Ahmed Khalifa

Generics: Java vs. C++

Bachelor's Thesis

Computer Science and

Information Systems

2.2.2005

University of Jyväskylä

Department of Computer Science and Information Systems

Jyväskylä

ABSTRACT

Khalifa, Ahmed Aziz
Generics: Java vs. C++ / Ahmed Khalifa.
Jyväskylä: University of Jyväskylä, 2005.
28 p.
Bachelor's Thesis

Generic programming is a widely appreciated and strongly affecting paradigm in software development. A number of programming languages now support generic programming, such as C++, Ada, and Haskell. *Generics* are the most recent enhancement to the type system of the Java programming language. This advanced feature of Java 1.5 is most notably similar to *templates* in C++, as some might like to claim. Templating, genericity, parameterized types, or parametric polymorphism refer to the same technique; basically, instantiating versions of classes using some type as a parameter, such as integer, float, or a class. This thesis work gives a comparison of generics in C++ against the newly introduced genericity extension to the Java programming language.

KEYWORDS: Generics, templates, Generic programming, C++, Java.

Table of Contents:

1. INTRODUCTION

Support for object-oriented programming has become an integral part of mainstream languages, and more recently generic programming has gained widespread acceptance. A natural question is how these two paradigms, and their underlying language mechanisms, should interact. (Järvi et al., 2003.) For example, C++ supports generic programming by providing parametric polymorphism via templates in addition to utilizing some other object-oriented features, such as function overloading. Java is an example of a language that utilizes subtyping to constrain type parameters.

The degree to which a language could support generic programming is varying from language to another. Some languages provide sufficient support merely for implementing type-safe polymorphic containers; some others went beyond what is actually needed for developing such containers. For example, whereas C++ class templates are enough for implementing polymorphic containers, function templates represent generic algorithms in a way by which the scope of C++ genericity became actually wider.

Before JDK 1.5, Java had no support for parameterized types in spite of the fact that the Collections Framework was introduced in JDK 1.1, and then redesigned more thoroughly in JDK 1.2. The fact is, for the purpose of making the Collections Framework a general-purpose tool, collections were designed to hold handles to objects of type ***Object***, which is the root of all ***classes*** (Eckel, 1998). As a result, the type information will be lost as soon as an object is shoved into a collection, since the only thing a collection knows it holds is a handle to an ***Object*** (ibid.). Therefore, it is essential to perform a cast to the correct type before an object can be used (ibid.). That is, there had been collections, or containers, as well as the technique to patronize a deceptive

genericity. JDK 1.5 introduces generics as a new extension to Java. Certainly, Java generics provided additional compile-time type safety to the Collections Framework, but the fact is that the technique of taking ***Object*** handles provided Java genericity with one of its defining characteristics, relative to C++, which is the single compilation of the generic type declaration; and therefore, Java generics are restricted to use only classes and interfaces as type arguments. So much unlike C++ templates, as it will be explained in the subsequent sections.

Due to the fact that generic programming is attaining momentum day after day, this thesis work is trying to clarify and achieve perception of the language issues in terms of generic programming. In doing so, the above mentioned two widely used and appreciated programming languages are to be compared in an attempt to present the language features which are necessary to support generic programming. The comparative study of this thesis work is limited only to the new extension to Java with type parameters, against C++ templating.

The next section provides background information about generic programming and its terminology. In section 3, the paper would deal more with the comparison intention of the study, trying to find out how far object-oriented language features would interact with generics to provide a degree of support for generic programming. Aside from the one-to-one comparison, section 4 describes two other language features, a specifically C++ one, and the other is Java related. Then finally the conclusion.

2. GENERIC PROGRAMING

Generic programming can simply be described as a programming technique or pattern that lets algorithms to be coded for one time, and then be used when needed, time and again, using data types arbitrarily as parameters. This is the so-called code reuse or software reuse that generic type definitions can aid. Typically, generic programming involves type parameters for data types and functions (Garcia et al., 2003). For example, a possible declaration to create a queue using generics would be as ***Queue<Type>***; then it could be instantiated as ***Queue<int>*** or ***Queue<Employee>***. So, the queue container could be handled with whatever type is defined. However, there is no one concrete definition associated with generic programming, definitions vary. While it is true that type parameters are required for generic programming, there is much more to generic programming than just type parameters (ibid.).

Musser (2003) defines generic programming as "programming with concepts," where a ***concept*** is defined as a family of abstractions that are all related by a common set of requirements. A large part of the activity of generic programming, particularly in the design of generic software components, consists of concept development--identifying sets of requirements that are general enough to be met by a large family of abstractions but still restrictive enough that programs can be written that work efficiently with all members of the family. (ibid.)

In generic programming parlance, the requirements on an argument (or actual parameter) are a concept (Mueller & Jensen, 2004). Generic algorithms are fundamentally concerned with the capability of an actual parameter to satisfy specific requirements. These requirements can be as simple as being able to be compared to other values of the same type or as complicated as having to define many operations and other types that work with the value during the

execution of the algorithm (ibid.). Types that meet the requirements of a concept are said to ***model*** the concept (Garcia et al., 2003). A concept may incorporate the requirements of another concept, in which case the first concept is said to ***refine*** the second (ibid.). At a high level, the concept/model/refine relationship is analogous to the class/instance/inherit relationship in object-oriented programming (Mueller & Jensen, 2004).

Due to the way a concept could be modeled, by whichever concrete type that satisfies its requirements, concept related algorithms should be implemented in a way that allows them to function using multiple types. Thus, generic algorithms must be polymorphic (Garcia et al., 2003). Not all languages have explicit features to support concepts; this is the case with C++. However, by reasons of the flexibility of C++ templates, programmers are likely to code in nearly the way that it would be if the feature were supported. The following two examples from Garcia et al. (ibid.) show how a generic algorithm could be realized as a ***function template*** and how template parameters could be named to identify a concept such as ***Comparable***, which is defined to represent types that may be used with the generic algorithm ***pick:***

```
template <class Comparable>
const Comparable&
pick(const Comparable& x, const Comparable& y) {
if (better(x, y)) return x; else return y;
}
```

Example 1. (Garcia et al., 2003, p. 117).

So, for the concept ***Comparable*** to say, for example, "if the arguments given to ***pick*** are of type ***int***, use this implementation of the ***better*** function; if they are of type ***Apple***, use that other implementation of the ***better*** function", the following code can be implemented:

```
bool better(int i, int j) { return j < i; }

struct Apple {
 Apple(int r) : rating(r) {}
 int rating;
};
bool better(const Apple& a, const Apple& b)
{ return b.rating < a.rating; }

int main(int, char*[]) {
int i = 0, j = 2;
Apple a1(3), a2(5);

int k = pick(i, j);
Apple a3 = pick(a1, a2);

return EXIT SUCCESS;
}
```

Example 2. (Garcia et al., 2003, p. 117).

Thus, ***Apple*** is said to be ***modeling*** the ***Comparable concept*** implicitly via the existence of the ***better*** function for the ***Apple*** type.

Standard practice, in the case of C++, is to express ***concepts*** in documentation. For example:

```
concept Comparable :
bool better(const Comparable&, const Comparable&)
```

For languages that have explicit features to support concepts, concepts are used to ***constrain*** type parameters (ibid.).

The following inclusive definition, by Jazayeri et al. (1998), demonstrates the efficiency restrictions necessary for implementing generic programming:

"Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. Key ideas include:

- Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as interoperable as possible.

- Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.

- When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.

- Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient."

Jazayeri et al (1998) p. 2

While Garcia et al. (2003) state that they are stimulated by the STL force to choose citing such a definition that takes a broader view of generic programming, Mueller & Jensen (2004) deem the definition as being derived, to a very great degree, from the C++ approach to generic programming, which is basically concerned with generating instances of algorithms at compile-time.

On the other hand, the current fact is that the JDK 1.5 permits a type or method to function on objects of different types with compile-time type safety. This recent extension to the Java programming language provides additional compile-time type safety to the Collections Framework and gets rid of the tedious unpleasant work of casting. This could be clarified with examples from Bracha (2004) as follows:

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

Example 3. (Bracha, 2004, p. 2)

The cast on line 3 is essential. The compiler can only guarantee that an ***Object*** will be returned by the iterator. To ensure the assignment to a variable of type ***Integer*** is type safe, the cast is required. (ibid.)

What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics. (ibid.) Another version of the above example using generics follows:

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); //2'
Integer x = myIntList.iterator().next(); // 3'
```

Example 4. (Bracha, 2004, p. 2)

The type declaration for the variable ***myIntList*** specifies that this is not just an arbitrary ***List***, but a ***List*** of ***Integer***, written ***List<Integer>***. We say that ***List*** is a generic interface that takes a *type parameter* - in this case, ***Integer***. We also specify a type parameter when creating the list object. Also the cast is gone on line 3'. (ibid.)

In addition to the primary aspects of generic programming, i.e., generic algorithm, concepts, refinement, modeling, and constraints, Garcia et al. (2003) came up with the following eight supplemental language features, which are used to support generic programming. Table 1 shows the level of support for these features in C++ and Java.

Multi-type concepts: indicates whether multiple types can be simultaneously constrained. In Java generics, concepts are approximated by interfaces. The modeling relation between a type and a concept is approximated by the subtype relation between a type and an interface. The refinement relation between two concepts is approximated by interface extension. (ibid.) A constraint signifies a requirement that a type should satisfy so that a generic

type can be constructed. Constraints based on interfaces and subtyping, however, cannot correctly express constraints on multiple types (ibid.).

	C++	Java Generics
Multi-type concepts	-	○
Multiple constraints	-	●
Associated type access	●	◉
Retroactive modeling	-	○
Type aliases	●	○
Separate compilation	○	●
Implicit instantiation	●	●
Concise syntax	●	◉

Table 1: The level of support for important properties for generic programming in C++ & Java generics.

- ● Fully supported.
- ◉ Partially supported.
- ○ Not supported.
- \- While C++ does not support the feature, one can still program as if the feature were supported due to the flexibility of C++ templates. (Garcia et al., 2003, p.117.)

Multiple constraints: indicates whether more than one constraint can be placed on a type parameter. In Java, type parameters can be constrained using subtyping, including multiple constraints on one parameter. (ibid.) For a given type parameter, any number of interfaces can be specified as constraints. In the following example, the ***T*** type parameter has two interface constraints:

```
public class java_multiple_constraints {
  public static <T extends I_fac_1, I_fac_2>
  void generic_algorithm(T a) { /* ... */ }
}
```

Example 5.

Associated type access: rates the ease in which types can be mapped to other types within the context of a generic function (ibid.). The *associated types* of a concept specify mappings from the modeling type to other collaborating types (such as the mapping from a container to the type of its elements). In practice it is convenient to separate the data types of a module into two groups: the main types and the associated types. An example of this is an iterator (the main type)

and its element type (an associated type). Associated type constraints are a mechanism to encapsulate constraints on several functionally dependent types into one entity. C++ can represent associated types as member typedefs or *traits classes*[1] but cannot express constraints on them. Java does not provide a way to access and place constraints on type members of generic type parameters. However, associated types can be emulated using other language mechanisms. A common idiom used to work around the lack of support for associated types is to add a new type parameter for each associated type. The main problem with this technique is that it fails to encapsulate associated types and constraints on them into a single concept abstraction. Every reference to a concept, whether it is being refined or used as a constraint by a generic function, needs to list all of its associated types, and possibly all constraints on those types. In a concept with several associated types, this becomes burdensome. (Järvi et al., 2003.)

Retroactive modeling: indicates the ability to add new modeling relationships after a type has been defined. Type arguments to a generic algorithm must model the concepts that constrain the algorithm's type parameters. To establish modeling relationships, Java generics use subtyping at the point of class definition; C++ provides no language feature for establishing modeling relationships, type arguments are required only to provide the functionality that is used within a function template's body. The modeling mechanism used for Java relies on named conformance. An explicit declaration links a concrete type to the concepts it models. Once a type is defined, the set of concepts that it models is fixed. Without modification to the definition, modeling relationships cannot be altered. (Garcia et al., 2003.)

[1] Introduced in Myers (1995).

Type aliases: indicates whether a mechanism for creating shorter names for types is provided. Java does not support type aliases, such as the ***typedef*** statement in C++. Type aliasing is a simple but crucial feature for managing long type expressions commonly encountered in generic programming. In addition to avoiding repetition of long type names, type aliases are useful for abstracting the actual types without losing static type accuracy. (ibid.)

Separate compilation: indicates whether generic functions are type-checked and compiled independently from their use. In Java, Generic components and their uses can be compiled separately. A generic algorithm's implementation is type-checked once, rather than for each instantiation as in C++. Uses of a generic algorithm are checked to ensure that the concept requirements are satisfied, independently from the implementation of the generic algorithm. This allows for faster compilation, as well as for catching type errors as early as possible, and is a major advantage relative to C++. (ibid.)

Implicit instantiation: indicates that type parameters can be deduced without requiring explicit syntax for instantiation (ibid.). That is, the actual type parameters will be deduced based on the types of the actual arguments. Thus, calling a template function or generic method is not differing from calling a non-template function or non-generic method.

Concise syntax: indicates whether the syntax required to compose layers of generic components is independent of the scale of composition (ibid.). C++ and Java have concise syntaxes. However, since everything in Java is a class, lengthy syntax may sometimes be inevitable. An example of a newly introduced feature for Java generics, which contributes to a more concise syntax, is "*Wildcards*". Furthermore, JDK 1.5 makes as if it allows built-in types as type arguments because of the Autoboxing/Unboxing feature.

3. JAVA VS. C++

For the reason that Java generics represent a new extension to the language, a simple Java generic is to be defined first in the next example, followed by its corresponding C++ version; then comparing the language features necessary to implement generic programming is to be done subsequently.

```
public class List<T> {
  public void add(T  i) { /* … */ }
  public int size() { /* … */ }
  public T firstElement() { /* … */ }
// …
}
```

Example 6.

```
template<class T> class List {
public:
      void add(T  i) { /*  … */ }
      int size() { /*  … */ }
      T firstElement() { /*  … */ }
// …
};
```

Example 7.

In both Java and C++, ***List*** can be instantiated like this:

```
List<Employee> employeeList;
```

Example 8.

But only in C++, not in Java, ***List*** can be instantiated like this:

```
List<int> intList;
```

Example 9.

Although Java has the primitive data type ***int,*** only its *"wrapper"* class ***Integer,*** which is contained in the Java standard library, can be passed as a type argument, since primitive data types are not references. Only classes and

interfaces can be used as type arguments, since they are, in fact, inherited from the type ***Object***. Therefore, all primitive data types have their corresponding *"wrapper"* classes. As mentioned above, the early technique of taking only ***Object*** handles into a collection led to the Java's capability to perform generic type declaration's single compilation. However, this would inevitably result in verbose code. The following example from Garcia et al. (2003) shows the syntax for setting the weights of a graph edge using the *"wrapper"* classes, it is:

weight_map.set(new adj_list_edge(new Integer(3), new Integer(5)),
new Double(3.4))

Example 10. (Garcia et al., 2003, p. 128).

If built-in types were allowed as type arguments, the code would be simpler (ibid.):

weight_map.set(new adj_list_edge(3, 5), 3.4)

Example 11. (Garcia et al., 2003, p. 128).

According to Bracha and Bloch (2002), the Autoboxing/Unboxing feature, which automates the conversion process from primitive data types to their *"wrapper"* classes and vice versa, would solve some of these problems.

While Java interface, which are the means to approximate concepts, do not allow method bodies inside them, and while methods in Java can only be defined within a class, C++ template classes allow member functions to be defined inside and outside them. When such a member is defined outside its class, it must explicitly be declared a template (Stroustrup, 1997). For example:

template<class T> T List<T> :: firstElement() { /…*/ }*

Example 12.

Java interface, as is often the case, establishes the form for a class; the keyword ***implements*** is used as usual to make the class that conforms to the interface. The interface name, that should be prefixed by the keyword ***implements*** in the

class definition, must be postfixed—only if the class is not generic—by a type argument bracketed by <>, in the case of genericity as in example 13.

After all, and from a generic programming perspective, the language features used to support generic algorithms are ***function templates*** in C++, and ***generic methods*** in Java. The Java keyword ***implements*** is used to ***model concepts***, whose notion is supported by the keyword ***interface.*** In C++, any other primary aspect of generic programming than generic algorithms can be expressed in documentation as a standard practice.

3.1 Defining Generics

One can say that even before the introduction of the new Java generics, Java's ***interface*** keyword, which basically contributes to an alternative form of subtyping, can be looked at as an early preparing step towards the notion of generic programming ***concepts.*** The ***interface*** keyword can simply be used instead of the ***class*** keyword in order to create what we can call a pure abstract class, whose function is to establish a policy or protocol between classes. Example 13, which is the Java version of examples 1 & 2, shows how the ***concept Comparable*** could be realized as an ***interface*** and how ***Apple*** could ***model*** the ***Comparable concept.*** As for the interface ***Comparable***, according to Bracha (2004), other than what is in angle brackets is not new to the Java programming language; it is the declaration of the *formal type parameter* of the interface **Comparable.** Then, throughout the generic declaration, the type parameter can be used in any place in which ordinary types could be used.

So, what makes C++ templates such a flexible feature for generic programming, while the language itself does not demonstrate any level of support for generic programming ***concepts*** and ***modeling***? It is the magic of *overloaded function names*, as is the case in example 2 with the ***better*** function, whose name is used

for operations on two different types (***int*** & ***Apple***) to ***model*** implicitly the ***Comparable concept***. On the other hand, due to the way generic programming is handled in Java, and due to the fact that everything is a class in Java, method overloading cannot actually go beyond its prime necessity as an object-oriented programming mechanism.

```
interface Comparable<T> {
  boolean better(T x);
}
class pick {
  static <T extends Comparable<T>>
  T go(T a, T b) {          // modified[1]
     if (a.better(b)) return a; else return b;
  }
}
class Apple implements Comparable<Apple> {
  Apple(int r) { rating = r; }
  public boolean better(Apple x)
     { return x.rating < rating;}
  int rating;
}
public class Main {
   public static void main(String[] args) {
     Apple a1 = new Apple(3),
           a2 = new Apple(5);
    Apple a3 = pick.go(a1, a2);
  }
}
```

Example 13. (Garcia et al., 2003, p. 118).

[1] With reference to other examples in Garcia et al. (2003), as well as the method call in this example, the method name "go" is the correct replacement of the method name "pick" in the original example.

The declaration of a generic in Java is never actually expanded like in this example:

```
Interface IntegerComparable {
  boolean better(Integer x);
}
```

Example 14.

There aren't multiple copies of the code: not in source, not in binary, not on disk and not in memory (ibid.). This is unlike a C++ template, as explained later on. A generic type declaration is compiled once and for all, and turned into a single class file, just like an ordinary class or interface declaration (ibid.).

Type parameters are analogous to the ordinary parameters used in methods or constructors. Much like a method has *formal value parameters* that describe the kinds of values it operates on, a generic declaration has formal type parameters. When a method is invoked, *actual arguments* are substituted for the formal parameters, and the method body is evaluated. When a generic declaration is invoked, the actual type arguments are substituted for the formal type parameters. (ibid.)

3.2 Subtyping

Against the immediate apprehension by one's object-oriented sense that a ***List<Manager>*** is a ***List<Employee>***, this is a serious logical error based on the assumption that ***Manager*** is a subtype of ***Employee.*** This holds in both C++ and Java. It is logically unaccepted to use a list of ***Managers*** as a list of ***Employees*** because the list of ***Managers*** guarantees that the members of the list are ***Managers***, while the list of ***Employees*** does not. The following C++ example from Stroustrup (1997) explains how the compiler supports such a restrictive rule:

```
class Shape { /*…*/ };
class Circle : public Shape { /*…*/ };
class Triangle : public Shape { /*…*/ };

void f(set<Shape*>& s)
{
  // …
  s.insert(new Triangle());
  // …
}
void g(set<Circle*>& s)
{
  f(s);      // error, type mismatch: s is a set<Circle*>, not a set<Shape*>
}
```

Example 15. (Stroustrup, 1997, p. 348, 349).

This won't compile because there is no built-in conversion from *set<Circle*>&* to *set<Shape*>&*. Nor should there be. (ibid.) The same applies to Java.

3.2.1 Subtype-based Constraints

The writing of generic code is partially supported by subtype polymorphism. With subtype polymorphism, one can define a family of related types: a supertype defines the behavior common to all its subtypes, which extend or specialize that behavior. Programs are then generic with respect to the defined family, since code written in terms of an object of type T is also usable for an object whose type S is a subtype of T. Alternatively, A where clause lists the names and signatures of required methods for the parameters. For example, the where clause in the specification of set indicates that any legal parameter must have an equal method. Where clauses allow the compiler to type check instantiations and implementations independently; thus this approach supports separate compilation. (Day et al., 1997.) Neither C++ nor Java has where clauses.

One particular design option, that of using subtyping to constrain the type parameters of generic functions, has been chosen for the generics extensions to

Java. An alternative to subtype-based constraints is to use concepts, as they are called in the C++ generic programming community. (Järvi et al., 2003.)

C++ does not have means for constraining template parameters or for separate type checking. To check that an actual instantiation of a parameterized routine or type is correct, the compiler rewrites the body of the routine or type, replacing the type parameter with the actual type, and then checks the result (Day et al., 1997). Utilizing subtyping for constraining type parameters in Java demands explicit definitions of type interfaces in order for the compiler to check all method calls and to recognize the type hierarchy.

According to Stroustrup (1994) as cited in Garcia et al. (2003), several mechanisms were proposed for constraining template parameters, including subtype-based constraints. All proposed mechanisms were found to either undermine the expressive power of generics or to inadequately express the variety of constraints used in practice. (ibid.) However, according to Siek and Lumsdaine (2000), techniques for checking constraints in C++ can be implemented as a library. These techniques, however, are distinct from actual language support and involve insertion of what are essentially compile-time assertions into the bodies of generic algorithms. (Garcia et al. 2003.)

3.3 Generic Algorithms

Generic algorithms are most straightforwardly expressed as functions. In languages without functions, they can be emulated by static methods. There are two choices for how to parameterize the method: either parameterized methods in non-parameterized classes or non-parameterized methods in parameterized classes. The first alternative has the advantage of implicit instantiation, while the second alternative requires the more verbose explicit specification of type arguments. (Garcia et al. 2003.)

In example 13, the generic algorithm is realized as a static parameterized method, called ***go***, in a non-parameterized class, called ***pick***. Example 1 demonstrates the C++ version of the generic algorithm, which is realized as a function template called ***pick***. As mentioned above, C++ has no built-in support that could allow templates to be separately type checked, and therefore type checking can only be carried out following a referral from an object instantiation at each call site. In Java, both arguments to generic algorithms and the bodies of those algorithms are checked separately against the concept requirements, leading to good separation between types used in generic algorithms and the implementations of those algorithms (ibid.).

Generic programming is basically supported in C++ by bringing templates and function overloading into harmonious union, as explained in subsection 3.1. This actually facilitates a broader version of generic programming than just implementing containers. Although generics were added to Java to provide support for implementing type safe containers, it could provide enough support for generic programming in some cases, such as the generic graph library, according to Garcia et al. (ibid.).

4. LANGUAGE SPECIFICS

This section describes two features, each of which adds a distinctive quality to its language. First, C++ template specialization, and then the Java generics' wildcards.

4.1 User-defined Specialization in C++ Templates

As a matter of fact, specialization had to be available in C++ to exercise control over what is known as template-generated code in order to prevent critical enlargements in the entire code. By default, a template gives a single definition to be used for every template argument (or combination of template arguments) that a user can think of, and then the compiler replicates the code for template functions using the actual type at each call site, which is good for run-time performance (Stroustrup, 1997). Yet, providing alternative definitions of a template for specific data types is feasible. The compiler can then determine the appropriate template definition according to the template argument at the call site. Such alternative definitions of a template are called *user-defined specializations*, or simply, *user specializations* (ibid.).

The following couple of subsections explain, with reference to examples from Stroustrup (ibid.), how containers such as ***Vectors*** of pointers can share a single implementation. Afterwards, template function specialization is to be explained. Since the specialization should be of an existing template, a *general vector type* is to be defined first:

```
template<class T> class Vector {        // general vector type
        T* v;
        int sz;
public:
        Vector ();
        Vector (int);

        T& elem (int i)         { return v[i]; }
        T& operator[] (int i);

        void swap (Vector&);
        // …
};
```

Example 16. (Stroustrup, 1997, p. 341).

4.1.1 Complete Specialization[1]

According to the fact that examples are the best way to convey ideas, we first define a *complete specialization* of ***Vector*** for pointers to ***void***. That is, a *specialization* that has no template parameter to specify or deduce as we use it:

```
template<> class Vector<void*> {
  void** p;
  // …
  Void*& operator[] (int);
};
```

Example 17. (Stroustrup, 1997, p. 341).

The ***template<>*** prefix says that this is a specialization that can be specified without a template parameter. The template arguments for which the specialization is to be used are specified in <> brackets after the name. That is, the ***<void*>*** says that this definition is to be used as the implementation of every ***Vector*** for which ***T*** is void*. (Stroustrup, 1997.) So, the following statement is an instance with a definition of its own:

[1] This is according to Stroustrup (1997); some authors call it "explicit specialization", some others call it "full specialization".

```
Vector<void*> vpv;
```

Example 18. (Stroustrup, 1997, p. 342).

The above specialization is to be used in the next example as the common implementation for all ***Vectors*** of pointers, although it is not a prerequisite to have defined a *complete specialization* before defining *a partial specialization.*

4.1.2 Partial Specialization

A specialization is likely to be defined for a wide variety of template arguments. For example, a specialization for ***Vector<T*>*** can be defined for any ***T;*** this is known as *partial specialization,* while ***<T*>*** is the specialization pattern. That is, for ***Vector<char*>, T*** is ***char*** not ***char****. So, to define a specialization that can be used for every ***Vector*** of pointers, we need partial specialization:

```
template<class T> class Vector<T*> : private Vector<void*> {
public:
   typedef Vector<void*> Base;

   Vector() : Base() { }
   explicit Vector (int i) : Base(i) { }

   T*& elem (int i) { return static_cast<T*&> ( Base :: elem(i) ); }
   T*& operator[] (int i) { return static_cast<T*&> ( Base :: operator[] (i) ); }

   // …
};
```

Example 19. (Stroustrup, 1997, p. 342).

Given this partial specialization of ***Vector***, we have a shared implementation for all ***Vectors*** of pointers. The ***Vector<T*>*** class is simply an interface to ***void**** implemented exclusively through derivation and inline expansion. (Stroustrup, 1997.)

4.1.3 Template Function Specialization

One can declare several function templates with the same name and even declare a combination of function templates and ordinary functions with the same name. When an overloaded function is called, overload resolution is necessary to find the right function or template function to invoke. (Stroustrup, 1997.) So, straightforwardly, we can not partially specialize template functions because they can simply overload; they can only be fully specialized.

```
//      A function template
template<class T> bool less(T a, T b) { return a<b; }                    // (1)

//      Different forms of declarations for specializing (1) for const char*
template<> bool less<const char*> (const char* a, const char* b)      // (2)
{
return strcmp ( a, b ) < 0;
}

template<> bool less<> (const char* a, const char* b) { /* … */ }      // (3)

template<> bool less(const char* a, const char* b) { /* … */ }         // (4)
```

Example 20. (Stroustrup, 1997, p. 344).

The ***template<>*** prefix as well as the ***<const char*>*** after the name in (2) declare the same as for *class template complete specialization*. However, the declarations in (3) and (4) are equivalent to (2) since the template argument can be deduced from the function argument list.

So, what would be the resolution in the case of defining either a template function or ordinary function to overload with (1) in the existence of (2)? The ordinary non-template function which matches the parameter types, has the highest priority. Template functions will then be compared to select the best match regardless of any associated specializations. Finally, specialization declarations will be looked at, if there is any, according to the selected template function.

4.2 Wildcards with Java Generics

As demonstrated above, while ***Object*** is the base class of all ***classes*** in Java**,** ***Collection<Object>*** cannot be a supertype of all kinds of collections. According to Bracha (2004), the supertype of all kinds of collections is ***Collection<?>*** and is pronounced "collection of unknown", that is, a collection whose element type matches anything (ibid.). Here is a routine to print all elements in a collection; as for the new ***for*** loop, whose syntax is ***for (variable : collection)***, it iterates over collections to get elements in a sequential manner:

```
void printCollection(Collection<?> c) {
        for (Object e : c) {
                System.out.println(e);
} }
```

Example 21. (Bracha, 2004, p. 5).

Certainly, the types of all elements to be read from ***c*** are inherited from ***Object***; and therefore, they can be read as of type ***Object***. However, we cannot add arbitrary objects to a collection of unknown. ***?*** is not a supertype, it symbolizes an unknown type.

4.2.1 Bounded Wildcards

As said in subsection 3.2, ***List<Manager>*** is not a ***List<Employee>***. We should consider first the following example:

```
public void printAll(List<Employee> employees) { /* … */ }
```

Example 22.

We cannot call ***printAll()*** on lists of other than ***Employee***; for example, ***List<Manager>***, although we know that ***Manager*** is a subtype of ***Employee***. According to Bracha (2004), *Wildcards* can ease the matter of allowing such methods to accept collections of any subtype of ***Employee***, for example. Accordingly, the ***List<Employee>*** in the above example could be replaced with

a *bounded wildcard* such as ***List<? Extends Employee>*** so that ***printAll()*** can accept lists of any subtype of ***Employee.*** That is, it is possible now to call ***printAll()*** on a ***List<Manager>.***

What is really interesting here is that we know that ***?*** symbolizes an unknown type, which is in fact a subtype of ***Employee***. We say that ***Employee*** is the *upper bound* of the wildcard. The unknown type could be ***Employee*** itself, or some subtype. Since we do not know what type it is, we do not know if it is a supertype of ***Manager***; it might or might not be such a supertype, so it is not safe to add a ***Manager*** to a ***List<? Extends Employee>.*** (Bracha, 2004.)

5. CONCLUSION

It is difficult to know if one should describe Java as a dialect of C++. Although the two have superficial surface similarities, the underlying differences are substantial enough to warrant calling Java an entirely new language. (Budd, 1997.) This is very much the case when it comes to generic programming; Java is not a dialect of C++. It is hard to tailor a notion for such a derivation relationship reasonably or to make it make sense in terms of genericity; neither with the previous Java related approaches to cast objects at run-time, nor with the newly introduced compile-time type safety.

According to Stroustrup (1994), templates were considered essential for the design of proper container classes. Greater emphasis was placed on clean and consistent design than restriction and policy (Garcia et al., 2003). For example, although function templates are not necessary to develop type-safe polymorphic containers, C++ has always supported classes and standalone functions equally; supporting function templates in addition to class templates preserves that design philosophy (ibid.). That is why C++ generics seem to be, by design, going beyond the limits of just implementing containers, despite the fact that providing a sufficiently general facility for defining such container classes was the prime motivation.

Likewise, the new Java extension is intended, for the most part, to support polymorphic containers. Although the early introduction of the Collections Framework led to a compilation advantage when it came to genericity, the type system remained with limitations that would result in wordy and inefficient code. However, the Autoboxing/Unboxing feature would solve some of the problems related to wordiness.

While Java uses subtyping to constrain generics, which is sufficient for polymorphic containers, C++ rejected this approach because it lacks expressiveness and flexibility (Garcia et al., 2003). Moreover, constraints based on interfaces and subtyping, in Java, cannot correctly express constraints on multiple types. That is, to support a flexible version of generic programming, a language needs to be less restricted than a mere object-oriented programming language. Both C++ and Java are object-oriented programming languages, but the flexibility and expressiveness that C++ provides for adopting generic programming are in fact taking the language beyond being described as an object-oriented programming language.

While Java is a mere object-oriented programming language that has been extended with generics where everything is a class, C++ can be described as a language that provides a great support for imperative, object-oriented, and generic programming.

Supporting powerful and flexible versions of generic programming needs language designs that are not restricted or confined within the requirements of mainstream languages.

REFERENCES

Bracha, G. & Bloch, J. 2002. JSR 201: Extending the Java Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import, http://www.jcp.org/en/jsr/detail?id=201. [Checked: 2.2.2005]

Bracha, G., Cohen, N., Kemper, C., Marx, S., et al. 2001. JSR 14: Add Generic Types to the Java Programming Language, http://www.jcp.org/en/jsr/detail?id=014. [Checked: 2.2.2005]

Bracha, G. 2004. Generics in the Java Programming Language. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf. [Checked: 2.2.2005]

Budd, T. 1997. An Introduction to Object-Oriented Programming. 2nd Edition, Addison-Wesley.

Day, M., Gruber, R., Liskov, B. & Myers, A. 1995. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In Proceedings of the OOPSLA '95, ACM, New York, 156 – 168.

Eckel, B. 1998. Thinking in Java. Prentice-Hall, Upper Saddle River, New Jersey.

Garcia, R., Järvi, J., Lumsdaine, A., Siek, J. & Willcock, J. 2003. A Comparative Study of Language Support for Generic Programming. In Proceedings of the OOPSLA '03, ACM, New York, 115 – 134.

Jazayeri, M., Loos, R., Musser, D. & Stepanov, A. 1998. Generic Programming. In Followup Report of the Dagstuhl Seminar on Generic Programming, Schloss Dagstuhl, Germany.

Järvi, J., Lumsdaine, A., Siek, J. & Willcock, J. 2003. An Analysis of Constrained Polymorphism for Generic Programming. In Kei Davis and Jörg Striegnitz,

editors, Multiparadigm Programming in Object-Oriented Languages Workshop (MPOOL) at OOPSLA '03, Anaheim, CA.

Mueller, C. & Jensen, S. 2004. The Java Generic Programming System. http://www.osl.iu.edu/~chemuell/classes/b629/GenericJava.pdf.
[Checked: 2.2.2005]

Musser, D. 2003. Generic Programming. http://www.cs.rpi.edu/~musser/gp/.
[Checked: 2.2.2005]

Myers, N. 1995. Traits: a new and useful template technique. C++ Report. http://www.cantrip.org/traits.html. [Checked: 2.2.2005]

Siek, J. & Lumsdaine, A. 2000 Concept checking: Binding parametric polymorphism in C++. In Proceedings of the First Workshop on C++ Template Programming, Erfurt, Germany.

Stroustrup, B. 1994. The Design and Evolution of C++. Addison-Wesley.

Stroustrup, B. 1997. The C++ programming Language. 3rd Edition, Addison-Wesley.