International Journal of Software Engineering and Knowledge Engineering 2(3):375-402, September 1992.

# Implementing a Domain Model for Data Structures[1,2]

**Don Batory, Vivek Singhal, and Marty Sirkin**
**Department of Computer Sciences**
**The University of Texas**
**Austin, Texas 78712**

**Abstract:** We present a model of the data structure domain that is expressed in terms of the GenVoca domain modeling concepts [Bat91]. We show how familiar data structures can be encapsulated as realms of plug-compatible, symmetric, and reusable components, and we show how complex data structures can be formed from their composition. The target application of our research is a precompiler for specifying and generating customized data structures.

**Keywords:** software building-blocks, domain modeling, software reuse, data structures.

## 1.0 Introduction

A fundamental goal of software engineering is to understand how software components fit together to form complex systems. Domain modeling is a means to achieve this goal; it is the study of a domain of similar software systems to identify the primitive and *reusable* components of that domain and to show how compositions of components not only explain existing systems but also predict families of yet unbuilt systems that have interesting and novel properties. In essence, domain models can be blue-prints for the as-yet-to-be-achieved software building-block technologies.

Domain modeling is presently an immature discipline [Pri91]. Besides the general skepticism that characteristically accompanies new areas of research, domain modelers face three difficult barriers to the development and popularization of their ideas:

- Domain models must be expressed in terms of constructs and software organization principles that are domain-independent. Domain-specific constructs are, by definition, not applicable to other domains. Models based exclusively on domain-specific ideas are both difficult to understand and contribute little to helping other researchers understand how other domains can be modeled.
- In order to understand a domain model, one must be intimately familiar with the domain itself. It is often hard for non-experts (and even experts) to appreciate the difficulty of a domain modeling effort and its contributions.
- Domain models are nontrivial. One cannot accept a domain model on face value; it is essential that there be an accompanying implementation for model validation. Unfortunately, because of the time and expense involved, few models are validated. Without validation, however, the significance of a model is questionable.

In this paper, we attempt to cross all three barriers (or alternatively, we show why each is hard to cross!). First, we review the GenVoca domain modeling concepts, which have been validated on the com-

---

1. This research was supported in part by grants from Texas Instruments and Digital Equipment Corporation.

2. This paper also appeared in *International Journal of Software Engineering and Knowledge Engineering*, 2(3):375-402, Septermber 1992.

plex and disparate domains of database and network software [Bat91][3] Next, we apply these concepts to develop a model of data structures, a domain that should be familiar to all readers. Finally, we explain how we are validating our domain model by explaining the mechanics of a precompiler for data structures. As we are covering a broad sweep of issues, we present in the introduction of every section the "big picture" of what we are about to do.

We believe our research makes two contributions. First, our domain model outlines the beginnings of a technology for synthesizing customized object base software from prewritten building-blocks. Persistency is but one of many options (i.e., building-blocks) that can be selected. Our paper will stress the non-persistent (i.e., data structure) aspects of this technology. Second, we see our work as an example of the types of activities and problems that are commonly encountered in a domain modeling effort. Other researchers may benefit in their domain modeling efforts by following our approach.

## 2.0 The GenVoca Domain-Modeling Concepts

As mentioned earlier, it is important for domain models to be expressed in terms of domain-independent concepts. Object-oriented software design models, as examples, offer a collection of concepts (e.g., entities, attributes, relationships, and inheritance) that software engineers can use to express the designs of their systems. These concepts are domain-independent as they have been shown to be applicable to a wide-variety of problems in very different domains [Rum91, Boo87, Teo86]. The ER model is a common graphical representation of object-oriented concepts [Che76, Kor91]. We will use the ER model as the vehicle for discussing OO concepts in this paper.

The concepts of the ER model are necessary, but not sufficient, to explain important kinds of software components and to express hierarchical software systems as compositions of these components. To do so requires additional concepts that transcend traditional ER/object-oriented models. We set the stage for introducing these concepts in Section 2.1 where we explain the relationship of ER/OO models with hierarchical system designs. We then introduce the concepts of components, realms, and type expressions for modeling hierarchical systems in Section 2.2. We conclude the section by explaining how our notion of parameterized components is different than traditional notions of parameterized types.

### 2.1 Hierarchical Software Systems and Domain Modeling

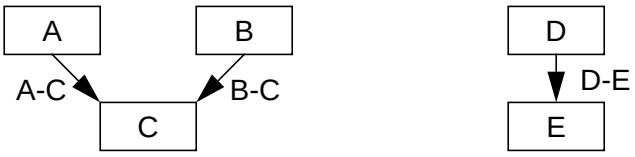
Hierarchical software systems are designed in terms of *levels*, where the interface of each level is a virtual machine. All operations on level `i+1` are expressed in terms of operations of level `i`. A layer is the *mapping* of operations between levels. The idea is to localize or encapsulate specific complexities of a system within layers, thereby simplifying overall system design [Dij68, Hab76].

Object-oriented programming languages and design methodologies have shown that interface specifications should encompass both objects and operations, rather than operations alone. An *object model* or *object-oriented virtual machine (OOVM)* is, in general, a set of classes and their interrelationships. ER diagrams depict object models using boxes to represent classes and arrows to represent relationships between classes. Figure 1 shows two object models: Model `R` exhibits classes `A`, `B`, and `C` with relationships `A-C` and `B-C`; Model `S` exhibits classes `D` and `E` with relationship `D-E`.

A hierarchical system design, from an object-oriented perspective, is a set of object-oriented virtual machines, one for each level of a system. Figure 2 shows the object model interfaces for levels `i` and `i+1` in a hierarchical system; the interface for level `i+1` is object model `R` and object model `S` is the interface for level `i`.

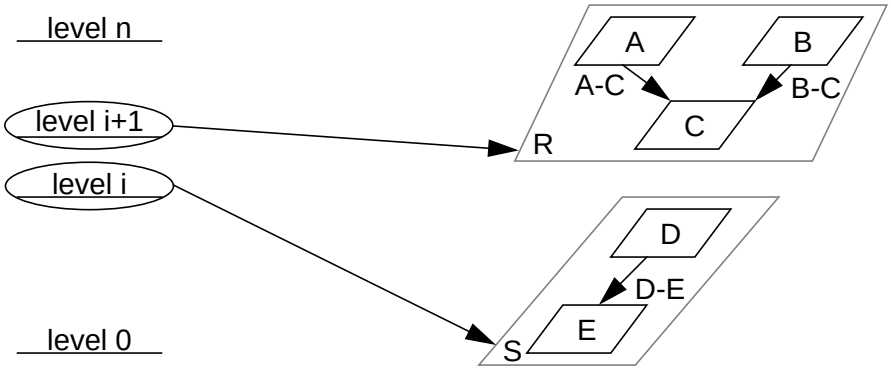
---

3. GenVoca is the model that resulted from the merging of the design philosophies of two different domain-specific software system generators, Genesis and Avoca.




(a) Model R (b) Model S

**Figure 1: ER Diagrams and Object Models**




(a) levels of a hierarchical system (b) expanded views of the OOVMs of two levels

**Figure 2: Levels and their OOVMs in a Hierarchical System**

As a general rule, no two independently-designed systems of a domain exhibit exactly the same sets of layers and virtual machines. As a consequence, layers in one system cannot be swapped with those of another; their interfaces are incompatible. Domain modeling enters the picture as an in-depth study of a domain to determine a *common* decomposition of these systems, where object-oriented virtual machines and their implementing layers are *standardized*. This leads to the following concepts of hierarchical software system construction.

## 2.2 The GenVoca Modeling Concepts

The fundamental unit of large scale software construction is the *component* or *layer*. A component is a software module with a well-defined external interface. The interface of a component is an object-oriented virtual machine that usually consists of several interrelated classes. A component `q`, for example, might present object model `S` (in Figure 1b) as its interface.

All components that export the same interface define a *realm*. Using set notation for specifying realm membership, suppose the realms for Models `R` and `S` are:

```
Realm_S = { q, r, s, t }
Realm_R = { f[x:Realm_S], g[x:Realm_S], h[x:Realm_S] }
```

That is, `Realm_S` consists of four components (`q-t`) and `Realm_R` consists of three (`f-h`). It is important to distinguish realms from their OOVM interfaces. OOVM `R` consists of three *classes*: `A`, `B`, and `C`. `Realm_R` consists of three *components*: `f`, `g`, and `h`. Each component in `Realm_R` implements the `R` OOVM; that is, each component of `Realm_R` provides a (typically unique) implementation for each of the classes `A`, `B`, and `C`.

Because every component within a realm exports exactly the same interface, all components are *plug-compatible* and *interchangeable*. Moreover, components may have parameters. Consider component `f[x:Realm_S]`. `f[ ]` exports interface R (because `f[ ]` belongs to `Realm_R`) and imports interface S (because `f[ ]` has parameter `x:Realm_S`). Moreover, `f[ ]` translates operations and objects of OOVM `R` to objects and operations of OOVM `S` *without depending on how OOVM* `S` *is implemented*. How OOVM `S` is implemented is specified by parameter `x`.

Two useful results follow. First, a *software system* of potentially enormous complexity is represented as a type expression (i.e., a composition of components). The following three systems present OOVM `R` as their interface:

```
system1 = f[r]
system2 = f[t]
system3 = g[t]
```

The above syntax was borrowed from the parameterized type notation in [Gog84, Gog86]. While the syntax is similar, we will show in Section 2.3 that the semantics of our components are rather different.

Second, component *reuse* corresponds to common subexpressions. Whenever different systems (different expressions) reference a common subsystem (subexpression), then that subsystem (and its components) is being reused. Thus, `system1` and `system2` reuse component `f[]`, and `system2` and `system3` reuse component `t`.

A fundamental concept of GenVoca is symmetric components, i.e., components that can be composed in arbitrary orders. More precisely, a *symmetric* component of realm `W` has a parameter of type `W`. Components `m` and `n` of realm `W` below are symmetric, while components `p` and `q` are not:

```
W = { m[x:W], n[x:W], p, q[y:R], ...}
```

Since m[ ] and n[ ] are symmetric, compositions `m[n[...]]` and `n[m[...]]` are possible. As a general rule, different composition orders have different meanings. While symmetric components seem strange, readers are probably familiar with a classical example of symmetric components: UNIX file filters. File filters present the same byte-stream interface for both their input and output; composing UNIX file filters together in a pipe is an example of composing symmetric components. As in the general case, the order in which one composes file filters (symmetric components) makes a difference in both semantics and performance.

The concepts of type expressions and realms of plug-compatible and parameterized components are not part of the ER/OO models. These concepts are domain-independent extensions to these models and have been validated in two rather different domains: databases and network software [Bat85-91, Oma90, Hut91]. The domain models for both databases and networks had their own sets of realms of interchangeable components. As this paper develops, we will show that the domain of data structures has a similar organization.

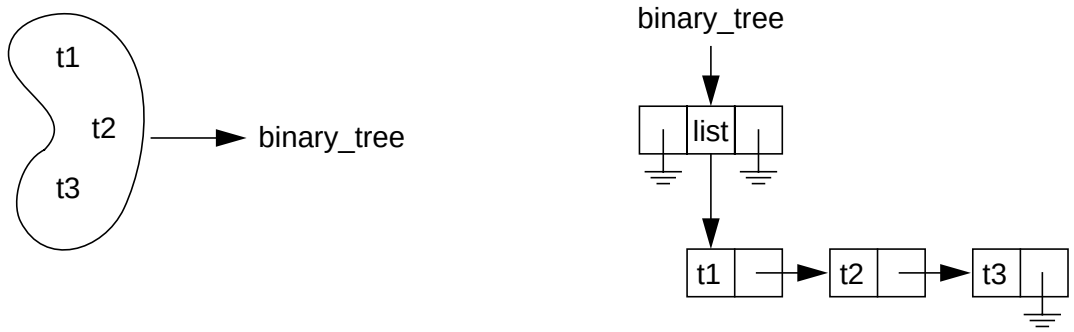
## 2.3 On the Relationship of Components to Parameterized Types

Our type expression notation was borrowed from research on parameterized types [Gog84]. However, the semantics of the components that we examine in this paper are very different than traditional constructs (e.g., templates in C++ [Str91], ADA generics [Ghe82], OBJ sorts [Gog86]).

Common to all parameterized types (both ours and traditional) is the concept of a *container*, i.e., a set of generic objects. Stacks, lists, binary trees, queues, etc. are specialized containers whose algorithms are defined independently of the types of objects that are to be stored.

**Traditional**. Traditional parameterized types take the container idea a step further. An instance of a *traditional parameterized type* (TPT) (e.g., stack, queue) is itself an object (i.e., a container). The essential idea is that a TPT maps a *container* of objects to a *single* object.

Consider the `LIST[*]` and `BINARY_TREE[*]` parameterized types. An instance of `LIST[T]` is a single object (a list) that represents a list of objects of type `T`. An instance of `BINARY_TREE[LIST[T]]` is a single object (a binary tree), where each tree node references a list object (which itself is a container of objects of type `T`). Figure 3 shows how a container of three objects of type `T`, namely { t1, t2, t3 }, is mapped by `BINARY_TREE[LIST[T]]` to a binary tree with a single node. (We depict containers whose implementation is partially specified as a set of objects/records within an enclosed boundary - i.e., as in Figure 3a. When a container implementation is fully specified, the container's objects are drawn without an enclosed boundary, as in Figure 3b).




(a) `BINARY_TREE[LIST[T]]` Mapping (b) Resulting Data Structure

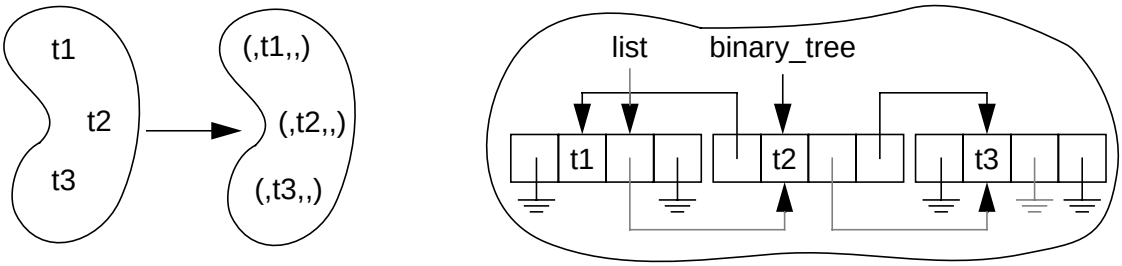
**Figure 3: Traditional Parameterized Type Mappings**

**Nontraditional**. We propose different semantics for parameterization; a *nontraditional parameterized type* (NPT) maps a *container* of abstract objects to one or more *containers* of concrete objects. The container abstraction and interface remain unchanged by the mapping.

Consider the `LIST'[*]` and `BINARY_TREE'[*]` NPTs. `LIST'[T]` maps the objects $\{t_1 ... t_n\}$ of container `T` to the set $\{(t_1, next_1) ... (t_n, next_n)\}$, where each $next_i$ is a pointer to the object $(t_{i+1}, next_{i+1})$. `BINARY_TREE'[T]` maps the objects $\{t_1 ... t_n\}$ of container `T` to the set $\{(left_1 , t_1, right_1) ... (left_n, t_n, right_n)\}$, where $left_i$ and $right_i$ are pointers to the left and right tree objects of $t_i$.

`BINARY_TREE'[LIST'[T]]` maps a container of `T` objects to a container of 4-tuples of the form $(left_i, t_i, next_i, right_i)$ (see Figure 4a). Figure 4b shows how the container of Figure 3a is mapped by `BINARY_TREE'[LIST'[T]]` to a container of objects that are independently interconnected by both a binary tree and list. Note that the resulting container can be implemented in a variety of ways (e.g., the 4-tuples could be stored by the `LIST[*]` or `BINARY_TREE[*]` TPTs). It is worth noting that the situation

depicted in Figure 4b arises frequently in practice, i.e., programmers often maintain two distinct orders of a single set of objects using different data stuctures.




(a) `BINARY_TREE'[LIST'[T]]` Mapping (b) Resulting Container of Interlinked Objects

**Figure 4: Nontraditional Parameterized Type Mappings**

The general paradigm that has surfaced from our research is that NPTs transform abstract *schemas* (i.e. sets of containers and their relationships) to concrete (albeit simpler) schemas. Traditional parameterized types (TPTs) do not have this capability and appear to have a rather restricted subset of the capabilities offered by NPTs [Bat85]. The NPT concept is the basis of our domain model for data structures.

# 3.0 The Domain of Data Structures

We explain in this section the steps that we followed in developing a domain model for data structures using the GenVoca concepts. We begin by analyzing a subdomain of data structures, namely data structures that implement containers. We define more precisely the container abstraction in Section 3.1 along with an OOVM that we are proposing to be the "standard" container interface. We then review a realm of plug-compatible implementations of this OOVM in Section 3.2, and show how to apply these implementations to real-world problems in Section 3.3. In Section 3.4 we outline extensions to our domain model that account for data structures that don't implement containers.

## 3.1 The Container Abstraction

As mentioned earlier, a container is a collection (set) of objects. Our container abstraction has been strongly influenced by work on relational and object-oriented databases [ACM91, Kor91, Coh85]. In these systems, users declare the relations/classes of their applications and write programs that update and query these relations/classes without ever knowing how tuples/instances are stored and retrieved. We take this idea one step further. We assume that there are many types of containers; all container types present exactly the same interface, but different types offer different ways of storing objects. Thus there could be a `list_container` type (which stores objects in a container on a list) and a `binary_tree_container` type (which stores objects in a binary tree) among others. In principle, users can swap container types without impacting the correctness of their programs. Of course, performance may be affected if container implementations are altered.

For purposes of exposition, we will present data structure algorithms in a C++ notation. It should be understood, however, that we are using C++ only for convenience. None of the GenVoca concepts rely on any C++ syntactic construct; any other high-level programming language could have been used.

Figure 5 outlines a generic class `T` in C++ syntax [Str91]. `T` has zero or more private class variables, private instance variables, and private functions.[4] `T` also has zero or more public instance variables $\{A_1 ... A_n\}$ called *attributes*, and zero or more public functions, including a constructor function (`T`) and a destructor function (`~T`).

```
class T
{
 private:
  // private class variables, functions and instance variables
  ...
 public:
  // public instance variables and functions
  T1 A1;          // T1 is the type for attribute A1
  T2 A2;
  ...
  Tn An;
  T ();           // constructor
  ~T ();          // destructor
  ...             // other public functions
};
```

**Figure 5: A General Definition of a C++ Class**

Two different containers `C1` and `C2` of `T` objects would be declared as instances of a C++ template [Str91]:

```
list_container<T> C1, C2;
```

That is, the objects of both `C1` and `C2` are stored in `list_container` data structures, which are parameterized by the type `T` of objects that they store. Containers are first-class objects and may themselves be stored in containers. Recalling the example of Figure 4, the container `Directory` is a binary tree container of list containers:

```
binary_tree_container<list_container<T>> Directory;
```

A run-time mechanism called an *iterator* or *cursor* [Ghe82, Kor91, ACM91] is used to enumerate subsets of objects in a container.. Cursors for the above containers are declared by:

```
cursor<list_container<T>> c1(C1), c2(C2);

cursor<binary_tree_container<list_container<T>>> c3(Directory);
```

That is, cursors `c1` and `c2` are of the same type; both traverse containers of type `list_container<T>`. Furthermore, cursor `c1` is bound to container `C1` and cursor `c2` is bound to container `C2`. Cursor `c3` has a similar interpretation. We'll see in the next section that programming with cursors is straightforward.

### 3.1.1 Programming with Cursors

In general, operations on objects inside a container must be performed through cursors. In order to update an attribute of an object, for example, one must first position a cursor on the object before the update operation is issued. Figure 6 summarizes operations that can be performed on a cursor `c`. This list is not intended to be comprehensive; it represents our first attempt to define a standardized interface for containers.

---

4. Recall that a class variable is effectively a shared attribute; there is only one copy of each class variable and all instances of the class share that copy [Str86].

| | |
|---|---|
| `cursor<K> c(C)` | constructor #1 for a cursor `c` on objects in container `C` of type `K`. |
| `cursor<K> c(C, predicate)` | constructor #2 for a cursor `c` on objects in container `C` of type `K`. `c` ranges only over objects that satisfy `predicate`. |
| `~cursor<K>()` | destroy a cursor. |
| `c.reset()` | `c` is positioned prior to the first object that satisfies `predicate`. |
| `c++` | advance cursor `c` to the next object of container `C` that satisfies the cursor predicate. `c.status = EOR` (end of retrieval) or `OK`, depending if a next object was found. |
| `c.status` | returns the status `OK` or `EOR` of the last advance (`++`) operation on `c`. |
| `c.ref(Ai)` | attribute `Ai` of the object referenced by `c`. |
| `c.Ai` | syntactic sugar for `c.ref(Ai)`. |
| `c.newobj()` | create a new object in the container referenced by cursor `c`; `c` is positioned over the new object. |
| `c.del()` | delete the object referenced by `c`. |
| `c.oid` | the identifier of the object referenced by `c`. |
| `c.pos(oid)` | reposition cursor `c` over an object whose identifier is `oid`. |
| `c.upd(Ai, value)` | update attribute `Ai` of the object referenced by `c` to be `value`. |
| `c.Ai = value` | syntactic sugar for `c.upd(Ai, value)`. |

**Figure 6: Operations on Cursors**

To make our discussions concrete, consider the `emp` class and a container `EMP`:

```
class emp
{
 public:
  char name[12];
  int  age;
  char city[15];
  char department[25];

  // emp operations ...

};

typedef list_container<emp> Emp_Container;

Emp_Container EMP;
```

The following program fragments illustrate how cursors are used.

**Example 1.** Insert an object for '`Batory`' into container `EMP`.

```
cursor<Emp_Container> e(EMP);     // constructor #1

e.newobj();                       // create new EMP object
e.name = "Batory";                // fill in attribute values
e.age  = 38;
e.city = "Austin";
e.dept = "Software";
```

**Example 2.** List the names of all employees in the Software department.

```
cursor<Emp_Container> e(EMP, dept=="Software");       // constructor #2

for (e.reset(), e++; e.status != EOR; e++)
{
   printf("%s\n", e.name);
}
```

Note that the predicate constructor binds `e` to range over selected `EMP` members. The `for` loop iterates over each qualifying member and prints his/her name.

**Example 3.** Increment the age of all employees and delete those younger than 35.

```
cursor<Emp_Container> e(EMP, TRUE);                   // constructor #2

for (e.reset(), e++; e.status != EOR; e++)
{
   e.age++;
   if (e.age < 35)
      e.del();
}
```

Note that the ugly details of searching and updating arbitrarily complex data structures are hidden by the container abstraction. This *significantly* simplifies program development.

### 3.1.2 The Container OOVM

Containers, objects inside containers, and cursors are the entities that are exported by a container OOVM. The classes of these three object types are: `container<T>`, `element<T>`, and `cursor<K>`, where `T` is the type of objects stored and `K` = `container<T>`. We encountered `container<T>` and `cursor<K>` in the last section but `element<T>` is new. For performance reasons, a data structure may use a concrete representation of `T` objects that is very different from its abstract `T` counterpart. `element<T>` denotes the type of objects that are actually stored in containers of type `container<T>`.

These three classes define the OOVM interface of a container. Each container<T> object can have multiple cursors ranging over its objects, and each object can be referenced by multiple cursors. Figure 6 shows an ER model of this OOVM.

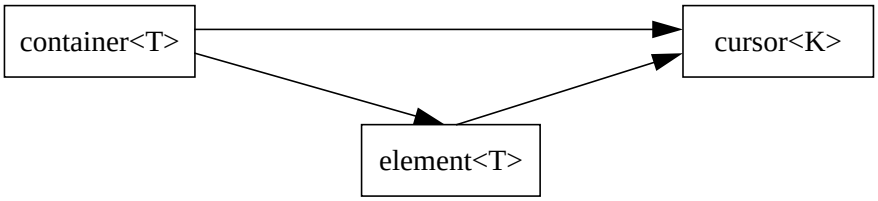



**Figure 7: The OOVM for DS containers.**

As a general rule, users are not aware of the `element<T>` class or the implementation details of the `cursor<K>` and `container<T>` classes. Supplying implementations for each class turns out to require a more complex mechanism than that provided by C++ templates. (A special preprocessor to C++ is needed to provide these mechanisms; the details of these mechanisms are deferred until Section 3.2). Initial definitions for all three classes are presented in Figure 8. Note that the `container<T>` class only has constructor and destructor functions. These functions allow any number of containers of type `container<T>` to be created and deleted. The `element<T>` class contains all public functions and attributes of `T` as private members. The once public members now appear as operations on cursors over

```
template<class T> class container
{
 // cursors are friends
 friend class cursor <container<T>>;

 private:
  // nothing initially

 public:
  container<T> ();  // constructor
  ~container<T> (); // destructor
```

(a) Definition of `container<T>`

```
template<class T> class element
{
 // cursors on element<T> are friends
 friend class container<T>;

 private:
  // private class variables,
  // functions, & instance variables
  ...

  // formerly public instance
  // variables and functions
  T1 A1;
  T2 A2;
  ...
  Tn An;

  element<T> ();  // constructor
  ~element<T> (); // destructor
  ...       // user-defined functions

  // no public variables, functions
};
```

(b) Definition of `element<T>`

```
template<class K> class cursor
// parameterized by container
{
 // cursors have access to private
 // members of their containers and
 // container elements
 // note K = container<T>
 friend class K;
 friend class element<T>;

 public:
  // instance variables
  K *k;   // the container
  enum {OK, EOR} status; // cursor status

  // public cursor operations
  cursor<K> (container); // constructor 1
  cursor<K> (container, predicate);
                        // constructor 2
  ~cursor<K> ();      // destructor
  newobj ();          // new instance
  del ();             // delete instance
  operator ++ ();     // advance cursor
  pos (oid);          // position cursor
  upd (attr, value);  // update instance

  // operations to reference instance
  // variables of elements in container
  A1;                 // attribute A1
  A2;                 // attribute A2
  ...
  An;                 // attribute An

  // previously user-define function
  ...
};
```

(c) Definition of `cursor<K>`

**Figure 8: Definition of Class T and its Cursor Definition**

`element<T>` instances. That is, all operations on objects within a container are now accomplished through operations on cursors (i.e., `newobj` is called for object creation instead of the `T` constructor `T()`, and so on).

We emphasize that that the definitions in Figure 8 are *not* in their final form. When data structures are viewed as a composition of components, each component may transform each of these class definitions by adding or removing attributes and possibly creating new container, cursor, and element types. As none of these transformations are particularly obvious, we need to consider specific examples of data structure components to show that these class definitions are indeed modified. This is one of the subjects of the next section.

## 3.2 The DS Realm

Let DS be the realm of components that implement our container OOVM of Figure 7 and Figure 8. Each DS component encapsulates a primitive data structure. Clearly there is an enormous number of such components; listing them all is impossible. Rather than enumerating realm membership, we will review specific examples and discuss the basic paradigm that accounts for a wide spectrum of useful data structures. Having done so, it is a simple intuitive leap to recognize the scope of DS realm. The specific members of DS that we will examine are:

```
DS = { DEL_FLAG[x:DS], ARRAY, MALLOC, DLIST[x:DS], BINTREE[x:DS],
       SEGMENT[p,s:DS], INDEX[d,i:DS], ... }
```

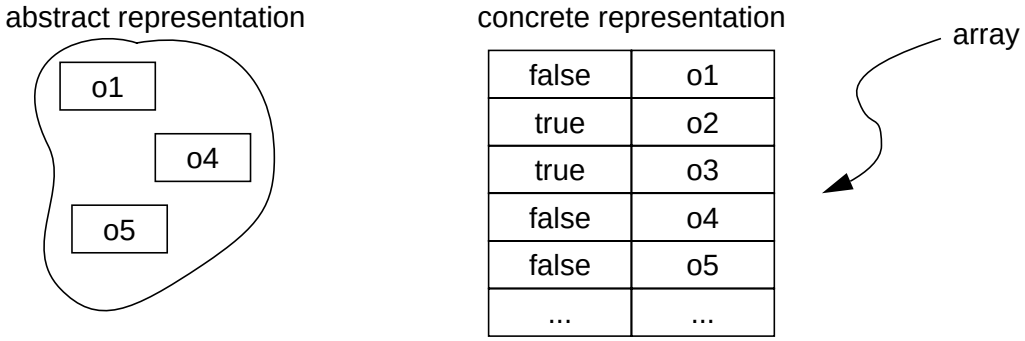
The 'big picture' that we want to convey in this section is that DS components are NPTs. Compositions of DS components implement TPTs. The container types `list_container` and `binary_tree_container` that we encountered in Section 3.1 are TPTs that can be constructed from DS components. In general, users can specify their own container types using the `container_def` statement, which defines TPTs as a composition of NPTs (DS components). Three examples are shown below:

```
container_def Simple_Array    = DEL_FLAG[ARRAY];
container_def list_container  = DLIST[MALLOC];
container_def binary_tree_container = BINTREE[DEL_FLAG[ARRAY]];
```

Of the three container types, `Simple_Array` is the most elementary. We will first examine a monolithic implementation (i.e., one without layering) of `Simple_Array` as implementations of the `element<>`, `container<>`, and `cursor<>` classes. Then we will show how this implementation is actually a composition of the two independently definable components, `DEL_FLAG[]` and ARRAY.

### 3.2.1 A Close Look At A Simple (But Composite) Data Structure: `Simple_Array`

Suppose a container is implemented by a preallocated array. Every object that it contains is augmented with a boolean attribute DF (delete flag). When an object is inserted, it is placed at the end of the array and its delete flag is set to `false`. When an object is deleted, its delete flag is set to `true` and the space the object occupies is not reclaimed. When a user requests the retrieval of all objects that satisfy a predicate P, the predicate that is actually applied to objects is ($P \wedge \neg DF$). (That is, retrieved objects must both satisfy the user's request and must not be deleted.) The figure below shows both the abstract and concrete representations of a small container of objects. Note that two of the concrete objects have been deleted.




Recall that we said in Section 3.1.2 that the definitions of the `container<>`, `element<>`, and `cursor<>` classes are transformed by container implementations. `Simple_Array` specializes the `container<>` class definition with the additional attributes `array` (which is the actual array) and

```
// new attributes for container<T> objects

element<T> array[MAX_ARRAY_SIZE]; // array for storing objects
int        free_index = 0;        // index to first free row

// new attributes for element<T> objects

bool       DF;                    // boolean delete field

// new attributes for cursor <container<T>> objects

pred       pred;                  // selection predicate
int        oid;                   // index of current object
```

```
newobj()
{
  oid = k->free_index++; // ARRAY
  DF = FALSE;            // DEL_FLAG
}

del()
{
  DF = TRUE;             // DEL_FLAG
}

update(attr, val)
{
  attr = value;          // ARRAY
}

cursor<K> (p)       // Constructor #2
{
  pred = p && !DF;       // DEL_FLAG
}

ref(attr)
{
  return(attr);          // ARRAY
}

 pos (newoid)
 {
   oid = newoid;          // ARRAY
 }
```

```
operator ++ ()
{
  element<T> *t;

  loop                    // ARRAY
  {
    if (oid++ >= k->free_index)
    {
       status = EOR;      // ARRAY
       return;            // ARRAY
    }
    t = &k->array(oid);   // ARRAY
    if (pred(t))          // ARRAY
    {
       status = OK;       // ARRAY
       return;            // ARRAY
    }                     // ARRAY
  }
}

reset()
{
  oid = -1;               // ARRAY
}
```

**Figure 9: Class Specializations and Methods for Cursor Operations**

`free_index` (the first unused row). The `element<>` class is specialized by the attribute `DF` and `cursor<>` is specialized by the attributes `oid` (the index of the current row) and `pred` (the selection predicate). Figure 9 summarizes these additions and lists the methods for each major cursor operation.

Alhough this data structure is rather simple, it is actually a composition of two independently definable components, one that encapsulates the primitive concept of "logical deletions" and another for "preallocated array". We define both components in the next two sections.

### 3.2.2 DEL_FLAG[x : DS] : DS

The `DEL_FLAG` component encapsulates the concept of logical deletion of objects. It augments each object with a boolean delete flag `DF`. When new object is created, its `DF` attribute is `false`. When it is deleted, `DF` is set to `true`; no physical deletion of the object takes place. When objects are retrieved, the selection predicate `P` is mapped to ($P \wedge \neg DF$). The figure below shows the abstract and concrete represen-

tations of a small set of objects. Note that two of the concrete objects have been logically deleted. The DEL_FLAG component is parameterized (x:DS) because its abstract-to-concrete mappings are not in any way dependent on how its concrete objects are stored.

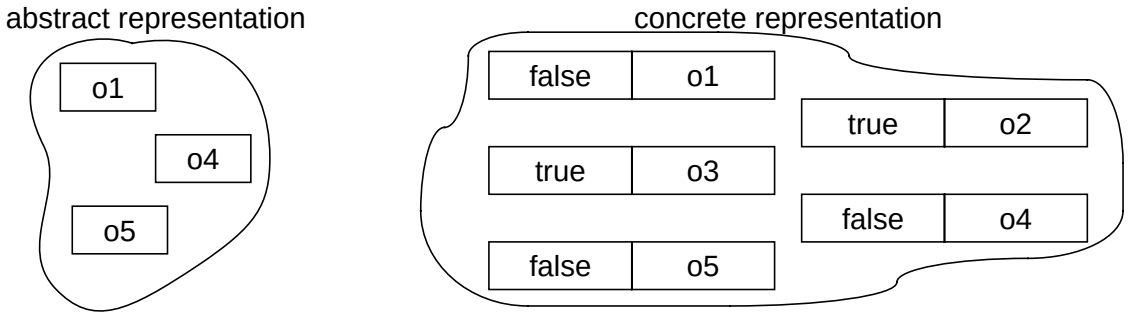



Figure 10 shows the class specializations and methods of DEL_FLAG. Note that we have taken two liberties with our C++ notation. Abstract functions have the name of the component (_DEL_FLAG) appended; concrete functions (i.e., calls to the component beneath DEL_FLAG) have the name _conc appended. Our motivation for doing so is to clearly distinguish functions on abstract objects from functions on concrete objects. Later we will see that this distinction is useful when components are composed.

```
// new attribute for element<T>
boolean DF;              // boolean delete flag
```

```
newobj_DEL_FLAG()
{
  // make new concrete instance
  // and flag if not deleted
  newobj_conc();
  upd_conc(DF, TRUE);
}

del_DEL_FLAG()
{
  // flag instance deleted
  upd_conc(DF, TRUE);
}

upd_DEL_FLAG(attr, val)
{
  upd_conc(attr, val);
}

ref_DEL_FLAG(attr)
{
  return(ref_conc(attr));
}
```

```
cursor<K>_DEL_FLAG(p)
{
  // modify selection predicate
  cursor<K>_conc((p) && !DF);
}

operator++_DEL_FLAG ()
{
  ++_conc;
}

pos_DEL_FLAG(newoid)
{
  pos_conc(newoid);
}

reset_DEL_FLAG()
{
  reset_conc();
}
```

**Figure 10: Class Specializations and Methods for DEL_FLAG**

### 3.2.3 ARRAY : DS

ARRAY is the component that encapsulates the concept of a preallocated array. ARRAY appends new objects to the end of an array; objects *cannot* be deleted (as there is no way to distinguish deleted objects from nondeleted objects). As ARRAY makes no calls to lower-level components, it has no DS parameters.

```
// new attributes for container<T> objects
element<T> array[MAX_ARRAY_SIZE]; // array for class objects
int        free_index = 0;        // index to first free row

// new attributes for cursor<T> objects
pred       pred;                  // selection predicate
int        oid;                   // index of current object
```

```
newobj_ARRAY()
{
  // find first free row
  // and remember ref to
  // object.  no array
  // bounds error checking
  oid = k->free_index++;
}

del_ARRAY(c)
{
  // not implemented by
  // this component
}

upd_ARRAY(attr, value)
{
  // direct update of attr
  attr = value;
}

cursor<K>_ARRAY(p)
{
  // remember the predicate
  pred = p;
}

ref_ARRAY(attr)
{
  return(attr);
}
```

```
operator++_ARRAY()
{
  element<T> *t;

  // examine next object; return EOR
  // if end of array or OK if object
  // is found that satisfies predicate
  loop
  {
    if (oid++ >= T.free_index)
    {
       status = EOR;
       return;
    }
    t = &k->array[oid];
    if (pred(t))
    {
       status = OK;
       return;
    }
  }
}

pos_ARRAY(newoid)
{
  // no error checking on oid
  oid = newoid;
}

reset_ARRAY()
{
  oid = -1;
}
```

**Figure 11: Class Specializations and Methods for** ARRAY

ARRAY specializes the definition of the `container` class with the addition of attributes `array` and `free_index`, which are the preallocated array and the index to the first free row. It also specializes the definition of `cursor` with the additional attributes `oid` and `pred`, which are the row number of the current object and the selection predicate. Figure 11 summarizes these specializations and lists the methods for each major cursor operation.

### 3.2.4 The Composition DEL_FLAG[ARRAY]

Keeping explicit interfaces to components is a bad idea for data structures: component methods are often so small that the overhead of function calls can seriously degrade performance. Compositions of DS components must be accomplished primarily by function renaming and inline expansion. The monolithic code for `Simple_Array` can be generated from DEL_FLAG and ARRAY by the following steps.

First, the abstract operations of DEL_FLAG are stripped of their _DEF_FLAG extensions. The renamed interface is the abstract interface (OOVM) of containers, which users will be calling from their programs. Second, each concrete function call in DEL_FLAG methods is replaced with the corresponding calls to abstract ARRAY operations (again, this is simply function renaming). Third, ARRAY methods are expanded inline. These three steps yield the algorithms given earlier in Figure 9. In that figure, we labeled each statement with the name of the component from which it arose.[5]

### 3.2.5 Other DS Components

One of the benefits of looking at the 'details' of the Simple_Array container is to get an insight into the mechanisms needed to compose DS components. (These mechanisms would be embodied in a data structures precompiler for C++). Fortunately, one does not need to know the minutia details of specific algorithms and specific variables that a component uses in order to understand what it does. Informal descriptions that explain the basic concept that is encapsulated by a component's abstract-to-concrete mappings is sufficient. In this section, we outline other DS components; it is a straightforward exercise to develop their class transformations and methods as we have done for DEL_FLAG[] and ARRAY in Section 3.2.2 and Section 3.2.3.

#### 3.2.5.1 MALLOC : DS

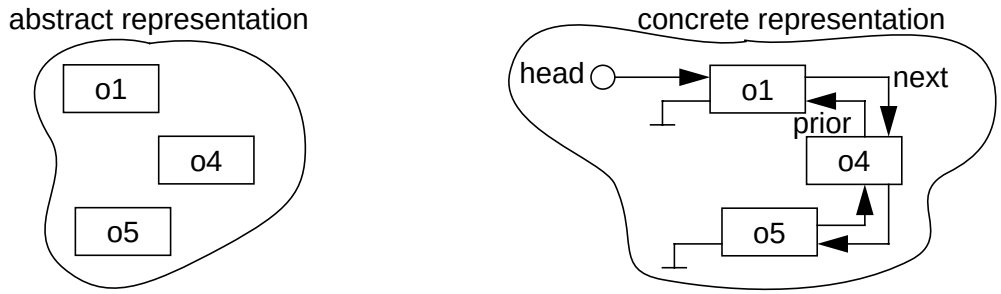
MALLOC is an alternative to ARRAY. Rather than storing objects in an array, memory for each object is allocated dynamically. Unlike ARRAY, objects can be deleted by simply reclaiming their memory. However, objects are not linked together, so scanning a container is not possible (i.e., the advance (++) operation on cursors is undefined, analogous to the delete operation for ARRAY). Also like ARRAY, the MALLOC component has no parameters as it makes no calls to lower-level components. The figure below shows abstract and concrete representations of a small set of objects that have been mapped by MALLOC.


#### 3.2.5.2 DLIST[x : DS] : DS

DLIST is the component that links together all objects of a container onto a doubly-linked list. Every object is specialized with the addition of two attributes, next and prior, that are pointers to adjacent objects. DLIST also specializes the container<> class with the attribute head, which points to the head of the list of objects. New objects are placed at the head of the list; deleted objects are removed from the list. A retrieval examines objects in list order, and returns those that satisfy the given predicate. As the mappings of DLIST are not dependent upon how concrete objects are stored, DLIST has a single parameter (x:DS) for their implementation. The figure below illustrates a DLIST mapping.

5. In general, macro expansion is not always sufficient to produce optimal code. That further optimization is sometimes necessary is not surprising when component composition is viewed as a compiler technology [Aho88].

Note that there are many other ways objects could be linked together on lists; each could be encapsulated as a distinct DS component.

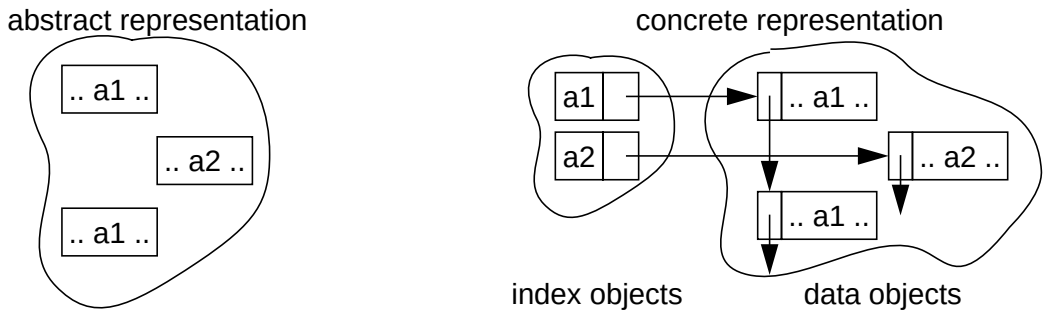



### 3.2.5.3 `INDEX[data, indx : DS] : DS`

`INDEX` is the component that maps a nonindexed container to an indexed container; that is, it creates secondary indices over selected attributes of stored objects. For each attribute that is to be indexed, a container of index objects is created. The attributes to index are conveyed to the `INDEX` component by a special class variable called `indexed_attr`, which is assigned the list of names of attributes to be indexed. As an illustration, the `emp` class below would have two index containers created: one for the `name` attribute and another for `age`.

```
class emp
{
 public:
  static ATTR_LIST indexed_attr = {name, age};
  char             name[12];
  int              age;
  char             city[15];
  char             department[25];
};
```

There is an index object in an index container for every distinct value that the indexed attribute assumes. (Thus, indexing a boolean attribute would yield a container of two index objects). There are many possible structures that can be used to interconnect an index object with its data objects. The method illustrated below shows each index object at the head of a list of data objects that have the same attribute value as the index object. Note that only one index container is shown in this figure. Also note that this is




the first example of a component that creates new containers (i.e., index containers) as part of its class definition transformation.

#### 3.2.5.4 SEGMENT[p, s : DS] : DS

SEGMENT is the component that partitions each abstract object of a container into a pair of interconnected concrete objects that are stored in separate concrete containers. Figure 12 shows an abstract class of five attributes plus an additional class variable part. The transformation of the element definition performed by SEGMENT results in two concrete element classes. The primary concrete element class contains all attributes up to but *not* including the attribute identified by class variable part. (In the figure, this is attribute A4). The secndary concrete element class contains the remaining abstract attributes. Moreover, an additional pointer ptr is stored in each primary object to connect it to its corresponding secondary object.

Segmenting records in database applications is occasionally useful [Bat78]; one need not retrieve from disk both halves of a record if only one half is needed. The performance advantages for SEGMENT in a data structure context are different. An example (from Genesis) is the partitioning of buffer header attributes from the page-buffer itself. Users write programs using a BUFFER class, which contains all attributes (headers + page buffer) of a buffer. For increased performance, page buffers should be aligned along page boundaries to minimize page faults when reading and writing buffers. Since the length of a BUFFER instance is greater than the size of a page, this alignment can only be guaranteed if the page-buffer attribute is stored separately from buffer header attributes. SEGMENT hides the ugly details of this partitioning, thereby simplifying the development of user programs [Roy91].

```
class element<T>
{
  static ATTR part = { A4 };

  T1 A1;
  T2 A2;
  T3 A3;
  T4 A4;
  T5 A5;
};
```

(a) Abstract Element Definition

```
class element_primary<T>
{
  static ATTR part = { A4 };

  T1 A1;
  T2 A2;
  T3 A3;
  element_secondary<T> *ptr;
};

class element_secondary<T>
{
  T4 A4;
  T5 A5;
};
```

(b) Concrete Element Definitions

**Figure 12: Class Definition Mapping of SEGMENT**

#### 3.2.5.5 More Components

Hopefully by now, it should be evident that the cardinality of the DS realm is very large. Every DS component maps a container of abstract objects to one or more concrete containers of concrete objects. Classical data structures (BINARY_TREE[x:DS], AVL_TREE[x:DS], HASH[x:DS]) add extra attributes to objects to interlink them together to provide fast retrievals for certain types of queries. (Primary key information for these structures would be conveyed using a class variable key, similar to the way attributes that are to be indexed are conveyed). They are parameterized in that their mappings do not depend on how their concrete objects are stored.

There are other broad classes of mappings which are not normally recognized as ‘data structures’. Perhaps the most important of these are components that realize persistent storage (e.g. [Mos88]). That is, abstract objects appear to be main-memory resident, while their concrete counterparts reside in persistent

storage. Components that encapsulate compression and encryption algorithms (e.g., RUN_LENGTH[x:DS]) translate uncompressed abstract objects to compressed concrete objects, and vice versa, also belong to the DS realm. Concepts for distributed systems can be encapsulated as DS components. Users may request objects that are not stored locally; client-server components hide the fact that objects are stored at remote sites [Hil91]. There are many other possibilities.

### 3.3 DS Expressions

In a typical application, software designers will have particular data structures in mind that they want to use. The problem is how to express this request as a composition of available DS components. We believe this problem can be addressed from two perspectives: forward engineering and reverse engineering. Through a series of examples, we will explore both perspectives to give readers an insight into the research problems that a domain model exposes.

#### 3.3.1 Forward Engineering

The forward engineering approach assumes that designers are familiar with components and can specify data structures directly in their terms. In effect, this means that designers have no difficulty interpreting DS type expressions and inferring the algorithms that would be generated. Consider the following examples.

```
container_def D1 = DLIST[DEL_FLAG[MALLOC]]
```

D1 defines the data structure where abstract objects are linked together onto a doubly-linked list. Objects that are deleted are removed from the list and are marked deleted. Memory space for each object is allocated dynamically and is never deallocated.

```
container_def D2 = DEL_FLAG[DLIST[MALLOC]]
```

D2 is almost identical to D1 except DLIST and DEL_FLAG are composed in reverse order. In D2, objects are logically deleted and are *not* removed from a doubly-linked list. Unlike D1, list traversal will encounter logically deleted objects (although such objects are never returned as a result of a retrieval).

```
container_def D3 = INDEX[SEGMENT[D1,D1], D1]
```

D3 reuses the D1 data structure. Objects in a D3 container are first indexed and then segmented. Three different types of concrete containers are created by this mapping: containers holding (1) the primary segment of abstract objects, (2) the secondary segment of abstract objects, and (3) index objects. Each concrete container (and their objects) are stored in a D1 container.

To illustrate the mappings of D3, consider the emp class:

```
class emp
{
 public:
  static ATTR_LIST indexed_attr = { age };
  static ATTR      part = city;

  char name[12];
  int  age;
  char city[15];
  char department[25];
};
```

D3 would first create a container of index objects for attribute age. D3 would partitions emp into a primary segment (containing attributes name and age) and a secondary segment (containing attributes city and department). The resulting class definitions for the index, primary segment, and secondary

segment objects, as mapped by D3, are listed in Figure 13. Keep in mind that each of the class (actually element) definitions of Figure 13 would be created by a preprocessor and are hidden from users. Users always write programs based on unmapped class definitions.

```
class element_primary<emp>
{
 public:
  static ATTR_LIST        indexed_attr = {age};
  static ATTR             part = city;

  // augmented attributes
  element_primary<emp>   *next_index; // INDEX ptr
  element_primary<emp>   *next;       // DLIST ptr
  element_primary<emp>   *prior;      // DLIST ptr
  element_secondary<emp> *ptr;        // SEGMENT ptr
  boolean                DF;          // DF delflag

  // original emp attributes
  char name[12];
  int  age;
}
```

(a) element_primary Class Definition

```
class element_secondary<emp>
{
 public:
  // augmented attributes
  element_secondary<emp> *next;      // DLIST ptr
  element_secondary<emp> *prior;     // DLIST ptr
  boolean                DF;         // DF delflag

  // original emp attributes
  char                    city[15];
  char                    department[25];
}
```

(b) element_secondary<emp> Class Definition

```
class age_index
{
  public:
   int                  age;
   element_primary<emp> *ptr;       // INDEX list head
}
```

(c) age_index Class Definition

**Figure 13: Mappings of the D3 Data Structure**

### 3.3.2 Reverse Engineering

Reverse engineering assumes that designers are not familiar with components. Rather, designers outline the concrete class definitions and the algorithms for basic cursor operations. The problem is to infer what composition of DS components yields this target data structure. An example of this type of problem is given below.

Consider Figure 14 which shows a container of objects of type R. A binary tree maintains objects in key order, where nodes of the tree store pointers to objects rather than containing the objects themselves.

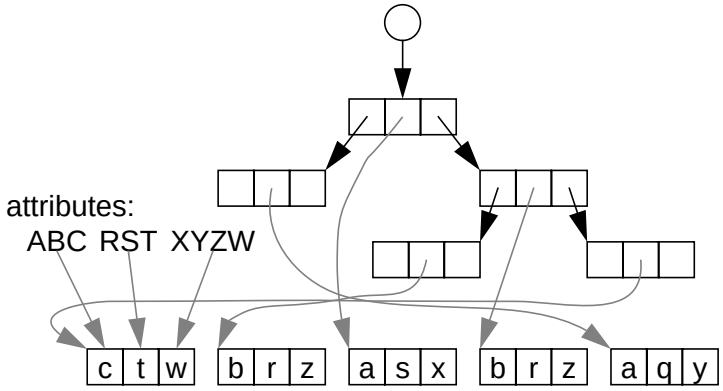



**Figure 14: A Layered Data Structure**

Binary tree nodes are dynamically allocated. What is the DS type expression for this data structure/container?

There are three steps in reverse engineering.

1. Identify the abstract class and its objects that are to be stored.

   In this example, abstract 3-tuples are being stored. The class definition for these tuples is shown below. (For the moment, ignore the class variables in the definition).

```
class R
{
  static ATTR key  = ABC;         // class variables
  static ATTR part = ABC;

  enum {a, b, c}    ABC;
  enum {r, s, t}    RST;
  enum {x, y, z, w} XYZW;
}
```

2. Recognize the mappings/components that are used.

   From the informal description, we suspect that the mappings of `MALLOC, BINARY_TREE[x:DS]`, and `SEGMENT[x,y:DS]` are referenced. Since the binary tree maintains objects in attribute `ABC` order, we convey key information to `BINARY_TREE` through the `key = ABC` class variable. Also, since all attributes appear to be stored in secondary objects of a `SEGMENT` mapping, the `part = ABC` class variable must also be added.

3. Propose a combination of components from those identified in previously and confirm or refute that the combination produces the desired implementation.

   As a first guess, consider the expression:

```
E1 = SEGMENT[BINARY_TREE[MALLOC], MALLOC]
```

E1 immediately segments objects into primary and secondary partitions. The primary partition consists of a single pointer to its secondary partition. Primary objects are stored in a binary tree, whose nodes are dynamically allocated. Secondary objects are stored in an MALLOC structure.

The concrete class definitions generated by E1 are what we want. Unfortunately, the resulting algorithms are not. In particular, note that BINARY_TREE maintains its objects in ascending order. In our case, the order that would be maintained is *pointer* order, not ABC attribute order. Thus, E1 does not meet our specifications.

Now consider the expression:

```
E2 = BINARY_TREE[SEGMENT[MALLOC, MALLOC]]
```

E2 produces identical concrete class definitions as that of E1. However, this time the desired algorithms are generated. E2 organizes objects of a container onto a binary tree using key ABC. SEGMENT then removes attributes ABC, RST, XYZW from each node and replaces them with a pointer to a secondary object that contains their values. Binary tree nodes and secondary objects are stored in an MALLOC structure.

In general, reverse engineering problems are challenging. We believe that graphical tools to guide the process of developing DS type expressions from concrete requirements may be needed to aid designers. Note also the possibility that related design tools may be able to "walk" the space of all type expressions to suggest the data structures (DS expressions) that are likely to be the most efficient for a given workload. Developing such tools is an interesting and nontrivial open problem.

### 3.4 Extensions to the Domain Model

If a domain is at all complicated, its domain model will have multiple realms. This is the case for data structures. Until now, we have been concentrating on data structures that implement containers. There are other data structures that do not deal with container implementations.

In any reasonable application objects of different containers will need to be related. These relationships, such as inheritance and link relationships, do not fit the DS abstractions we have presented so far, and not surprisingly, nor do their implementations. In this section we outline extensions to our domain model. A more complete survey of possible extensions is given in [Bat91].

A *link* is (the classical database) relationship between two (typically distinct) containers. One container of a link is called the *parent*, the other is the *child*. Typically, each parent object is associated with zero or more child objects. A link is defined by its name and a *join predicate* which relates parent objects to child objects. Consider the following container and link definitions:

```
class emp
{
 public:
  char name[12];
  int  age;
  char city[15];
  char department[12];
}

class dept
{
 public:
  char dept_name[12];
  char manager_name[12];
  int  building_number;
}
```

```
D1<emp>  EMP;
D2<dept> DEPT;

RLIST  works_in(DEPT.dept_name == EMP.depatment);
PARRAY mgr(DEPT.manager_name == EMP.name);
```

EMP is a D1 container of emp objects. DEPT is a D2 container of dept objects. Each object in DEPT is connected to zero or more objects in EMP via the works_in link. Moreover, each object in DEPT is connected to an object in EMP via the mgr link. Just as there can be different implementations of containers, so too can there be different implementations of links. works_in is implemented by the RLIST (ring-list) data structure and mgr is implemented by the PARRAY (pointer array) data structure.

Traversing links (i.e., retrieving the child objects of a parent and vice versa) is the primary link operation, and is not an operation of the DS interface. The link abstraction gives rise to another OOVM and to a realm of link implementations as indicated above. Users would program much in the same way as before, except that now retrieval predicates may contain link names or join predicates. For example, to retrieve all employees that work in departments located in building #5 might be specified as:

```
cursor<EMP> e(works_in && building_number == 5);

for (e.reset(), e++; e.status != EOR; e++)
{
 ...
}
```

Link components such as ring-lists require more complex forms of class transformations than we have encountered so far. Each parent object is augmented with a pointer to the head of a list of child objects, and each child object is augmented with a pointer to the next object on the list. What this means is that ring-list simultaneously specializes objects in *both* the parent and child containers by adding pointer attributes to each [Mor89, New89].

Transformations are even more complicated in the case of inheritance. Subclass-superclass (or actually, subcontainer-supercontainer) relationships simultaneously hold for many different containers. Components that augment an attribute to objects in one container must also augment that same attribute to all objects in subcontainers of that container. Implementations of these ideas are presented in [Vas91].

## 4.0 Conclusions

The underlying premise of the GenVoca framework is that well-understood domains of hierarchical software systems can be standardized in terms of realms of plug-compatible and reusable components, and that complex systems are compositions of these components. We have outlined a domain model of data structures that has been cast in these terms. We explained how familiar data structures could be expressed as realms of plug-compatible, symmetric, and reusable primitives, and showed how complex data structures can be constructed from their compositions.

We indicated how our data structure components are related to traditional parameterized types (TPTs): they are the building-blocks of TPT implementations. We also suggested that the realm of data structures is broader than that describable by TPTs (e.g., links). More work is needed to define precisely the expressibility of TPTs and compositions of our components.

The approach that we have taken to express our domain model is nonstandard. Recognizing the primitives of a domain, and identifying object-oriented virtual machines that are suitable for standardization is by no means simple or obvious. It is through a detailed analysis of a domain that justification for specific abstractions and interfaces is possible.

On a broader note, widespread recognition is lacking that domain-independent concepts can serve as a framework for defining large-scale building-block software technologies. We stated earlier that the GenVoca concepts were validated previously on two different domains; this paper outlines work on a third. It is our belief that as more domains are analyzed in this manner, the more we will understand the generality and contribution of the GenVoca framework.

As mentioned in the introduction, the complexity of domain models means that they cannot be taken at face value; they must be validated through implementations. A prototype compiler has been written in C that has validated the concepts in this paper. However, more work needs to be done, for example, to show that generated code is indeed efficient. This is the subject of future work.

## 5.0 Acknowledgements

We thank the referees for their thorough and insightful comments; their input significantly improved this paper.

## 6.0 References

[ACM91] ACM, 'Next Generation Database Systems', *Communications of the ACM*, October 1991.

[Aho88] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1988.

[Bat79] D.S. Batory, 'On Searching Transposed Files', *ACM Transactions on Database Systems*, 4 #4 (December 1979).

[Bat85] D.S. Batory, 'Modeling the Storage Architectures of Commercial Database Systems', *ACM Trans. Database Syst.*, 10 #4 (Dec. 1985), 463-528.

[Bat86] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise, 'GENESIS: An Extensible Database Management System', *IEEE Trans. on Soft. Engr.*, November 1988.

[Bat88] D.S. Batory, 'Concepts for a DBMS Synthesizer', in [Pri91] and *ACM PODS*, 1988.

[Bat91] D.S. Batory and S.W. O'Malley, 'The Design and Implementation of Hierarchical Software Systems Using Reusable Components', University of Texas TR-91-22, submitted for publication.

[Boo87] G. Booch, *Software Components with Ada*, Benjamin Cummings, 1987.

[Coh85] D. Cohen and N. Goldman, 'AP5 Manual', Information Science Institute, University of Southern California, 1985.

[Che76] P.P. Chen, 'The Entity-Relationship Model: Toward a Unified View of Data', *ACM Transactions on Database Systems*, 1 #1 (January 1976).

[Dij68] E.W. Dijkstra, 'The Structure of THE Multiprogramming System', *Comm. ACM*, 11 #5 (May 1968).

[Ghe82] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, John Wiley & Sons, 1982.

[Gog84] J. Goguen, 'Parameterized Programming', *IEEE Trans. Soft. Engr.*, SE-10 #5 (September 1984).

[Gog86] J.Goguen, 'Reusing and Interconnecting Software Components', *IEEE Computer*, 19 #2 (February 1986). Also in [Pri91].

[Hab76] A.N. Habermann, L. Flon, and L. Cooprider, ‘Modularization and Hierarchy in a Family of Operating Systems’, *Communications of the ACM*, 19 #5 (May 1976).

[Hil91] D.K. Hildebrand, ‘Building Relational Client-Server DBMSs from Components’, M.Sc. Thesis, Dept. of Computer Sciences, University of Texas, 1991.

[Hut91] N.C. Hutchinson and L.L. Peterson, ‘The *x*-Kernel: An Architecture for Implementing Network Protocols’, *IEEE Trans. on Soft. Engr.*, 17 #1 (January 1991).

[Kor91] H.F. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, 1991.

[Mor89] E. Mora, ‘Design and Implementation of Link Layers for the Genesis Database Management System’, M.Sc. Thesis, Dept. Computer Sciences, University of Texas, 1989.

[Mos88] J.E.B. Moss and S. Sinofsky, ‘Managing Persistent Data with MNEME: Designing a Reliable Shared Object Interface’, *Second International Workshop on Object-Oriented Database Systems*, 1988.

[New89] D. Newman, ‘Link Implementations as Software Building Blocks’, M.Sc. Thesis, Dept. Computer Sciences, University of Texas, 1989.

[OMa90] S.W. O’Malley and L. Peterson, ‘A New Methodology for Designing Network Software’, University of Arizona TR 90-29 (Sept. 1990). Submitted for publication.

[Pet90] L. Peterson, N. Hutchinson, and H. Rao, and S.W. O’Malley, ‘The *x*-kernel: A Platform for Accessing Internet Resources’. *IEEE Computer (Special Issue on Operating Systems)*, 23 #5 (May 1990), 23-33.

[Pri91] R. Prieto-Diaz and G. Arango, *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

[Roy91] J. Roy, ‘Design and Use of the Jupiter File Management System’, M.Sc. Thesis, Dept. of Computer Sciences, University of Texas, 1991.

[Rum91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[Str91] B. Stroustrup, *The C++ Programming Language*, 2nd edition, Addison-Wesley, 1991.

[Teo86] T.J. Teorey, D. Yang, and J.P. Fry, ‘A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model’, *Computing Surveys*, 18 #2 (June 1986).

[Vas91] D. Vasavada, ‘Design and Implementation of Structural Inheritance for the Genesis Database Management System’, M.Sc. Thesis, Department Computer Sciences, University of Texas, 1991.