce o
       JReduce o TParse o SParse o
       JParse
```

Q: How is this mapping done?
Q: Are they consistent?

A: can't be answered by inspection

Questions to Answer

- What is a model of gluons that
  - produces consistent equations
  - explains facets
- How do we use model to build IDE generators?
- That's next...

Origami

a model of gluons and facets

Change Notation

- Instead of writing:

  Eqn = A(B(C(D)))

- We will write:

  Eqn = A o B o C o D

- Where o is composition operator

Model of Gluons and Facets

- GenVoca models are 1-dimensional
  - set of constants and functions
- Gluon models are inherently 2-dimensional
  - or more generally n-dimensional
  - view them accordingly

Origami Matrix

- Rows are all language features;
  Columns are all tool features;
  Gluons are entries

	Parser	Harvest	Doclet	Reduce	Print	...
Java	JParser	JHarvest	JDoclet	JReduce	JPrint	...
Sm	SParser	SHarvest	SDoclet	SReduce	-	...
Template	TParser	THarvest	TDoclet	TReduce	-	...
DS	DParser	DHarvest	DDoclet	DReduce	-	...
...	...	...	...	...	-	...

- Filling in this matrix is easy, facets

Extending the Matrix

- New row requires gluons for all columns
- New row cross-cuts all column "features"

	Doclet	Harvest	Parser	Reduce	Print	...
Java	JDoclet	JHarvest	JParser	JReduce	JPrint	...
Sm	SDoclet	SHarvest	SParser	SReduce	-	...
Template	TDoclet	THarvest	TParser	TReduce	-	...
DS	DDoclet	DHarvest	DParser	DReduce	-	...
...	...	...	...	...	-	...

Facets

Extending the Matrix

- New column requires gluons for all rows
- New column cross-cuts all row "features"

	Doclet	Harvest	Parser	Reduce	Print	...
Java	JDoclet	JHarvest	JParser	JReduce	JPrint	...
Sm	SDoclet	SHarvest	SParser	SReduce	-	...
Template	TDoclet	THarvest	TParser	TReduce	-	...
DS	DDoclet	DHarvest	DParser	DReduce	-	...
...	...	...	...	...	-	...

Facets

Origami

- Compositions produced by "folding" Matrix:
  - compose rows by composing corresponding gluons in each column
  - compose columns by composing corresponding gluons in each row

Application is Equation

- Identify language, tool features to compose – ex: Jedi

	Doclet	Harvest	Parser	Reduce	Print	...
Java	JDoclet	JHarvest	JParser	JReduce	JPrint	...
Sm	SDoclet	SHarvest	SParser	SReduce	-	...
Template	TDoclet	THarvest	TParser	TReduce	-	...
DS	DDoclet	DHarvest	DParser	DReduce	-	...
...	...	...	...	...	-	...

Discard Non-Selected Entries

	Doclet	Harvest	Parser
Java	JDoclet	JHarvest	JParser
Sm	SDoclet	SHarvest	SParser
Template	TDoclet	THarvest	TParser

Fold Rows and Columns

- in Design Rule order
  - Java then { Sm, Templates } in any order
  - Parser then Harvest then Doclet

	Doclet	Harvest	Parser
Java	JDoclet	JHarvest	JParser
Sm	SDoclet	SHarvest	SParser
Template	TDoclet	THarvest	TParser

compose Java row with Sm row

Fold Rows and Columns

- in Design Rule order
  - Java then { Sm, Templates } in any order
  - Parser then Harvest then Doclet

	Doclet	Harvest	Parser
Java Sm	JDoclet o SDoclet	JHarvest o SHarvest	JParser o SParser
Template	TDoclet	THarvest	TParser

compose Parser col with Harvest col

Fold Rows and Columns

- in Design Rule order
  - Java then { Sm, Templates } in any order
  - Parser then Harvest then Doclet

	Doclet	Harvest, Parser
Java Sm	JDoclet o SDoclet	(JHarvest o SHarvest) o (JParser o SParser)
Template	TDoclet	THarvest o TParser

compose with Doclet column

Fold Rows and Columns

- in Design Rule order
  - Java then { Sm, Templates } in any order
  - Parser then Harvest then Doclet

	Doclet, Harvest, Parser
Java Sm	(JDoclet o SDoclet) o (JHarvest o SHarvest) o (JParser o SParser)
Template	TDoclet o THarvest o TParser

compose with Template row

Fold Rows and Columns

- in Design Rule order
  - Java then { Sm, Templates } in any order
  - Parser then Harvest then Doclet

	Doclet		Harvest		Parser
Java	JDoclet		JHarvest		JParser
	o		o		o
Sm	SDoclet	o	SHarvest	o	SParser
			o		
Template	TDoclet	o	THarvest	o	TParser

done!

To Yield Equation

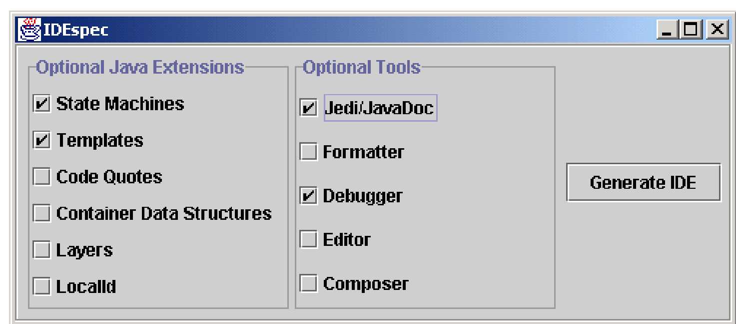
```
Jedi = (TDoclet o THarvest o TParser) o
       (SDoclet o JDoclet) o
       (SHarvest o JHarvest) o (SParser o JParser)
```

- Other constraints may preclude certain foldings
  - but this is the essential idea

Use Origami to Generate Language-Extensible IDEs

yields generator for a product-line of product-families

Recall IDE Generator GUI



Origami Matrix

- Selected language features trims rows

	Parser	Harvest	Doclet	Reduce	Print	...
Java	JParser	JHarvest	JDoclet	JReduce	JPrint	...
Sm	SParser	SHarvest	SDoclet	SReduce	-	...
Template	TParser	THarvest	TDoclet	TReduce	-	...
DS	DParser	DHarvest	DDoclet	DReduce	-	...
...	...	...	...	...	-	...

Effect on Matrix

- Selected language features trims rows

	Parser	Harvest	Doclet	Reduce	Print	...
Java	JParser	JHarvest	JDoclet	JReduce	JPrint	...
Sm	SParser	SHarvest	SDoclet	SReduce	-	...
Template	TParser	THarvest	TDoclet	TReduce	-	...

- Easy to determine order of row composition

Effect on Matrix

- Now compose the rows

	Parser	Harvest	Doclet	Reduce	Print	...
Java Sm	JParser o SParser	JHarvest o SHarvest	JDoclet o SDoclet	JReduce o SReduce	JPrint	... ...
Template	TParser	THarvest	TDoclet	TReduce	-	...

Effect on Matrix

- Now compose the rows

	Parser	Harvest	Doclet	Reduce	Print	...
Java Sm Template	JParser o SParser o TParser	JHarvest o SHarvest o THarvest	JDoclet o SDoclet o TDoclet	JReduce o SReduce o TReduce	JPrint	... ...

Resulting Row

- Note its semantics!

	Parser	Harvest	Doclet	Reduce	Print	...
Java	JParser	JHarvest	JDoclet	JReduce	JPrint	...
	o	o	o	o		
Sm	SParser	SHarvest	SDoclet	SReduce		...
	o	o	o	o		
Template	TParser	THarvest	TDoclet	TReduce		...

```
Parser  = TParser  o SParser  o JParser

Harvest = THarvest o SHarvest o JHarvest

Doclet  = TDoclet  o SDoclet  o JDoclet
```


Resulting Row

- Is GenVoca model for IDE product-line!
  - each constant, function is a feature of tool

```
IDE_Model = { Parser, Harvest, Doclet, Print, Reduce, ... }
```

  - folding defines an eqn for each feature
  - and we know equations for each program of product family!

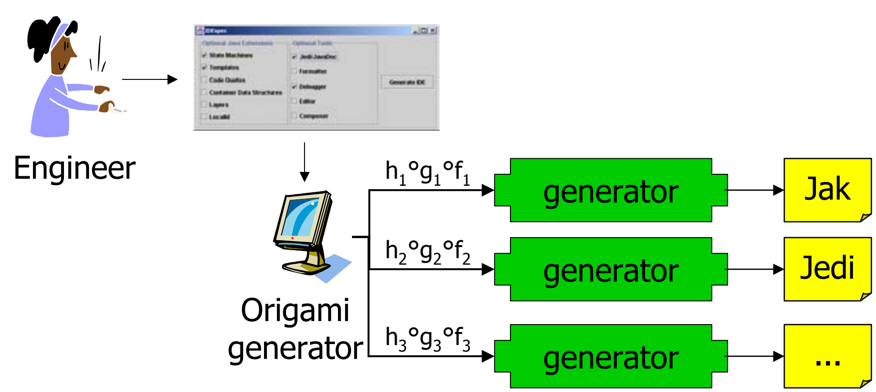
```
Jak  = Print o Reduce o Parser

Jedi = Doclet o Harvest o Parser
...
```


IDE Generator is Simple

- For each selected tool, evaluate its eqn

And generate the code for each tool automatically!

Generator of IDE Prod-Line (Generator of Product-Family)



Implementing Origami in AHEAD

Origami – Idea 1

- Need 4 ideas
- Equation files (Jak.eqn)
  - another artifact file type
  - specifies a single equation

  Jak = print ° reduce ° parse

Origami – Idea 2

- There are LOTS of other operators, besides °, for collectives and units
- One is evaluation Φ
  - applied to a model, all .eqn files are evaluated

M = { parse, reduce, print, harvest, doclet, Jak.eqn, Jedi.eqn }

Φ(M) generates Jak and Jedi tools

Origami – Idea 3

- Metamodel is a model whose instances are models

M = { a, b, c } // model M

MM = { AA, BB, CC, DD } // metamodel
= { { a }, { b }, { c }, { d } }

M = AA ° BB ° CC // eqn defining M

Origami – Idea 4

- Origami is a metamodel!
  - recall matrix:

	Parser	Harvest	Doclet	Reduce	Print	...
Java	JParser	JHarvest	JDoclet	JReduce	JPrint	...
SM	SParser	SHarvest	SDoclet	SReduce	-	...
Tmpl	TParser	THarvest	TDoclet	TReduce	-	...
DS	DParser	DHarvest	DDoclet	DReduce	-	...
...	...	...	...	...	-	...

- Rows are units of metamodel
  - collective with an .eqn file for each IDE tool

Origami (Cont)

- IDE metamodel

Equation collectives

IDEMM = { Java, Sm, Tmpl, DS, ... Jedi, Jak, ... }

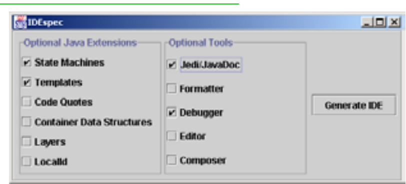
origami rows

Java = { Parser, Harvest, Doclet, Reduce, ... } // std names
Sm = { Parser, Harvest, Doclet, Reduce, ... }
Tmpl = { Parser, Harvest, Doclet, Reduce, ... }

Jedi = { Jedi.eqn }
Jak = { Jak.eqn }

Origami (Cont)

- Use selected language features and selected tools to compose model from metamodel



M = Jedi ∘ Jak ∘ Tmpl ∘ Sm ∘ Java

= { Parser, Harvest, Doclet, Reduce, ...
Jedi.eqn, Jak.eqn }

- Φ(M) generates Jak, Jedi tools

Questions?