SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **33**:121–142 (DOI: 10.1002/spe.499)

A comparative evaluation of generic programming in Java and C++

Hossein Saiedian* and Steve Hill

Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS 66045, U.S.A.

SUMMARY

Generic programming has been defined as 'programming with concepts' where a concept refers to a family of abstractions. The criteria for generic programming include independence of collections from data types, independence of algorithms that operate on the collection, and the adaptability of the collections. This paper examines and evaluates the support for generic programming in the Java Development Kit (JDK) in comparison to C++'s Standard Template Library (STL). The evaluation will consider both the 'qualitative' factors as well as certain 'quantitative' factors (i.e. factors that can be measured). The qualitative factors that are considered include: 1. a comparison of the structure and APIs; 2. homogeneity versus heterogeneity; and 3. ease of use (including ease of converting to collection classes, ease of changing collection type, and ease of error handling).

The quantitative factors include: 1. compiled size; 2. runtime memory usage; and 3. performance. The results of our evaluative comparisons based on the above factors and certain other criteria are presented at the end. Based on the results, we conclude that the support provided for generic programming in C++'s STL is superior to that provided by JDK. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: generic programming; Java; JDK; C++; STL

1. INTRODUCTION

The objective of our work has been to make an impartial comparative evaluation of generic programming in C++ and the Java Development Kit (JDK). To the best of our knowledge, no such evaluation has yet been made. To perform our evaluation, we will consider both the qualitative as well as quantitative factors. The qualitative factors include:

1. structure and APIs;
2. homogeneity versus heterogeneity; and

*Correspondence to: Hossein Saiedian, Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, KS 66045, U.S.A.

Received 4 June 2001
Revised 18 September 2002
Accepted 18 September 2002

3. ease of use (including ease of converting to collection classes, ease of changing collection type, and ease of error handling).

The quantitative factors include:

1. compiled size;
2. runtime memory usage; and
3. performance.

One of the best definitions of generic programming is Musser's 'programming with concepts' [1]. The 'concepts' are the ideas or building blocks (or a 'family of abstractions') that we use to write software such as data structures or containers (e.g. an array, vector or string) and algorithms (e.g. sort and find) that operate on data structures. Stroustrup expands on this definition by stating that data structures are general concepts and are independent of element type; therefore they should be represented independently and, furthermore, algorithms which can be expressed independently of a specific data structure, efficiently and without difficulty, should be emphasized [2]. Additionally, Musser and Saini state that the 'adaptability of components is essential to generic programming' [3]. From these definitions, one can define the criteria for generic programming to be as follows:

1. collections are independent of type;
2. algorithms are defined independently (when possible) of a collection;
3. components are adaptable.

C++ supports the concepts of generic programming in the Standard Template Library (STL). This library defines a set of standard containers and algorithms that can operate on the containers. Genericity in containers comes from utilizing C++'s template mechanism. Genericity in the algorithms comes from defining algorithms in terms of iterators on containers.

The JDK provides (partial) support for generic programming through its class library and, most notably, the collection classes. The collection classes in the JDK are generic in the sense that they hold objects of any type. This is accomplished by defining the containers in terms of the JDK's object class, which is the root of the JDK's class hierarchy. Algorithms are then defined as part of the collection itself or as part of the JDK's collections class.

We examine both approaches based on the criteria defined above and make concluding comments about their support for generic programming. The approach taken in this paper is to develop a simple application and then to enhance it to utilize the STL container and JDK collection classes. While doing this, features of both environments for generic programming are compared. As the qualitative support for generic programming should not change between containers, we have strictly limited the number of containers that we compare. Although we also present quantitative factors as part of our results, the results can only be used to understand the impact of generic programming within that environment.

As noted by one of the reviewers, a fundamental difference in generic programming between the JDK and STL is the language support provided by their corresponding languages. C++ provides generic types, Java does not. This difference influences the two libraries to some extent, but we have decided to avoid direct feature-by-feature language comparison and instead focus on their externally observable support for generic programming and not necessarily what leads to or causes such support differences.

The organization of this paper is as follows. Section 2 discusses the organization of the class hierarchies. Section 3 presents the experimental programs and changes made to transitions between

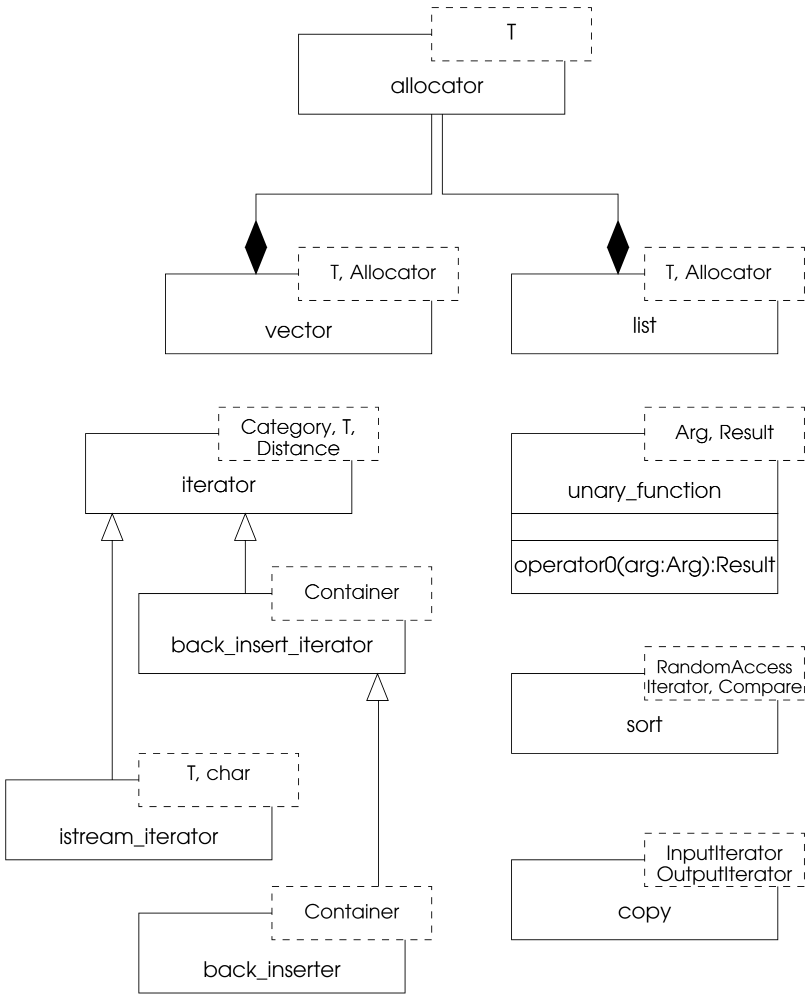

Figure 1. C++ STL class diagram.

them. Section 4 discusses the heterogeneity/homogeneity aspects of the containers. Section 5 discusses some error handling aspects of the containers. Section 6 presents the quantitative metrics recorded from the programs. Section 7 discusses the results and concludes the paper.

Throughout the paper, the term container(s) is used to refer to the STL containers (or in generic contexts). The term collections or collection classes is used to specifically refer to the JDK's collection classes. Class names are capitalized as they are defined in their respective libraries.

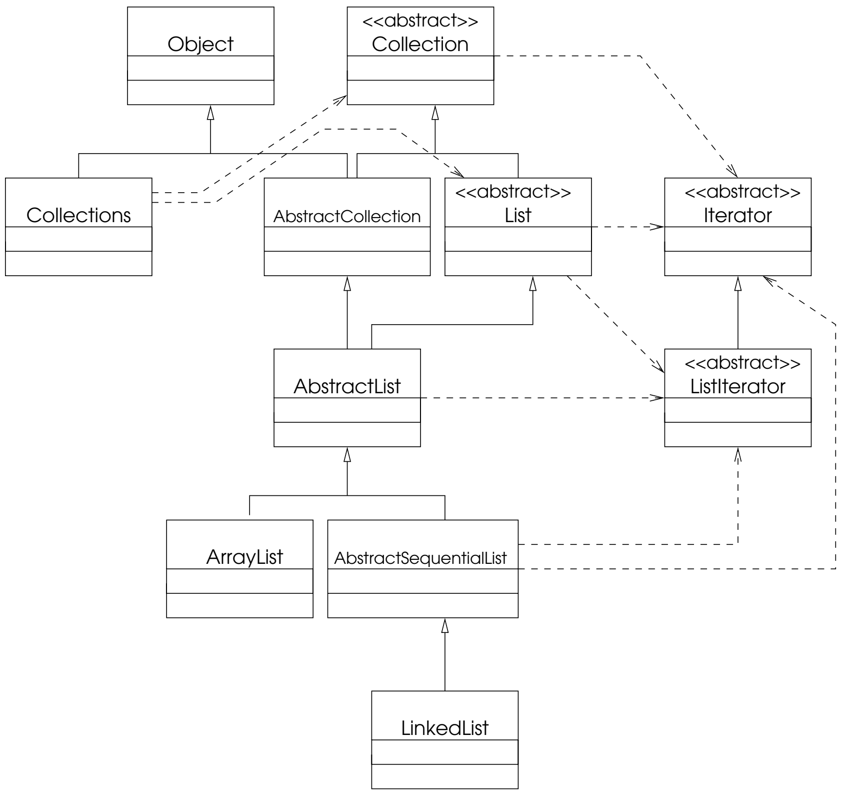

Figure 2. JDK collection classes.

2. CLASS HIERARCHY

Simplified class diagrams for the portions of the STL and the JDK container classes are shown in UML-like diagrams in Figures 1 and 2. The class diagram for the STL was reverse engineered from the pre-processor output of the C++ projects. The class diagram for the JDK collection classes was built by referring to the JDK class documentation in source [4].

2.1. STL class hierarchy

The class hierarchy of the STL shows a relatively flat inheritance hierarchy with all but the specialized iterators (`istream_iterator` and `back_inserter`) having no parent classes. What is not apparent from the class diagram is that each container defines its iterator within its own namespace and that this iterator conforms to a standardized iterator interface according to its type.

Also, from the STL class diagram one notices that the only specified associations between classes is between the containers and their memory allocator. An iterator obviously must have access to its container's data store, but this is not done through a direct association with the container class. This lack of associations is a product of C++'s template mechanism and the STL's design. The algorithms are template-based global functions that take iterator objects as parameters. Also, the library user must implement an appropriate subclassed `unary_function` class for a comparison function.

The template mechanism used in the STL provides for type independence, and the typing rules in C++ make the collections homogeneous.

2.2. JDK collections hierarchy

The class hierarchy of the JDK collection classes shows a more monolithic structure emphasizing inheritance and interfaces. As one can see, the two container classes used in the test programs are at a depth of 4 and 5 in the hierarchy.

The JDK also provides iterators over collections. However, iterators in the JDK are provided by realizing the appropriate iterator interface. As Java is an object-oriented language, the collection classes cannot provide stand-alone algorithms as in the STL. The algorithms for the JDK are primarily found in the utility class `Collections`. The JDK does provide a `Comparator` class to allow custom comparison routines. The `unary_function` class is used to provide this functionality in the STL.

The JDK collection classes provide type independence by being defined in terms of the JDK's Object class. This prevents collections of primitives and makes the collections heterogeneous.

3. EXPERIMENTAL PROGRAMS

The program used for comparison is quite simple. It will perform three basic operations:

1. read in a file of 5000 random integers and place them into a container;
2. sort the container into an ascending order;
3. write all elements to standard out.

In the sections that follow, portions of the C++ and Java programs are compared and their generic programming support is examined. Full source code may be found in Appendix A. Information on programming with the STL may be found in [2,3]. Information on programming with JDK collection classes may be found in [4].

3.1. Conversion to a vector

3.1.1. Reading in the values

The first step in the conversion is to utilize the collection object instead of the array used in the base project. This step is simply creating an object of the appropriate type. The next major step is loading the data into the collection object. The Java program uses the JDK's `ArrayList` class which replaces the `Vector` class for the JDK collection classes. The code in Listing 1 shows the differences between the base and vector projects.

C++ Base
```
for (i = 0; i < 5000; i++) {
    inputFile >> myNumbers [i];
}
```

C++ Vector
```
copy (
    istream_iterator<int>(inputFile,
    istream_iterator<int>(),
    back_inserter(myNumbers)
);
```

Java Base
```
i = 0;
while (inputFile.nextToken() !=
       StreamTokenizer.TT_EOF) {

    myNumbers[i] = new
       Integer((int)inputFile.nval);
    i++;
}
```

Java Vector
```
while (inputFile.nextToken() !=
       StreamTokenizer.TT_EOF) {

    myNumbers.add (new
     Integer((int)inputFile.nval));
}
```

Listing 1. Reading in the values.

The C++ program changed from making an assignment to an array element to utilizing a generic `copy` algorithm. The prototype of the `copy` algorithm, which is defined in terms of iterators, is

```
OutputIterator copy (InputIterator first, InputIterator last,
                     OutputIterator result)
```

The only major change in the Java code is that the `add` method on the collection is used instead of an assignment to an array element.

3.1.2. Sorting the values

The next step is the sorting of the collection. The code in Listing 2 shows the differences between the base and vector projects. The conversion to the collections classes in both projects provide for dramatically cleaner code. The C++ vector uses the generic sort algorithm. The prototypes for the sort algorithms are

```
void sort (RandomAccessIterator first, RandomAccessIterator last)
void sort (RandomAccessIterator first, RandomAccessIterator last,
          StrictWeakOrdering comp)
```

The Java vector project uses the generic sort algorithm defined as part of the Collections class. This class provides methods that operate on containers. The prototypes of the sort algorithms that the JDK provides are

```
public static void sort(List list)
public static void sort(List list, Comparator c)
```

C++ Base

```
for (int i = 0; i < 5000; i++) {
    for (int j = i + 1; j < 5000; j++) {
        if (myNumbers[j] < myNumbers[i]){
           int temp = myNumbers[i];
           myNumbers[i] = myNumbers[j];
           myNumbers[j] = temp;
        }
    }
}
```

C++ Vector

```
sort (myNumbers.begin(),
      myNumbers.end());
```

Java Base

```
for (int i = 0; i < 5000; i++) {
    for(int j = i + 1; j < 5000; j++) {
        if (0 > myNumbers[j].compareTo
                 (myNumbers[i])) {
            Integer temp = myNumbers[i];
            myNumbers[i] = myNumbers[j];
            myNumbers[j] = temp;
        }
    }
}
```

Java Vector

```
Collections.sort(myNumbers);
```

Listing 2. Sorting the values.

The first form of the sort function assumes that the type supports `operator<` for C++ and the Comparable interface for Java. The second form of the sort functions takes a comparison object.

The primary difference between the two environments are the terms in which the algorithm is defined. Since the STL's generic sort function is defined in terms of iterators, it will work with anything that supports random access iterators. In contrast, Java's sort function is defined in terms of a list data structure.

3.1.3. Printing the values

The final step in the experiment is to iterate over the collection and print out all the elements. The code in Listing 3 shows the differences between the base and vector projects.

The C++ vector code uses the `for_each` generic algorithm to iterate over the list. The algorithm receives a function object which is applied to each element in the range of `[first, last)`. The prototype of the `for_each` algorithm is

```
void for_each (InputIterator first, InputIterator last, UnaryFunction f)
```

This approach requires the definition of a subclass of `unary_function`. This class is `C_ColPrinter` in the experimental program.

The Java vector program is very similar in structure to the Java base program. It utilizes an iterator object to walk and access the elements of the list. This approach can be simpler than the generic

C++ Base

```
for (i = 1; i <= 5000; i++) {
    cout.setf (ios::left,
               ios::adjustfield);
    cout.width(20);
    cout << myNumbers [i];
    if(i % 4 == 0) cout << endl;
}
```

C++ Vector

```
class C_ColPrinter:public
      unary_function<int, void> {
private:
    int m_currCol;
    int m_numCols;
public:
     C_ColPrinter (int numCols = 1){
        m_numCols = numCols;
        m_currCol = 0;
    }

    void operator()(const int i) {
        cout.setf(ios::left,
                  ios::adjustfield);
        cout.width(20);
        cout << i;
        m_currCol++;
        if (m_currCol % m_numCols == 0)
             cout << endl;
    }
};

C_ColPrinter myPrinter(4);
for_each (myNumbers.begin(),
    myNumbers.end(), myPrinter);
```

Java Base

```
for (i = 1; i <= 5000; i++) {
    System.out.print(myNumbers [i-1]);
    if (i % 4 == 0)
        System.out.println("");
}
```

Java Vector

```
Iterator iter;
for (int i=1, iter = myNumbers.iterator();
              iter.hasNext(), i++) {
    System.out.print ((Integer)iter.next());
    if (i % 4 == 0)
       System.out.println("");
}
```

Listing 3. Printing the values.

algorithm approach, but does not provide much opportunity for reuse. The C++ program could have been written in the same manner as the code shown in Listing 4.

3.2. Converting from vector to a linked list

The process of converting to the linked list class required only one change in the C++ program. The list class does not support random access iterators and could not be used with the generic sort algorithm. Therefore, the sort had to be changed to utilize the list class' sort method.

There were no changes required in the Java program.

```
    my_iter iter;
    int i;
    for (iter = myNumbers.begin(), i = 0; iter != myNumbers.end(); iter++, i++) {
        cout.setf (ios::left, ios::adjustfield);
        cout.width(20);
        cout << *iter;
        if(i % 4 == 0)
            cout << endl;
    }
```

Listing 4. Alternative C++ code.

4. HETEROGENEITY VERSUS HOMOGENEITY

Since C++'s templates are bound to types during compilation, the compiler can catch any typing errors. This ensures that the STL containers are homogeneous. The following piece of code illustrates a typing problem that can be found at compile time:

```
vector<int> vect1;
int id = 1;
string name = "Fred";
...
vect1.push_back(id);
vect1.push_back(name);
```

However, the JDK's collection classes are collections of the object type and object is the ancestor of all the JDK classes. This means that the collection classes are heterogeneous as illustrated in the following code which will compile and run successfully.

```
ArrayList myNumbers = new ArrayList();
myNumbers.add (new Integer(1));
myNumbers.add (new String(''Fred''));
```

5. DEFINING ALGORITHMS IN TERMS OF ITERATORS

The STL's practice of defining algorithms in terms of iterators allows one to do the following:

```
vector<int> vect1, vect2;
...
sort(vect1.begin(), vect2.end());
...
```

As long as the collections are of the same type the code will compile and run until the program causes a memory segment violation when it goes past the end of `vect1`. The JDK's practice of defining algorithms in terms of the collection and utilizing only a single iterator avoids this problem. However,

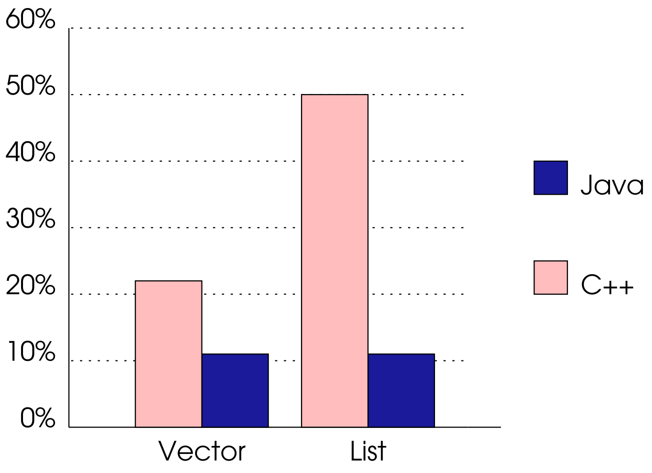

Figure 3. Percentage change in size (bytes) of complied programs.

Table I. Complied program sizes in bytes.

	C++	Java
Base	32 768	1471
Vector	40 960	1650
List	49 152	1656

the heterogeneous nature of the JDK collection classes means that it is possible to have runtime errors due to an incorrect type, as shown in the following code:

```
ArrayList myNumbers = new ArrayList();
myNumbers.add (new Integer(1));
myNumbers.add (new String(''Fred''));
...
for (iter = myNumbers.iterator(); iter.hasNext();) {
    total += (Integer) iter.next();
    ...
```

The above code will run until an attempt is made for the string to be typecast to an integer.

6. QUANTITATIVE METRICS

The quantitative aspects of the programming environments cannot be directly compared. To provide a fair comparison, the percentage change from the base project is compared. The data shown is an average of five runs per test. The raw data may be found in Appendix B.

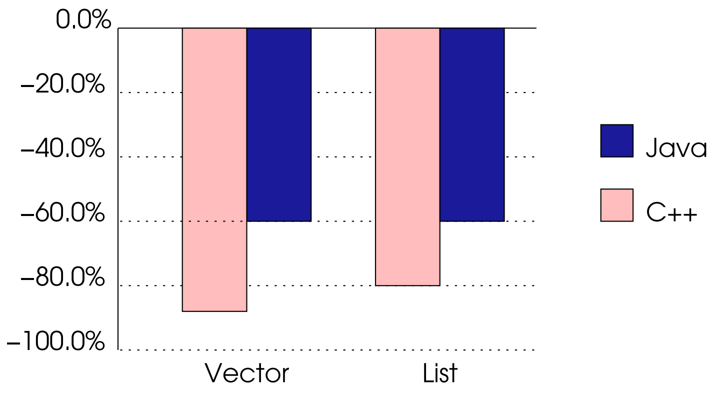

Figure 4. Percentage change in runtime (seconds).

Table II. Runtimes in seconds.

	C++				Java			
	Total	Load	Sort	Print	Total	Load	Sort	Print
Base	0.330	0.009	0.280	0.045	3.628	1.156	2.119	0.329
Vector	0.051	0.004	0.002	0.046	1.589	1.195	0.058	0.333
List	0.073	0.004	0.017	0.053	1.586	1.180	0.064	0.345

The C++ projects were developed on a Compaq XP1000 professional workstation running Tru64 Unix 4.0F and version 6.2-024 of the `cxx` compiler using default optimization. Compaq Tru64 Unix uses RogueWave's implementation of the STL. The Java projects were developed on an IBM ThinkPad 390E running Windows NT version 4 with service pack 5 and version 1.3.0 of the JDK and version 1.3.0-C of the Java HotSpot client.

6.1. Compiled size

Figure 3 shows the percentage change in the compiled programs. The data, in bytes, is shown in Table I. The Java program shows little change from the base program and little difference between the different collection classes. The C++ programs show a significant increase in size including a very marked difference between the vector and linked list projects.

6.2. Time efficiency

The STL and JDK containers use a quick sort or similar algorithm for the sorting. Since the base project uses a very basic bubble sort, it was expected that there would be significant runtime differences.

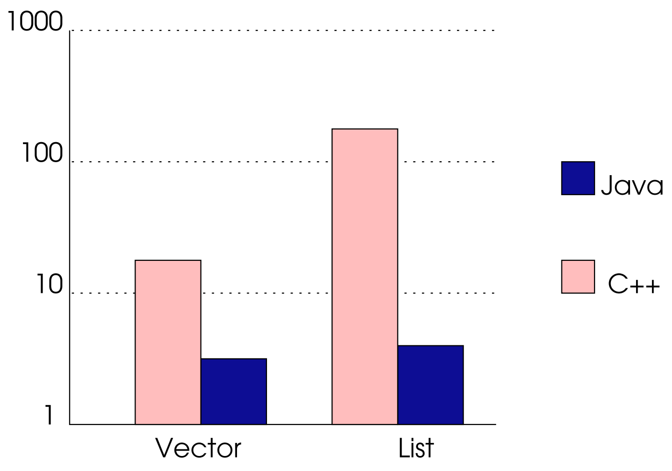

Figure 5. Percentage change in memory usage.

Table III. Memory usage in kbytes.

	C++			Java
	Total	Heap	Stack	Total
Base	315.4	312	3.4	4356
Vector	372.1	368	4.1	4476
List	816.5	808	8.5	4580

Figure 4 shows the percentage change in the total runtimes for the projects. The runtime data is shown in Table II. The data includes the runtimes for each portion of the program as well as the total.

6.3. Space efficiency

The memory utilization for the C++ programs was obtained by using Compaq's Atom tool to profile the programs for memory use. The memory utilization for the Java programs was obtained from sampling the memory usage from Windows NT's task manager. The percentage change in memory usage is shown in Figure 5. The memory usage data is shown in Table III.

The STL linked list program shows a very large increase in memory usage. This increase may be attributed to the fact that the STL list is a dynamically allocated doubly linked list. Just the pointers for the list on a 64-bit platform will use approximately 78 kbytes of additional memory.

7. DISCUSSIONS AND CONCLUSIONS

This paper has examined several qualitative and quantitative issues for the support of generic programming in C++ STL and the JDK. In conclusion we will examine first the qualitative and then the quantitative issues.

If the criteria for generic programming include:

1. collections are independent of type;
2. algorithms are defined independently (when possible) of a collection;
3. components are adaptable;

how do the JDK collection classes compare to the STL?

7.1. Collections are independent of type

The JDK collection classes provide type independence by being defined in terms of the JDK's object class. This prevents collections of primitives and makes the collections heterogeneous. C++ allows collection of primitives and the template mechanism used in the STL provides for type independence, and the typing rules in C++ make the collections homogeneous.

7.2. Algorithms are defined independently of a collection

In the JDK the algorithms are defined as members of the utility class `Collections` or as part of the container and are defined primarily in terms of the `List` interface. In comparison, the algorithms in the STL are defined in terms of iterators, where an iterator provides navigation over a collection and access to its elements. Defining algorithms in this manner allows an algorithm to work with any collection that supports the required type of iterator. For example, the `copy` algorithm used to load the containers in the test programs works with both the vector and list containers and the `sort` algorithm, which requires a random access iterator, only works with the vector.

7.3. Adaptability of components

Adaptability may come in two forms. The first is the ability to work across a variety of types. Both Java and C++ meet this form of adaptability. The second form of adaptability covers extension and reuse. One may extend by adding new collections or algorithms, reuse an existing algorithm with a new collection, or *vice versa*.

Since the JDK collection class hierarchy is predefined, one has to work within that hierarchy to extend it in a reusable manner. Thus, a new collection would have to implement one of the base classes and an iterator such as the `List` and `ListIterator` interfaces. Also, new algorithms cannot be transparently added to the collections hierarchy. Although these are not significant issues they do cause a problem in a lack of consistency. An example of a lack of consistency would come from having multiple utility classes implementing different algorithms.

The design style used in the STL is extensible to any type of container or algorithm without the constraints of working with an existing class hierarchy. Thus, if one wanted, they could create a new container and by defining the appropriate iterators; the container could be used with any compatible

Table IV. A qualitative comparison of C++ and JDK.

	C++	JDK
Structure (class hierarchy)	Relatively flat; iterators have no parents (except those that are very specialized)	A more monolithic structure emphasizing inheritance
Homogeneity versus heterogeneity	Homogeneous	Collections classes are a collection of the object type (the ancestor of all JDK classes), thus the collections are heterogeneous
Modifications (when converting between collections—for ease of use)	Minor modifications may be necessary in certain situations	None (because the JDK containers realized the `ListIterator` interface, but runtime errors are possible)

Table V. A criteria-based comparison of C++ and JDK.

	C++	JDK
Collections independence of type	Yes	Partial—defined in terms of object class; no collection of primitives
Algorithms independent of collections	Yes (algorithms are presented in terms of iterators)	Algorithms primarily in terms of the `List` interface; certain run-time errors are possible
Adaptability of components (for extensibility and/or reuse)	Yes	Partial support (within the hierarchy)—interfaces may have to be implemented; also, new algorithms may not be transparently added to the collection hierarchy

generic algorithm. Also, any new algorithm that one creates will work with every collection that supports the appropriate type of iterator.

A comparative summary of ‘qualitative’ features (as defined in the early part of this paper) of the JDK and the C++ STL is shown in Table IV. A comparative evaluation of the two languages based on the criteria deduced from [2] and [3] is shown in Table V.

According to the three criteria listed, one may conclude that programming in the STL is superior to that provided by the JDK, and that Java does not provide a generic programming environment. However, support for generic programming is a qualitative issue and one should also consider the quantitative issues and design criteria when evaluating different environments.

Some of the design goals of the Java language and class library (JDK) are [5]:

1. downloadable modules;
2. portability;
3. simplicity.

Some of the design goals for C++ and the STL [6]:

1. runtime efficiency;
2. type safe;
3. support for generic programming.

In order to allow modules to be downloadable, the Java environment must put a priority on the compiled size of the module. The compiled size of the Java test programs ranged from 1471 to 1656 bytes and the C++ programs from 32 768 to 49 152 bytes. In comparison, there is a large increase in size in the C++ programs. Templates are a well-known source of code bloat in C++, and the designers of the STL worked to minimize the affect of code bloat in the STL [1].

Since the Java environment provides much of its functionality in the JDK and virtual machine, ostensibly for smaller compiled size, one may ask what the tradeoffs are. The primary tradeoff seen in the metrics gathered is the memory usage of the programs. The Java programs used 4356 to 4580 kbytes on a 32-bit system and the C++ programs 315 to 816 kbytes on a 64-bit system. As Java is interpreted and has a large class library that must be loaded during runtime it utilizes a lot of resources. Of interest in Java is the small amount of growth in resource usage compared to C++ in the different projects. The C++ list project used approximately 158% more memory than the base project as compared to the approximately 5% growth in the Java project.

The final metric is the runtime efficiency of the environments. If we ignore the differences in the sorting times, which is due to different implementation algorithms, the collection classes of both environments added little to no performance impact. In fact the STL provided approximately 125% improvement in the time required to load the data structure.

Although Java does not provide a true generic programming environment, the designers did provide a robust and workable solution for their design constraints. Many of the generic programming shortfalls in Java could be overcome by the addition of generic types to Java and the addition of generic types is being actively researched. A Java Specification Request, JSR #000014, has been submitted to the Java Community Process Program for the inclusion of generic types into the Java language [7]. Two of the leading projects under consideration as part of JSR #14 are Generic Java (GJ) and `PolyJ`. Both of these projects are conservative extensions to the language to support generic types. Further information on GJ can be found in [8] and information on `PolyJ` can be found at MIT's `PolyJ` site [9] and also [10].

APPENDIX A. COMPLETE SOURCE CODE LISTINGS

Base project in C++

```
int main(int argc, char **argv) {
    fstream inputFile;
    int *myNumbers;
    int i;

    inputFile.open("testFile", ios::in);
    myNumbers = new int[5000];

    // build the array
    for (i = 0; i < 5000; i++) {
        inputFile >> myNumbers[i];
    }

    // sort the array
    for (i = 0; i < 5000; i++) {
        for (int j = i + 1; j < 5000; j++) {
            if (myNumbers[j] < myNumbers[i]) {
                int temp = myNumbers[i];
                myNumbers[i] = myNumbers[j];
                myNumbers[j] = temp;
            }
        }
    }

    // print the array out in 4 columns
    for (i = 1; i <= 5000; i++) {
        cout.setf(ios::left, ios::adjustfield);
        cout.width(20);
        cout << myNumbers[i - 1];
        if (i % 4 == 0)
            cout << endl;
    }

    cout << endl;
    return 0;

}
```

Base project in Java

```
static void main(String[] argv) {
    Integer[] myNumbers;
    int i;

    myNumbers = new Integer[5000];
    FileReader fileReader = new FileReader (new File("testFile"));
    StreamTokenizer inputFile = new StreamTokenizer(fileReader);
    // build the array
    i = 0;
    while (inputFile.nextToken() != StreamTokenizer.TT_EOF) {
        myNumbers[i] = new Integer ((int)inputFile.nval);
        i++;
    }

    // sort the array
    for (i = 0; i < 5000; i++) {
        for (int j = i + 1; j < 5000; j++) {
            if (myNumbers[j].compareTo(myNumbers[i]) < 0) {
               Integer temp = myNumbers[i];
               myNumbers[i] = myNumbers[j];
               myNumbers[j] = temp;
            }
        }
    }

    // print the array out in 4 columns
    for (i = 1; i <= 5000; i++) {
        System.out.print(myNumbers[i-1] + "\t\t");
        if (i % 4 == 0)
            System.out.println("");
    }

}
```

C++ Vector

```
// define a function object to print elements of
// the collection in columns to stdout.
//
class C_ColPrinter : public unary_function<int, void> {
private:
    int m_currCol;
    int m_numCols;
public:
    C_ColPrinter(int numCols = 1) {
        m_numCols = numCols; m_currCol = 0;
    }

    void operator()(const int i) {
        cout.setf(ios::left, ios::adjustfield);
        cout.width(20); cout << i;
        m_currCol++;
        if (m_currCol % m_numCols == 0)
           cout << endl;
    }
};

typedef vector<int> my_collection;
typedef my_collection::iterator my_iter;

int main (int argc, char **argv) {
    my_collection myNumbers;
    fstream inputFile;
    inputFile.open("testFile", ios::in);

    // build the vector
    copy (istream_iterator<int>(inputFile), istream_iterator<int>(),
          back_inserter(myNumbers));

    // sort the vector
    sort(myNumbers.begin(), myNumbers.end());

    // print the vector out in 4 columns
    C_ColPrinter myPrinter(4);
    for_each(myNumbers.begin(), myNumbers.end(), myPrinter);

    return 0;
}
```

Java Vector (Array List)

```
static void main(String[] argv) {
    ArrayList myNumbers;
    int i;

    myNumbers = new ArrayList();
    FileReader fileReader = new FileReader (new File("testFile"));
    StreamTokenizer inputFile = new StreamTokenizer(fileReader);

    // build the array
    while (inputFile.nextToken() != StreamTokenizer.TT_EOF) {
        myNumbers.add (new Integer ((int)inputFile.nval));
    }
    // sort the array
    Collections.sort(myNumbers);

    // print the array out in 4 columns
    Iterator iter;
    for (i = 1, iter = myNumbers.iterator(); iter.hasNext(); i++) {
        System.out.print ((Integer)iter.next() + "\t\t");
        if (i % 4 == 0)
            System.out.println("");
    }
}
```

C++ Linked List

```
// define a function object to print elements of
// the collection in columns to stdout.
//
class C_ColPrinter : public unary_function<int, void> {
private:
    int m_currCol;
    int m_numCols;
public:
    C_ColPrinter(int numCols = 1) {
        m_numCols = numCols; m_currCol = 0;
    }
    void operator()(const int i) {
        cout.setf(ios::left, ios::adjustfield);
        cout.width(20); cout << i;
        m_currCol++;
        if (m_currCol % m_numCols == 0)
            cout << endl;
    }
};

typedef list<int> my_collection;
typedef my_collection::iterator my_iter;
int main(int argc, char **argv) {
    my_collection myNumbers;
    fstream inputFile;
    inputFile.open("testFile", ios::in);

    // build the list
    copy (istream_iterator<int>(inputFile), istream_iterator<int>(),
          back_inserter(myNumbers));
    // sort the list
    myNumbers.sort();

    // print the list out in 4 columns
    C_ColPrinter myPrinter(4);
    for_each(myNumbers.begin(), myNumbers.end(), myPrinter);
    return 0;
}
```

Java Linked List

```
static void main(String[] argv) {
    LinkedList myNumbers;
    int i;

    myNumbers = new LinkedList();
    FileReader fileReader = new FileReader (new File("testFile"));
    StreamTokenizer inputFile = new StreamTokenizer(fileReader);

    // build the array
    //
    while (inputFile.nextToken() != StreamTokenizer.TT_EOF) {
        myNumbers.add (new Integer ((int)inputFile.nval));
    }

    // sort the array
    //
    Collections.sort(myNumbers);

    // print the array out in 4 columns
    //
    Iterator iter;
    for (i = 1, iter = myNumbers.iterator(); iter.hasNext(); i++) {
        System.out.print ((Integer)iter.next() + "\t\t");
        if (i % 4 == 0)
            System.out.println("");
    }
}   // end main
```

APPENDIX B. RAW DATA FOR THE EXPERIMENTS

The quantitative aspects of the programming environments cannot be directly compared. To provide a fair comparison, the percentage change from the base project is compared. The data shown illustrated in the paper is an average of five runs per test. The raw data for the experiments is given below.

JAVA programs

```
                LOAD     SORT     PRINT    TOTAL
Trial 1 BASE    1.131    2.114    0.38     3.625
        LIST    1.201    0.06     0.331    1.591
        VECTOR  1.132    0.07     0.38     1.582

Trial 2 BASE    1.151    2.114    0.374    3.639
        LIST    1.151    0.06     0.331    1.542
        VECTOR  1.162    0.06     0.351    1.573

Trial 3 BASE    1.142    2.103    0.371    3.616
        LIST    1.212    0.05     0.331    1.593
        VECTOR  1.202    0.06     0.341    1.603

Trial 4 BASE    1.142    2.113    0.32     3.575
        LIST    1.212    0.06     0.34     1.612
        VECTOR  1.212    0.07     0.321    1.603

Trial 5 BASE    1.212    2.153    0.32     3.685
        LIST    1.201    0.061    0.33     1.592
        VECTOR  1.192    0.061    0.33     1.583

Average BASE    1.156    2.119    0.329    3.628
        LIST    1.195    0.058    0.333    1.586
        VECTOR  1.18     0.064    0.345    1.589
```

Memory usage in Java

	Total
Base	4356 kbytes
List	4580 kbytes
Vector	4476 kbytes

C++ programs

```
                LOAD     SORT     PRINT    TOTAL
Trial 1 BASE    0.345    0.012    0.289    0.044
        LIST    0.071    0.004    0.017    0.051
        VECTOR  0.052    0.004    0.001    0.047

Trial 2 BASE    0.325    0.003    0.28     0.042
```

```
        LIST    0.074   0.004   0.018   0.053
        VECTOR  0.05    0.004   0.002   0.044

Trial 3 BASE    0.331   0.003   0.283   0.045
        LIST    0.069   0.005   0.016   0.05
        VECTOR  0.051   0.004   0.001   0.046

Trial 4 BASE    0.325   0.002   0.276   0.047
        LIST    0.074   0.004   0.017   0.054
        VECTOR  0.052   0.004   0.002   0.046

Trial 5 BASE    0.325   0.003   0.275   0.047
        LIST    0.075   0.005   0.016   0.055
        VECTOR  0.051   0.004   0.002   0.045

Average BASE    0.33    0.009   0.28    0.045
        LIST    0.073   0.004   0.017   0.053
        VECTOR  0.051   0.004   0.002   0.046
```

Memory usage in C++

	Heap	Stack	Total
Base	319 488 (312 kbytes)	3488 (3.4 kbytes)	322 976 (315.4 kbytes)
List	827 392 (808 kbytes)	8720 (8.5 kbytes)	836 112 (816.5 kbytes)
Vector	376 832 (368 kbytes)	4208 (4.1 kbytes)	381 040 (372.1 kbytes)

REFERENCES

1. Musser D. *Generic Programming*. www.cs.rpi.edu/~musser/gp.
2. Stroustrup B. *The C++ Programming Language* (3rd edn). Addison-Wesley: Reading, MA, 1997.
3. Musser D, Saini A. *STL Tutorial and Reference Guide*. Addison-Wesley: Reading, MA, 1996.
4. Sun Microsystems, Inc. *Java Standard Edition Platform Documentation*. java.sun.com/docs/index.html.
5. Gosling J, McGilton H. *The Java Language Environment: A White Paper*, May 1996. java.sun.com/docs/white/langenv.
6. Hamilton S. The real Stroustrup interview. *IEEE Computer* 1998; **31**(6):110–114.
7. Sun Microsystems, Inc. *JSR #000014: Add Generic Types to the Java Programming Language*. java.sun.com/aboutJava/communityprocess/jsr/jsr_014_gener.html.
8. Bracha G, Odersky M, Stoutamire D, Wadler P. Making the future safe for the fast: Adding genericity to the Java programming language. *Proceedings of the 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*. ACM Press: New York, 1998; 183–200.
9. MIT. *PolyJ: Java with Parameterized Types*. www.pmg.lcs.mit.edu/polyj.
10. Myers A, Bank J, Liskov B. Parameterized types for Java. *Proceedings of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press: New York, 1997; 132–145.