Product-Line Architectures, Aspects, and Reuse

Don Batory

Department of Computer Sciences
University of Texas
Austin, Texas 78712

batory@cs.utexas.edu
512-471-9713
www.cs.utexas.edu/users/dsb/tutorial.html

©dsbatory

My Background

In 1983, I began studying how database management systems could be synthesized from plug-compatible components. First journal paper on this subject appeared in *ACM Transactions on Databases*, December 1995

Modeling the Storage Architectures of Commercial Database Systems

D. S. BATORY
The University of Texas at Austin

Modeling the storage structures of a DBMS is a prerequisite to understanding and optimizing database performance. Previously, such modeling was very difficult because the fundamental role of conceptual-to-internal mappings in DBMS implementations went unrecognized.

In this paper we present a model of physical databases, called the *transformation model*, that makes conceptual-to-internal mappings explicit. By exposing such mappings, we show that it is possible to model the storage architectures (i.e., the storage structures and mappings) of many commercial DBMSs in a precise, systematic, and comprehendible way. Models of the INQUIRE, ADABAS, and SYSTEM 2000 storage architectures are presented as examples of the model's utility.

We believe the transformation model helps bridge the gap between physical database theory and practice. It also reveals the possibility of a technology to automate the development of physical database software.

Topic	**Mainstream Date**
Components (“Aspects”)	1989 (1997)
Domain-Specific Software Architectures	1993
Product-Lines	1998
Automation/Generation	??

©dsbatory

Introduction

Central problems in software engineering stem from:

$\boxed{\textit{\textbf{software is written by hand}}}$

- easy to understand complaints about software quality, performance, reliability, maintainability, evolvability, …

Software *quality* is a function of:

- quality of application design

  designs improve with experience

  design is difficult — build several times to get it “right”

- quality of the programming staff

Answers to both are highly variable. So…

- how can we do better?
- how can we better exploit previous systems, experiences?

©dsbatory

The Future (and this Tutorial)

The future of software development lies in *standardization* and *automated production* of well-understood software

- major improvements in quality, reliability, performance…
- technology based on:

Domain-Specific Component Technologies

Principled engineering approach that standardizes:

- expert-approved designs (programming problems)
- expert-approved implementations (programming solutions)
- component compositions define target systems

In this way, we improve key factors of software quality:

- using "expert" designs, "expert" implementations

©dsbatory

Key Features of Vision

Scale of "component" encapsulation/reuse is critical:

SSR	small scale reuse	algorithm, function reuse
MSR	medium scale reuse	suites of related functions (classes)
LSR	large scale reuse	suites of related classes (subsystems, frameworks)

Using LSR components is key; challenge is *how*?

Tutorial answers following questions:

- what is a *large-scale component* or building-block for a domain of applications?
- how are component-based *product-line architectures* defined?
- how can complex, efficient, and *extensible* applications be constructed from components?

We review answers distilled from experiences and prototype *generators*:

- tools that assemble families of systems from expert-designed & expert-implemented components for well-understood domains

©dsbatory

So What?

Product-Line Architectures (PLAs)

- producing family of applications requires definition of a PLA; component-based generators are exemplars to study

Domain Modeling / Software Modeling

- how do you model a family of applications? state-of-art modeling/programming paradigm

Software Reuse

- component-based generators are success stories; reveal secrets of success

Relevance to Research

- *Aspect Oriented Programming (AOP)*, Microsoft's IP Intensions, Collaboration-Based Design, Perry's Light Semantics, …

Relevance to Practice

- ideas of tutorial are being used in industry; Microsoft's COM is a special case of this technology

©dsbatory

Organization of Tutorial

Sequence of short lectures (with question/answer periods):

	Lecture / Period	Length
	Introduction to GenVoca	40 min
	Mixin-Layer Implementations of Refinements	50 min
	Design Rule Checking	40 min
	Recap & Open Discussion	
Expanded Tutorial	Domain Modeling Methodology	40 min
	Metaprogram Implementations of Refinements	40 min
	Architectural Connectors as Refinements	30 min
	Design Wizards and Automatic Selection of Components	30 min

Qualification: rich subject area that requires familiarity with many topics!

©dsbatory

GenVoca

Name given to a general theory and principles for software "legos":

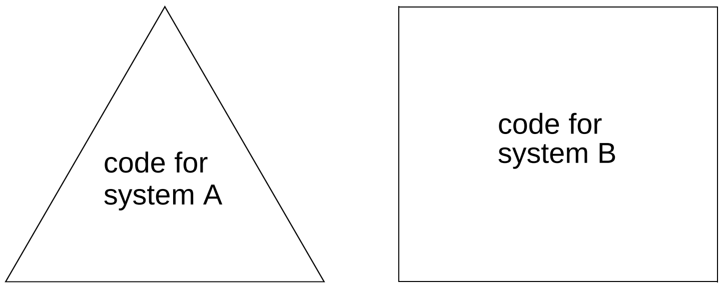
- ideas have surfaced (i.e., rediscovered, reimplemented) in many different contexts
- *goal of tutorial is to avoid reinvention of these ideas*
- name is a merging of the names "Genesis" and "Avoca" (the first two systems built on these principles)

idea of components that export and import standardized interfaces taken to its logical conclusion

GenVoca exposes fundamental role of domain modeling in large scale reuse, PLAs, and generation:

- what is domain modeling (reference architectures)?
- what is relationship to large scale reuse?

A Domain Modeling Thought Experiment



Rich Set of Lessons Learned

Ad Hoc Software Designs/Decompositions

- don't work for component-based generators
- consequence of conventional 1-of-kind system designs
- not suitable for assembling product-lines economically

Large Scale Software Reuse

- is a consequence of premeditated design; standardization of recurring "shapes" within a domain

  programming abstractions and implementations are standardized

- components must be designed to be interoperable and composable

  components will not have these properties otherwise

©dsbatory

Lessons Learned (Continued)

Domain Modeling

- is retrospective study of a family of systems
- differs from application modeling (i.e. point designs)
- process of standardizing well-understood domain
- parametric model or blue-print of a PLA

Component-based Generators

- can significantly increase productivity
- especially if all required components are available

Explain GenVoca from first principles

- goes beyond OO concepts
- OO is medium scale, not large scale

Resist Interpretation!!

Look at syntax now, semantics later!!

©dsbatory

Background on Hierarchical Software

Virtual machines (Dijkstra 1968)

- design each level of a hierarchical system independently
- each level defines a *virtual machine*

  all operations on level i+1 defined in terms of operations on level i

Refresh using OO ideas:

- define interface as objects, classes, and methods
- hierarchical design is set of:

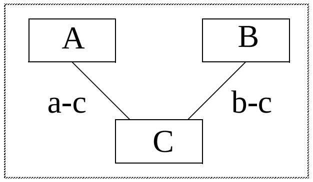
  object oriented virtual machines (OOVM)

  one OOVM for each level

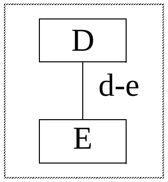
Background

Object model (or *object-oriented virtual machine*) is set of classes and their interrelationships

Use ER-like notation:



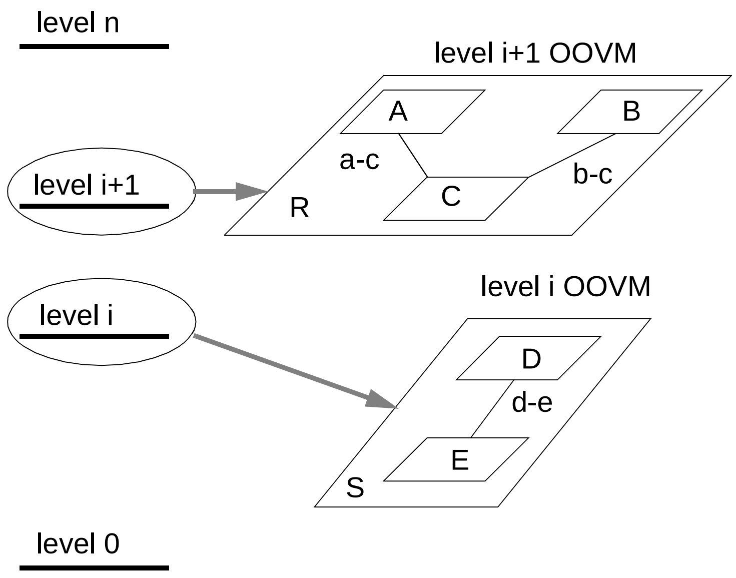
object model R



object model S

GenVoca not dependent on specific OO notation (choose your own …)

Hierarchical Design



Layer, component, “aspect” is:

- a ***consistent refinement*** (*mapping* or *transformation*) between virtual machines
- ***large scale*** **—** simultaneous and consistent refinement of multiple classes, objects, and methods

What is New?

Large scale components are ***layers*** *or* ***aspects***

- are encapsulated suites of interrelated classes *or fragments of related classes*
- are new: larger units of abstraction and encapsulation (see next lecture on **Mixin-Layers**)

How is large scale reuse achieved?

- standardize fundamental abstractions, OOVMs of a well-understood domain
- *layers export & import standardized interfaces*
- *layers are designed to be plug-compatible, interchangeable, interoperable*

latter two points are trademarks of GenVoca designs

- new: contrary to traditional library/reuse paradigms

More difficult to achieve:

- substantial step beyond design of OOVMs
- key to component-based software generation/synthesis

The GenVoca Model

Component (layer, aspect) is fundamental unit of large scale software construction

- interface of component is an object model

  (multiple classes, relationships)

- component **w** exports OOVM interface S

All components that export the same interface (OOVM) belong to a *realm*

- realm is a *library* of plug-compatible, interoperable, and interchangeable components

- OOVM S and R define realms **S** and **R**

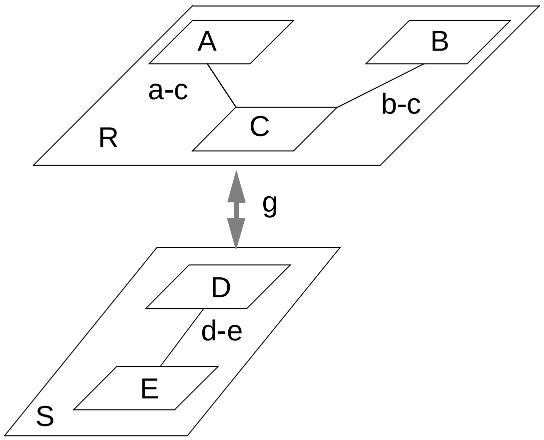
```
S  =  {  y, z, w  }

R  =  { g[ x:S ], h[ x:S ], i[ x:S ]  }
```

- note: components may be parameterized

Components with Parameters

Consider g[x:S] : R

- g exports interface R; g imports interface S



- g translates operations and objects of R to S; translation called a *refinement*
- *parameter* x:S *means that translation doesn't depend on a specific implementation of* S

Applications and Type Equations

An application/subsystem is named composition of components called a *type equation*:

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

R  =  { g[ x:S ], h[ x:S ], i[ x:S ] }


app1 = g[ y ];

app2 = g[ w ];

app3 = h[ w ];
```

now possible to precisely define family of applications that can be built

- modeling applications as equations is hallmark of *parameterized programming* (Goguen 1986)

©dsbatory

Interpretation

Interpretation critical!

- relates domain features to components

```
S  =  {  y,         // program with feature y
         z,         // program with feature z
         w          // program with feature w
      }

R  =  { g[ x:S ],   // adds feature g to x
        h[ x:S ],   // adds feature h to x
        i[ x:S ]    // adds feature i to x
      }
```

So, type equations define programs with known features!

```
app1 = g[ y ];     // program w. features g,y

app2 = g[ w ];     // program w. features g,w

app3 = h[ w ];     // program w. features h,w
```

can reason about applications in terms of their components

©dsbatory

Grammars and Families of Applications

Realms and components define a *grammar* whose sentences (component compositions) are applications

Parameterized component representation:

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

R  =  { g[ x:S ], h[ x:S ], i[ x:S ] }
```

Grammar representation:

```
S  :=   y   |   z   |   w

R  :=  g S  |  h S  |  i S
```

The set of all sentences defines a language (product-line):

- Parnas family of systems (1976)
- connection with grammars goes further....

©dsbatory

Symmetric Components

Just as recursion is fundamental to grammars, *symmetric components* are fundamental to GenVoca

- export and import same interface
- composable in virtually arbitrary orders

  order of composition affects semantics & performance

- *symmetric* components of realm `W` have parameters of type `W`:

```
W  =  { m[ x:W ], n[ x:W ], p }
W :=    m  W   | n  W   |  p
```

- examples:

  `m[n[p]], n[m[p]], m[m[p]], n[n[p]]`

- familiar example: Unix file filters

©dsbatory

Scalability and Component Reuse

Adding a new component to a realm is equivalent to adding a new rule to a grammar

- the family of applications enlarges exponentially (in length of type equation)
- because large families can be built using relatively few components, GenVoca models are *scalable*

Component *reuse* obvious: different systems/equations reference the same component...

```
application1 = g[ y ];

application2 = g[ w ];

application3 = h[ w ];
```

- components **g** and **w** are reused...
- reuse is common subexpressions, common terms

©dsbatory

Design Rules and Domain Models

Given realms below, in principle any component of **R** can be composed with any component of **S**:

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

R  =  { g[ x:S ], h[ x:S ], i[ x:S ]  }
```

- although equations may be type correct, there are always combinations of components that don’t make sense
- domain-specific constraints called *design rules* preclude illegal component combinations

GenVoca domain model is:

- realms of components
- design rules that restrict compositions
- can be expressed as an *attribute grammar*

See lecture on Design Rule Checking

©dsbatory

Important Special Case

Microsoft's *Component Object Model (COM)*

- components can export and import "standardized" interfaces
- applications are compositions of COM components

Differences:

- supports multiple-inheritance among interfaces (like Java)
- allows components to export *multiple* standardized interfaces, and import components that implement *multiple* interfaces (**R** and **S**, **R** or **S**)
- very useful indeed, but not critical to our PLA model

COM can be used to create GenVoca product-lines, but that isn't how COM is used today

- most interfaces implemented by a single component (IExplorer, Windows Media Player)
- different implementations arise from versioning
- generally don't have the GenVoca plug-and-play

©dsbatory

Special Case (Cont)

- Key restriction: COM interfaces limited to a *single class*

  we will look at components that have more complicated (i.e., multi-class) interfaces. Our components are Java packages or Microsoft DLLs that are typed (e.g. have interfaces) — concept not present in today's operating systems and programming languages

- Key restriction: COM components are binaries

  binaries are NOT the only way refinements can be implemented: there are lots of other ways (see next few slides).

  In fact, if one limits components only to (COM) binaries, there are many domains for which PLAs couldn't be built — the assembled applications would run so slowly that no one would ever use them.

  This doesn't mean that PLAs can not be implemented for these domains, it simply means that refinements for this domain have to be implemented differently …

©dsbatory

Why GenVoca Important?

The simplest “building-blocks” model of software construction has components that import and export standardized interfaces

- idea that has arisen independently many times

System	Domain	Year
Genesis	Database Management	1988
Avoca/x-kernel	Network Protocols	1989
Ficus	File Systems	1990
Rosetta	Database Data Languages	1994
ADAGE	Avionics	1994
ASP	Audio Signal Processing	1995
JTS	Extensible Precompilers	1997
P3	Data Structures	1997
LavaLamp	Radio Software	1998
FSATS99	Command-and-Control Simulator	1999
…	…	…

- there common and deep problems of conceptualization and implementation that arise in every one of these domains/generators and that take *years* to sort out…

  This tutorial allows you to avoid costly reinvention…

©dsbatory

Perspective

GenVoca advocates that the atomic building blocks of product-lines are *refinements* (i.e., mappings between standardized virtual machines)

- model doesn’t say *how* refinements are implemented or *when* refinements are composed
- all that is known is that refinements have parameters and can be composed via parameter instantiation

Lack of specificity makes GenVoca general…

Refinements can have vastly different implementations:

- *object*
  (Java object or COM component)
- *template*
  (mixin-layer)
- *metaprogram*
  (a program that generates another program)
- *rule set of a program transformation system*
  (transformations of abstract specifications into efficient programs through rewrites)

©dsbatory

Perspective (Cont)

In addition to multiple implementations, refinements can be composed at different times:

- *statically* at application compile time (build time) (for applications whose type equation is fixed)
- *dynamically* at application execution time (allows type equations to change during program execution)

Look at known GenVoca PLAs:

Refinement Implementation	**Refinement Composition Time**	
	Static	**Dynamic**
Object	Genesis Ficus	Avoca LavaLamp
Template	JTS FSATS, ADAGE	
MetaProgram	P3 LavaLamp	ASP
Transformation	?	?

©dsbatory

Perspective (Cont)

Refinements can have different implementations and can be composed at different times, there is yet another variability — there are different kinds of refinements!

- *Virtual Machine Refinements*
  imported interface(s) are not visible
  consistent with standard notion of “virtual machines”

- *Subjective (Extension) Refinements*
  imported interfaces are visible
  enhancing lower-level VMs with more capabilities and exporting this enhanced VM — ***see mixin-layers***

- *Optimizing Refinements*
  map less-efficient programs to more efficient programs

```
i = 3 + 4 - 5;
```

```
i = 2;
```

```
sum = 0;     // n > 0
for (j=1; j<=n; j++)
   sum = sum + j;
```

```
j = n+1;
sum = (n*j)/2;
```

note: subjective and optimizing refinements are symmetric

©dsbatory

Perspective (Cont)

How to chose among implementations?
Ans: depends on application requirements

- dynamic for run-time reconfigurations
- static for optimizations
  although different technologies have different optimization possibilities:

  templates — none (except for C++)
  metaprograms — lots (but need language support)
  program transformation systems — ∞ (need rule engine)

GenVoca offers a single way in which to conceptualize a domain and its building blocks in a largely implementation-independent way

- ex: don’t use rule-sets when templates or objects suffice

Conceptual economy of GenVoca is a big win …

©dsbatory

Conclusions — Lessons Learned

- ***GenVoca takes idea of components exporting and importing standardized interfaces to logical conclusion***
- ***refinements are basic building blocks***

  abstract design entities
  implementations are multi-class encapsulations
  layers export and import multi-class virtual machine interfaces

- ***standardizing abstractions, their interfaces and implementations as plug-compatible components***

  different than conventional library paradigms

- ***applications are modeled by equations***

  parameterized programming

- ***a variant of COM is an instance of GenVoca***

Rest of tutorial will explain GenVoca in more detail

©dsbatory

Further Reading

D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.

T. Biggerstaff, "An Assessment and Analysis of Software Reuse", in *Advances in Computers,* Volume 34, Academic Press, 1992.

T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse", *Third International Conference on Software Reuse, Rio de Janeiro*, November 1-4, 1994, 102-110.

K. Czarnecki and U.W. Eisenecker, "Synthesizing Objects", *ECOOP 1999*.

E.W. Dijkstra, "The Structure of THE Multiprogramming System", *Communications of ACM*, May 1968, 341-346.

J.A. Goguen, "Reusing and Interconnecting Software Components", *IEEE Computer*, February 1986.

R.E. Johnson and B. Foote, "Designing Reusable Classes", *Journal of Object-Oriented Programming*, June/July 1988.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier and J. Irwin, "Aspect-Oriented Programming", *ECOOP 97*, 220-242.

C. Krueger, "Software Reuse", *ACM Computing Surveys*, June 1992, 131-183.

D. McIlroy, "Mass Produced Software Components", *Software Engineering: Report on a Conference by the Nato Science Committee*, Oct 1968, P. Naur and B. Randell, eds. 138-150.

G. Novak, "Software Reuse by Specialization of Generic Procedures through Views", *IEEE Trans. Software Engineering*, July 1997, 401-417.

D.L. Parnas, "Designing Software for Ease of Extension and Contraction", *IEEE Transactions on Software Engineering*, March 1979.

R. Prieto-Diaz and G. Arango (ed.), *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press 1991.

©dsbatory

Mixin-Layer Implementations of Refinements

Understanding GenVoca requires looking at example domains where symmetry, scalability, encapsulation, and composition issues can be studied carefully.

In this lecture, we examine:

- *collaboration based designs (CDBs)* as GenVoca components
- how to encode and statically compose components
- interesting case studies:

  Graph Algorithm PLA

  FSATS99 — Army Fire Support Simulator

  Jakarta Tool Suite (JTS) — Extensible Java Compilers

©dsbatory

Collaboration-Based Designs

Fundamental, but not well-known, technique for creating reusable OO designs

Collaboration-Based Designs (CDBs) idea:

- define a set of interrelated classes that collaborate to implement some “feature” or “aspect” of a program
- each class actually represents an individual *role* in that collaboration
- methods define generic interactions among class/roles

Graph Collaboration:


```
add_node()         belongs_to()
remove_node()      linked_to()
graph_name()       node_name()
roots()            connect_to()
...                ...
```


©dsbatory

Collaboration Based Designs (Cont)

Note the following about CDBs:

- define generic relationships among classes/objects playing “roles”

  methods, instance variables, etc. that are needed to capture the desired relationships
  (in our case, node connectivity of a graph)

whenever one is dealing with “Graph-Collaboration” these methods will need to be written!

CDBs generally aren’t stand-alone; that’s why roles exist

- *role is a parameter that must be instantiated*

Examples of ‘re’-using Graph-Collaboration:

- map plays role of graph, city corresponds to a node
- communication network is graph; site corresponds to a node

©dsbatory

Mixin-Layer Implementation of Refinements

Collaboration Based Designs (Cont)

Problem with CDBs:

- not that well understood …
- known implementation techniques were not scalable (e.g., parameter instantiations of exponential length)
- no idea how they were related to component-based designs or refinements…

Let’s address these points first by examining how static refinements can be expressed in OO…

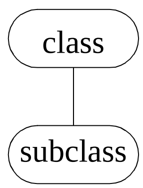
©dsbatory

OO Components

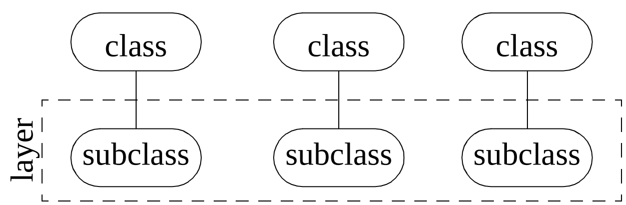
How are static refinements expressed in OO?

- *refinement* of a class adds new data members, new methods, and/or overrides existing methods

expressed
as a subclass



- a *GenVoca* or *large-scale refinement* adds new data members, methods, etc. simultaneously to several classes



note: there can be any number of "horizontal" or "collaboration" relationships among subclasses — here we show only inheritance relationships

©dsbatory

Mixin-Layer Implementation of Refinements

Insights

Ever add a new feature to an existing application?

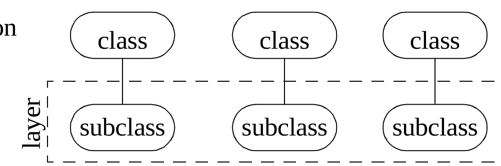
- changes aren't localized!
- multiple classes of an application must be updated
- if feature is removed, updates must be simultaneously removed from all classes

A "feature" or "aspect" can be expressed as a CBD:

- consists of a number of collaborating classes/roles
- roles of a collaboration must be bound to classes of the actual application itself

  accomplished via parameter instantiation

application
classes

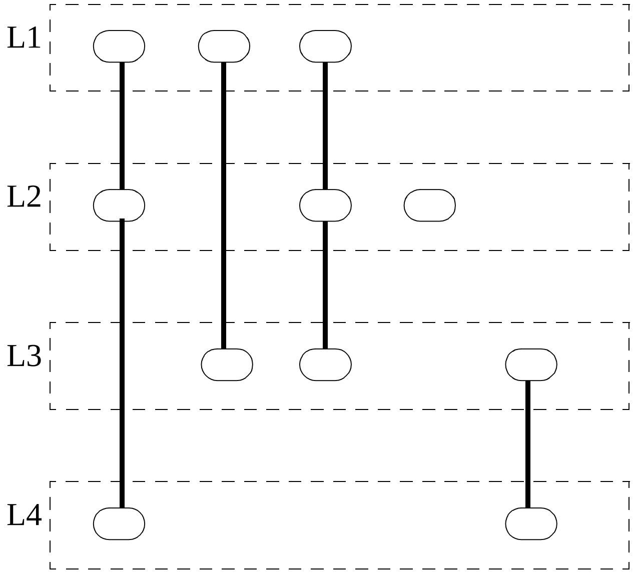


a layer defines a collaboration;
layer instantiation defines role/class bindings!

©dsbatory

Composition Insights

When CBDs (GenVoca components) are composed, a forest of inheritance hierarchies is created that gets progressively broader and deeper!



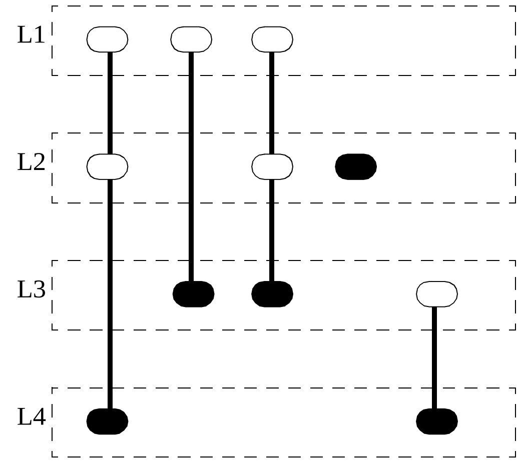
note: L_{i+1} inherits all the classes from L_i, recursively. Thus, the above instance of L4 has a total of 5 classes, each the terminals of their refinement chains.

©dsbatory

Composition Insights (Cont)

The classes that are instantiated by an application are the terminals of these refinement chains

- *nonterminal classes define intermediate derivations of application classes*



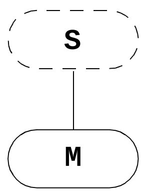
- in our example, “generated” application consists of 5 classes (shaded in black); the white classes are “intermediate” derivations of the black classes

©dsbatory

1st Idea: Mixins

How can we encode refinements in OO languages?

Mixin is a class whose superclass is specified via a parameter



Can express mixin **M** as Java “template” with parameter **S**

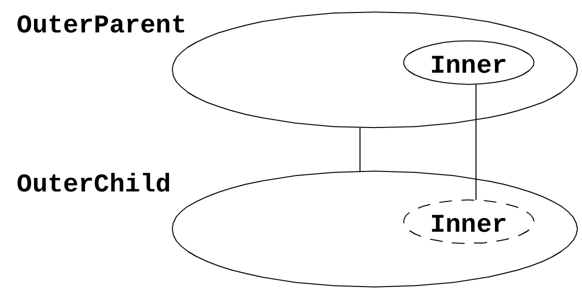
```
class M <AnyClass S> extends S { ... }
```

• note: “mixin” means something different in C++, CLOS literature, so beware! We use Bracha’s definition…

Mixin-Layer Implementation of Refinements

2nd Idea: Nested (Inner) Classes

Nested (*inner*) classes behave (e.g., access control, scoping) like regular class members




```
class OuterParent { class Inner { ... } }
class OuterChild extends OuterParent { }
```

• no **OuterChild.Inner** explicitly defined, but it does exist … **Inner** is inherited from **OuterParent**

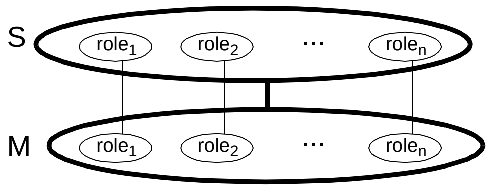
Nested classes emulate package encapsulation

• *representation allows “packages” to appear as nodes in inheritance hierarchies*

Combining Ideas = Mixin-Layers

Experience has shown that:

- different collaborations use the same names for roles
- classes with same role names refine each other when their collaborations are composed



Express as a *mixin-layer* **`M`** with parameter **`S`**:

```
class M <AnyClass S> extends S {
   class role1 extends S.role1 { ... }
   class role2 extends S.role2 { ... }
   ...
   class rolen extends S.rolen { ... }
}
```

Buy where do realms fit in?

©dsbatory

Realms

Remember: realms have interfaces that components export and import

- so component **`M[x:I]:E`** expressed as:

```
interface E { ... }        // export interf.
interface I { ... }        // import interf.

class M<I x> extends x implements E {
   class role1 extends x.role1 { ... }
   class role2 extends x.role2 { ... }
   ...
   class rolen extends x.rolen { ... }
}

// above is “standard” mixin-layer pattern
```


©dsbatory

Connection to GenVoca

Straightforward connection to type equations:

```
// type equation notation

Application = L4[ L3[ L2[ L1 ] ] ];


// extended-Java notation

class Application extends L4< L3< L2< L1 >>>;
```

If we followed same derivation, except that objects are being refined, not classes, you'll discover that OO frameworks are dynamic counterparts to Mixin-Layers

```
// type equation notation

Application = L4[ L3[ L2[ L1 ] ] ];


// extended-Java notation

Application =
       new L4( new L3( new L2( new L1())));
```

- OO framework implements a realm of components, refinements…

Mixin-Layer Implementation of Refinements

Case Study: Graph Algorithm PLA

Look at a simple domain for which a PLA can be created

- to reinforce ideas just presented
- see "complex" example afterward (and in Appendix)

Domain: programs that implement different graph algorithms over directed and undirected graphs

- product line programs described by a "feature" menu:

graph algorithm	**search algorithm**	**graph type**
vertex numbering cycle checking ...	depth-first breadth-first	directed graph undirected graph
choose one or more	**choose one**	**choose one**

- example programs of product-line:

  `ex1` — vertex numbering using a depth-first search on an undirected graph
  `ex2` — vertex numbering and cycle checking using a breadth-first search on a directed graph

GenVoca Model

Define a realm **G** of components that implement each individual “feature”:

```
G = {   undirected,        // undirect graph
        directed,          // directed graph
        dft[x:G],          // depth-first
        bft[x:G],          // breadth-first
        cycle[x:G],        // cycle checking
        number[x:G],       // vertex number
        ...
    }
```

Composition restrictions (i.e. “choose one”, ordering) realized by design rules (discussed in next lecture)

Compositions (from previous page):

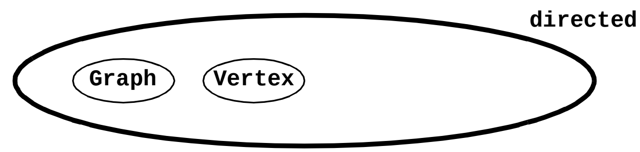
```
ex1 = number[ dft[ undirected ] ]
ex2 = number[ cycle[ bft[ directed ] ] ]
```


©dsbatory

Implementation as Mixin-Layers

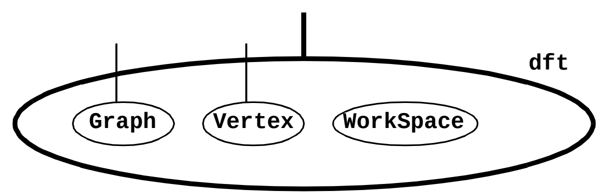
directed and **undirected** encapsulate two classes **Graph** and **Vertex**

- methods support vertex addition and removal from graphs; no traversals



dft and **bft** encapsulate a pair of refinements of **Graph** and **Vertex**, and add new abstract class **WorkSpace**.

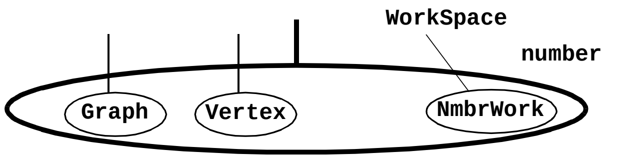
- traversal methods (**GraphSearch**, **VertexSearch**) added to **Graph**, **Vertex** with a **WorkSpace** object as parameter
- **WorkSpace** object has 3 abstract methods: **init_vertex**, **preVisitAction**, **postVisitAction**



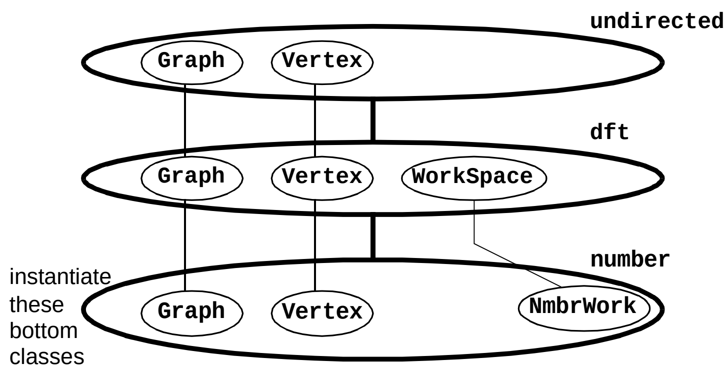
©dsbatory

Implementation (Cont)

`cycle` and **`number`** encapsulate refinements of **`Graph`** and **`Vertex`** (by adding algorithm-specific methods — ex: **`vertexNumber`**) and define a subclass of **`WorkSpace`** with appropriate init, pre-, post-action methods)



Example: **`ex1 = number[ dft[ undirected ] ]`**



Mixin-Layer Implementation of Refinements

Recap

Graph Domain illustrates the ability to:

- application of product-line defined by features it offers
- implement each feature as a mixin-layer

  each layer encapsulates multiple classes

  many domains have more complex features (e.g., features within features) — handled in a simple, parametric way rather than decomposing these features into a composition of layers…
- easy to implement in C++ and mixin-extended versions of Java

  see JTS example in Appendix of this lecture
- define product-line app as a composition of mixin-layers

  see “Object-Oriented Frameworks and Product-Lines”, SPLC1, 2000.

Now let’s look at a more complex domain…

Case Study: FSATS99

FSATS (Fire Support Automated Test System)

- command-and-control simulator for Army fire support
- first-generation system (9 years to build)
- difficult to understand & maintain current code base
- difficult to debug
- new capabilities are projected, old capabilities revamped
- implementation team wants to expand capabilities, but not current version
- beginnings of a product-line (but no PLA yet)

Upcoming topics:

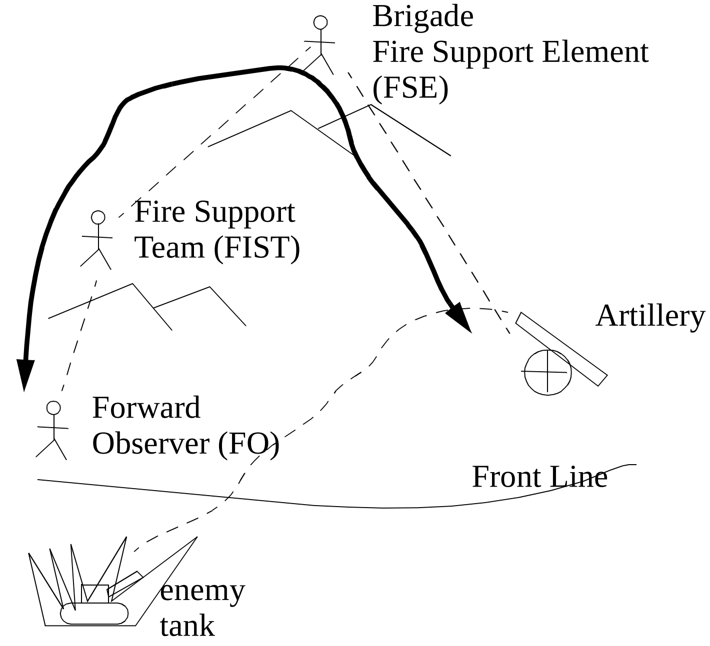
- explain domain of fire support
- quickly review current implementation of FSATS
- review component-based redesign (FSATS99) to see how mixin-layers were used to build an FSATS99 PLA

©dsbatory

Domain of Fire Support

Battlefield layout:

OPFAC (operational facility — command post)
command hierarchy



©dsbatory

Vanilla Distributed Application

Set of collaborating objects that work collectively toward achieving a given mission

All sorts of different kinds of mission types:

- *when-ready-fire-for-effect (WRFFE)* - mortars
- when-ready-fire-for-effect - artillery
- *adjust-fire (AF)* - mortars
- adjust-fire - artillery
- about 20 mission types in all;
  more mission types projected in future

For each mission type, each OPFAC (e.g., FO, FIST, FSE, …) takes different actions

- actions are coordinated with-respect to particular mission

An OPFAC can be simultaneously processing any number of mission instances

- e.g., 2 WRFFE-mortars, 3 AF-artillery, etc.

Current Implementation

Is monolithic; each OPFAC is an Ada program that sends and receives tactical messages

When a message is received, it (and the current state of the OPFAC) is processed by a sequence of rules:

if (conditions$_1$) do-action$_1$;

if (conditions$_2$) do-action$_2$;

if (conditions$_3$) do-action$_3$; ...

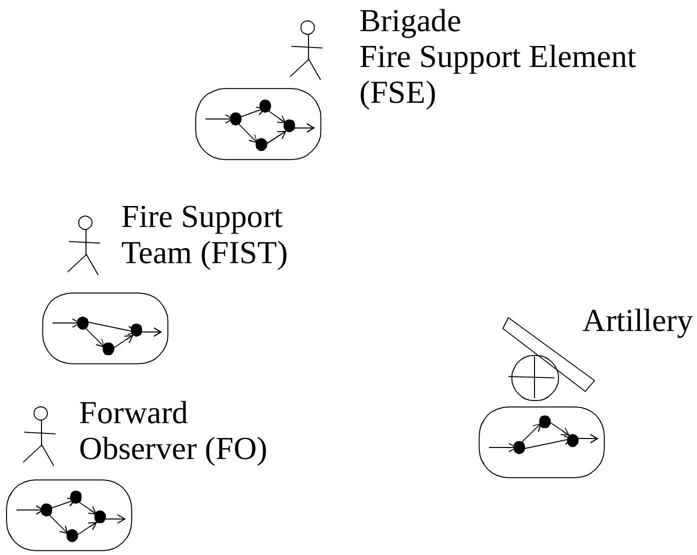
Rules encode conditions (5-10 primitives) to fire actions for one or more missions

- 200-1000+ rules per OPFAC
- difficult to write, understand, debug rules
- hard to see what rules apply to given mission
- typical example of non-extensible system; started out small and understandable, but didn’t stay that way

FSATS99

Re-engineering FSATS to be a GenVoca PLA

- *key idea: each Mission Type is a collaboration*



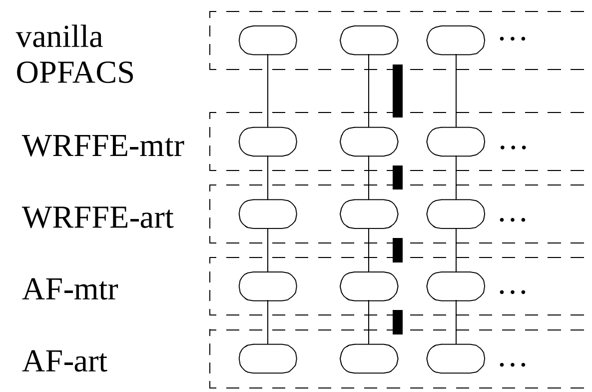
- actions of each OPFAC is defined by a *protocol* (state machine) that it follows to do its part in processing an instance of a mission type

FSATS99 (Cont)

Each OPFAC may play a role in different mission types

- for each mission type, it has a protocol to follow to process instances of that type

  thus, new code is added to process a mission (whenever a new mission type is added to FSATS99)



Most refined OPFAC classes “understand” capabilities and responsibilities of all mission types —
only bottom classes are instantiated!

Advantages of Layered Design

Mission complexity is encapsulated in one spot — the layer definition

Mission types can be debugged separately, in isolation from each other

- substantially simplifies mission development (over current version of FSATS)
- using common OO design techniques to define collaborations (layers) — e.g., state machines — it is much easier to determine if all possibilities are accounted for

New mission types can be added/removed from FSATS99 through component composition/reconfiguration

- extensible, creates PLA for FSATS simulators

©dsbatory

Introspection

Could we have built FSATS99 in a different way, using standard OO techniques?

- ans: maybe, but generally no.

Insights:

- OO design methodologies look for objects, classes, and their relationships; GenVoca seeks the fundamental *refinements* in a domain — find them first, choose their implementation later
- *OO design methodologies direct you toward the design of the most refined classes of an application, and do not expose the intermediate derivation classes. That is, an application is one big collaboration, instead of a composition of smaller reusable collaborations*
- want a higher-level ability to specify applications as type equations, rather than writing code. *We want to program at the architectural level, not code level.*

©dsbatory

GenVoca Again

FSATS99 PLA model has single realm; all components refine basic OPFAC abstraction

```
F = {   vanilla,           // basic opfacs
        wrffe-common[F],   // shared by wrffe
        wrffe-mtr[F],
        wrffe-art[F],
        af-common[F],      // shared by af
        af-art[F],
        af-mrt[F],
        …
    }
```

Compositions yield FSATS simulator. Ex:

```
fsats101 =
    wrffe-mtr[ af-art[
        wrffe-common[ af-common[
              vanilla ] ] ] ];
```


©dsbatory

Big Picture

Graph Algorithm PLA, FSATS99, JTS aren't isolated examples. Similar GenVoca architectures exist for:

- avionics (ADAGE)
- database systems (Genesis)
- network protocols (Ensemble)
- …

Graph PLA, FSATS99, JTS components are mixin-layer (template-based representations) of refinements.

Remember, there are lots of other representations!

©dsbatory

Conclusions — Lessons Learned

- ***collaboration-based designs correspond to GenVoca layers or aspects; compositions of CBDs correspond to GenVoca type equations***

  connection with important and under-appreciated OO design technique

- ***a mixin-layer is a template-based construct that implements a GenVoca refinement***

  novel mixture of parameterized inheritance and nested classes;

  mixin-layers provide scalable implementions of collaboration-based designs

- ***case studies: FSATS99, JTS (appendix)***

  nontrivial examples of scalable PLAs using mixin-layers

©dsbatory

Further Readings

D. Batory, B. Lofaso, and Y. Smaragdakis, “JTS: Tools for Implementing Domain-Specific Languages”. *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

D. Batory, C. Johnson, Bob McDonald, and Dale von Heeder, “Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study”, *6th International Conference on Software Reuse*, Vienna, Austria, 2000.

D. Batory, R. Cardone, and Y. Smaragdakis, “Object-Oriented Frameworks and Product-Lines”, *1st Software Product-Line Conference*, Denver, Colorado, 2000.

G. Bracha and W. Cook, “Mixin-Based Inheritance”, *ECOOP/OOPSLA 90*.

R. Findler and M. Flatt, “Modular Object-Oriented Programming with Units and Mixins”, ICFP 98.

M.G. Hayden, “The Ensemble System”, Ph.D. dissertation, Dept. Computer Science, Cornell, January 1998.

Y. Smaragdakis and D. Batory, “Implementing Layered Designs with Mixin Layers”. *12th European Conference on Object-Oriented Programming, (ECOOP '98)*, July 1998.

Y. Smaragdakis and D. Batory, “Implementing Reusable Object-Oriented Components.” *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

D.S. Wile, “POPART: Producer of Parsers and Related Tools”, USC/ISI, November 1993.

M. VanHilst and D. Notkin, “Using Role Components to Implement Collaboration-Based Designs”. *OOPSLA 1996.*

M. VanHilst and D. Notkin, “Decoupling Change From Design”, *ACM SIGSOFT* 1996.

©dsbatory

Appendix: JTS Case Study

JTS (Jakarta Tool Suite)

- compiler tool suite to create extensible Java languages

- motivation: need tool suite to help write domain-specific languages and domain-specific extensions to host programming languages

```
// ex #1 - metaprogramming addition to Java

AST_Exp x = exp{ q > z }exp;          // ast constructors

AST_Stm s = stm{ if (x.alpha()) foo();
                 else bar();   }stm;


// ex #2 - type equations to extend Java

class Application extends L4< L3< L2< L1 >>>;
```

- want a product-line of Java dialects where each optional feature encapsulated as a component; don’t want to build monolithic precompilers, as there would be an exponential number of them

- particular dialect specified by composition of components of desired features

Mixin-Layer Implementation of Refinements

How JTS Works

Programs are represented as parse trees (syntax trees)

- parse trees are infinitely extensible

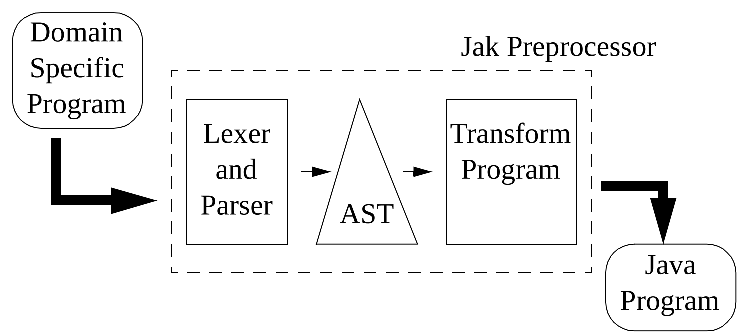
Microsoft IP’s “Intentions”

- add AST nodes with domain-specific semantics

  ex: AST constructors, type equations

- at reduction time, intention nodes are replaced with their Java (or host language) implementations

  ex: replace code constructors, type equations with their pure Java counterparts



Bali

How are lexers, parsers, and transform programs created?

- ans: Bali tool

Bali is a GenVoca generator of (Java) preprocessors

- assembles variants of *Jak (extensible Java)* from components
- components encapsulate primitive Java extensions

  ex: AST constructors
  hygienic macros
  P3 data structure generators
  layer definition and composition
  etc.

A *Bali component* is an ordered pair (syntax, semantics)

- syntax — grammar extensions to Java
- semantics — meaning given to grammar extensions

©dsbatory

Bali Syntax

Are extended, annotated BNF grammars

- extended with repetitions (see POPART)

```
StatementList : ( Statement )+
              ;

ArgumentList  : Argument ( ‘,’ Argument )*
              ;
```

- annotated by class to instantiate when production is recognized (e.g., see POPART)

```
SelectionStmt

   : IF ‘(’ Expr ‘)’ Statement  :: IfStm
   | SWITCH ‘(’ Expr ‘)’ Block  :: SwitchStm
   ;
```

from grammar, can infer constructors:

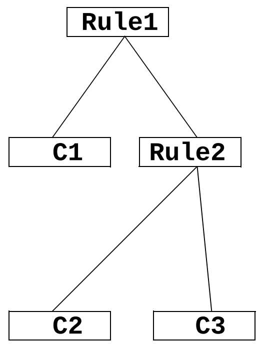
```
IfStm( token, token, AST_Exp, token, AST_Stm )

SwitchStm( token, token, AST_Exp, token, AST_Blk )
```


©dsbatory

Inheritance Hierarchies

Can be deduced from grammar specifications:

```
Rule1 : pattern1 :: C1
      | Rule2
      ;
```




```
Rule2 : pattern2 :: C2
      | pattern3 :: C3
      ;
```

Grammar specifications used for:

- defining host grammar (e.g. Java)
- defining additional rules, lexical tokens for grammar extensions

©dsbatory

Mixin-Layer Implementation of Refinements

Bali Grammar Specifications

```
// bali spec

lexical
patterns


%%

grammar rules
```

```
Jakarta grammar
- 150 tokens
- 350 productions
- 740 lines
```

What can be generated automatically:

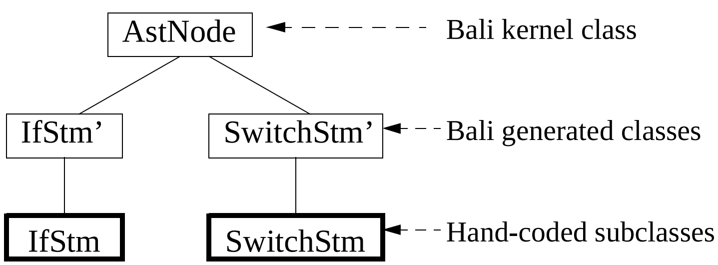
- lexical analyzer
- parser — now using JavaCC
- inheritance hierarchy and AST classes
- class constructors, unparsing, tree editing methods

©dsbatory

What Can’t be Generated?

Type checking, reduction, and optimization methods

- AST node specific
- hand code as subclasses to Bali-generated classes

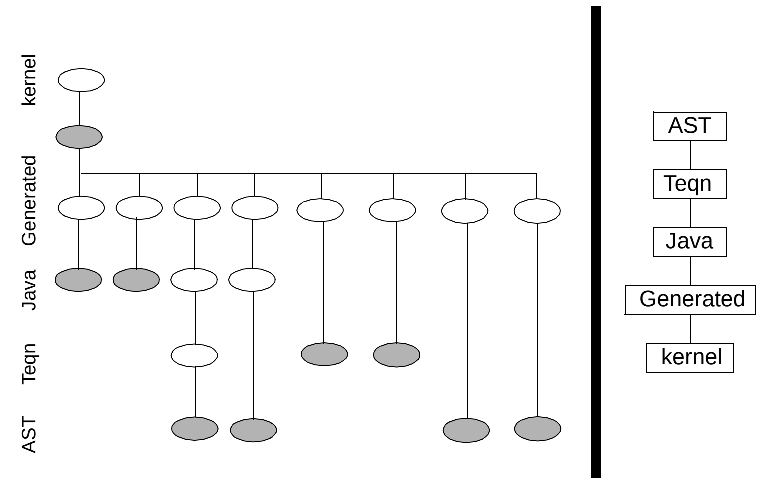


- *Bali generates a mixin layer that encapsulates all generated AST classes*
- *separate mixin-layer encapsulates the hand-coded subclasses of each class defined in a Bali-grammar file*

©dsbatory

Mixin-Layer Implementation of Refinements

Relationship to GenVoca/Mixin-Layers



Notes:

- terminals of refinement chains are the classes that are instantiated
- *type equation to scale*:

```
Jak = AST[ Teqn[ Java[ Generated[ kernel ] ] ] ];
```

- *inheritance hierarchy not drawn to scale*: 500+ classes, some mixin layers have over 100 classes each

©dsbatory

GenVoca Again

JTS PLA model for Java dialects has 2 realms:

```
K = {   kernel   }
J = {   java1_0[K],    // Java 1.0
        java1_1[J],    // Java 1.1 ext to 1.0
        AST[J],        // metaprogramming
        gscope[J],     // hygienic macros
        layerdef[J],   // layer definitions
        Teqn[J],       // type equations
        …
    }
```

Compositions yield a Java dialect. Ex:

```
JavaPlusPlus =
    Teqn[ layerdef[
          AST[ java1_1[ java1_0[
                 kernel ] ] ] ];
```


©dsbatory

Design Rule Checking

Not all syntactically correct combinations of GenVoca components are semantically correct. Some components work only in the presence (or absence) of other components

- fundamental problem: impossible for generator users to debug generated code; need automated help to debug component compositions
- design rules are domain-specific constraints that specify illegal configurations of components. *Design rule checking (DRC)* is the process of (automatically) applying design rules

In this lecture, we present:

- a model of DRC based on attribute grammars
- has been used in every GenVoca PLA
- relate DRC to research on software architectures

©dsbatory

Motivating Example: P3

Generator for container data structures

- relies on two realms containing 50 components:

```
ds  = { bintree[ ds ],      // binary tree
        dlist[ ds ],        // unordered list
        odlist[ ds ],       // ordered list
        avail[ ds ],        // free list manager
        array[ mem ],       // sequential storage
        malloc[ mem ],      // random storage
        inbetween[ ds ],    // common delete code
        … }

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

Data structures are modeled by type equations

- reference 5 to 15 components
- too elaborate to validate by inspection
- some components have obscure rules for their use

©dsbatory

Example P3 Design Rule

`inbetween` component encapsulates:

- algorithms shared by many data structure components (e.g., `bintree` and `dlist`)
- deals with positioning of cursor after element is deleted
- details complex, hidden from user

Correct usage of `inbetween` requires:

- one copy in TE that has 1+ data structure components &
- precede all such components in equation

```
right = ...inbetween[ ... dlist[ bintree[ ... ]]];
wrong = ... dlist[ inbetween[ bintree[ ... ]]];
```

Such rules should not be borne by programmers

- too easy to forget and be misapplied

want rules tested automatically

©dsbatory

Results from Software Architectures

Perry's *Inscape* (1989) is environment for managing evolution of software systems:

- novel aspects: obligations and consistency checking

light semantics

Components have pre-, post-conditions, and obligations

bank loan example

- *obligations* are conditions that must be satisfied by system that uses the component
- require "action-at-a-distance" — nonlocally satisfied
- propagated to enclosing modules where they are eventually satisfied by some postcondition

Full-fledged verification not attempted

- primitive predicates declared (but informally defined)
- pre-, post-, obligations expressed in terms of primitives
- practical & powerful form of "shallow" consistency checking using pattern matching and simple deductions

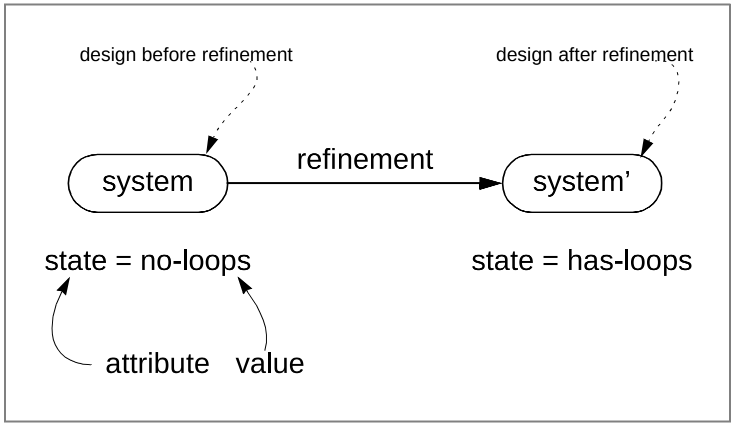
©dsbatory

Design Rule Checking

Adapt and generalize Inscape consistency checking to DRC by exploiting the semantics of GenVoca layers

(1) DRC models states of system (TE) *design*

- *not states of system execution*
- model states / properties of system design by assigning values to attributes
- *exploit refinement interpretation of layers*



©dsbatory

DRC Basics

(2) Preconditions and obligations of component K are satisfied “at-a-distance” by components that lie either:

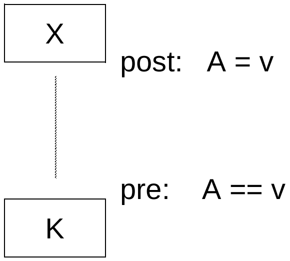
- (far) beneath K or
- (far) above K

constraints typically not satisfied by adjacent components (c.f. Goguen, Tracz, Sitaraman references)

Properties exported to “higher” layers generally not the same as properties exported to “lower” layers

Leads to 2 kinds of design rules:

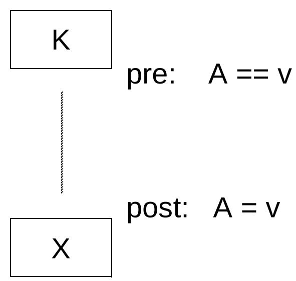
- *#1: preconditions for component usage*



©dsbatory

DRC Basics (Continued)

- *#2: preconditions for parameter instantiation*



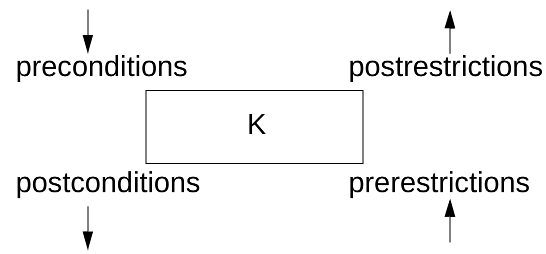
new names: preconditions called *prerestrictions*

postconditions called *postrestrictions*

note: prerestrictions correspond to Inscape obligations

DRC Basics (Continued)

Components have:



Design rule checking involves:

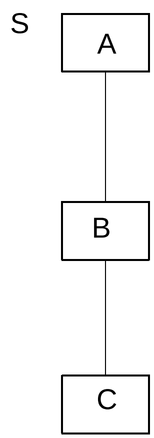
- top-down propagation of postconditions and testing component preconditions
- bottom-up propagation of postrestrictions and testing of parameter prerestrictions

In following, we assume no restriction on complexity of predicates, but will later show that very simple predicates suffice for P3 *(and other domains as well)*.

Top-Down DRC

Components have preconditions and postconditions

top -- initial conditions for composition S



top $\Rightarrow$ precondition-A

postcondition-A $\oplus$ top = top'

top' $\Rightarrow$ precondition-B

postcondition-B $\oplus$ top' = top"

top" $\Rightarrow$ precondition-C

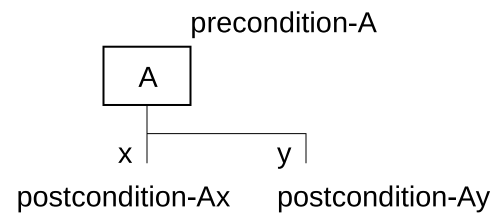
postcondition-C $\oplus$ top"

- postconditions propagated by $\oplus$ operator
- conditions tested by $\Rightarrow$ operator

A Twist...

Consider component with multiple parameters: A[x,y]

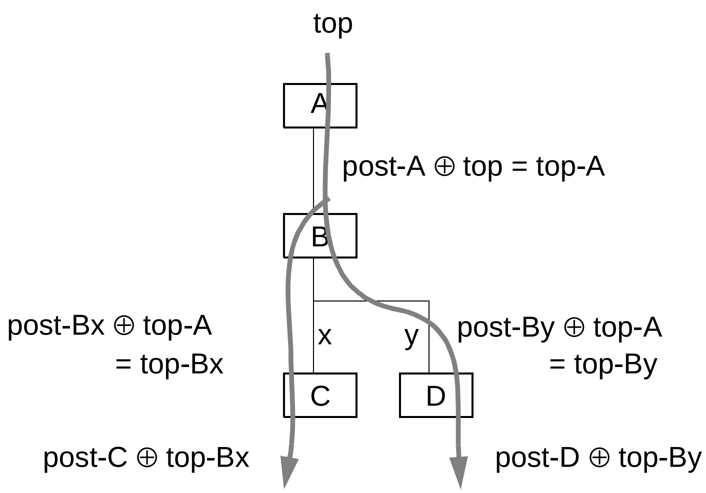
- gives rise to trees (dags)
- twist: each parameter has its own postcondition



i.e., conditions for parameter x may be different for those of y in A[x,y]

- example: the realm of a parameter could be expressed as a postcondition; realms for x and y could be different

Top-Down Algorithm



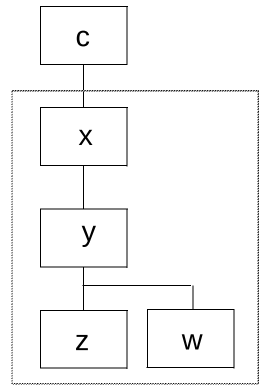
simple recursive algorithm for top-down propagation of conditions and testing component preconditions

©dsbatory

Bottom-Up DRC

Conditions must also be propagated upwards...

- parameters of components have *prerestrictions* for instantiations to be correct



***prerestrictions for* c *are generally not satisfied by the component* x *that instantiates its parameter, but rather by components deep within the system rooted by* x**

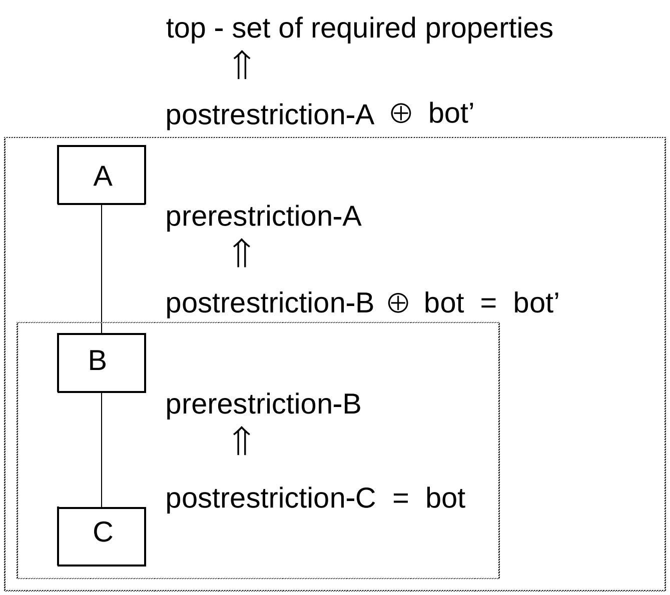
- *systems instantiate parameters, not components*

- exported states (called *postrestrictions*) propagated upwards so that prerestrictions can be tested

©dsbatory

Bottom-Up DRC Continued

Every component has *postrestrictions*, i.e., exported states, and *prerestrictions* for each parameter



- use same operators ⊕ and ⇒ for bottom-up DRC
- simple recursive algorithm for bottom-up DRC

©dsbatory

Attribute Grammars

McAllester observed *attribute grammars* unify realms, attributes, top-down & bottom-up DRC algorithms

- realms of components modeled by grammars
- attributes model program development states
- postconditions are *inherited attributes* (values determined by ancestors)
- postrestrictions are *synthesized attributes* (values determined by descendants)

Bonus: common tools (lex, yacc) well-suited for implementing design rule checkers

Need to supply definitions & representations for:

- attributes
- predicates
- operators ⊕ and ⇒

For the P3 and Genesis generators…

©dsbatory

P3 Attributes

Each attribute models a property that exposes a composition constraint

Attributes have restricted values:

Value	Interpretation
any	nothing is known about property
assert	property is asserted
negate	property is negated
inherit	attribute value inherited, but is otherwise unconstrained

- example:

attribute: component_belongs_to_realm_A

attribute value: assert (or negate)

- see Batory and Geraci *IEEE TSE 1997* paper (and report UTCS TR-94-03 for other values…)

©dsbatory

P3 Predicates

Preconditions & prerestrictions request specific attribute values (any, assert, negate), but not how they were determined (i.e., inherit)

- only 4 different *primitive* predicates:

Predicate	Interpretation
P-any	true (no constraints)
P-assert	attribute has assert value
P-negate	attribute has negate value
P-false	false (unsatisfiable)

- complex predicates are typically conjunctions of primitives, one primitive predicate for each attribute
- encode as vector of predicates indexed by attribute

$$P \equiv P_1 \wedge P_2 \wedge \ldots \equiv [\ P_1,\ P_2,\ \ldots\ ]$$

©dsbatory

Postcondition Propagation Operator $\oplus$

Component postconditions assert or negate values, or may propagate values

- table below defines the condition propagation operator + for a single attribute:

component postcondition + existing condition		component postcondition		
		inherit	assert	negate
existing condition	any	any	assert	negate
	assert	assert	assert	negate
	negate	negate	assert	negate

- given $P = [\, P_1, P_2, \ldots \,]$ and $E = [\, E_1, E_2, \ldots \,]$

$$P \oplus E \;=\; [\; P_1 + E_1,\; P_2 + E_2,\; \ldots \;]$$

Implication Operator $\Rightarrow$

The implication operator $\rightarrow$ for a single attribute is:

Existing Condition $\longrightarrow$ Precondition		Precondition			
		P-any	P-assert	P-negate	P-false
Existing Condition	assert	true	true	false	false
	negate	true	false	true	false
	any	true	false	false	false

- given $E = [\, E_1, E_2, \ldots \,]$ and $P = [\, P_1, P_2, \ldots \,]$

$$E \Rightarrow P \;=\; (\, E_1 \rightarrow P_1 \,) \wedge (\, E_2 \rightarrow P_2 \,) \wedge \ldots$$

Implementation Notes

Straightforward implementation: 1500 lines in **`lex & yacc`**

DRC algorithm is efficient: O(mn)

m = # of attributes, n = # of components

Example domain models:

Generator (Domain)	**# of Realms**	**# of Components**	**# of Attributes**
Genesis (databases)	9	52	14
JTS (Java precompilers)	1	10	10
P3 (data structures)	2	50	7

Some P3 attributes:

attribute	**property description**
logical_deletion	"a logical deletion layer"
retrieval	"a retrieval layer"

Design Rule Checking

Straightforward Specifications

Example component & design rule declaration:

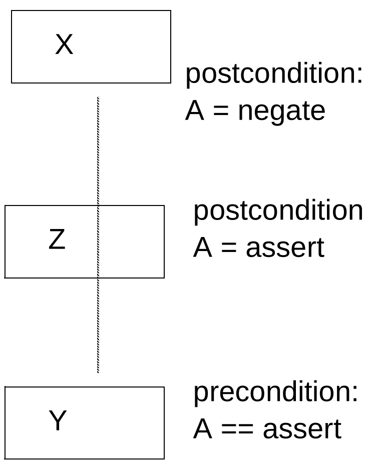
name of component
realm of component
realm parameters of component
design rules

```
array : ds [ mem ] {

   # logical del. layer required above array

   precondition    assert   logical_deletion

   # assert that array is a retrieval
   # layer to all descendants and ancestors

   postcondition   assert   retrieval

   postrestriction assert   retrieval

}
```

Explanation Based Error Reporting

In addition to detecting errors, we can extend DRC algorithms to suggest how to repair a type equation

- *precondition ceilings* from Inscape:



- error located in between components X and Y
- similar technique for obligations/prerestrictions

©dsbatory

Example

Example: want container implementation that stores elements onto a binary tree, whose nodes are stored sequentially in transient memory

First attempt at composition:

```
first = top2ds[ bintree[ array[ transient ]]];
```

DRC response:

```
Precondition errors:

   an inbetween layer is expected between top2ds
       and bintree
   a logical deletion layer is expected between
       top2ds and array

Prerestriction error:

   parameter 1 of top2ds expects a subsystem with
   a qualification layer
```


©dsbatory

Explanation Based Error Reporting

Clumsy fix…

```
second = top2ds[ inbetween[ bintree[ qualify[
               delflag[ array[ transient ]]]]]];
```

DRC response:

```
Precondition error:

  a retrieval layer (bintree) not expected above
         qualify
```

Correct type equation — swap **`qualify`** with **`bintree`**:

```
third = top2ds[ inbetween[ qualify[ bintree[
            delflag[ array[ transient ]]]]]];
```

DRC directs users to modify TE to the "nearest" correct TEs in space of all TEs

©dsbatory

Design Rule Checking

Insights

Why isn't DRC a challenging problem in program verification?

- solution unlikely to be automatable

Inscape work and our work have observed:

- problem is straightforward
- solution automatable and efficient

… why? … 2 reasons

Reason #1:

- shallow consistency checking goes a very long way
- in general, most logical errors are shallow errors

  conjecture: all errors at component composition level should be shallow

- remaining errors must be dealt with by component implementors

©dsbatory

Insights (Cont)

Reason #2: important distinction:
- Inscape components are functions
- GenVoca components are subsystems

Large applications consist of tens of thousands of lines of code
- hundreds or thousands of functions

  $\Rightarrow$ hundreds or thousands of primitive predicates

- TEs rarely have more than 50 components

  $\Rightarrow$ modest # of primitive predicates in a domain ~10-40

seems counterintuitive

Why?
- modeling states of development (not execution) reduces number of properties to examine
- and GenVoca is a methodology for designing reusable components ...

©dsbatory

Design Rule Checking

The Key

What makes OO designs so powerful and attractive?
- Ans: ability to manage and control software complexity

Standardization is powerful way of managing and controlling software complexity in product-line architecture

Standardization makes some problems tractable that would otherwise be very difficult

- ex: composing off-the-shelf components
- composition of components is simple in GenVoca
- standardization seems to limit the ways in which components can constrain each other’s behavior
  $\Rightarrow$ make DRC tractable

Historical perspective…

©dsbatory

Additional Insights

We understand software in terms of implementation-independent refinements

- enhances power of design rule checking
- DRC tells you whether two refinements (concepts) can be composed *regardless how they are implemented*

  ex: **`bintree[ encrypt[...] ]`** may be correct
  ex: **`encrypt[ bintree[...] ]`** is always incorrect

- design rules allow you to state whether certain combinations of *concepts* or *features* (i.e., refinements) are possible

©dsbatory

Design Rule Checking

Conclusions — Lessons Learned

- ***GenVoca domain models (realms + design rules) are attribute grammars***

  can use existing tools (lex, yacc) to express models

- ***simple, efficient algorithms for DRC***

  constraints imposed on higher layers (preconditions)

  constraints imposed on lower layers (prerestrictions)

  don’t need formal methods, theorem provers

- ***components that are designed to be interoperable, plug-compatible, and interchangeable often makes complex problems much easier to solve***

  standardization of programming abstractions is a powerful way of controlling the complexity of a product-line (i.e., family of systems)

©dsbatory

Further Reading

D. Batory and J. Barnett, “DaTE: The Genesis DBMS Software Layout Editor”, in *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*, P. Loucopoulos and R. Zicari, editors, Wiley, 1992.

D. Batory and B.J. Geraci, “Validating Component Compositions and Subjectivity in GenVoca Generators”, *IEEE Transactions on Software Engineering*, February 1997, 67-82.

L. Blaine and A. Goldberg, “DTRE - A Semi-Automatic Transformation System”, in *Constructing Programs from Specifications*, Elsevier Science Publishers, 1991.

J.A. Goguen, “Reusing and Interconnecting Software Components”, *IEEE Computer*, February 1986.

J. Neighbors, “Draco: A Method for Engineering Reusable Software Components”, in T.J. Biggerstaff and A. Perlis, eds., *Software Reusability*, Addison-Wesley/ ACM Press, 1989.

M.D. Katz and D.J. Volper, “Constraint Propagation in Software Libraries of Transformation Systems”, *Int. Journal Software Engineering and Knowledge Engineering*, Vol. 2 #3 (1992), 355-374.

D. McAllester. “Variational Attribute Grammars for Computer Aided Design.” ADAGE-MIT-94-01.

M. Moriconi and X. Qian, “Correctness and Composition of Software Architectures”, *ACM SIGSOFT 1994*.

D.E. Perry and A.L. Wolf, “Foundations for the Study of Software Architecture”, *ACM SIGSOFT Software Engineering Notes*, October 1992, 40-52.

D.E. Perry, “The Inscape Environment”, *Proc. ICSE 1989*, 2-12.

D.E. Perry, “Software Interconnection Models”, *Proc. ICSE 1989*, 61-69.

©dsbatory

Further Reading (Continued)

D.E. Perry, “The Logic of Propagation in The Inscape Environment”, *ACM SIGSOFT 1989*, 114-121.

M. Sitaraman and B. Weide, “Component-Based Software Using RESOLVE”, *ACM Software Engineering Notes*, October 1994.

Y. Smaragdakis and D. Batory, “Implementing Layered Designs with Mixin Layers”, *ECOOP 1998*.

W. Tracz, “LILEANNA: A Parameterized Programming Language,” *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*. Lucca, Italy. R. Prieto-Dìaz and W.B. Frakes, eds. IEEE Computer Science Press, March 24-26, 1993.

©dsbatory

Reading and Reference List

1 Background

1.1 Historical Precedence

McIlroy was among the first to identify the problem of library scalability; the notion virtual machines is due to Dijkstra, and families of systems is due to Parnas.

D. McIlroy, “Mass Produced Software Components”, *Software Engineering: Report on a Conference by the Nato Science Committee*, Oct 1968, P. Naur and B. Randell, eds. 138-150.

E.W. Dijkstra, “The Structure of THE Multiprogramming System”, *Communications of ACM*, May 1968, 341-346.

D.L. Parnas, “Designing Software for Ease of Extension and Contraction”, *IEEE Transactions on Software Engineering*, March 1979.

1.2 Parameterized Programming

The concepts of horizontal parameterization (i.e., parameterizing interfaces) and vertical parameterization (i.e., layering) have been expressed elegantly by Goguen and Tracz. The certification of parameterized components has been examined in the RESOLVE project.

J.A. Goguen, “Reusing and Interconnecting Software Components”, *IEEE Computer*, February 1986. Also in Prieto-Diaz and Arango text (below).

W. Tracz, “LILEANNA: A Parameterized Programming Language,” *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability*. Lucca, Italy. R. Prieto-Dìaz and W.B. Frakes, eds. IEEE Computer Science Press, March 24-26, 1993.

M. Sitaraman and B. Weide, “Component-Based Software Using RESOLVE”, *ACM Software Engineering Notes*, October 1994.

©dsbatory

1.3 Large System Development

References that survey the problems of large system development (and indirectly, problems that arise in domain modeling) are:

B. Curtis, H. Krasner, and N. Iscoe, “A Field Study of the Software Design Process for Large Systems”, *Communications of the ACM*, November 1988.

G. Booch, *Object Solutions: Managing the Object-Oriented Project*, Addison-Wesley, 1995.

1.4 Object-Oriented Frameworks

GenVoca components encapsulate suites of interrelated classes. So too do object-oriented frameworks; they are suites of interrelated abstract classes that have multiple concrete class implementations, which describe different implementations of what we have called “subsystems”. A framework is an OO way of representing a realm of components; the abstract classes define both the interface of a realm and code that is common across all components, whereas concrete subclasses provide component-specific implementations. What frameworks lack are parameterizations and method wrappers that are needed to express GenVoca compositions (See **Subjectivity** and **Method Wrappers**).

R.H. Campbell and N. Islam, “A Technique for Documenting the Framework of an Object-Oriented System”, *IEEE 2nd International Workshop on Object Orientation in Operating Systems (1992)*, 288-300.

R.H. Campbell, N. Islam, and P. Madany, “Choices, Frameworks and Refinement”, *Computing Systems*, 5(3), 1992.

R.E. Johnson and B. Foote, “Designing Reusable Classes”, *Journal of Object-Oriented Programming*, June/July 1988. Also in Prieto-Diaz and Arango text (below).

R.E. Johnson, “Documenting Frameworks using Patterns”, *OOPSLA 1992*, 63-76.

R.E. Johnson, “How to Design Frameworks”, Tutorial Notes, 1993.

G.C. Murphy and D. Notkin, “The Interaction Between Static Typing and Frameworks”, TR 93-09-02, Dept. Computer Science and Engineering, University of Washington, Seattle, 1993.

©dsbatory

1.5 Role-Based Designs and Mixins

Role-based designs is a design technique that encapsulates features of applications through a set of classes that perform specific roles. Role-based designs are an object-oriented design technique that can be used to design GenVoca layers. Implementing role-based designs is through the use of mixins, classes whose superclass are specified via a parameter. The following papers survey recent work on mixins and on role-based designs.

G. Bracha and W. Cook, "Mixin-Based Inheritance", *ECOOP/OOPSLA 90*, 303-311.

G. Bracha and D. Griswold, "Extending Smalltalk with Mixins", *Workshop on Extending Smalltalk* at OOPSLA 96. See `http://java.sun.com/people/gbracha/mwp.html`.

M. Flatt, S. Krishnamurthi, M. Felleisen, "Classes and Mixins". ACM *Symposium on Principles of Programming Languages*, 1998 (PoPL 98).

R.B. Findler and M. Flatt, "Modular Object-Oriented Programming with Units and Mixins", *ICFP 98*.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", *ECOOP 97*, 220-242.

C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects", *ECOOP* 97, 419-443.

Y. Smaragdakis and D. Batory, "Implementing Reusable OO Components", *International Conference on Software Reuse, 1998*.

Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers", *ECOOP 1998*.

M. VanHilst and D. Notkin, "Using C++ Templates to Implement Role-Based Designs", *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 22-37.

M. VanHilst and D. Notkin, "Using Role Components to Implement Collaboration-Based Designs". *OOPSLA 1996*.

M. VanHilst and D. Notkin, "Decoupling Change From Design", *ACM SIGSOFT* 1996.

M. VanHilst, "Role-Oriented Programming for Software Evolution", Ph.D. Dissertation, University of Washington, Computer Science and Engr., 1997.

©dsbatory

Reading and Reference List

1.6 Domain Modeling

A key problem in modeling domains is choosing the right abstractions. The following references offer a variety of practical perspectives on this topic. (See also papers on **Frameworks**, and **Role-based Designs and Mixins**).

D. Batory, L. Coglianese, M. Goodwin, and S. Shafer, "Creating Reference Architectures: An Example From Avionics", *ACM SIGSOFT Symposium on Software Reusability*, Seattle, 1995, 27-37.

H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I. Tavakoli, "A Prototype Domain Modeling Environment for Reusable Software Architectures", *Third International Conference on Software Reuse, Rio de Janeiro*, November 1-4, 1994, 74-83.

R. Prieto-Diaz and G. Arango (ed.), *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press 1991.

1.7 Software Reuse

Good overviews of the state-of-art results and problems in software reuse are:

P.G. Bassett, *Framing Software Reuse*, Yourdon Press Computing Series. 1996.

T. Biggerstaff and C. Richter, "Reusability Framework, Assessment and Directions", *IEEE Software*, March 1987.

T. Biggerstaff, "An Assessment and Analysis of Software Reuse", in *Advances in Computers*, Volume 34, Academic Press, 1992.

T. Biggerstaff, "A Perspective on Generative Reuse", *Annals of Software Engineering*, 5 (1998), 169-226.

C. Krueger, "Software Reuse", *ACM Computing Surveys*, 24(2), June 1992, 131-183.

H. Mili, F. Mili, A. Mili, "Reusing Software: Issues and Research Directions", *IEEE Transactions on Software Engineering*, June 1995, 528-562.

1.8 Transformation Systems

GenVoca domain models can be implemented by program transformation systems. Simonyi's paper describes an innovative approach to the integration of transformation systems and compilers to create extensible programming languages. Weigert's paper describes an impressive transformation system that is being used at Motorola

©dsbatory

to produce customized software for different radio-products. Griswold's paper deals with semantics preserving program transformations. The papers by Roberts, et al and Tokuda et al describe innovative approaches to creating tools for editing OO programs by transformations. Pu's paper is an example of a dynamic generative generator, and the other papers deal with transformations in the context of "parameterized layers" of rewrite rules:

I. Baxter, "Design Maintenance Systems", *CACM* April 1992, 73-89.

I. Baxter, "Transformation Systems: Theory, Implementation, and Survey", Tutorial Notes, 1996.

T. Biggerstaff, "A Perspective on Generative Reuse", *Annals of Software Engineering*, 5 (1998), 169-226.

L. Blaine and A. Goldberg, "DTRE - A Semi-Automatic Transformation System", in *Constructing Programs from Specifications*, Elsevier Science Publishers, 1991.

W.G. Griswold, "Direct Update of Data Flow Representations for a Meaning-Preserving Program Restructuring Tool", *ACM SIGSOFT 1993*.

J. Neighbors, "Draco: A Method for Engineering Reusable Software Components", in T.J. Biggerstaff and A. Perlis, eds., *Software Reusability*, Addison-Wesley/ACM Press, 1989.

H. Partsch and R. Steinbruggen, "Program Transformation Systems", *Computing Surveys*, March 1983, 199-236.

C. Pu, H. Massalin, and J. Ioannidis, "The Synthesis Kernel", *Computing Systems*, 1(1):11-32, Winter 1988.

D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk", University of Illinois at Urbana-Champaign, Dept. Computer Sciences, 1997.

C. Simonyi, "The Death of Computer Languages, the Birth of Intentional Programming", Microsoft Corporation, Sept 1995.

D.R. Smith, "KIDS: A Semiautomatic Program Development System", *IEEE Transactions on Software Engineering*, Sept. 1990, 1024-1043.

L. Tokuda and D. Batory, "Evolving Object-Oriented Architectures with Refactorings", *Automated Software Engineering Conference*, October 1999.

T.J. Weigert, J.M. Boyle, T.J. Harmer, "Knowledge-Based Derivation of Programs from Specifications", *Artificial Intelligence in Automation*, World Scientific Press, 1996.

©dsbatory

1.9 Software Architectures

Software architectures deal with issues of "programming-in-the-large" and building software systems from components. A variety of popular perspectives are:

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

D. Garlan and M. Shaw, "An Introduction to Software Architecture", in *Advances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific Publishing Company, 1993.

D. Garlan, et al, "Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts", *ICSE 1995*.

M.M. Gorlick and R.R. Razouk, "Using Weaves for Software Construction and Analysis", *Proc. ICSE* 1991, 23-34.

D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture", *ACM SIGSOFT Software Engineering Notes*, October 1992, 40-52.

J. Udell, "Componentware", *BYTE*, May 1994.

E. White and J. Purtilo, "Integrating the Heterogeneous Control Properties of Software Modules", *ACM SIGSOFT 1992*.

1.10 Product-Line Architectures

A natural out-growth of research on architectures is product-line architectures. The idea of building families of applications isn't new (see Parnas) nor is building families of applications from components (that's GenVoca, Draco). However, some fresh ideas have begun to surface. You may want to consult the Proceedings of the *1st Software Product-Lines Conference*, Denver, Colorado in August 2000.

J. Bosch, "Product-Line Architectures in Industry: A Case Study", *1999 International Conference on Software Engineering.*

Software Engineering Institute, "The Product-Line Practice Initiative", **http://www.sei.cmu.edu/plp/plp_init.html**

D. Batory, "Product-Line Architectures", Invited presentation, *Smalltalk und Java in Industrie and Ausbildung*, Erfurt, Germany, October 1998.

D. Batory, R. Cardone, and Y. Smaragdakis, "Object-Oriented Frameworks and Product-Lines", *1st Software Product-Line Conference*, Denver, Colorado, 2000.

©dsbatory

1.11 Subjectivity

When modeling families of related applications, objects do not have single interfaces. Rather, they are described by families of related interfaces. The interface that is appropriate for an application is application specific (i.e., subjective). The following references give an overview of current thinking on the topic. Also see **Role-Based Designs and Mixins**.

D. Batory and B.J. Geraci, "Validating Component Compositions and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, February 1997, 67-82.

W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", *OOPSLA 1993*, 411-428.

W. Harrison, H. Ossher, R.B. Smith, and D. Ungar, "Subjectivity in Object-Oriented Systems: Workshop Summary", *Addendum to OOPSLA 1994*.

H. Ossher and W. Harrison, "Combination of Inheritance Hierarchies", *Proc. OOPSLA 1992*, 25-40.

H. Ossher, et al., "Subject-Oriented Composition Rules", *OOPSLA 1995*, 235-250.

Y. Smaragdakis and D. Batory, "Implementing Reusable OO Components", *International Conference on Software Reuse, 1998*.

Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers", *ECOOP 1998*.

M. Van Hilst and D. Notkin, "Using Role Components to Implement Collaboration-Based Designs", *OOPSLA 1996*, 359-369.

M. Van Hilst and D. Notkin, "Using C++ Templates to Implement Role-Based Designs", *Proc. JSSST Int. Symposium on Object Technologies for Advanced Software*, Springer-Verlag 1996, 22-37.

M. Van Hilst and D. Notkin, "Decoupling Change from Design", *SIGSOFT 1996*.

1.12 Component Design

The following papers give an overview of techniques on how components can be implemented. Please refer to the **Subjectivity** lecture for an overview. (See also **Method Wrappers** and **Role-based Designs and Mixins**).

J.S. Heidemann and G.J. Popek, "File-System Development with Stackable Layers", *ACM Transactions on Computer Systems*, 12(1), 58-89.

©dsbatory

N. Hutchinson and L. Peterson, "The *x*-kernel: an Architecture for Implementing Network Protocols", *IEEE Trans. Software Engineering*, January 1991.

M. Van Hilst and D. Notkin, in **Subjectivity** above.

Y. Smaragdakis and D. Batory, in **Subjectivity** above.

1.13 Method Wrappers

The encapsulation of method wrappings within components is an important part of the GenVoca model. The following references provide a state-of-the-art look at current ideas in method wrappings:

D. Batory, "Subjectivity and GenVoca Generators", *Fourth International Conference on Software Reuse, Orlando, Florida*, April 1996.

D. Batory and Y. Smaragdakis, "Another Look at Architectural Styles and ADAGE", Loral FSD Owego T.R. ADAGE-UT-95-02.

S. Danforth and I. Forman, "Reflections on Metaclass Programming in SOM", *OOPSLA 1994*, 440-452.

I.R. Forman, S. Danforth, and H. Madduri, "Composition of Before/After Metaclasses in SOM", *OOPSLA 1994, 427-439*.

P. Graham, *ANSI Common Lisp*, Prentice Hall, 1995.

J.S. Heidemann and G.J. Popek, "File-System Development with Stackable Layers", *ACM Transactions on Computer Systems*, 12(1), 58-89.

G. Kiczales, J. des Rivieres, and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

1.14 Validating Component Compositions

Shallow consistency checking is an important technique used in software architectures and software system generators in order to validate compositions of components. It offers a practical approach to consistency checking without requiring full-fledged verification.

D. Batory and B.J. Geraci, "Validating Component Compositions and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, February 1997, 67-82.

©dsbatory

M.D. Katz and D.J. Volper, “Constraint Propagation in Software Libraries of Transformation Systems”, *Int. Journal Software Engineering and Knowledge Engineering*, Vol. 2 #3 (1992), 355-374.

M. Moriconi and X. Qian, “Correctness and Composition of Software Architectures”, *ACM SIGSOFT 1994*.

D. E. Perry, “The Inscape Environment”, *Proc. ICSE 1989*, 2-12.

D.E. Perry, “Software Interconnection Models”, *ICSE 1987*.

D.E. Perry, “The Logic of Propagation in The Inscape Environment”, *ACM SIGSOFT 1989*.

Y. Smaragdakis and D. Batory, “Implementing Layered Designs with Mixin Layers”, *ECOOP 1998*.

1.15 Scalability

Papers that put forth similar arguments for library scalability are:

D. Batory, V. Singhal, M. Sirkin, and J. Thomas, “Scalable Software Libraries”, *ACM SIGSOFT* 1993.

T. Biggerstaff, “The Library Scaling Problem and the Limits of Concrete Component Reuse”, *Third International Conference on Software Reuse, Rio de Janeiro*, November 1-4, 1994, 102-110.

I.R. Forman, S. Danforth, and H. Madduri, “Composition of Before/After Metaclasses in SOM”, *OOPSLA 1994, 427-439.*

D. McIlroy, “Mass Produced Software Components”, *Software Engineering: Report on a Conference by the Nato Science Committee*, Oct 1968, P. Naur and B. Randell, eds. 138-150.

H. Ossher and W. Harrison. “Combination of Inheritance Hierarchies”, *Proc. OOPSLA 1992*, 25-40.

1.16 Generators

GenVoca is not the only way in which software generators are implemented. Other approaches include:

B. Abbott, T. Bapty, T., C. Biegl, G. Karsai, J. Sztipanovits: "Model-Based Approach for Software Synthesis," *IEEE Software*, pp. 42-53, May, 1993.

©dsbatory

P.G. Bassett, *Framing Software Reuse*, Yourdon Press Computing Series. 1996.

K. Czarnecki and U.W. Eisenecker, “Synthesizing Objects”, *ECOOP 1999*.

H. Gomaa, L. Kerschberg, et. al., paper in **Domain Modeling** (above).

M.L. Griss and K.D. Wentzel, “Hybrid Domain-Specific Kits for a Flexible Software Factory”, *Proc. ACM SAC’94*, March 1994, 47-52.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, *ECOOP 97*, 220-242.

R. Kieburtz, L. McKinney, J. Bell, J. Hook, A.Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith and L. Walton, “A Software Engineering Experiment in Software Component Generation”, *ICSE*1996.

J.C.S. do Prado Leite, M. Sant’Anna, and F.G. de Freitas, “Draco-Puc: A Technology Assembly for Domain Oriented Software Development”, *Third International Conference on Software Reuse, Rio de Janeiro*, November 1-4, 1994, 94-101.

K.J. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996.

J. Neighbors, “The Draco Approach to Constructing Software from Reusable Components”, *IEEE Trans. Software Engineering*, Sept. 1984, 564-574.

J.Q. Ning, K. Miriyala, and W. Kozaczynski, “An Architecture-Driven, Business-Specific, and Component-Based Approach to Software Engineering”, *Third International Conference on Software Reuse, Rio de Janeiro*, November 1-4, 1994, 84-93.

G. Novak, “Creation of Views for Reuse of Software with Different Data Representations”, *IEEE Transactions on Software Engineering*, December 1995, 993-1005.

2 GenVoca

The first paper covers many of the basic concepts of GenVoca. The second paper focuses on composition validation and subjectivity. The third paper presents a broad overview of results and experiences using GenVoca generators. The “Design Wizards” paper shows how type equations can be optimized. The Czarnecki and Eisenecker paper gives a very good overview of the goals of layering, software synthesis, and product-lines.

©dsbatory

D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.

D. Batory, "Component Validation and Subjectivity in GenVoca Generators", *to appear IEEE Trans. Software Engineering, 1997*.

D. Batory, "Intelligent Components and Software Generators", Invited presentation to the Software Quality Institute Symposium on Software Reliability, Austin, Texas, April 1997. Technical Report 97-06, Department of Computer Sciences, University of Texas at Austin, February 1997.

D. Batory, G. Chen, E. Robertson, and T. Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators", to appear in *IEEE Transactions on Software Engineering*, March 2000.

K. Czarnecki and U.W. Eisenecker, "Synthesizing Objects", *ECOOP 1999*.

Also see **http://www.cs.utexas.edu/users/schwartz/index.html**

2.1 ADAGE (Avionics)

D. Batory, Y. Smaragdakis, and L. Coglianese, "Architectural Styles as Adaptors Software Architecture", Kluwer Academic Publishers, Patrick Donohoe, ed., 1999.

D. Batory, "Subjectivity and Software System Generators", *Fourth International Conference on Software Reuse*, Orlando, Florida, April 1996.

D. Batory, L. Coglianese, M. Goodwin, and S. Shafer, "Creating Reference Architectures: An Example From Avionics", *ACM SIGSOFT Symposium on Software Reusability*, Seattle, 1995, 27-37.

L. Coglianese and R. Szymanski, "DSSA-ADAGE: An Environment for Architecture-based Avionics Development", *Proceedings of AGARD 1993*.

D. McAllester, "Variational Attribute Grammars for Computer Aided Design." ADAGE-MIT-94-01.

W. Tracz and L. Coglianese, "An Adaptable Software Architecture For Integrated Avionics", *Proceedings of NAECON'93*, May 1993.

©dsbatory

2.2 Avoca/*x*-kernel (Communication Networks)

N. Hutchinson, L. Peterson, S. O'Malley, and M. Abbott, "RPC in the *x*-Kernel: Evaluating New Design Technique", *Symposium on Operating System Principles*, (December 1989), 91-101.

N. Hutchinson and L. Peterson, "The *x*-kernel: an Architecture for Implementing Network Protocols", *IEEE Trans. Software Engineering*, January 1991.

S. O'Malley and L. Peterson, "A Dynamic Network Architecture", *ACM Transactions on Computer Systems*, 10, 2, May 1992.

S. O'Malley, H. Orman, E. Menze III, and L. Peterson, "A Fast General Implementation of Mach IPC in a Network", *Proceedings of the USENIX Mach III Symposium*, April 1993.

Also see: **http://www.cs.arizona.edu/xkernel/www/index.html**

2.3 Ficus (File Systems)

R.G. Guy, J. Heidemann, W. Mak, T. Page, G. Popek, and D. Rothmeier, "Implementation of the Ficus Replicated File System", *USENIX Conference*, June 1990.

R.G. Guy, G.J. Popek and T.W. Page, "Consistency Algorithms for Optimistic Replication", *Proceedings of the First International Conference on Network Protocols*, IEEE Press, October 1993.

J.S. Heidemann and G.J. Popek, "File-System Development with Stackable Layers", *ACM Transactions on Computer Systems*, 12(1), 58-89.

P. Reiher, J. Heidemann, D. Ratner, and G. Popek, "Resolving File Conflicts in the Ficus File System", *Proceedings of the 1994 Summer USENIX Conference*.

Also see **http://www.isi.edu/~johnh/WORK/index.html**

2.4 Genesis (Database Management Systems)

D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C.Twichell, T.E. Wise, "GENESIS: An Extensible Database Management System", *IEEE Transactions on Software Engineering*, Vol. 14 #11 (November 1988), 1711-1730.

©dsbatory

D.S. Batory, “Concepts for a Database System Synthesizer”, *ACM Principles Of Database Systems Conference* 1988, 184-192. Also in Prieto-Diaz and Arango text (above).

D. Batory and J. Barnett, “DaTE: The Genesis DBMS Software Layout Editor”, in *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*, P. Loucopoulos and R. Zicari, editors, Wiley, 1992.

D. Batory and D. Vasavada, “Software Components for Object-Oriented Database Systems”, *International Journal of Software Engineering and Knowledge Engineering*, 2 #3, 1993, 165-192.

Also see `http://www.cs.utexas.edu/users/schwartz/index.html`

2.5 P2 and P3 (Data Structures)

M. Sirkin, D. Batory, and V. Singhal, “Software Components in a Data Structure Precompiler”, *Proc. ICSE 1993*.

D. Batory, V. Singhal, M. Sirkin, and J. Thomas, “Scalable Software Libraries”, *ACM SIGSOFT* 1993.

D. Batory, J. Thomas, and M. Sirkin, “Reengineering a Complex Application Using a Scalable Data Structure Compiler”, *ACM SIGSOFT* 1994.

D. Batory and J. Thomas, “P2: A Lightweight DBMS Generator”, to appear *Journal of Intelligent Information Systems*, 1997.

D. Batory, G. Chen, E. Robertson, and T. Wang, “Web-Advertised Generators and Design Wizards”, *Internation Conference on Software Reuse 199*8.

G. Jimenez-Perez and D. Batory, “Memory Simulators and Software Generators”, to appear in *1997 Symposium on Software Reuse*.

Also see `http://www.cs.utexas.edu/users/schwartz/index.html`

2.6 Jakarta Tool Set

D. Batory, B. Lofaso, and Y. Smaragdakis, “JTS: Tools for Implementing Domain-Specific Languages”, *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

Also see `http://www.cs.utexas.edu/users/schwartz/index.html`

©dsbatory