To appear in the Proceedings of the 5th European Software Engineering Conference, September 25-28, Sitges, Spain.

Component Programming— a fresh look at software components

Mehdi Jazayeri

Distributed Systems Department
Technical University of Vienna
A-1040 Vienna, AUSTRIA
jazayeri@tuwien.ac.at
http://www.infosys.tuwien.ac.at/

Abstract. All engineering disciplines rely on standard components to design and build artifacts. The key technical challenge in software engineering is to enable the adoption of such a model to the development of software. The transformation from line-by-line development to component-based development will address many of the industry's persistent productivity and quality problems. Indeed, component-based software development has been a long-standing dream of the software industry, prompting a search for both technical and nontechnical solutions. A successful approach to component-based development requires a comprehensive solution that draws on advances in programming languages, programming paradigms, algorithm analysis, and software design. This paper presents an approach based on the C++ Standard Template Library. More than a traditional library, STL embodies a concrete approach to software design based on a well-defined taxonomy and theory of software components. I present the fundamental contributions of STL to a paradigm of *component programming*—a component-based software development paradigm in which there is a clear separation between component development and application development. I motivate component programming, give the requirements for components and catalogs, and give an example of component programming applied to the standard Keyword in Context (KWIC) problem. I then summarize the implications of component programming for the software industry and for software engineering education.

Keywords and phrases: Software components, genericity, component programming, programming paradigms

1 Introduction

A fundamental weakness of software engineering is the lack of a scientific foundation. There are no laws that govern the decomposition of a software design into a set of components or the selection of a set of components to implement a given design. The practical implication of this problem is that we still develop software one line at a time. Indeed, the essential technical challenge in software engineering is to transform the industry from relying on line-based development to using component-based development. All other engineering disciplines use standard components. We must adopt a similar model in software engineering to enable an engineering approach in software development.

I use the term *component programming* to refer to a software development paradigm based strictly on the use of standard software components. Various programming paradigms such as object-oriented programming and functional programming have provided partial solutions to this problem, although they neither insist on the use of *standard* components nor on the *exclusive* use of components. The ultimate goal of component programming is to develop a scientific foundation for 1) the design of software based on software components and 2) the discovery and development of those components. As I shall argue later, component programming depends on advances in programming language technology, algorithms and data structures, and programming methodology. At the present time, enough progress has been made in all these areas to make component programming feasible. This paper presents the practical requirements for component programming, speculates on why it has not worked so far, and gives examples based on the C++ Standard Template Library (STL) to show the promise of component programming. It concludes with a summary of what needs to be done to accomplish the vision of component programming.

2 Requirements on components and catalogs

Component programming postulates that software must be developed from components found in standard software catalogs. This by itself is not a new goal. Indeed, it was the vision presented by McIlroy at the NATO conference in 1968[6]. Yet that vision has not been realized to date because we have not concentrated on the essential requirements that components must satisfy. To enable software development primarily from component catalogs, components and catalogs must meet the following fundamental requirements: 1) components in a catalog must form a systematic taxonomy both to guide the design of an application and to enable the search for, and the selection of, components; 2) components should be *generic* so that they have wide applicability; 3) components should be *efficient* so that they meet the demands of real applications; and 4) catalogs must be comprehensive, that is, they must cover a significant portion, if not all, of the taxonomy mentioned in requirement 1. In this section I will explain each of these requirements in general with appropriate references to existing software libraries and how they succeed or fail to meet some of these requirements. The following section will discuss an example catalog that meets all four requirements. By use of examples from this catalog, I show how McIlroy's vision can be realized.

Components in a catalog must form a systematic taxonomy. Many existing libraries are collections of loosely-related, or worse, unrelated, components. The successful "component" catalogs have been mathematical libraries and standard libraries supported for particular languages, for example, the *stdio* C library for buffered input-output. Such libraries contain closely-related components that cover a well-defined domain of functionality. A user knows the functionality provided and the cost of using the components. The components are designed and implemented to support the advertised functionality as efficiently as possible. Few people would venture to write their own buffered i/o routines or linear algebra functions.

The components in a buffered i/o or linear algebra library are related intrinsically. This relationship is not abstract or artificial in any way. We do not need to relate the components at some meta-level, for example with an inheritance relationship, to make them understandable to users. What matters is the contents of the catalog. In successful catalogs, the components support a related set of concepts. If these concepts are understood and valued by users, and the components are implemented well, the catalog will be useful to users.

A systematically developed set of catalogs, each supporting a related set of concepts, is the first step towards a component-based software development paradigm. If the concepts are chosen right, they provide the vocabulary used by the software designer. In mathematical libraries, the concepts are well-known from mathematics, e.g. matrix computations. Buffered i/o and random access i/o are examples closer to computer science. This requirement shows why it is difficult to develop useful catalogs for arbitrary domains: we need to know the concepts first. It shows why the so-called organization-wide reusable library attempts have failed. A collection of modules randomly contributed by—albeit well-meaning—programmers does not produce a catalog of tightly-related components. To be useful as a design tool, the designer must find it worthwhile to spend the time to study the concepts supported by the catalog. The designer must foresee opportunities for repeated uses of the catalog. Otherwise, the effort of studying the catalog is an overhead that may not be recovered.

In short, a systematic taxonomy makes it possible for the catalog designer to decide what components must be included in the catalog and it tells the catalog user whether the catalog may contain the components required by the user. Without a taxonomy, neither the developer nor the user can be sure.

Components should be as generic as possible. The basic motivation for component programming is to reduce the number of lines of code that we have to write, to think about, and to maintain for each new project or application. Fewer lines of code means more productivity in initial development and less effort during maintenance. The same argument holds for the development of component catalogs. A catalog that has fewer components but supports the same functionality is better than one that has more components. The goal of minimality is even more important in the case of standard components because they are used repeatedly. Fewer components makes it easier for users to find what they need and it makes it easier for component developers to devote the effort needed to perfect the components.

To make it possible to have fewer components means that each component must be usable in more contexts, that is, it must make minimal assumptions about the context in which it is used. Generic components can be used to meet this requirement. Generic data structures such as stacks or lists of arbitrary data types are available in various libraries. Such genericity allows us to write one stack component rather than n stacks, one for each supported type. These kinds of components may be written in languages that support a generic facility such as Ada, Eiffel, or C++. Indeed, most C++ libraries are now template libraries. C++ templates can be used to write not only generic data structures but also generic functions and procedures.

But the notion of genericity can be pushed much further: algorithms can be written generically to make minimal assumptions about the structures on which they operate. For example, the component *find* in the STL library searches for an element in a sequence. The same component can be used to search arrays or different kinds of lists. As long as the implementation of the sequence provides a way to step from one element of the sequence to the next, and to examine each element, we can use the component *find*. I will examine how this is done in the next section. The point here is that components must be made as generic as possible to make them as universally usable as possible. And genericity may be applied not only to data structures but also to algorithms and, in general, to any component type.

Writing generic components is not straightforward. Each generic component captures the essential properties of a large number of specific contexts in which it is used. Identifying the contexts from which a generalization can be made, and inventing a mechanism that may be used in all those contexts requires a taxonomy of concepts and careful interfaces to those concepts. The concepts point out a proper modularization of the software components. The STL sequence algorithms, including *find*, use the concept of an *iterator*, which is a generalization of the familiar *pointer.* An iterator is used to traverse and examine the contents of a data structure. Algorithms take iterators as parameters. This way, an algorithm can be more generic because it does not depend on the structure of a data structure. The algorithm's assumptions about the data structure are captured by the iterator, which provides the means for accessing the elements of the data structure. An algorithm can therefore apply to a family of data structures that support a particular type of iterator. At the same time, many data structure implementations can be considered "equivalent" as long as they support the same iterator categories.

STL contains a comprehensive set of algorithms for several types of sequences and associative containers. A sequence is represented by two iterators, one that points to the first element of the sequence and one that points to the position past the last element of the sequence. This is the interface used by most STL algorithms, including *find*. This interface allows the algorithm to be independent of both the types of elements in the sequence and the structure of the sequence.

Components should be as efficient as possible. Genericity and efficiency appear to be contradictory requirements. It is rather easy to write generic routines if we don't care about efficiency. We could, for example, encode the type information in each data structure and have each algorithm check the code and do the appropriate thing based on the type. Such an approach is neither maintainable nor efficient. In practice, a programmer will not use standard components unless they are as efficient as those the programmer can produce. Users must be able to rely on performance guarantees from standards components. Typically, the components will be used to build even more components. The only way to be able to predict the performance of these higher-level components is if we are guaranteed the performance of the used components.

A unique feature of the STL library is that for each algorithmic component, its run-time costs are specified. For example, the generic *find* component is guaranteed to run in linear time for unsorted sequences, which is the best one can do. In turn, the *find* component makes an assumption about the data structure on which it works: that stepping from one element of the sequence to the next takes a fixed cost. This requirement is met by all the data structures provided by STL. But it is a requirement on the users as well: if you are going to build your own data structure, then you are required to provide a fixed-cost next-element operator. Such a set of requirements—*laws*—are necessary for a component-based paradigm. If we are to rely on components as the primary means to building applications, the semantics of a component must include not only its functional behavior but also its performance. For example, Common Lisp, which originated many of the early ideas in generic programming, has a generic function *elt sequence index*, which returns an element in position *index* in *sequence*. The sequence may be either a vector or a list. While this generic function makes it easier from the programming point of view to write generic algorithms that work on both arrays and lists, it can be disastrous in terms of runtime efficiency, because *elt* will take linear time for lists. A sort routine written with the help of *elt* will certainly work correctly for both arrays and lists, but it will run much slower for lists. In general, we may replace one component with another only if it has the same interface and the same performance characteristics.

The efficiency requirement is quite serious, even if not academically popular: much effort in software development is involved with finding faster ways of providing the same functionality. Indeed, delivering a given functionality is not usually difficult. The delay in many projects is due to trying to deliver the functionality at acceptable performance. Often, delivery of a software product is delayed while special performance teams solve the performance “problems.” It is during this performance improvement phase of a project that many of the other software goals such as maintainability are compromised. The overriding concern in this phase is performance and nothing is ruled out, not even rewriting pieces of code in assembly language. Unless standard components are as efficient as possible—both at the algorithm level and at the coding level—they will be discarded when it is time to address the performance problems.

STL components not only provide a uniform interface for both built-in and user-defined structures, they guarantee that if the components are used with built-in types, they do not incur any execution overhead, such as extra procedure calls or indirect references. This is a problem in many existing libraries.

Writing generic components is hard but writing *efficient* generic components is even harder! Component production is a specialized activity. It requires special concern for abstraction and generalization, and study of data structures and algorithms. Components cannot be expected to be produced as a by-product of application development. In fact, the differences between component development and application development appear to be similar to the differences between chip design and circuit design.

Catalogs must be comprehensive. Our first requirement was that a catalog must cover a taxonomy of concepts in a given domain. For the use of components to become pervasive, not only each catalog must be comprehensive, but there must be many comprehensive catalogs. It must be worthwhile for the programmer to study a particular catalog. If the catalog is not comprehensive, the programmer will not find the needed components and therefore will stop using it. If there are not enough catalogs to cover most of an application, then using catalogs will not become a routine activity because line-by-line development will still be required.

Thus, as is usual in any design, the design of catalogs requires a trade-off between not including enough and including too much. The tradeoff must be resolved by appealing to the concept taxonomy.

3 STL: A model component catalog

The Standard Template Library is a library of templates adopted recently by ANSI as a standard for C++. It meets all the requirements stated in the previous section. In this section, I examine the key characteristics of this library, and discuss the unique features that make it promising as a foundation for component-based software engineering.

3.1 Kinds of components

STL contains five kinds of components: algorithms, containers, iterators, function objects, and adaptors. Algorithms and containers provide many of the usual algorithms and data structures. Iterators provide different ways of traversing through containers. Function objects are a mechanism for packaging a function so that it can be used by other components. Adaptors are mechanisms for modifying the interface of a component. To achieve maximum genericity, the library separates algorithms and data structures and uses iterators as an intermediary. This interesting modularization allows the algorithms to encapsulate computational procedures, the containers to encapsulate memory management policies, and the iterators to encapsulate container traversal policies. Thus, algorithms make assumptions about the capabilities of iterators, for example, whether an iterator is able to move forward or backward through a container; algorithms make no assumptions about a container's memory organization. All containers that support a forward iterator are operable by an algorithm that requires a forward iterator. Iterators can be thought of as abstracting the generic properties of containers of interest to certain algorithms. In general, containers are classified by the type of iterators they are able to support and algorithms are classified by the type of iterators they require. Bidirectional iterators support traversal in both directions and random access iterators support access to an arbitrary element in a container in constant time. Thus, if an algorithm requires a random access iterator, it cannot work on a standard list. Obviously, it is possible to write a routine to simulate random access to arbitrary elements of a list by repeatedly stepping through the list. But this will violate the complexity assumptions required by the algorithms. Two other more limited iterators, input iterators and output iterators, are used to include input and output in the same framework. These iterators extend the applicability of algorithms to input and output files and devices.

The notion of an iterator is the most important building block of STL. A forward iterator supports the two operations of dereferencing (* in C++) and step (++ in C++), each in constant time. Thus, a C pointer may be used as an iterator (of an array). This way, STL algorithms will work on C++ built-in pointers as well as on user-defined iterators, such as the STL-defined iterators for list containers. Treating built-in and user-defined types uniformly is a key design principle of STL. It implies that STL algorithms may be used with user containers, and STL containers may be used with user algorithms. More importantly, the user does not have to sacrifice efficiency to use the STL components.

A pair of iterators is used to represent a range of elements in a container. Most algorithms use this representation for a sequence. By convention, the first iterator points to the first element in the range and the second iterator points to one element beyond the last element. In C++, this address is always guaranteed to be defined. This representation of a range allows easy representation for an empty range—two iterators are equal—and a natural structure for algorithms—loop as long as the first iterator has not reached the second. Together with a taxonomy of iterators, this representation is the basis for the interface to sequences. Such a "published" interface enables others to produce other algorithm and container components.

An adapter is used to coerce a component into providing a different interface. For example, instead of implementing stacks and queues as new containers, stack and queue templates are provided that adapt the interface of other containers such as STL vectors and lists. The requirements that state which containers are convertible are given precisely—in terms of supported iterators—and are checked at compile-time. Using adapters reduces the amount of code that needs to be written for the library and is a useful technique for the programmer. It is a way to avoid overcrowding a component catalog with many different types of components that are similar and are derivable from each other. It is also a design technique for the programmer to reduce the number of lines to be written and maintained. Proper iterators do not introduce runtime overhead because the "adaption" is done at compile-time. Adapters may be used for components other than containers as well. For example, a reverse iterator may be manufactured out of a bidirectional iterator to allow a container to be traversed in reverse order.

3.2 Generic examples

In this section, I use some small examples to show the generality and versatility of the STL components. In the next subsection, I will give a solution to the standard KWIC example in STL to allow comparison with other approaches.

```
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last,  const T& value) {
   while (first != last && *first != value) first++;
   return first;
}
```

Fig. 1. The find component from STL

Figure 1 shows the code for the *find* component in STL. This component is an algorithm that searches a sequence linearly for a desired value. It is a good example of how components can be written that are generic, powerful, and efficient. The code is simple: the range is represented by *first* and *last* (*first* and *last* are passed by value—they are local variables in *find* and are initialized from the parameter values); *first* is repeatedly incremented until it is either equal to *last*, or it points to something equal to *value*. Finally, *first* is returned, either pointing to the desired value or equal to *last*. The following principles are used to enable so few lines to accomplish so much:

1. C++ templates are used to make the component generic with respect to the data type of the element being sought. This kind of genericity is now common in C++ template libraries. In this example, the class T captures the requirements on the *value* we are looking for: it may be a built-in or user-defined type and we need only read access to it. Perhaps most surprising is the absence of any specific type declarations in this algorithm. The two template parameters, *InputIterator* and *T*, are declared to be class parameters. All this says is that they are types—either built-in or user-defined types. The requirements on these types are deduced by the compiler based on the operations in the algorithm.
2. The range interface, consisting of two iterators, is used to achieve genericity with respect to the structure of the container being searched. Without indicating what an iterator is, or what container we are searching, we are able to state that we are searching for something in a collection of things, sequentially, until the range is exhausted. It is the responsibility of the iterators to know how to step through the container. By distributing the responsibility this way, we have made it possible for the algorithm to state only the essence of linear search through a sequence. It is hard to imagine how *find* can be written in less code.
3. By convention, the class of iterators required for specifying the range is stated here as InputIterator, which is the least restrictive category of iterator. This means that the sequence may even be read from an input device. All that is required of the iterator can be seen in the code: comparison, dereference, and increment. To guarantee the complexity of the algorithm, it is also required that these three operations take constant time.
4. Because the code is a template, it will be compiled together with the user program, and may be expanded in-line, avoiding the overhead of a procedure call.

Let us say we have defined an array of integers in our program:

```
int a[100];
```

We can check for some value, say 5, in the array:

```
f = find(&a[0], &a[100], 5);
if (f == &a[100]) //not found...
```

Or we can look only through part of the array:

```
f = find(&a[2], &a[50], 5);
```

But *find* is generic. If we have written a list container, or use the one from STL, containing elements of a previously-defined *student_t* type, we can look for a particular student, say *georg*:

```
list<student_t> students;
...insertion into students...
f = find(students.begin(), students.end(), georg);
```

Each container is required to provide the two iterators *begin* and *end* that point to the beginning and one position beyond the end of the container.

We can also use *find* to search a list being read from an input device. But first we must attach an iterator to an input stream. STL provides an istream_iterator component for this purpose. The code requires that an input operation for the student type is defined.

```
f = find(istream_iterator<student_t>(cin), istream_iterator<student_t>(), georg);
```

The *istream_iterator* constructs an iterator that works on an input stream. This allows all the algorithms to work with input streams. Input iterators are limited to providing access to successive elements of the stream. An *istream_iterator* without a parameter constructs an end-of-stream iterator object. Each ++ on an input iterator reads a value from the input stream for which it was constructed and stores the value. Every dereference of the iterator returns the value stored, as a constant value. No other operations are allowed on input iterators.

We can see the versatility of STL components from the *merge* component which merges two ordered input sequences into an ordered output sequence. Because it relies on an iterator interface, the same component is able to operate with any combination of containers. For example, we may merge a vector and a sequence read from input into an output list, or we may merge two lists and produce the result on an output stream.

The code for merge is shown in Figure 2. Again, we see that the two input sequences are represented by two iterators and the output sequence is represented by a single iterator. The code is compact and simple. It uses another STL component, *copy*, to copy the remainder of one of the sequences into the output after the other input sequence has been exhausted. Because *copy* returns a pointer beyond its resulting output sequence, *copy* can be combined easily with other components such as itself. We see an example of this in the last line of *merge*. Indeed, many STL components are built from other STL components, reinforcing the usability and efficiency requirements of STL components.

Here is a final example of the power of component composition. These statements sort two sequence and merge the results onto a third sequence:

```
sort (f1, l1);
sort (f2, l2);
```

```
template <class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result) {
  while (first1 != last1 && first2 != last2)
          if (*first2 < *first1)
              *result++ = *first2++;
          else
              *result++ = *first1++;
  return copy(first2, last2, copy(first1, last1, result));
}
```

Fig. 2. The Merge component from STL

```
merge (f1, l1, f2, l2, f3);
```

f1, l1, f2, l2, and *f3* are iterators into any combination of STL-defined or user-defined sequences, including input or output streams. The same *merge* component will work for all combinations. *Merge* requires that the elements in the containers being merged are comparable.

3.3 Software design with STL: a KWIC example

To demonstrate component-oriented software design and decomposition, in this section I give a component-based solution to a problem that has been used as a standard example in the software literature. In 1972, Parnas [10] showed a novel decomposition and modularization technique called information hiding. He demonstrated the benefits of the new modularization technique over traditional procedural modularizations with an example of a Keyword in Context program. I will show here one possible component programming approach to the KWIC example.

The KWIC index generation problem is defined informally as follows: The input consists of an ordered set of lines; a line is an ordered set of words; a word is an ordered set of characters. Each line may be circularly shifted repeatedly by moving the first word of the line to the end of the line. The output is the sorted listing of all the circular shifts of all the input lines.

We can solve this problem using the following components:

1. *titles*: a container that holds the input lines—this is a simple linear list of titles.
2. *KWICindex*: a container that contains the sorted list of permuted indexes. For this container, we will use a multimap of pairs of strings. The first element of a pair is the permuted title and the second is a pointer into the titles list, pointing to the original title whose permutation is the first element of the pair. A multimap is an STL container that keeps its elements sorted. At the time the multimap is created (instantiated), we must supply a comparison operator for ordering the container elements.

3. *CircularShift*: A function object that, when applied to a title, produces all the circular shifts of that title.
4. $<<$: Definition of an output operator for permuted lines. This operator can perform the necessary formatting.

```
#include <iostream.h>
#include <pair.h>
#include <list.h>
#include <multimap.h>
#include <algo.h>
#include <vector.h>

typedef vector<char> string;
typedef multimap<string, string *, less<string> > permutedTitles_t;
typedef list<string> Titles_t;
typedef pair<const string, string *> TitlesPair_t;

int main (int argc, char *argv[]) {
   Titles_t titles;
   permutedTitles_t KWICindex;

   copy(istream_iterator<string, ptrdiff_t>(cin), istream_iterator<string, ptrdiff_t>(),
      back_inserter(titles));

   for_each(titles.begin(), titles.end(), CircularShift(KWICindex));

   copy(KWICindex.begin(), KWICindex.end(),
      ostream_iterator<titlesPair_t>(cout));

   return 0;
}
```

Fig. 3. Main program for KWIC index program

The main program appears in Figure 3. The program `includes` the relevant STL files. In the subsequent programs, I will not show the `include` statements. The complete program appears in the appendix in one file. The processing part of the program consists of three lines: copying from an input file into the titles list; shifting the list and producing the KWICindex; and printing the results by copying the resulting index to output. In the first statement, we use *copy* to read the file into an in-memory list. We use the adapter *back_inserter* to modify the interface to the *titles* list so that assignments to the list will actually do an insert instead of the usual overwriting. This adapter allows *copy* and other algorithms to work both with targets that have memory pre-allocated and those that do not. The third statement also uses *copy*, this time to produce the results onto the output stream. Here, we use *ostream_iterator* adapter to get an output iterator that writes onto the output stream. The same copy operation is being used for moving data, either from input or to output. Each time, it uses the appropriate output operation associated with the relevant stream. The second statement uses the *for_each* algorithm of STL which applies a function object to every element in a range. In the function object *CircularShift,* we see the use of a component that encapsulates both an algorithm and its state. *CircularShift* forms all the circular shifts of its argument and inserts them into the multimap with which it was instantiated—*KWICindex*, in this case. The code for the operator<< and the function object *CircularShift* is shown in Figure 4.

The distinctive feature of a function object is that it defines an application operator (), which allows the object to be applied. Such an object is used in functional programming extensively and is known as a *closure*. A closure encapsulates a function together with a state. In the case of *CircularShift*, the state of the computation is kept in the multimap *index*, a private variable in *CircularShift*. C++ templates allow the use of this useful technique in a traditionally imperative setting.

CircularShift produces the circularly-shifted lines by iterating through a “virtual” container and inserting each circularly-shifted line into its *index*. The container is virtual because we have actually defined an iterator, *permute*, which reads the original input line, maintaining the state of iteration through the container. Only if the iterator is dereferenced, does it actually build and return the next circularly-shifted configuration. As with any other iterator, we have to define ++ and *. The code for permute is shown in Figure 5. The constructor for *permute* with a parameter sets up the initial configuration of a shifted line. A constructor with no parameters returns an iterator that matches the end of a shifted configuration. This is the mechanism used for terminating the iteration through the permutations. This component illustrates the use of iterators in developing new components.

The complete code for the program is given in Appendix I. Appendix II shows a sample input file and the output produced by our program. At this point, I summarize the essential characteristics of this solution.

First, the program is quite short, owing to our approach of using standard components. The brevity of the program enables us to reproduce the entire program here. It is important in software engineering to discuss and compare different approaches by referring to the resulting code. Without code, there is the danger of discussing only abstract ideas and glossing over important details. The program being short actually has a deeper implication: it shows that we have met our goal of changing our paradigm from line-by-line programming to component programming.

Second, the solution is modular, based on the kinds of components available in STL. The most far-reaching contribution of STL is its identification of the kinds of components needed for software construction. We have decomposed the KWIC problem into containers, algorithms, iterators, and function objects. We found some of the needed components in the catalog, and others we had to build ourselves. Modularization according to this classification of components seems natural and general.

Third, to reach the complete goal of component programming, there would have to be many catalogs. For example, we would need a catalog for text processing, file management, window management, and so on. If we had a text processing catalog, we may not have had to write any new code at all for this program because *CircularShift* would probably have been available.

```
class CircularShift {
public:
   CircularShift(permutedTitles_t& store) : index(store) {}

   void operator()(const string& str) {
      for(permute x(str); x != permute(); x++)
         index.insert(TitlesPair_t(*x, &str));
   }
private:
   permutedTitles_t& index;
};

inline ostream& operator<< (ostream& out, const TitlesPair_t& p)
{
   return out << p.first << ":\n\t " << (*(p.second)) << '\n';
}
```

Fig. 4. Code for function object CircularShift and <<

Fourth, this solution combines elements from imperative and functional programming together with abstract data types and information hiding. Just as the STL catalog contains a variety of different types of components, the design approach is also multi-paradigm, combining the salient features of the different approaches. In particular, containers provide *generic abstract data types* and function objects provide *closures*.

Each component in this solution encapsulates a particular design decision and therefore meets the goal of design-for-change. Furthermore, because we are using mainly standard components, it is feasible to try several different design alternatives without exorbitant implementation cost. For example, we can generate a permuted index first in an unsorted list and then sort it. We would use a list container and a sort algorithm from STL. We can then compare the performance of both solutions before selecting a final solution.

4 Skepticism about component programming

In the previous section, I have shown an example of component-based software design. I have tried to show the benefits to be gained from such an approach to software engineering. But because the vision of component programming is as old as the field of software engineering itself, there is some built-in skepticism about the probability of success of any new solution. In this section, I address two specific concerns that may arise for the reader in encountering component programming initially: one concern is specific to the approach

I have presented here—the apparent basic level of the components—and the other is a more general one about the feasibility of the whole approach.

Concern 1: The components such as I have described are "low-level"; instead, we need application-oriented large-grained components. There are two answers to this concern. First, the granularity of components is a tricky issue. The apparent simplicity of STL components is deceptive: The genericity of the components renders them quite general-purpose and powerful. They are appropriate for decomposing many common software problems. We can get larger components by combining such powerful components in different ways. For example, we can build a permuted index component from the KWIC program in the previous section. Unix has already demonstrated the usefulness of small tools as building blocks for more powerful, special-purpose, tools. The next step in component programming should be to develop other catalogs, dealing with different domains, such as window management.

But a deeper answer to the first concern is that to create a component-oriented approach, we must first start with the basic software components. More important than the components themselves is a science and theory of software components that enables the creation of standard components that are guaranteed to work together. We must develop such a theory and use it in the development of components and a component-based design methodology. The STL catalog shows the elements of such a theory: a classification of components and a compatible component interface design. We should not expect to be able to take large pieces of software and connect them to each other without a firm scientific foundation at the most basic level. The lack of such a foundation is responsible for the lack of success in component-based software development.

One of the principles underlying STL is a consistent set of requirements on components. The notion of user-defined types in programming languages was a fundamental idea that has evolved over time. An important lesson learned in language design has been to ensure that user-defined types are treated by the language exactly as built-in types. STL extends this notion to user-defined components. For example, an algorithm component can operate equally well on user-defined containers—built from language built-in types such as arrays—as on STL-defined containers. Such a property requires concrete requirements from component designers. Every type of component is required to provide a specific set of interfaces. For example, every container is required to provide an equality operator as well as an inequality operator. More importantly, each required interface must meet stated complexity requirements. These requirements are stated explicitly and in detail in the STL definition document. These kinds of requirements are the elements of a theory that enables us to write components that can work both with existing components and components that will be built in the future.

There is other work to be done as well, such as trying the approach on real applications, developing catalogs for other domains, defining a complete software process, etc. But everything depends on pinning down the details of what I have referred to as a "theory of software components" and this is best done at the level of fundamental components.

Concern 2: The idea of component-based software construction is an old one and it hasn't worked before. Why should it work now? There are several answers to this concern. First, a comprehensive solution has many ingredients: systematic study of algorithms and data structures, combination of programming techniques from multiple paradigms, sufficient programming language support for genericity—all of these available in a mainstream language. Indeed, there has been progress along all these lines and only recently have all reached a level of maturity so that they can be combined. Second, component programming relies on proper software modularization. Our understanding of modularity has been increasing over time. The seminal work by Parnas on information hiding and modularization in general, the efficient support of abstraction facilities in a mainstream programming language such as C++, functional programming techniques such as closures, and compiler techniques aiding in type inference, are some of the important milestones in this progression. The modularity approach of STL is in a way more conservative than previous approaches. Rather than starting with a blank piece of paper and deciding what modules are needed, STL postulates that all software must be built out of a few types of components. But this is the way it has to be! Only by limiting the number of kinds of components can we hope to educate engineers in the design and use of components. By analogy to circuit design, we only need components such as resistors and capacitors, not thousands of specially-crafted ones. We may need many different kinds of resistors—provided by different vendors—but the number of component types is limited.

In short, progress in many areas of computer science has laid the foundation for successful component programming. More important, a component programming paradigm is a key technical challenge to software engineering that must be met. Without confronting it, we will continue to use individual programming language statements as our software components.

5 Implications of component programming

Changing our software paradigm from line-by-line programming to component programming has profound and far-reaching implications for software engineering both in terms of what is needed to make the change possible and in its impact on other aspects of the field. In this section, I review what is required to make it succeed and the benefits to be gained.

5.1 Software industry

The practical benefit of component programming is the conversion of the software industry into a component-based industry. Such a transformation is necessary to solve the industry's problems: even the most successful software organizations consistently suffer from poor quality and late delivery. We need a software components industry analogous to the chip industry that has propelled the growth of the computer industry. By fixing the kinds of components and the interface to those components, as in STL, it is possible for different vendors to provide software catalogs based on their special expertise—encryption, image processing, and so on. Application vendors—system integrators—then build solutions using standard catalogs. Component vendors specialize in algorithms and data structures, application builders specialize in architectures and application requirements. As in other engineering disciplines, we will be able to develop different specialties in software engineering. The division of the industry into component builders and system builders will reinforce the development and adoption of standard interfaces. The discovery of good interfaces in software is as hard, if not harder, than in hardware engineering. Indeed, modularization techniques and interface design are two sides of the same coin and dividing our concerns into component design and system design will speed advances in both.

5.2 Software design

One of the questions in software design methodology, to which a satisfactory answer has not been found, is whether design is a top-down or bottom-up activity? Component programming answers this question concretely and pragmatically: You must know the types of components that are available. You must be familiar with all the standard catalogs. Depending on your specialty, you must also be familiar with several special-purpose catalogs. The task of design is to devise a particular configuration—architecture—that enables the interconnection of appropriate components. In the implementation step, you simply look in catalogs to find the components. Occasionally, you may need to write a component yourself, but this should be the exception.

A complete software process based on component programming is yet to be defined. Component programming addresses the design and implementation steps of the process. It limits the design space and gives the designer a vocabulary for decomposing the design in terms of this vocabulary. For example, STL offers several types of linear and associative containers and adapters for obtaining variations of these containers. Having analyzed the problem and identified the kinds of objects and operations you need, the design process starts by choosing the needed containers. The choice is based on the kinds of objects in the application and the needed operations. For example, may there exist several copies of an object? Is fast access to arbitrary objects required? The operations required by the design are made up from individual STL algorithms or by combining several appropriate ones. The primary design guideline provided by STL is the decomposition of the design in terms of the five types of abstractions: containers, algorithms, adapters, function objects, and iterators.

5.3 Language design

One of the key requirements for the success of component programming is the availability of components that are generic and efficient. In STL, this goal is achieved by using C++ templates. The template facility of C++ combines the advantages of highly generic code, type safety, and code efficiency. This is due to requiring minimum type information from the template definition, and completing the type checking at template instantiation time—still at compile-time. This approach strikes the right balance between compiling versus interpreting or static versus dynamic languages: the compiled code is efficient and contains no type checking yet the programmer does not overspecify the type requirements. The compiler's template processor does a fair amount of type inference.

For example, we cannot write the program *find* shown in Figure 1 so simply in a language like ML which has a more restricted static typing rule. ML [15]would require that the equality test for values be resolved by the programmer to resolve the ambiguity of the equality operator, for example to *integer* equality. This, of course, immediately reduces the genericity of the component and makes it special-purpose.

An important direction for the design of programming languages is to support the writing of generic and efficient components. The right kind of support will give the programmer a lot of power at little cost in language complexity. The C++ template is a good example of such a facility. Of course, language features alone are not enough and appropriate tools will be necessary. Currently, debugging of generic programs is rather difficult both due to generally poor compiler messages and due to lack of debugger support. One particularly useful tool would be a template "analyzer" that can check the compatibility of a template component with your particular component. It would have to check whether all the operations required by the template are provided by your component. Currently, this compatibility is checked by the compiler.

5.4 Software engineering education

The decision of what we should teach in a software engineering course is a very difficult one today. The same lack of a scientific foundation that causes the problems of the software industry is the source of the education dilemma as well. A component-based approach implies that we must teach component-based design and implementation from the beginning. We must teach the progression of how to build components, how to build catalogs, and how to build applications using catalogs. The taxonomy of components into algorithms and containers also motivates a more systematic study of algorithms and data structures. The taxonomies of containers, algorithms, and iterators, with emphasis on appropriate interfaces and complexity of the implementation, allow us to approach the field in a systematic way.

The component-based approach enables a concrete treatment of software engineering—solving one of software engineering education's long-standing problems. For example, many of the principles that we believe are important are in fact difficult to demonstrate to a beginning student. Many of these principles can be demonstrated rather well with components from STL. For example, the *find* component in Figure 1, despite its simple appearance, demonstrates several principles[2]:

1. Separation of concern: this algorithm only is concerned with processing based on a minimum of assumptions—no memory organization (the concern of the iterator) and no memory allocation (the concern of the container). Actually, STL even removes the memory allocation concern from the container by parameterizing all containers with an additional *allocator* parameter. An allocator captures the memory model being used. A default allocator is provided and requirements for user-defined allocators are given. An allocator defines the concepts of memory locations, their sizes and addresses, and

allocation and deallocation policies. Allocators extend the genericity of STL components to different memory models.

2. Generality: this algorithm is the most generic description of linear search and makes as few assumptions as possible.
3. Abstraction: the algorithm is as abstract as possible; it uses the abstract operations of equality, dereference, and increment. The iterators it uses in its interface are abstractions of container organizations.
4. Modularity: the separation of the iterator is responsible for the simplicity of the algorithm.

Component programming will take hold only if component-based software engineering is part of the student's education. A catalog such as STL is a good source of examples and a good starting point. We can teach software engineering concretely. Current textbooks are either only abstract or concentrate on line-by-line development.

6 Relationship to other work

The work reported here is based directly on STL [12] and the work leading to it, such as [9]. The need for a multiplicity of abstractions is clearly indicated in the title of Wirth's book, "Algorithms + Data Structures = Programs" [17] and is further developed in [8]. Some design issues of STL have been discussed in [5, 4, 3, 13]. The use of STL is illustrated in [16].

In general, the work here is a step in the direction of finding appropriate modularization techniques. For example, with information hiding [10], Parnas introduced the notion of decomposing software based on the design decisions that we want a module to hide. Component programming requires that each catalog contains a small number of types of modules. The modularization in STL is based on generic containers, algorithms, iterators, and function objects. This modularization allows the same algorithms to apply to a large number of containers, thus reducing the number of modules that need to be written. The approach is loosely related to Parnas's program families [11]. Parnas's approach regards a program being developed as a member of a family of possible programs. The generic programming approach views a component as implementing a family of components all sharing the same design decisions about the context in which they will be used. This information is captured by the generic (template) interface.

Component programming can also be contrasted with other paradigms such as object-oriented and functional programming. A complete comparison requires a long discussion and even some research but some brief comparisons are possible. The most obvious difference between component programming and the other paradigms is that component programming relies on components of different types, that is, the component space contains different types of abstractions. It is difficult to compose most industrial applications from only a single type of abstraction whether it is objects or functions.

Another difference between object-oriented programming and component programming is that object-oriented programming relies on inheritance whereas component programming relies on genericity. Traditional object-oriented languages treat genericity only at the data type level. But the separation of components into containers, algorithms, and iterators allows genericity to be used to a much deeper extent. In particular, generic algorithms may be written that work on many different kinds of data structures. It is, of course, possible to combine component programming and object-oriented programming techniques.

Object-oriented programming insists on the consistency of interfaces but this consistency can be called "syntactic." A subclass may reimplement an interface as long as the parameters maintain the same types as before. The **requires** clause of Eiffel [7] is a way of strengthening the semantic content of an interface. In component programming, the execution complexity of an interface is also part of the contract between the component provider and component user.

Since the Turing paper of Backus[1], there has been a significant amount of research in functional programming. The function objects of STL make it possible to use some important functional techniques in an imperative language. But, rather than insisting on the lack of side-effects, STL uses side-effects in a disciplined way. Component programming, in general, must adopt the salient features from each paradigm as long as they can be combined efficiently and naturally. STL containers represent ideas from object-oriented programming while algorithms and function objects represent ideas from functional programming.

Much of the work in "software reuse" is aimed at finding so-called *reusable components*. A significant theme in that area is that the problems are nontechnical (see Myth#1 in [14]). The emphasis of our work is that 1) the underlying problem is technical; 2) the difficult problem is in discovering useful taxonomies of components and interfaces; 3) components must be designed as part of a family that forms a catalog; and 4) without the technical foundation, the nontechnical problems are not well-defined.

7 Summary and conclusions

In this paper, I have argued in general about the necessity of a component-based approach to software engineering, defined exactly what kinds of components are needed, and given a component-based solution to the standard KWIC problem—using the STL standard C++ library. The development of a component-based paradigm is the major technical challenge facing software engineering today. There may be many nontechnical issues (e.g. organizational) facing us as well but unless we solve the technical problem, the benefits of solutions to the nontechnical issues will be illusory. I have outlined an approach for solving the technical problem.

This approach is based on a taxonomy of types of software components and a set of laws that must be obeyed by components. Such an approach can provide a much needed scientific foundation for the field of software engineering. But much work remains to be done, both at the fundamental level and at the empirical level. Because only a comprehensive solution will work, it is not possible to address all the issues in a single paper. I have tried to paint the big picture and identify the key issues. I have also tried to indicate the key requirements that the solution must meet.

In my group, we are currently running several projects to address some of the issues:

- We are rewriting ("reengineering") some small to medium-sized applications to use generic components. The goal is to gain a quantitative evaluation of using standard components both in terms of static metrics such as code size and in terms of runtime performance;
- We are designing some standard applications to help define a software process based on standard components. One of these is for an industrial partner;
- We are designing a generic catalog for window management; and
- We are examining the applicability of the approach to distributed and parallel applications.

The last two are multi-year research projects.

Acknowledgments. Alex Stepanov not only explained to me the intricacies of STL but the whole philosophy and history of the work—several times! He convinced me that a right technical approach exists and that it has to be comprehensive. Most, if not all, of the ideas in this paper were clarified for me in conversations with Alex. Georg Trausmuth wrote the KWIC program. Milon Mackey, René Klösch, Robert Barta, Georg Trausmuth, Meng Lee and Alex Stepanov provided valuable comments on previous drafts of this paper.

STL is available by anonymous ftp from bulter.hpl.hp.com in the directory stl.

References

[1] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–41, Aug. 1978.

[2] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[3] A. Koenig. File iterators. *Journal of object-oriented programming*, pages 59–62, November December 1994.

[4] A. Koenig. Generic iterators. *Journal of object-oriented programming*, pages 69–72, Sept. 1994.

[5] A. Koenig. Templates and generic algorithms. *Journal of object-oriented Programming*, pages 45–7, June 1994.

[6] D. McIlroy. *Mass-produced software components*, pages 88–98. Petrocelli/Charter, 1976.

[7] B. Meyer. *Object-oriented software construction*. Prentice-Hall, Englewood Cliffs,

NJ, 1993.

[8] D. R. Musser and A. A. Stepanov. *The Ada Generic Library: Linear List Processing Packages*. Springer Verlag, 1989.

[9] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software—Practice and Experience*, 24(7):623–42, July 1994.

[10] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8, Dec. 1972.

[11] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2:1–9, Mar. 1976.

[12] A. A. Stepanov and M. Lee. The standard template library. Technical report, Hewlett-Packard Laboratories, May 1994.

[13] B. Stroustrup. Making a vector fit for a standard. *C++ Report*, Oct. 1994.

[14] W. Tracz. Software reuse myths. *ACM SIGSOFT Software Engineering Notes*, 13(1), Jan. 1988.

[15] J. D. Ullman. *Elements of ML programming*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[16] M. J. Vilot. An introduction to the standard template library. *C++ Report*, 6(8):22–9, Oct. 1994.

[17] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1976.

APPENDIX 1. Source for the KWIC program

```
//   kwic.cc
//   the KWIC program
//   Georg Trausmuth, May 1995
//
#include <iostream.h>
#include <pair.h>
#include <list.h>
#include <multimap.h>
#include <algo.h>
#include <vector.h>

typedef vector<char> string;
typedef multimap<string, string *, less<string> > permutedTitles_t;
typedef list<string> Titles_t;
typedef pair<const string, string *> TitlesPair_t;

inline istream& operator>>(istream& istr, string& mystr)
{
   char titleBuffer[256];
   char eatNewline;

   istr.get(titleBuffer, 256, '\n');
   istr.get(eatNewline);

   if(istr && eatNewline=='\n') {
      mystr.erase(mystr.begin(),mystr.end());
      copy (titleBuffer, find (titleBuffer, titleBuffer+256, '\0'),
          back_inserter(mystr));
   }
   return istr;
}

inline ostream& operator<<(ostream& ostr, const string& mystr)
{
   string::const_iterator si;
   for(si=mystr.begin(); si!=mystr.end(); ostr << *si++);
   return ostr;
}

class permute {
public:
   permute() : str('\0'), pos(0) {}
   permute(const string& s) : str(s), pos(str.begin()) {}
```

```
    string operator*()
    {
       if((pos == str.begin()) || (pos == 0)) return str;

       string ret;
       copy(pos+1, str.end(), back_inserter(ret));
       ret.push_back(' ');
       copy(str.begin(), pos, back_inserter(ret));
       return ret;
    }

    permute& operator++()
    {
       if(pos != 0) pos = find(pos+1, str.end(), ' ');
       if(pos == str.end()) pos=0;
       return *this;
    }

    permute operator++(int)
    {
       permute p = *this;
       operator++();
       return p;
    }

    friend int operator==(const permute& p, const permute& q);

private:
    string str;
    string::iterator pos;
};

inline int operator==(const permute& p, const permute& q)
{
    return ((&p == &q) || ((p.pos == 0) && (q.pos == 0)));
}

class CircularShift {
public:
    CircularShift(permutedTitles_t& store) : index(store) {}

    void operator()(const string& str) {
       for(permute x(str); x != permute(); x++)
          index.insert(TitlesPair_t(*x, &str));
    }
```

```
private:
   permutedTitles_t& index;

};

inline ostream& operator<<(ostream& out, const TitlesPair_t& p)
{
   return out << p.first << “:\n\t “ << (*(p.second)) << ‘\n’;
}

// main program

int main (int argc, char *argv[]) {
   Titles_t titles;
   permutedTitles_t KWICindex;

   copy(istream_iterator<string,ptrdiff_t>(cin),
      istream_iterator<string,ptrdiff_t>(),
      back_inserter(titles));

   for_each(titles.begin(), titles.end(),
         CircularShift(KWICindex));

   copy(KWICindex.begin(), KWICindex.end(),
      ostream_iterator<TitlesPair_t >(cout));

   return 0;
}
```

APPENDIX 2. //Sample input and output of the KWIC program

Input file:

Fundamentals of Software Engineering
Applicators, Manipulators, and Function Objects
An introduction to the Standard Template Library

Output file:

AN INTRODUCTION TO THE STANDARD TEMPLATE LIBRARY:
    AN INTRODUCTION TO THE STANDARD TEMPLATE LIBRARY
AND FUNCTION OBJECTS APPLICATORS, MANIPULATORS,:
    APPLICATORS, MANIPULATORS, AND FUNCTION OBJECTS
APPLICATORS, MANIPULATORS, AND FUNCTION OBJECTS:
    APPLICATORS, MANIPULATORS, AND FUNCTION OBJECTS
ENGINEERING FUNDAMENTALS OF SOFTWARE:
    FUNDAMENTALS OF SOFTWARE ENGINEERING
FUNCTION OBJECTS APPLICATORS, MANIPULATORS, AND:
    APPLICATORS, MANIPULATORS, AND FUNCTION OBJECTS
FUNDAMENTALS OF SOFTWARE ENGINEERING:
    FUNDAMENTALS OF SOFTWARE ENGINEERING
INTRODUCTION TO THE STANDARD TEMPLATE LIBRARY AN:
    AN INTRODUCTION TO THE STANDARD TEMPLATE LIBRARY
LIBRARY AN INTRODUCTION TO THE STANDARD TEMPLATE:
    AN INTRODUCTION TO THE STANDARD TEMPLATE LIBRARY
MANIPULATORS, AND FUNCTION OBJECTS APPLICATORS,:
    APPLICATORS, MANIPULATORS, AND FUNCTION OBJECTS
OBJECTS APPLICATORS, MANIPULATORS, AND FUNCTION:
    APPLICATORS, MANIPULATORS, AND FUNCTION OBJECTS
OF SOFTWARE ENGINEERING FUNDAMENTALS:
    FUNDAMENTALS OF SOFTWARE ENGINEERING
SOFTWARE ENGINEERING FUNDAMENTALS OF:
    FUNDAMENTALS OF SOFTWARE ENGINEERING
STANDARD TEMPLATE LIBRARY AN INTRODUCTION TO THE:
    AN INTRODUCTION TO THE STANDARD TEMPLATE LIBRARY
TEMPLATE LIBRARY AN INTRODUCTION TO THE STANDARD:
    AN INTRODUCTION TO THE STANDARD TEMPLATE LIBRARY
THE STANDARD TEMPLATE LIBRARY AN INTRODUCTION TO:
    AN INTRODUCTION TO THE STANDARD TEMPLATE LIBRARY
TO THE STANDARD TEMPLATE LIBRARY AN INTRODUCTION:
    AN INTRODUCTION TO THE STANDARD TEMPLATE LIBRARY