

The Discipline and Method Architecture for Reusable Libraries

Kiem-Phong Vo
AT&T Laboratories – Research
180 Park Avenue, Florham Parks, NJ 07932, U.S.A.
kpv@research.att.com

SUMMARY

Over the past several years, my colleagues and I have written a number of software libraries for fundamental computing tasks including I/O, memory allocation, container data types, and sorting. These libraries have proved to be good software building blocks and are used widely by programmers around the world. This success is due in part to a library architecture that employs two main interface mechanisms: *disciplines* to define resource requirements, and *methods* to parameterize resource management. Libraries built this way are called *discipline and method libraries*.

KEYWORDS: Discipline and method libraries, handle and operations architecture, resource-oriented library design, buffered I/O, region memory allocation, container data types.

INTRODUCTION

Building software libraries is a good way to capture domain knowledge in reusable code. The popularization of languages such as C and C++ and operating systems such as Unix and Windows has brought about a plethora of libraries covering a wide range of functions from basic I/O and memory allocation [1, 2] to container data types [3, 4, 5], sorting, and mathematical functions. Despite this apparent success, the usefulness of libraries are often not fully realized due in part to two problems: *resource rigidity* and *interface multiplicity*.

Resource rigidity is endemic in many traditional software libraries that are often geared toward specific resources even though their techniques are much more generally applicable. A well-known example is the ANSI-C Malloc [1] memory allocation library whose interface is aimed only at heap memory. This means that the implementations of various Malloc versions based on advanced allocation algorithms [6, 7] cannot be reused for allocating other memory types such as shared or persistent memories. Applications dealing with such alternative memories often end up inventing specialized allocators with inferior algorithms.

Interface multiplicity often arises when multiple methods and policies exist for the same resource management purposes. A common mistake in library design is to tightly bind the operational interfaces with such methods or policies, resulting in multiple semi-distinct interfaces. Container

data type packages such as the STL templates [4] or the Hanson's components [8] tend to exhibit this problem. These packages support the same basic container operations such as object insertion, deletion and search but ordered and unordered object collections often require different object types and, in some cases, even use different interfaces. This means that ad-hoc adaptation code must be invented whenever different collection types are needed for the same objects or object types.

Resource rigidity and interface multiplicity are symptoms showing that the resource requirements and management policies of a library have not been fully factorized and generalized. To address this problem, my colleagues and I have been experimenting with an alternative library architecture in our effort to build a collection of reusable libraries.^[9, 10] Our libraries make explicit two main interface devices: *disciplines* to define resource requirements and *methods* to encapsulate different ways to manage resources. Operations on resources (i.e., library functions) remain abstract and become concrete only after combining the desired disciplines and methods. We call libraries based on these interface devices *discipline and method libraries*.

To maximize reusability, we assert that a discipline type must appear uniformly across its corresponding methods so that the algorithms and data structures underlying such methods can be parameterized by the discipline type. This enables applications to define their own resource types and immediately reuse all implemented resource management methods and policies. Clearly, the major work in designing a library is to arrive at the proper discipline and method interfaces. Toward this end, we use a *resource-oriented methodology* to analyze library requirements from both resource and operational points of view.

Our libraries deal with common computing aspects including IO, memory allocation, data compression, and container data types. Some of them, such as Vmalloc, Sflo and Cdt,^[6, 11, 12] are widely used around the world in well-known applications including the Korn^[13] shell, the UWIN^[14] environment, and the Perl^[15] language. These libraries share the below characteristics:

- Each library provides a parsimonious and general interface that solves a range of problems traditionally requiring multiple libraries or tools,
- Each can be easily combined with other libraries,
- Each performs as well as or better than other comparable packages,
- Each is single source, yet works on diverse and incompatible platforms, and
- Each is single binary image per installation.

Thus, our libraries have achieved generality and ease of use with enhanced performance and portability. The rest of the paper discusses the overall discipline and method library architecture and the resource-oriented approach to analyzing library requirements and interfaces. Examples and experimental results will be given to show that disciplines and methods both enable flexible system construction and enhance performance. Finally, a discussion of related works and the benefits of designing libraries based on the discipline and method architecture will be presented.

Library architecture

Handle	Operations
Resource	Opening/closing handles
States	Abstract resource functions
Discipline	Methods
Resource definition	Algorithms
Resource acquisition	Management styles
Event handling	Special modes

Figure 1: The discipline and method architecture

Figure 1 summarizes the discipline and method library architecture. Four main interface constructs are used to manage resources:

- *Handle*: A handle holds resources and states that need to be maintained across operations.
- *Operations*: Library operations create/delete handles and access handle resources.
- *Discipline*: A discipline type publically and uniformly defines certain resource requirements.
- *Methods*: Methods encapsulate different operational or performance characteristics.

The top two parts of Figure 1 constitute the familiar handle and operations architecture of many traditional libraries such as the ANSI-C Stdio library [1] and the STL template libraries.[4] This architecture is simple and easy for both library writers and application developers to learn and use. However, its simplicity also means that the resource requirements of a library must be hardwired into the library functions and the implemented resource management policies must be fixed and kept hidden from applications. Thus, applications have no flexibility in defining alternative resource types or in selecting appropriate algorithms and policies for resource management.

To increase library usability, we extend the top two parts of Figure 1 with the bottom two parts, disciplines and methods. A discipline type encapsulates certain necessary resource requirements in an abstract data structure. Library functions are then parameterized by discipline instances, or disciplines for short. A library may provide default disciplines for certain standard resource types but disciplines are typically supplied by applications. Handle operations can either be made concrete or remain abstract and be parameterized by some method type. In the latter case, the library would provide a predefined set of method instances, or methods for short. Each method represents some aspect of resource management such as an algorithm or just some special policy.

Figure 2 shows the Vmalloc library as an example discipline and method library. Vmalloc allows applications to create *regions*, each of which holds one or more memory segments from which smaller pieces are allocated per request. A region is opened by calling `vmopen()` with a discipline, `disc`, to obtain raw memory segments and a method, `meth`, to define an allocation policy. Vmalloc provides a standard heap region, `Vmheap`, to allocate heap memory. This region uses `Vmdcsbrk`, one of the two standardly provided disciplines:

Region	Allocation operations
Memory segments Free lists	<pre> Vmalloc_t* vmopen(Vmdisc_t* disc, Vmethod_t* meth, int type) int vmclose(Vmalloc_t* vm) void* vmalloc(Vmalloc_t* vm, size_t size) void* vmresize(Vmalloc_t* vm, void* addr, size_t size, int type) int vmfree(Vmalloc_t* vm, void* addr) ... </pre>
Vmdisc_t	Vmethod_t
Raw memory manager Event handler	Vmbest, Vmpool, Vmlast Vmdebug, Vmprofile

Figure 2: The Vmalloc library architecture

- **Vmdcsbrk**: This discipline obtains raw memory segments via either the Unix `sbrk()` system call or the Windows `VirtualAlloc()` call.
- **Vmdcheap**: This discipline uses `Vmalloc` calls to recursively obtain memory from the heap region. This is an example of how `Vmalloc` can be used to organize memory in non-trivial ways, creating regions out of memory from other regions.

The allocation calls `vmalloc()`, `vmresize()` and `vmfree()` manage memory in a region based on the allocation policy defined by the given method `meth`. The available allocation methods are:

- **Vmbest**: a general purpose allocation method based on a best-fit policy (i.e., allocating from a smallest suitable free area) with additional heuristics to improve allocation speed and reduce space fragmentation. The heap region uses this method.
- **Vmpool**: a special allocation method suitable for dealing with objects of the same size.
- **Vmlast**: a special allocation method that disallows most freeing except on the last allocated block. It is good for building complex structures that always get deleted in whole.
- **Vmdebug**: a general purpose allocation method that issues alerts of memory usage errors.
- **Vmprofile**: a general purpose allocation method that tracks and profiles memory usage.

Figure 3 shows another example discipline and method library, `Cdt`, for manipulating container data types. A `Cdt` handle is called a *dictionary* and opened with the call `dtopen(disc, meth)`. The discipline argument `disc` defines object characteristics while the method argument `meth` specifies a particular container data type. The available container data types are ordered/unordered sets/multisets, lists, stacks, and queues. These methods are implemented on top of container data structures such as top-down splay trees, hash tables, and linked lists.^[16]

Dictionary	Operations
Objects Binary trees Hash tables Linked lists	<pre> Dt_t* dtopen(Dtdisc_t* disc, Dtmeth_t* meth) int dtclose(Dt_t* dt) void* dtinsert(Dt_t* dt, void* obj) void* dtdelete(Dt_t* dt, void* obj) void* dtsearch(Dt_t* dt, void* obj) void* dtfirst(Dt_t* dt) void* dtnext(Dt_t* dt, void* obj) void* dtlast(Dt_t* dt, void* obj) void* dtprev(Dt_t* dt, void* obj) ... </pre>
Dtdisc_t	Dtmeth_t
Object description, Comparator, Constructor, destructor, Allocator, event handler	<pre> Dtset, Dtbag Dtset, Dtobag Dtlist Dtstack, Dtqueue </pre>

Figure 3: The Cdt library architecture

A token counting example

To ease the discussion of disciplines, methods and their usage, we shall use as examples code fragments from a *token counting application*. This application reads text from the standard input, partitions it into tokens and computes token frequencies. Here, a token is any sequence of characters separated by blanks, tabs, carriage-return and new-lines. Figure 4 shows the main parts of the application. Lines 1-4 define a `Token_t` type associating a token to its frequency. Line 5 declares a Cdt discipline `Tkdisc`. The initialization code for `Tkdisc` will be discussed later. Lines 10-11 create a dictionary `dt` to store tokens. The Cdt method `Dtset` is used to store tokens in an ordered set. Lines 12-18 form the main processing loop to read tokens and update their frequencies. For example, the `if` clause inserts a new token into the dictionary. The function `tkread()` reads tokens from a stream. Its description will be omitted. Lines 19-20 output tokens and their frequencies. Line 21 releases all space associated with `dt`.

DISCIPLINES

The primary purpose of disciplines is to define resource types and methods to acquire, process and release resources. Interesting events or exceptions often accompany data processing so disciplines also often define functions to handle such events.

```

1. typedef struct
2. {   char*   name;
3.     int    freq;
4. } Token_t;

5. Dtdisc_t Tkdisc;

6. main()
7. {   Dt_t*   dt;
8.     char*   r;
9.     Token_t token, *tk;

10.     ...initialize Tkdisc;
11.     dt = dtopen(&Tkdisc, Dtoset);

12.     while((r = tkread(sfstdin)) )
13.     {   if(!(tk = dtmatch(dt, r)) )
14.         {   token.name = r;
15.             dtinsert(dt, &token);
16.         }
17.         else tk->freq += 1;
18.     }

19.     for(tk = dtfirst(dt); tk != 0; tk = dtnext(dt,tk) )
20.         sprintf(sfstdout,"%s\t%d\n", tk->name, tk->freq);

21.     dtclose(dt);
22. }

```

Figure 4: A token counting example

Resource definition

A main use of disciplines is to define characteristics of resources processed by a handle. For example, Figure 5 shows the `Cdt` discipline structure used to define characteristics of objects stored in a dictionary. `Dtdisc_t.key` defines where the key of an object resides in the object. In the absence of a comparison function, an object key is treated as a character array. In that case, if `Dtdisc_t.size` is non-negative, it defines the key length; otherwise, the key is treated as a null-terminated string. Each object is held inside the dictionary by a holder structure of type `Dtlink_t`. The `Dtdisc_t.link` field should be set to the offset in an object where `Dtlink_t` is embedded or `-1` if this structure should be separately allocated by `Cdt`. The rest of the fields define various functions to construct and delete objects, compare and hash keys, to allocate memory and to process events. Note that, in this design, `Cdt` can support both *set-like* and *map-like* containers. A set-like container identifies objects via object comparisons so it would set both `Dtdisc_t.key` and `Dtdisc_t.size` to `0` and supply

```

1. typedef struct _dtdisc_s Dtdisc_t;
2. struct _dtdisc_s
3. { int          key;
4.   int          size;
5.   int          link;
6.   void*        (*makef)(Dt_t* dt, void* obj, Dtdisc_t* disc);
7.   void         (*freef)(Dt_t* dt, void* obj, Dtdisc_t* disc);
8.   int         (*comparf)(Dt_t* dt, void* k1, void* k2, Dtdisc_t* disc);
9.   unsigned int (*hashf)(Dt_t* dt, void* key, Dtdisc_t* disc);
10.  void*        (*memoryf)(Dt_t* dt, void* addr, size_t size, Dtdisc_t* disc);
11.  int         (*eventf)(Dt_t* dt, int type, void* data, Dtdisc_t* disc);
12. };

```

Figure 5: The Cdt discipline structure

suitable `Dtdisc_t.comparf` and `Dtdisc_t.hashf` functions. On the other hand, a map-like container identifies objects via their keys as defined by the mentioned fields.

Figure 6 shows the use of a Cdt discipline to describe the `Token_t` structure defined in Figure 4. Lines 1-8 define a function `tkmake()` to construct a `Token_t` object based on the prototype given in the argument `arg`. Note that the frequency of a new token is set to 1. Lines 9-12 define a corresponding function `tkfree()` to free space associated with `Token_t` objects. Line 13 sets `Tkdisc.key` to the offset of the field `name` in the `Token_t` structure. Line 14 sets `Tkdisc.size` to -1 to mean that the key is a null-terminated string. Lines 16-17 set the object constructor and destructor. Other functions in the discipline structure are not defined so they will default to some internally provided Cdt functions. For example, memory will be allocated via `Malloc`. Line 10 in Figure 4 should be replaced with Lines 13-17 of Figure 6 to properly initialize the discipline `Tkdisc`.

Resource processing

Disciplines can be used to define special resource processing at the point where the resource is brought into a handle or moved out of it. For example, a `Sfio` stream can be equipped with a stack of disciplines, each of which acts as a filter to process input or output data. Figure 7 gives an example of how this may work. Lines 1-7 show the `Sfio` discipline structure `Sfdisc_t`. The first two fields define functions to read raw data or to write out data, the third field defines a function to set the I/O position in the raw data stream, and the last field defines an exception handler to deal with various I/O events. Lines 8-15 show a discipline function `lower()` that translates upper case characters to lower case. Line 11 calls the `Sfio` function `sfrd()` to recursively invoke a discipline function somewhere lower in the discipline stack to read data. Each `Sfio` discipline stack has at its bottom a virtual discipline consisting of the I/O system calls. Lines 12-13 loop through the input characters and translate them to lower case. Line 16 shows a discipline `Lowerdisc` defined in terms of `lower()`. The undefined fields delegate their functions to a discipline further down in the stack. Line 17 shows how `Lowerdisc` may be inserted into the standard input stream to translate characters.

```

1. void* tkmake(Dt_t* dt, void* arg, Dtdisc_t* disc)
2. { Token_t *tk, *obj = arg;
3.   tk = malloc(sizeof(Token_t));
4.   tk->name = malloc(strlen(obj->name)+1);
5.   strcpy(tk->name, obj->name);
6.   tk->freq = 1;
7.   return tk;
8. }

9. void tkfree(Dt_t* dt, void* tk, Dtdisc_t* disc)
10. { free(((Token_t*)tk)->name);
11.   free(tk);
12. }

13. Tkdisc.key = offsetof(Token_t,name);
14. Tkdisc.size = -1;
15. Tkdisc.link = -1;
16. Tkdisc.makef = tkmake;
17. Tkdisc.freef = tkfree;

```

Figure 6: A Cdt discipline describing `Token_t`

Back to the token counting example, suppose that we wish to count tokens regardless of cases, for example, treating “The” and “the” as being the same. The Cdt discipline `Tkdisc` could be changed to do case-insensitive comparisons but that is expensive since many pairs of tokens must be compared. A simpler way is to insert Line 17 of Figure 7 before the main processing `while` loop starting at Line 12 of Figure 4. Once that is done, the token reading function `tkread()` is guaranteed to see only lower case characters. Later on, we shall show how this strategy benefits overall performance.

Resource acquisition

Disciplines can be used to define data and functions to acquire and release the raw resources of a handle. Lines 1-6 of Figure 8 show the `Vmalloc` discipline `Vmdisc_t`. The discipline function `Vmdisc_t.memoryf` is used to obtain, resize, and free raw memory segments in a handle. Segment sizes are always multiples of `round`. The exception handler `Vmdisc_t.exceptf` processes exceptional events such as out of memory.

Disciplines allow applications to adapt library functionality to arbitrary resource types and to different operating environments. Lines 7-22 of Figure 8 present two example `Vmdisc_t.memoryf` functions, one for Unix systems and one for Windows systems. As shown, memory acquisition can be defined in a few lines of code by application programmers. Once that is done, all `Vmalloc` allocation algorithms embedded in the various methods can be reused without change.

```

1. typedef struct _sfdisc_s Sfdisc_t;
2. struct _sfdisc_s
3. { ssize_t (*readf)(Sfio_t* sf, void* buf, size_t n, Sfdisc_t* disc);
4.   ssize_t (*writef)(Sfio_t* sf, const void* buf, size_t n, Sfdisc_t* disc);
5.   Sfoff_t (*seekf)(Sfio_t* sf, Sfoff_t pos, int type, Sfdisc_t* disc);
6.   int      (*exceptf)(Sfio_t* sf, int type, void* data, Sfdisc_t* disc);
7. };

8. ssize_t lower(Sfio_t* f, void* arg, size_t n, Sfdisc_t* disc)
9. { size_t i;
10.  char  *buf = arg;
11.  n = sfrd(f, buf, n, disc);
12.  for(i = 0; i < n; ++i)
13.      buf[i] = tolower(buf[i]);
14.  return n;
15. }

16. Sfdisc_t Lowerdisc = { lower, 0, 0, 0};
17. sfdisc(sfstdin,&Lowerdisc);

```

Figure 7: An example Sfio discipline

Event handling

A standard part of any discipline type is a function to handle events such as the opening and closing of a handle, changing of disciplines or methods, and other library-specific events such as running out of memory or errors in some underlying system calls. It is beyond the scope of this paper to do so but the Vmalloc and Cdt references [6, 12] show how event handling is necessary for effective construction of shared and/or persistent memory regions and dictionaries.

When an event is raised by some operation, the event handler can dictate subsequent actions via return values. These values are treated based on the *RPR* or *Return-Proceed-Retry* [17] approach:

- < 0: the operation will return immediately with an error status,
- = 0: the operation will proceed in some library-defined way, and
- > 0: the operation will be retried as if the event did not happen.

Figure 9 shows a Vmalloc event handler `memexcept()`. Here, the event handler is restricted to handle only the out of memory event, `VM_NOMEM`, raised when an attempt to get more raw memory into the handle `vm` fails. Lines 3-4 try to perform garbage collection. If this is successful, the handler returns 1 to tell the library to retry the allocation call. Lines 5-8 print an appropriate error message when garbage collection fails, then cause the application to exit with an error status. Line 10 returns 0 for events other than `VM_NOMEM` so that some default actions as defined by the library will be taken.

```

1. typedef struct _vmdisc_s Vmdisc_t;
2. struct _vmdisc_s
3. { void* (*memoryf)(Vmalloc_t* vm,void* addr,size_t csz,size_t nsz,Vmdisc_t* disc);
4.   int  (*exceptf)(Vmalloc_t* vm,int type,void* data,Vmdisc_t* disc);
5.   size_t round;
6. } Vmdisc_t;

7. void* unixmem(Vmalloc_t* vm,void* addr,size_t csz,size_t nsz,Vmdisc_t* disc)
8. { unsigned char* seg;
9.   if(csz > 0 && sbrk(0) != (unsigned char*)addr+csz)
10.     return (void*)0;
11.   seg = sbrk((ssize_t)nsz - (ssize_t)csz);
12.   if(seg == (unsigned char*)-1)
13.     return (void*)0;
14.   else return csz == 0 ? addr : seg;
15. }

16. void* windowmem(Vmalloc_t* vm,void* addr,size_t csz,size_t nsz,Vmdisc_t* disc)
17. { if(csz == 0)
18.     return (void*)VirtualAlloc(0,nsz,MEM_COMMIT,PAGE_READWRITE);
19.   else if(nsz == 0)
20.     return VirtualFree(addr,0,MEM_RELEASE) ? addr : (void*)0;
21.   else return (void*)0;
22. }

```

Figure 8: Vmalloc discipline examples

Judicious uses of event handling can help to simplify application code and make it more efficient. Even if we ignore the garbage collection part in the above example, out of memory errors are being handled more elegant, accurate and efficient than in the usual convention of testing for a null return value from each allocation request.

Discipline extension

The reader may have noticed that a discipline function always has the discipline itself as the last argument. This feature allows applications to extend a discipline structure to hold certain private states. As an example, consider the code in Figure 4 and Figure 6. Allocating tokens via Malloc as shown in Figure 6 can incur significant space overhead. Further, significant time is used to close a dictionary since each token must be freed separately by a `tkfree()` call. Both problems can be solved by using a Vmalloc region allocating with the `VmLast` method.

Figure 10 shows how a Cdt discipline is extended to use a Vmalloc region for memory allocation. Lines 1-4 shows that the extended discipline `Tkdisc_t` includes `Dtdisc_t` as the first element and has a field `vm` to hold a Vmalloc region. Including `Dtdisc_t` as the first field enables C typecasting

```

1. int memexcept(Vmalloc_t* vm, int type, void* obj, Vmdisc_t* disc)
2. {   if(type == VM_NOMEM)
3.     {   if(garbage_collect() > 0)
4.         return 1;
5.     else
6.         {   sprintf(sfstderr,"Out of memory\n");
7.             exit(1);
8.         }
9.     }
10.    return 0;
11. }
```

Figure 9: An example Vmalloc exception handler

to fudge `Tkdisc_t` and `Dtdisc_t` pointers. Lines 5-7 show how the Cdt memory allocation function `tkalloc()` is implemented on top of the Vmalloc function `vmresize()`. Lines 8-12 show how an event handling function is defined to free the entire allocation region at the point when the respective dictionary is being closed. Lines 13-21 redefine the Cdt discipline function `tkmake()` of Figure 6 to allocate tokens using Vmalloc calls.

Figure 11 redefines the `Tkdisc` discipline based on the new extended discipline structure in Figure 10. Line 2 creates a Vmalloc region based on the standardly provided discipline `Vmdcheap` and the allocation method `Vmlast`. As the dictionary grows, `Vmdcheap` recursively calls Vmalloc functions to obtain large chunks of memory from the heap region. In turn, the various memory allocation calls in the Cdt discipline functions use the allocation policy defined in `Vmlast` to manage memory. Lines 6-8 set the new object constructor, memory allocator and event handler. Note that we no longer need a `tkfree()` function since `tkevent()` will deallocate the entire dictionary when the dictionary is closed.

The association of an allocator to a container is similar to the use of an `Allocator` class [18] in STL. However, `Allocator` was originally invented to deal with typing issues concerning different memory models such as normal and `far` pointers in certain flavors of C++ and not necessarily to deal with allocation strategies. In any case, without a discipline and method library like Vmalloc, constructing a specialized allocator can be time consuming. The joint use of Vmalloc and Cdt to save space and time is almost effortless here.

METHODS

The operations provided by a library may require different interpretations and/or implementations depending on needed functional or performance characteristics. In such a case, it is desirable to define the resource operations abstractly and parameterize them with some method type. For example, the abstract operations `vmresize()` and `vmfree()` of Vmalloc are parameterized by some selected method such as `Vmbest` or `Vmlast`. However, not all libraries require multiple methods. For example, the `Sfio` library implements all of its operations concretely. Also, unlike disciplines that are

```

1. typedef struct _tkdisc_s
2. {   Dtdisc_t   disc;
3.     Vmalloc_t* vm;
4. } Tkdisc_t;

5. void* tkalloc(Dt_t* dt, void* addr, size_t size, Dtdisc_t* disc)
6. {   return vmresize(((Tkdisc_t*)disc)->vm, addr, size, VM_RSMOVE|VM_RSCOPY);
7. }

8. int tkevent(Dt_t* dt, int type, void* data, Dtdisc_t* disc)
9. {   if(type == DT_CLOSE)
10.        vmclose(((Tkdisc_t*)disc)->vm);
11.    return 0;
12. }

13. void* tkmake(Dt_t* dt, void* arg, Dtdisc_t* disc)
14. {   Token_t   *tk, *obj = arg;
15.     Tkdisc_t *dc = (Tkdisc_t*)disc;
16.     tk = vmalloc(dc->vm, sizeof(Token_t));
17.     tk->name = vmalloc(dc->vm, strlen(obj->name)+1);
18.     strcpy(tk->name, obj->name);
19.     tk->freq = 1;
20.     return tk;
21. }

```

Figure 10: Extending a Cdt discipline

often supplied by applications, all method instances must be provided by a library. When multiple methods are available, each method instance serves some particular purpose as discussed below.

Algorithm encapsulation

A method instance may implement some particular algorithm and data structure to match performance requirements to operational characteristics. For example, the Cdt method `Dtoset` implements ordered sets as top-down splay trees and guarantees amortized time $O(\log n)$ per access operation. On the other hand, unordered sets have less stringent requirements so the method `Dtset` uses hash tables and guarantees average time $O(1)$ per access operation.

Semantics specialization

General data structures often have multiple usage semantics. For example, a splay tree as a container data structure can be used to implement either set semantics (no duplicated elements) or

```

1. Tkdisc_t Tkdisc;
2. Tkdisc.vm = vmopen(Vmdcheap, Vmlast, 0);
3. Tkdisc.disc.key = offsetof(Token_t,name);
4. Tkdisc.disc.size = -1;
5. Tkdisc.disc.link = offsetof(Token_t,link);
6. Tkdisc.disc.makef = tkmake;
7. Tkdisc.disc.allocf = tkalloc;
8. Tkdisc.disc.eventf = tkevent;

```

Figure 11: Extending a Cdt discipline

multiset semantics (duplicated elements allowed). The Cdt methods `Dtobag` and `Dtoset`, that represent ordered set and ordered multiset respectively, are implemented by the same splay tree code. The selection of a particular method tells the code which semantics are applicable.

Special performance modes

Engineering problems such as memory allocation [6, 19, 20, 21] often do not have generally optimal solutions. In such cases, it is desirable to identify important special cases with efficient solutions. Methods can be used to make such solutions available under a uniform interface. For example, `Vmalloc` provides a method `Vmpool` to allocate objects of the same size and another method `Vmlast` to construct complex objects that only get deallocated as a whole (e.g., a graph or a parse tree). These methods capture common ways of allocation and eliminate the need for application programmers to invent special implementations on top of a general purpose allocator such as `Malloc`.

Atypical usage modes

Methods can also be used to fit solutions for atypical but important usage modes into a general framework. For example, memory debugging and profiling are important for assessing the quality of programs using dynamic memory allocation. These activities are often done separately from normal memory allocation because the provided tools [22] alter code and data in ways not suitable for production use. By contrast, the `Vmalloc` library integrates such techniques under a uniform method interface so that applications can select per region either an efficient allocation method or a method suitable for error detection and memory usage tracking. Thanks to this flexibility, `Vmalloc` was able to provide an upward compatible `Malloc` interface that allows method selection at execution time via setting environment variables. Thus, the user of an application based on the `Malloc` provided by `Vmalloc` can run it efficiently in normal circumstances but also has the option to debug or profile memory usage by simply setting an appropriate environment variable.

DYNAMIC DISCIPLINES AND METHODS

A library may allow a handle to change its discipline and method dynamically if handle states are not critically dependent on these structures. For example, the Cdt library allows dynamic changing of disciplines and methods because such a change merely means reinserting the current set of objects based on the new semantics. The Sfio library uses disciplines as filters to process data so it allows each stream handle to have a stack of disciplines. On the other hand, the Vmalloc library does not allow changing disciplines and methods after a region is opened. This is because allocated memories carry state data (e.g., allocated sizes) or types (shared or heap memory) that depend on the particular methods or disciplines.

```

1. ssize_t depunct(Sfio_t* f, void* arg, size_t n, Sfdisc_t* disc)
2. { size_t i;
3.   char *buf = arg;
4.   n = sfrd(f, buf, n, disc);
5.   for(i = 0; i < n; ++i)
6.       if(buf[i] == ',' || buf[i] == ';' || buf[i] == '.')
7.           buf[i] = ' ';
8.   return n;
9. }

10. Sfdisc_t Depunctdisc = { depunct, 0, 0, 0};

11. sfdisc(sfstdin,&Lowerdisc);
12. sfdisc(sfstdin,&Depunctdisc);

```

Figure 12: Dynamic stacking of Sfio disciplines

To see how discipline stacking may be useful, consider again the token counting example. In addition to white spaces, it is also reasonable to treat commas, semicolons and periods as token separators. If the source code of the token reading function `tkread()` was available, it could be rewritten to include punctuation characters. However, even if the source code for `tkread()` was not available, the problem could still be solved as shown in Figure 12. Here, a new discipline `Depunctdisc` turns punctuation characters into blanks. Lines 11-12 show how `Lowerdisc` and `Depunctdisc` are stacked on top of each other to process the standard input stream. This example is somewhat contrived since it is equally simple to extend `Lowerdisc` to also translate punctuation characters. However, it shows how these small algorithms can be encapsulated in reusable disciplines. In fact, Sfio provides a number of disciplines for common tasks such as decompressing data compressed by the Unix `compress` or `gzip` commands or to make an unseekable stream appear seekable. These disciplines enable sophisticated data processing without complex programming.

Dynamic disciplines and methods can be used to match resource semantics to usage contexts. In the token counting example, we may wish to order tokens by frequency. Figure 13 shows how this is done. Lines 1-3 define a function to compare frequencies as integers in reverse order. Lines 4-7

```

1. int freqcmp(Dt_t* dt, void* k1, void* k2, Dtdisc_t* disc)
2. {   return *((int*)k2) - *((int*)k1);
3. }

4. Tkdisc.key = offsetof(Token_t, freq);
5. Tkdisc.size = 0;
6. Tkdisc.comparf = freqcmp;
7. ...

8. dtdisc(dt, &Tkdisc, DT_SAMEHASH|DT_SAMECMP);
9. dtmethod(dt, Dtobag);

```

Figure 13: Dynamic discipline and method changes

redefine `Tkdisc`. Lines 8-9 announce the discipline change, then change the method to `Dtobag` so that objects will be stored in an ordered multiset. The flags `DT_SAMEHASH` and `DT_SAMECMP` tell `Cdt` that both hashing and comparison semantics remain the same. This technical lie saves a reordering step that would be done anyway on Line 9. Lines 4-9 of Figure 13 should be inserted before the output loop in Figure 4 to affect the desired change.

PERFORMANCE

We have discussed how discipline and method libraries enable flexibility in structuring application code. The same flexibility also helps applications to tune for performance. This can be seen by comparing the below versions of the token counting application:

- **tk**: This is the original token counting example as shown in Figure 4 and Figure 6.
- **tk-v**: This version changes the `Cdt` discipline to allocate memory via a `Vmalloc` region using the allocator `Vmlast` as shown in Figure 10.
- **tk-1**: This version counts case-insensitive tokens by using the `Sfio` discipline `Lowerdisc` to do case translation as shown in Figure 7.
- **tk-c**: This version does the same as **tk-1** but by changing the `Cdt` discipline to use a string comparison function that ignores cases.

A single program was written with switches to select different versions of the token counting application. The program was run with five different input files:

- *ps*: PostScript source of a technical paper,
- *src*: an archive of C source code,

- *kjv*: a version of the King James bible,
- *city*: a database mapping cities to area codes.
- *host*: a database mapping IP addresses to machine hosts, and

File	Size	Tokens	Sensitive	Insensitive
<i>ps</i>	1,989,260	335,997	11,912	11,890
<i>src</i>	1,169,920	149,886	27,932	27,330
<i>kjv</i>	4,441,851	822,587	33,916	32,574
<i>city</i>	1,349,537	81,206	69,610	69,610
<i>host</i>	2,722,962	449,554	102,566	102,538

Table 1: Statistics of input files

Table 1 summarizes the file statistics, sizes in bytes, total numbers of tokens, and numbers of distinct tokens when cases are sensitive and when cases are insensitive. For example, *ps* and *src* contain many tokens but only a small number of them are distinct. This is typical of source code data. On the other hand, *city* and *host* are databases so their percentages of distinct tokens are much higher. The prose in *kjv* apparently has much redundancy.

File	tk		tk-v	
	Cpu+Sys	Space	Cpu+Sys	Space
<i>ps</i>	1.64	1,146,880	1.58	884,736
<i>src</i>	1.60	1,966,080	1.44	1,310,720
<i>kjv</i>	5.59	2,080,768	5.14	1,277,952
<i>city</i>	2.51	4,931,584	2.19	3,276,800
<i>host</i>	3.95	7,045,120	3.36	4,603,904

Table 2: Time and space comparisons for tk and tk-v

Table 2 compares tk and tk-v with both CPU+System times in seconds and space measurements in bytes. The experiment was done on an SGI-MIPS3 machine running the Irix6.2 operating system. Each time measurement was the median of five values obtained in consecutive runs on a lightly loaded machine. Space was measured by running `sbrk(0)` at the start and end of the `main()` function and computing the difference. Version tk-v which used a Vmalloc region for memory allocation clearly outperformed version tk. In particular, up to 38% of space was saved in processing the files *kjv* and *host*, confirming that avoiding the Malloc’s administrative space overhead was worthwhile.

Table 3 compares time performances among tk, tk-1 and tk-c. The results show that adding an Sfi discipline to translate upper case characters to lower case incurs little cost. On the other hand, changing the comparator in the Cdt discipline to perform case-insensitive comparisons can consume up to 25% more time as in the case of the input file *host*. This confirms the intuition that during dictionary construction many comparisons may be performed; therefore, doing case translation in string comparison is just a bad idea.

File	tk	tk-1	tk-c
<i>ps</i>	1.64	1.75	1.98
<i>src</i>	1.60	1.66	1.99
<i>kjv</i>	5.59	5.82	7.31
<i>city</i>	2.51	2.63	3.11
<i>host</i>	3.95	4.17	5.02

Table 3: Time comparisons for tk, tk-1 and tk-c

The experiment shows that matching disciplines and methods to usage contexts improves performance significantly. Examining the application code reveals the following observations:

- The main token processing code on Lines 12-20 of Figure 4 stays unchanged across the different versions of the token counting application. All the code tuning was done via changing discipline functions. This is nice because it shows that the library interfaces enable a high degree of modularity in the application architecture. Being able to protect high level application logic from tuning of low level resource manipulation code is a good step toward achieving effective code maintenance.
- Performance tuning may involve multiple libraries, for example, when tk-v changes a Cdt discipline so that memory can be allocated with a suitable Vmalloc method. This type of code tuning demands that the libraries expose their resource acquisition interfaces and make available suitable computing techniques in general and uniform ways. Discipline and method libraries do this naturally while traditional libraries such as Malloc and Stdio do not lend themselves well to this sort of optimization.

DESIGNING A LIBRARY

A resource-oriented approach

The resource management methods implemented by a library encapsulate efficient algorithms and/or important resource usage scenarios. Their utility is maximized when they can be applied against a wide variety of resources. Thus, the key to a good library design is to select the right set of methods and to define wisely the discipline type so that it appears uniformly across methods. Toward this end, we analyze library resources based on the below points (not necessarily in the order presented):

- *Defining resource operations and method type*: Investigate operations on resources at a general level regardless of how they may be implemented. This step results in: (i) a concise collection of resource operations and (ii) if multiple methods are necessary, a method type that parameterizes the abstract operations.

- *Defining the method collection*: Investigate usage scenarios to identify important subcases and variations. Concurrently, investigate suitable data structures and algorithms. The result of this step is a collection of methods.
- *Defining a discipline type*: Analyze the resources to be managed by a library to arrive at a general description of how they are acquired and released and how their key features are accessed. This analysis should reveal and include any dependencies on external functions and data. The result of this step is a definition of a discipline type. As mentioned, if multiple methods are to be made available, the discipline type must define resources in a way that appears uniformly across methods.
- *Defining library events*: Examine the exceptional events that may occur in manipulating resources and define when they should be raised. Raising an event means calling an application-provided function with the event type and any associated data in the context of the event. Such an event-handling function is a standard part of the discipline type.
- *Defining default disciplines*: Certain types of resources are standardly used (e.g., heap memory in Malloc or Vmalloc). In those cases, it is beneficial to applications if the libraries provide default disciplines defining such resources.

Applying the methodology

To see how the resource-oriented analysis approach may work, we shall use Cdt as an example. Regardless of container data types, the basic operations on objects in a dictionary are: insertion, deletion, searching, and iteration. Aside from a few other simple support functions, the basic dictionary operations are as shown in the top right part of Figure 3.

The abstract dictionary operations have different concrete semantics depending on particular container data types. For example, inserting an object into an ordered set means putting it at the correct position in the object list while inserting an object into a stack simply means putting it at the top. Since multiple container data types are involved, we need to define a common method type, `Dtmethod_t`, each instance of which implements a particular container data type. Note that this does not necessarily mean a separate implementation per method. For example, the Cdt unordered set and multiset methods, `Dtset` and `Dtbag`, share the same implementation with the right semantics selected by method names.

Next we analyze how objects may be used in a particular dictionary. Both ordered and unordered set methods must compare objects but they need different minimal requirements: sameness testing for unordered sets and order testing for ordered sets. Since objects must appear uniformly across methods, we specify a three-value discipline comparison function `Dtdisc_t.comparf()` whose return values are significant only in their signs: (1) negative for the first object being smaller than the second object, (2) zero for equal objects, and (3) positive for the first object being larger than the second object. In this way, unordered set methods can use non-zero return values to tell that objects differ while ordered set methods would use all three possibilities. This uniform object treatment increases programming flexibility by allowing dynamic matching of methods to usage contexts.

We would like to support multiple semantics for the same objects. For example, an object may need to be treated differently depending on usage contexts as shown in Figure 13. This leads to the inclusion of the discipline fields `DtDiscrT.key` and `DtDiscrT.size` to tell where the key is embedded inside an object and the size and type of the key. Dynamically definable keys means that the same interface can support both set-like and map-like dictionaries. Except for certain syntactic sugar, this eliminates the needs for separate set and map container types as provided, for example, in the Standard Template Libraries [4].

The above set-like vs. map-like discussion hints that the set of methods can be reduced by defining suitable object characteristics. Therefore, one may ask if ordered and unordered set methods could also be unified by adding object ordering as a discipline field. Cdt keeps these methods separate because they have distinct operational and performance characteristics and require different support container data structures (hash tables for unordered sets and binary trees for ordered sets). Further, many applications do not need both types of methods at the same time. Thus, keeping them separate not only makes clear the concepts but also reduces application code size on systems that use static linkage because only the required methods need be linked into an application.

DISCUSSION

Reusable libraries are essential assets for software construction. Much work has been done in increasing their usability. Below we discuss problem areas and compare and contrast solutions by the discipline and method libraries and other approaches.

A common fault in library design is to tie algorithms and interfaces so closely that similar operations appear completely different. Languages such as C++ [23] and Eiffel [24] provide constructs such as templates, classes, and inheritance that can be used to unify multiple implementations under some general interfaces. In particular, the *algorithm-oriented* [25] approach used in the Scalable Software Libraries [26] and the Standard Template Libraries [4] aims at analyzing generic algorithms and data structures, deciding the types and access operations needed for efficient execution, and encoding such types and access operations in macros or templates to be combined with actual object types in code generation. Although this approach is general and can generate efficient code, it is surprisingly often neither sufficiently flexible nor maximally efficient. For example, even though different STL container templates did achieve a high degree of uniformity, they still require different object specifications so that an ordered map cannot be dynamically changed to an unordered one and vice versa. By contrast, examples with the Cdt library showed how the library interface has been structured into general components to enable both flexible software construction and optimizing application performance.

Design patterns [27] are ways to institute architectural reuse. Each design pattern documents a common software problem and architectural guidances to solving it. For example, software modules often have conflicting interface requirements, so a common pattern in building integrated applications is to write adapters for different module interfaces. Patterns are similar to disciplines and methods in that they are architectural elements whose instantiation in concrete software requires

confinement to specific domains. However, patterns tend to aim at abstracting the mechanics of solving certain problem types while the discipline and method architecture aims at standardizing certain resource and operation interfaces in libraries to make such libraries both more general and more concrete. Of course, in a way, one may think of the discipline and method architecture in which handles, operations, disciplines and methods have the stylized forms shown throughout the paper as a particular design pattern for reusable libraries.

The need for documenting resource requirements in software has long been recognized in the software engineering community. In particular, Parnas and Clements [28] in 1986 discussed, among other things, the need to document input/output requirements and exception handling. However, relative little has been done since then in devising general formalisms to make explicit interfaces to resource definition and acquisition. In fact, traditional libraries often focus only on the operations and method interfaces as primary features and simply hardwire minimal resource requirements into their implementation. For example, we have talked about how the popular Malloc library provides memory allocation functions yet never says where raw memory comes from. Without explicit specifications of resources and their acquisition, it can be difficult to compose multiple libraries in application code. The mentioned adapter design pattern [27, 29] is a response to this lack of resource specification in reusable modules. However, adaptation can cost both in performance and, more importantly, in extra code design effort during application construction. The idea of using disciplines as explicit interface structures to define resource definition and acquisition is a step forward in enhancing the joint reuse of multiple libraries.

Exception handling is another area that traditional libraries pay little attention to. Most libraries, including those standardized by various standard bodies such as ANSI-C and POSIX, simply define broad categories of exceptions (e.g., out of memory or end of file) without saying how they should or could be handled. This leads to defensive programming in application code [17] with undesirable consequences in both awkward coding and poor performance. Libraries based on languages with built-in exception handling such as in C++ or Eiffel often delegate that to the language level. Our paradigm of Return-Proceed-Retry as a way to handle exceptional events has proved to be simple for applications to use yet sufficiently general to capture most application needs.

There are a number of further benefits in designing libraries based on the discipline and method architecture and the accompanying resource-oriented analysis approach. We discuss these next:

- *Generality:* A resource-oriented analysis can lead to a surprising unification of techniques from different domains. An example is the way that Vmalloc unifies interfaces to allocate, debug, and profile memory, activities usually accomplished in separate tools. In our ongoing work on a data compression library,[30, 31] a similar analysis led to the inclusion of other methods for data transformations including encryption. Such a unification is not readily transparent from an algorithm-oriented point of view because algorithms for compression and encryption are naturally different and studied in separate domains. However, the unification makes sense from a resource-oriented point of view because both compression and encryption are just ways to transform data.

- *Adaptability*: A desirable feature for a reusable software is the ability to adapt to new resource types. Defining resource requirements explicitly in discipline structures allows library users to take advantage of resource advances without changing library code. For example, only in the past few years that the Unix `mmap()` facility for memory mapping has become popular as a way to implement shared and persistent memory. Vmalloc users can write disciplines based on `mmap()` and reuse all allocation algorithms embedded in Vmalloc methods.
- *Portability*: In a similar vein, identifying external dependencies and abstracting them in disciplines makes it easy to port a library to a new environment. All the libraries mentioned in this paper have been ported to virtually all known popular platforms including various Unix and Windows flavors.
- *Concreteness*: The use of run-time disciplines to define resource types means that operations and resource types are not statically bound and library code images can remain concrete and dynamically linkable. This means, of course, that static type checking and any opportunity for code inlining are lost. Static type checking is relatively unimportant in libraries like Sflo and Vmalloc, which deal with simple resources such as streams of bytes or memory segments. Code inlining, even when applicable, tend to have little benefit since the function call cost of resource acquisition is often trivial comparing to the work in resource management. Even for the Cdt library, where the function call cost may matter because of frequent object comparisons, applications often deal with complex objects and good engineering practices would dictate writing functions anyway.
- *Bidirectional reuse*: Libraries such as Sflo and Vmalloc can be used to write other reusable code on top of them. On the other hand, disciplines allow and encourage construction of code reusable underneath them. The Sflo library includes standard disciplines for tasks ranging from making an unseekable stream seekable to automatic file decompression. Likewise, Vmalloc users have written disciplines for shared and persistent memories. The Cdt reference ^[12] shows how such Vmalloc disciplines can be used to build shared and/or persistent dictionaries. This composition of libraries both allows sharing and persistence of data structures to be treated properly as characteristics of the inherited memory models and eliminates the need to provide parallel versions of Cdt for different memory types.
- *Evolution*: Advances in computing research continue to open new ways to manage resources efficiently. By specifying library operations at an abstract level, parameterizable by methods, a library can take advantage of different algorithms by implementing them as separate methods. This is especially attractive for library users because experimentation with different algorithms can be done without too much disturbance in application code.
- *Efficiency*: Resource usage tends to be modal and it is hard to devise efficient algorithms that can adapt smoothly with diverse modes. For example, no general purpose allocators can come close to the performance of special purpose methods like `Vmpool` and `Vmlast` in Vmalloc when they are properly used. Discipline and method libraries make tools like these available in a form that is easily mixed and matched by applications to arrive at efficient solutions to particular resource usage modes.

- *Efficacy*: Modal resource usages extend beyond performance issues and include efficacy in application construction and maintenance. Vmalloc shows how methods such as `Vmdebug` and `Vmprofile` for memory debugging and profiling can be used to make the same abstract allocation operations perform computations that normally require separate tools [22]. Vmalloc uses such facilities to provide a Malloc interface with dynamic method setting. In this way, applications can both run fast normally and retain the ability to diagnose memory errors without recompilation, thus, improving the ability to maintain and service production code.
- *Scalability*: Biggerstaff [32] and Batory et al. [26] discussed limitations in current library construction approaches due to the combinatorial explosion of feature interactions. In particular, Biggerstaff discussed the difficulty in writing reusable software that simultaneously achieves vertical scaling (i.e., highly integrated libraries in narrow domains), horizontal scaling (i.e., small, general and widely applicable components), and performance. The discipline and method architecture is a step forward in alleviating this quandary. Factoring a library by abstract operations, disciplines, and methods both increases vertical and horizontal scaling and enhances performance. This is because efficient algorithms in the form of methods can be integrated under a general uniform interface based on suitable resource abstractions in the form of disciplines. This factorization is critical in controlling code size. For example, the `Cdt` library, which provides a full complement of container data types, is implemented in less than 1,800 lines of amply commented C code.

CONCLUSION

This paper presented a discipline and method library architecture and a resource-oriented approach to analyzing and designing library interfaces. Several libraries have been built using these techniques. Examples given throughout the paper show that the libraries built this way are general and efficient, and their usage enhances modularity and maintainability in application code.

Even with the guidance of the design methodology, the effectiveness of our libraries did not come easy. From a library designer's perspective, perhaps the hardest part in engineering a library was to come up with the right discipline interface resilient enough to deal with the variety of resource types as well as the requirements of different underlying computational data structures and algorithms. In many cases, the complexity of this task was compounded since the underlying algorithms and data structures were developed along with the engineering of their interfaces. The ability to design good interfaces only came with practice and experience. A consolation was that such exercises did get easier over time and they helped bring forward the awareness of any low-level dependencies so that, in the end, the libraries did achieve greater generality and portability than if done otherwise.

From a library user's perspective, the ability to mix and match resource types and management methods enabled faster application construction as well as more efficient applications. Defining or selecting the right disciplines was a good part of the effort. Here, the inherent modularity of disciplines made it easy to add functionality and tune performance without too much code rewriting. For libraries such as Vmalloc and Sflo, the standardly provided disciplines were adequate for most

tasks. When new disciplines must be written, it was satisfying that, by their nature, such disciplines tended to be reusable or could be made reusable with little extra effort. In fact, this was how many of the standard disciplines in our libraries came about - they were built along with our applications.

In ending, we note that architectures and design methodologies are just structured thought processes. As such, their usefulness can only be shown via examples of their applicability. Beyond Sflo, Cdt and Vmalloc, other discipline and method libraries have been built for sorting, data differencing and data transformation. Some of these libraries are used by programmers world-wide. Their success indicates that discipline and method libraries are on the right path.

Acknowledgement

Many colleagues, including Glenn Fowler, David Korn, and Stephen North, helped solidifying the ideas. Disciplines were first used in the Sflo library, a joint work with David Korn and Glenn Fowler.

Code availability

Sflo, Cdt, Vmalloc and other libraries mentioned here are part of an ongoing effort to build a repository of highly portable and reusable software tools [9, 33]. Much of the code is available at: <http://www.research.att.com/sw/tools>.

References

- [1] ANSI. *American National Standard for Information Systems - Programming Language - C*. American National Standards Institute, 1990.
- [2] D. Lea. lib++, the GNU C++ library. In *Proceedings of the USENIX C++ Conference*, 1988.
- [3] Andrew R. Koenig. Associative Arrays in C++. In *Proceedings of 1988 Summer USENIX Conference*, pages 173–186, 1988.
- [4] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1995.
- [5] Unix System Laboratories. *USL C++ Standard Components Programmer's Reference*. AT&T and Unix System Laboratories, Inc., 1990.
- [6] Kiem-Phong Vo. Vmalloc: A General and Efficient Memory Allocator. *Software—Practice and Experience*, 26:1–18, 1996.
- [7] B. Zorn and D. Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Notices*, 27(12):71–80, 1992.
- [8] David R. Hanson. *C Interfaces and Implementations*. Addison-Wesley, 1996.
- [9] Edited by B. Krisnamurthy. *Practical Reusable Unix Software*. John Wiley & Sons, Inc., 1995.
- [10] Kiem-Phong Vo. Concrete software libraries. In *Proceedings of the 1998 Extended Memory Algorithm Workshop*. DIMACS, 1998.
- [11] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. Sflo: A Buffered I/O Library. *Software—Practice and Experience*, Submitted for publication, 1998.

- [12] Kiem-Phong Vo. Cdt: A Container Data Type Library. *Software—Practice and Experience*, 27:1177–1197, 1997.
- [13] David G. Korn. ksh: An Extensible High Level Language. In *Proceedings of the Usenix VHLL Conference*, 1994.
- [14] David G. Korn. Porting UNIX to Windows NT. In *Proceedings of the 1997 Usenix Conference*. USENIX, 1997.
- [15] Larry Wall and Randal Schwartz. *Perl*. O’Reilly & Associates, 1990.
- [16] Robert Sedgewick. *Algorithms, 2nd Edition*. Addison-Wesley, 1988.
- [17] Kiem-Phong Vo, P.Y. Chung, Y. Huang, and Y.-M. Wang. Xept: A Software Instrumentation Method for Exception Handling. In *Proceedings of the 8th International Symposium on Software Reliability Engineering*. IEEE Computer Society Press, 1997.
- [18] Bjarne Stroustrup. Making a vector Fit for a Standard. In *The C++ Report*, 1994.
- [19] D.E. Knuth. *The Art of Computer Programming, Volume 1*. Addison-Wesley, 1968.
- [20] J.M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20:242–244, 1975.
- [21] J.E. Shore. On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies. *Communications of the ACM*, 18:433–440, 1975.
- [22] R. Hastings and R. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter 1992 Usenix Conference*, pages 125–136. USENIX Association, 1992.
- [23] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
- [24] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall.
- [25] D.R. Musser and A.A. Stepanov. Algorithm-oriented generic libraries. *Software—Practice and Experience*, 24(7):623–642, 1994.
- [26] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable Software Libraries. In *Proceedings of the Symposium on the Foundations of Software Engineering*, 1992.
- [27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.
- [28] David L. Parnas and Paul C. Clements. A Rational Design Process: How and Why to Fake It. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, 1986.
- [29] P.C. Clements, D.L. Parnas, and D. Weiss. Negotiated Interfaces for Software Reuse. *IEEE Trans. on Soft. Eng.*, 18(7):646–653, 1992.
- [30] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. An Empirical Study of Delta Algorithms. In *IEEE Software Configuration and Maintenance Workshop*, 1996.
- [31] David G. Korn and Kiem-Phong Vo. Vdelta: Differencing and Compression. In *Practical Reusable Unix Software*, Editor B. Krishnamurthy. John Wiley & Sons, Inc., 1995.
- [32] T.J. Biggerstaff. The Library Scaling Problem and the Limits of Concrete Component Reuse. In *Proceedings of the 3rd International Conference on Software Reuse*, 1994.
- [33] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. Principles for Writing Reusable Library. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability*, pages 150–160. ACM Press, 1995.