

An Example

Understanding GenVoca requires examining a domain where symmetry, scalability, encapsulation, and composition can be studied carefully.

In this lecture, we will examine:

- how an interesting domain can be modeled
- how individual components are defined
- how compositions are expressed
- importance of scalability
- fundamental generator implementation strategies
- lay groundwork for understanding OO implementations (see lecture on **Subjectivity**)

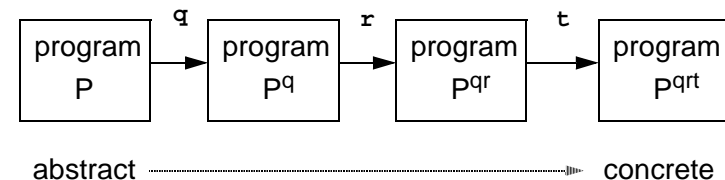
What is a Component?

A component is a *large scale refinement* that encapsulates consistent data & operation refinements between standardized OOVs

Suppose abstract program P imports interface S (written $P[x:S]$). Suppose further k implements S :

$$k = q[r[t]];$$

Program $P[k]$ is a concrete program P^{qrt} that is produced by applying the refinements q , r , and t in succession to P :

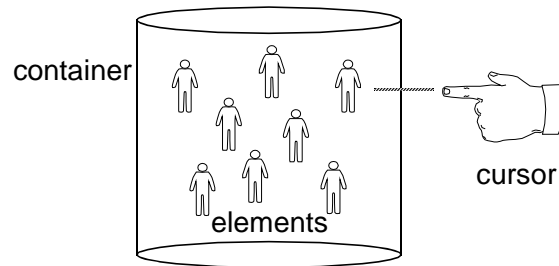


Note: *not functional evaluation!!!* apply outer first!!!

Example Domain: Container Data Structures

Few domains are familiar to all: container data structures

Binary trees, linked lists, etc. are implementations of a (well-known) container abstraction:



- *container* is a collection of *elements*
- all elements are instances of a single type
- *cursor* is run-time object used to reference, update elements of a container

Why Container Data Structures?

Compared to avionics, file systems, databases, network protocols.... isn't data structure domain trivial?

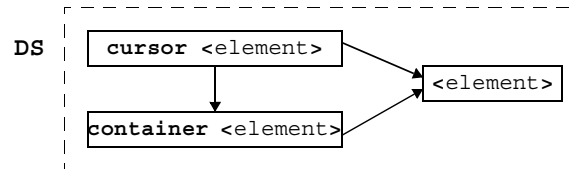
No!

- complexity of data structure *algorithms* simple
- key issues are *domain modeling*:
 - component definitions
 - component parameterizations
 - component compositions

... *the same for all domains* ...

Container data structures is an ideal domain to study

Container Data Structure OOVm



Container operations:

```

container ();           // constructor
boolean overflow ();    // overflowed?
...
  
```

Cursor operations:

```

cursor (container);    // constructor
void goto_first ();     // goto first elem
void goto_next ();      // goto next elem
boolean at_end ();      // no more elems?
element elem();         // pointer to elem

void insert (element);  // insert
void remove ();         // delete current
...
  
```

- methods (classes) parameterized by `element` type
- interface-only definition of realm

Upcoming Events

Examine DS (data structures) realm:

```

DS = {
    array,           // sequential allocation
    delflag[ DS ],  // logical deletion
    slist[ DS ],     // singly-linked list
    avail[ DS ],     // reuse of unused slots
    malloc,          // random allocation
    ...
}
  
```

Define as encapsulated, large-scale, consistent data and operation refinements

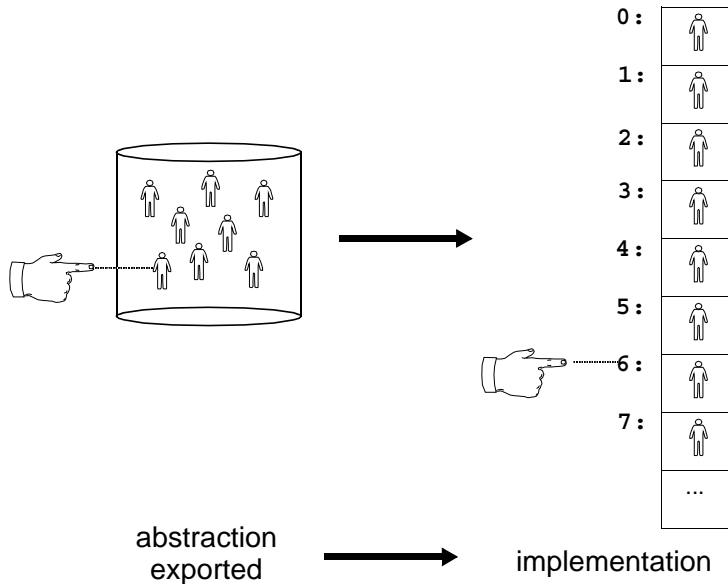
Note:

- we will rely on a simplistic Java-like language to convey ideas
- we will show later how these specifications can be implemented in fundamentally different ways

Array Component

Terminal component **array** : **DS**

- encapsulates storage of elements in array



- inserted elements appended to array
- deleted elements removed, shifting elements forward

Array Data Refinements

Data refinements add new data members to **Container**, **Cursor**, and **Element** classes:

Container:

```
element array[ Array_Size ];  
int next_free;
```

Cursor:

```
Container container;  
int index;           // index of current  
                     // element
```

Element:

```
// nothing added
```

- parameters: `element` and `Array_Size`

Hint: think of using inheritance (subclassing) as means of adding data members to classes

- refinement adds a subclass to existing **Container**, **Cursor**, and **Element** classes
- not always true, but close enough for now...

Array Cursor Operation Rewrites

Notation: operation \rightarrow algorithm

“operation is implemented by algorithm”

```
goto_first()  $\rightarrow$ 
    index = 0;
```

```
goto_next()  $\rightarrow$ 
    index++;
```

```
at_end()  $\rightarrow$ 
    index >= container.next_free
```

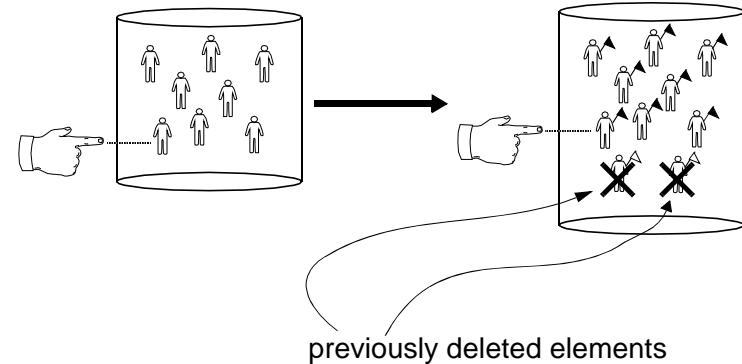
```
elem()  $\rightarrow$ 
    array[index]
```

```
insert( element )  $\rightarrow$ 
    index = container.next_free++;
    copy element into array[index];
```

Delflag Component

Symmetric component **delflag**[DS] : DS

- encapsulates deletion-tagged elements;
don't care how refined elements stored



- input elements refined into output elements (**element** + **deleted-flag**) that are stored
- insert** clears delete flag and **remove** sets delete flag
- retrievals return only undeleted elements

Delflag Data Refinements

The obvious...

Container:

```
LowerContainer lowerContainer;
```

Cursor:

```
LowerCursor lowerCursor;
```

Element:

```
boolean deleted;    // delete flag
```

Delflag Cursor Operation Rewrites

```
goto_first_nondeleted() →
    while ( !lowerCursor.at_end() &&
            lower.elem().deleted )
        lowerCursor.goto_next();
```

```
goto_first() →
    lowerCursor.goto_first();
    goto_first_nondeleted();
```

```
goto_next() →
    lowerCursor.goto_next();
    goto_first_nondeleted();
```

```
at_end() →
    lowerCursor.at_end()
```

Delflag Operation Rewrites (Cont)

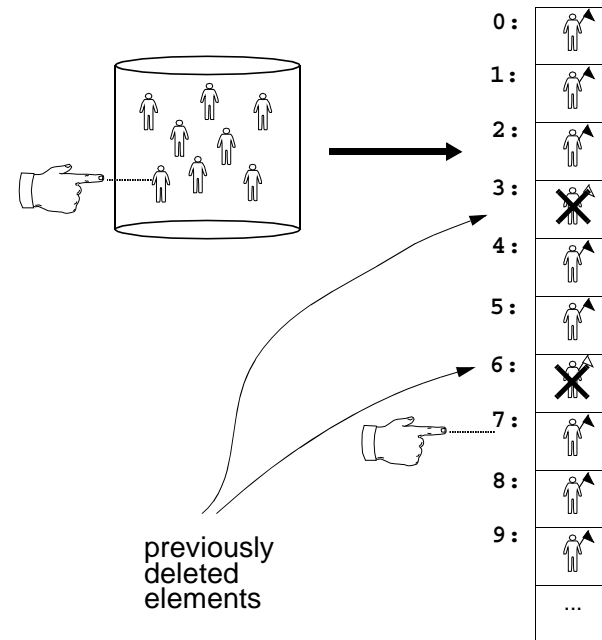
```
insert( element ) →
    lowerCursor.insert( element );
    lowerCursor.elem().deleted = false;
```

```
remove() →
    elem().deleted = true;
```

...

A Composition : delflag[array]

Tagged array implementation:



- elements tagged deleted; slots never reclaimed
- “macro” expansion of pictures shows implementation

Composition Data Refinement

“Macro” expand data type definitions to yield:

Container:

```
element array[ Array_Size ];
int next_free;    // array
```

Cursor:

```
int index;        // array
```

Element:

```
...                // original fields
boolean deleted;   // delete flag
```

- parameters: element and Array_Size

Composition Operation Rewrites

Look at one operation — *rest are the same*:

```
insert( element ) →
    index = container.next_free++;    //1
    copy element into array[index];  //1
    array[index].deleted = false;     //2
```

Note: this rewrite is a “macro” expansion of the **delflag** **insert** with the **array** **insert** and **elem** operations:

```
insert( element ) →delflag
    lowerCursor.insert( element );    //1
    lowerCursor.elem().deleted = false; //2
```

Composite rewrite is what you would expect to write by hand

- but it has been manufactured *automatically* by composing the primitive **delflag** and **array** operations
- statements of different layers are interwoven
- components tell us how to interweave code

Big Picture

Generators “expand” type equation:

- outermost component first, innermost last
- simplify & optimize to produce code

no simplification or optimization in this example though...

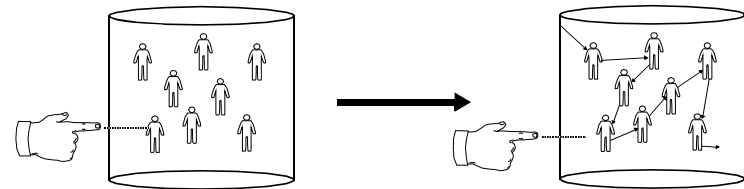
Composition yields another implementation of `DS`

Look briefly at 3 more `DS` components...

Slist Component

Symmetric component `slist[DS] : DS`

- links elements of container onto singly-linked list;
don't care how lower container is implemented

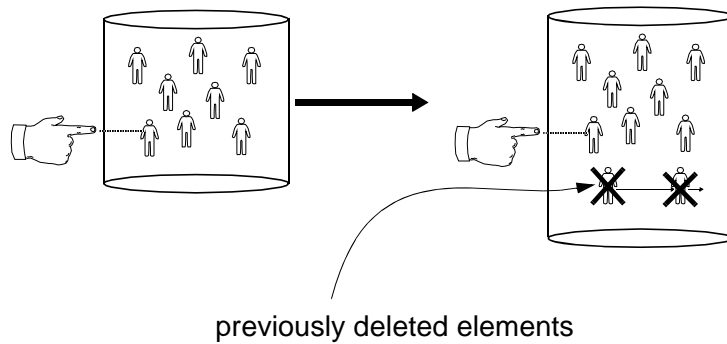


- refined element types have (next-on-list) pointer
- container augmented with pointer to first refined element
- inserted elements placed at head of list
- removed elements are unlinked and then deleted
- lower-level stores instances of refined element types

Avail Component

Symmetric component `avail[DS] : DS`

- layer that permits reuse of unused element slots

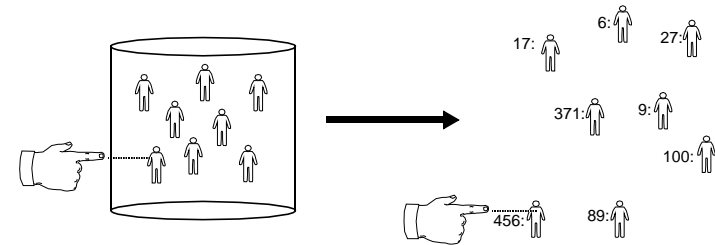


- when element is removed, slot placed on “avail” list
- when element is inserted, slot removed from “avail” list
- like `delflag`, uses lower component to advance to first undeleted element

Malloc Component

Terminal component `malloc : DS`

- encapsulates heap storage of elements

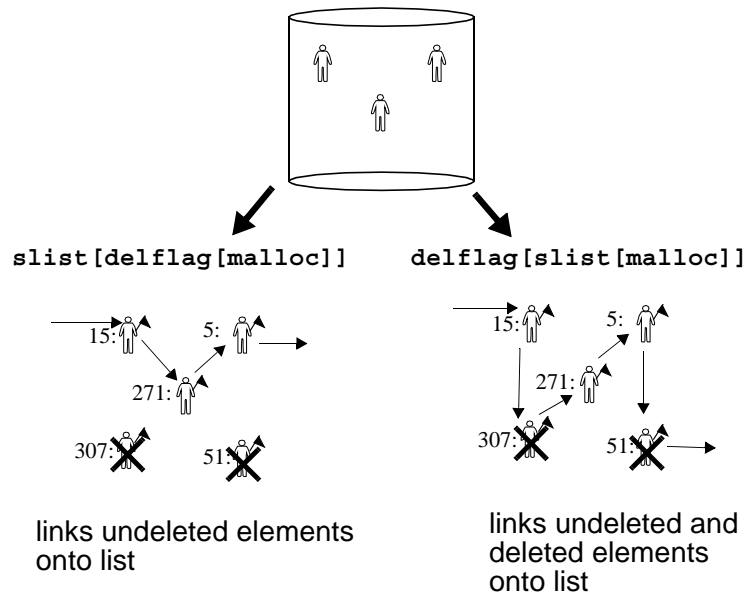


- heap storage allocated by calls to Unix `malloc()`
- elements aren't linked together; cursor operations `goto_first`, `goto_next`, `at_end` undefined
- must be used with other layers that implement `goto_first`, `goto_next`, `at_end` operations

Compositions and Type Equations

Given **ds** realm there are many compositions:

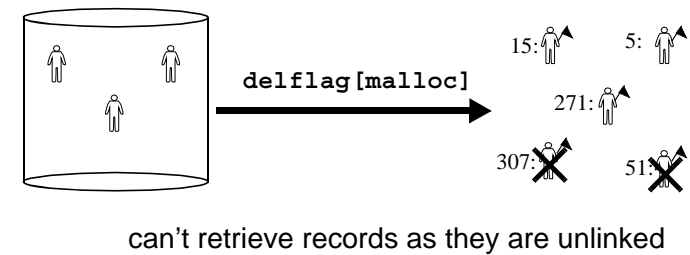
`delflag[array]`, `slist[malloc]`, ...



Compositions and Design Rule Checking

Generator produces code by “macro” expanding type equation, simplifying and optimizing

- refinement-like interpretation of type equation:
do this, then do this, then do this ...
- there are “incorrect” or “undesirable” combinations of components — ex:



- another example: `slist[delflag[delflag[array]]]`
(only one `delflag` needed)

Design rule checking — rules that define component combinations that are legal, illegal.

More Components

What other `ds` components are there?

```
DS = { // common container data structures

    dlist[ DS ], // doubly-linked list
    bintree[ DS ], // binary tree
    odlist[ DS ], // ordered dlist
    quadtree[ DS ], // quad tree
    ...

    // common data structure features

    delflag[ DS ], // delete flag
    avail[ DS ], // avail list
    compress[ DS ], // element compression
    ...

    // common storage schemes

    array, // transient sequential
    malloc, // transient heap
    parray, // persistent sequential
    pmalloc, // persistent heap
    remote // remote storage
    ... }
```

`ds` has a rather large (~100) set of components.

Scalability

How many distinct container data structures can be formed by composing `ds` components?

- Ans: infinite

of implementations increases *exponentially* in the length of a type equation; components can be nested arbitrarily deep:

```
array
odlist[malloc]
odlist[odlist[malloc]]
odlist[odlist[odlist[pmalloc]]]
bintree[odlist[slist[delflag[parray]]]]
...
```

Scalability — a small number of components can be composed to yield large families of systems

- `ds` is *scalable*
- each new `ds` component leads to vast numbers of new type equations/data structures

Why is Scalability Important?

*Don't build libraries
where components implement
combinations of features*

Ex: Booch Components: 400+ data structure templates

- 18 varieties of dequeues = $3 \times 3 \times 2$

concurrency (sequential, guarded, synchronized)

memory allocation (bounded, unbounded, dynamic)

ordering (ordered or unordered)

- size of library may double each time new feature is added (e.g., persistence)
- no conventional library could encompass enormous spectrum of data structures that are encountered

*let components encapsulate individual, largely
orthogonal features; model combinatorics
explicitly using type equations*

Result: Singhal Components

V. Singhal re-engineered the Booch Components (Version 1.47) using P++

- P++ enhanced C++ templates with realms, components

	Booch C++ Components	Singhal P++ Components
Number of abstract classes/realms	7	11
Number of concrete classes/components	82	22
Lines of Code	11,067	2,760
Cardinality of Data Structure Domain	169	208

Productivity results:

- approximately 1/4th size of Booch Components
- greater numbers of data structure implementations

Performance results:

- P++ data structures more efficient (no virtual dispatching)
- easier to add new P++ component to customize application, performance

How to Implement Generators?

Determined by two factors:

- when components are composed, and
- how components are implemented.

GenVoca components are parameterized:

```
bintree[ x:DS ]
```

Question: when are parameters instantiated?

model doesn't say — two known possibilities:

- *dynamic* — at application run-time
- *static* — at application compile-time
(or generator run-time)

How to Implement Generators

Question: How to implement components?

model doesn't say — three known possibilities

Compositional

- components are templates or packages encapsulating suites of classes/subclasses
- virtually no optimizations possible

Generative

- code synthesizers that compose locally-optimal code fragments
- simple, common optimizations possible by breaking component encapsulations

Transformational

- program transformation systems
- layers expressed by sets of rewrite rules
- very complex optimizations possible

Generator Implementations (Cont)

Survey of known generators:

Implementation	Composition Time	
	Static	Dynamic
Compositional	Genesis Ficus JTS ADAGE	Avoca
	P3	ASP
Generative	?	?
Transformational	?	?

Note: LavaLamp (radio-software domain) needs both:

- dynamic-compositional components and
- static-generative components

Choice of implementation is determined by:

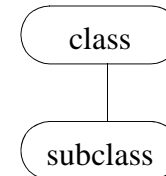
- performance
- application domain requirements

Compositional Generators

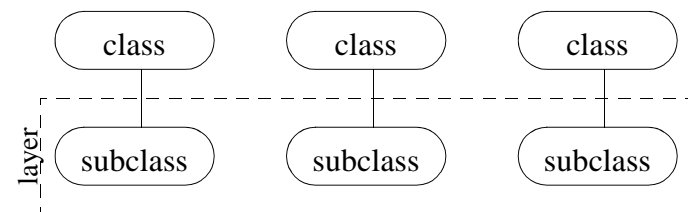
How are GenVoca refinements implemented as OO classes?

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

expressed
as a subclass



GenVoca or *large-scale* refinement adds new data members and methods simultaneously to *several* classes



Compositional Generators (Cont)

Consider cursor class of `array` component

- data and method refinements

```
Cursor:  Container container;
        int index;

goto_first() → index = 0;
goto_next() → index++;
...
```

- straightforward OO implementation (with class renaming):

```
class ArrayCursor {
    ArrayContainer container;
    int index;

    void goto_first() { index = 0; }
    void goto_next()  { index++; }
    ...
}
```

Compositional Generators (Cont)

Consider cursor class of `delflag` component:

```
goto_first() → lowerCursor.goto_first();
              goto_first_nondeleted();

goto_next() →  lowerCursor.goto_next();
              goto_first_nondeleted();
```

- straightforward OO implementation as a *mixin* (i.e., a class whose superclass is specified by parameter)

```
template < class superCursorClass >
class DelflagCursor extends superCursorClass {

    void goto_first() {
        super.goto_first();
        goto_first_nondeleted();
    }

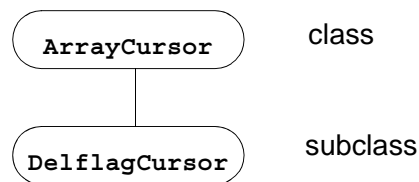
    void goto_next() {
        super.goto_next();
        goto_first_nondeleted();
    }

    void goto_first_nondeleted() {...}
    ...
}
```


Compositional Generators (Cont)

A cursor for a `tagged_array` is composition of the `ArrayCursor` and `DelflagCursor` classes:

```
typedef DelflagCursor< ArrayCursor > TA_Cursor;
```



Mixin/compositional implementations are so important that they are a focus of the next lecture (**Subjectivity**)

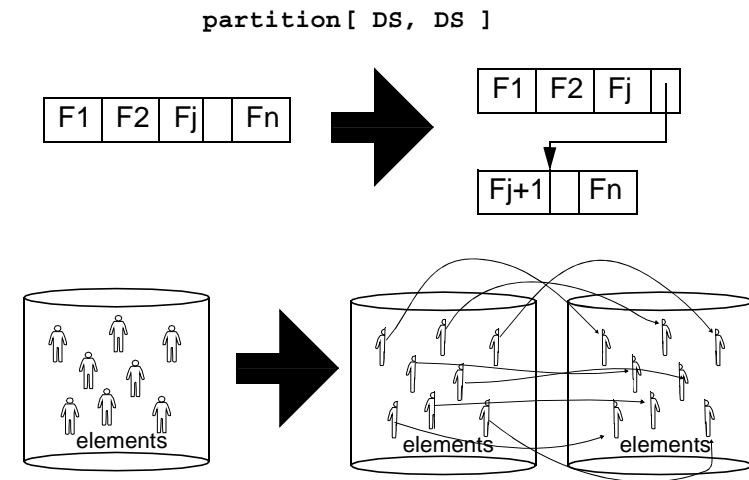
Compositional implementations (e.g., templates, packages) are appropriate:

- when refinements are simple
- and optimizations are not needed

Compositional Generators (Cont)

Not all refinements can be implemented compositionally

- ex: refinement that splits each element into two interlinked halves



- requires classes to be first-class objects
- actually, not all combinations of refinements can be implemented compositionally...

Generative Implementations

Are code synthesizers or *metaprograms*
(i.e., programs that generate other programs)

- normally requires programming language support
- *example JTS (Jakarta Tool Suite)* extends Java with *ASTs (Abstract Syntax Trees)*

```
y = exp{ 7*x }exp;
y.print();          // "7*x"

z = stm{ if ($exp(y) > 4 ) foo();
        else bar();
      }stm;

z.print();          // "if (7*x > 4) foo();
                    //  else bar();"

```

Distinguishes code generator is to execute from code that is to be generated (like LISP quote/comma)

Generative Implementations (Cont)

Generative implementations compose code fragments

- recall `delflag remove()` and `array elem()` rewrites

```
remove() →delflag elem().deleted = true;
elem() →array array[index];

```

- expressed in JTS as:

```
elem = exp{ array[index] }exp;
elem.print(); // "array[index]"

remv = stm{ $exp( elem ).deleted = true; }stm;
remv.print(); // "array[index].deleted = true;"

```

Basic idea is to implement layers as a set of methods that create and glue together code fragments

- can perform fairly sophisticated optimizations, such as **partition**

Generative Implementations (Cont)

Can't perform sophisticated optimizations such as code motion:

```
int cons, sum = 0;
for (i=1; i<100; i++) {
    foo(i);
    cons = 8;
    sum = sum + i;
}
```

Which can be optimized/rewritten as:

```
int cons = 8;
int sum = 5050;
for (i=1; i<100; i++) {
    foo(i);
}
```

Requires some heavy compiler machinery:

- constant folding
- data flow analysis
- arithmetic expression simplification
- ...

Transformational Implementations

Can accomplish such “sophisticated” optimizations using *program transformation systems (PTS)*

- PTS express “small-scale” refinements as pattern-based rewrites — ex: rewrite to move statements out of loops:

```
int ?var;                // left-hand pattern

for (i=?lower; i<=?upper; i++) {
    ?statements;
    var = ?const;
}
```

→

```
int ?var = ?const;       // right-hand pattern

for (i=?lower; i<=?upper; i++) {
    ?statements;
}
```

- can express code motion
- code rewrites that are much more complex than just composing code fragments together

Transformational Implementations

GenVoca layer is expressed as a *rule set*:

- consistent set of pattern-based rewrites
- ex: rewrites for array layer have rather simple left-hand patterns (method calls):

```
goto_first() → index = 0;
```

```
goto_next() → index++;
```

Domain-specific optimizations expressed as rewrites, too:

```
goto_first(); goto_first();
```

```
→
```

```
goto_first();
```

See Weigert et al. paper for illustration of approach

Why Important?

GenVoca defines a *single* way in which to model/
conceptualize a domain:

- in terms of large-scale refinements and their compositions
- *no a priori commitment to particular implementation strategy when creating a domain model*

- model can have radically different implementations

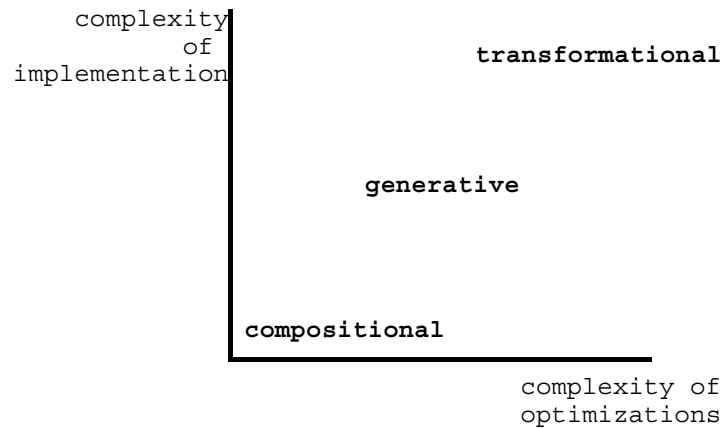
compositional or generative or transformational
dynamic or static

each with different capabilities and limitations

i.e., compositional is not as powerful as generative

generative is not as powerful as transformational

How to Choose Implementation?



Generator builder or domain-analyst determines the most appropriate way to implement the generator (depending on the degree of optimizations that need to be performed)

- ex: don't use transformational when templates or packages (compositional) implementation suffices

Conceptual economy of GenVoca is a big win ...

Conclusions — Lessons Learned

- *GenVoca components implement fundamental refinements of domain abstractions*

powerful, high-level model of families of systems as compositions of refinements

- *conventional means of building libraries are inherently unscalable*

make feature combinatorics explicit
(using symmetric components)

can define an astronomical number of systems from a small number of GenVoca components

- *single way to conceptualize domain in terms of implementation-independent refinements*

compositional, generative, or transformational implementations with dynamic or static compositions

Further Readings

- I. Baxter, "Design Maintenance Systems", CACM April 1992, 73-89.
- D. Batory, "Modeling the Storage Architectures of Commercial Database Systems", *ACM Transactions on Database Systems*, December 1985.
- 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.
- L. Blaine and A. Goldberg, "DTRE - A Semi-Automatic Transformation System", in *Constructing Programs from Specifications*, Elsevier Science Publishers, 1991.
- G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming", *ECOOP 97*, 220-242.
- 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.
- 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.
- G. Jimenez-Perez and D. Batory, "Memory Simulators and Software Generators", *1997 Symposium on Software Reuse*, 136-145.
- V. Singhal, "A Programming Language for Writing Domain-Specific Software System Generators", Ph.D. thesis, University of Texas at Austin, 1996.
- D.R. Smith, "KIDS: A Semiautomatic Program Development System", *IEEE Transactions on Software Engineering*, Sept. 1990, 1024-1043.
- T.J. Weigert, J.M. Boyle, T.J. Harmer, "Knowledge-Based Derivation of Programs from Specifications", *Artificial Intelligence in Automation*, World Scientific Press, 1996.