Ahmed Aziz Khalifa

Generics: Ada 95 vs. C++ vs. Java 1.5

Master's Thesis

Computer Science and

Information Systems

1.8.2005

University of Jyväskylä

Department of Computer Science and Information Systems

Jyväskylä

ABSTRACT

Khalifa, Ahmed Aziz
Generics: Ada 95 vs. C++ vs. Java 1.5/ Ahmed Khalifa.
Jyväskylä: University of Jyväskylä, 2005.
73 p.
Master's Thesis

Generic programming is a widely appreciated and strongly affecting paradigm in software development. Genericity has become an integral part of most widely known and used programming languages. Some have just most recently been extended with generics, as is the case with Java 1.5. Some are known to have generics from the very beginning and even before they were extended to support object-oriented programming, as is the case with Ada.

Genericity, templating, parameterized types, or parametric polymorphism refer to the same technique; basically, instantiating versions of an algorithm using built-in types, classes, objects, or program units as parameters. This Thesis work is to provide a comparison between three different language approaches to generic programming. The languages to be compared are Ada, C++, and Java.

The objective of this study is to understand how and why these languages employ certain features with generics to provide a degree of support for generic programming, and to understand the extent to which each language could support a powerful and flexible version of generic programming.

KEYWORDS: Generics, templates, generic programming, C++, Java, Ada.

Table of Contents:

1. INTRODUCTION

In spite of its name, today's software is usually not *soft* enough; adapting it to new uses turns out in most cases to be a harder endeavor than it should be. It is thus essential to find ways of enhancing such software quality factors as extendibility, reusability, and compatibility. (Meyer, 1986.) Genericity has turned out very useful in the development of reusable software. The fact is that languages, such as C++ and Java, claim reuse as one of their goals through object-oriented programming, and therefore the need for generics was primarily raised for the design of proper container classes, such as lists, vectors, and queues. Despite that, it is more appropriate to say that the need for generics was raised to empower the ability to code for reuse, especially when it comes to a language such as Ada; Ada primarily employs generics to support one of its strongest claims, which is the ability of writing general-purpose, reusable software components.

Generic programming can simply be described as a programming paradigm which makes it possible for an algorithm 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. While it is true that type parameters are required for generic programming, there is much more to generic programming than just type parameters (ibid.). There is no one concrete definition associated with generic programming, definitions vary, and can even be tailored to specific language capabilities as need be.

David Musser and Alexander Stepanov developed the methodology of generic programming[1] in the late 1980's and applied it to the construction of sequence and graph algorithms in Scheme, Ada[2], and C. Since then the methodology has evolved to meet the needs of new algorithms and problem domains, and to take advantage of new programming language features, such as templates in C++. (Siek & Lumsdaine, 2004.)

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.

This thesis work is to provide a comparison between three different language approaches to generic programming. The languages to be compared are Ada, C++, and Java. The objective of this study is to expose the true nature of genericity in these languages, to understand how and why these languages employ certain features with generics to provide a degree of support for generic programming, and to find out the extent to which each language could support a powerful and flexible version of generic programming.

Before Java 1.5, Java had no support for parameterized types; so, this study focuses on Java 1.5. As for Ada, Ada 83 is not an object-oriented programming language, by most definitions (Tang, 1992). Although it supports many object-oriented design principles, such as information hiding, encapsulation and reusability, it provides only a primitive form of inheritance and it does not

[1] Musser & Stepanov (1988).
[2] Musser & Stepanov (1987, 1989).

directly support polymorphism and dynamic binding (ibid.). There are a number of important improvements and extensions to the generic model in Ada 95 (Barnes, 1995). For example, a generic formal parameter can stand for a value, variable, type, or subprograms in Ada 83, but not for a package (Cohen, 1996). Moreover, there were neither tagged types nor classwide types in Ada 83. In Ada 95, the extensions are mainly concerned with providing appropriate new parameter mechanisms to match the additional functionality provided by tagged and other new types (Barnes, 1995). Because of the above and some other substantial differences between Ada 83 and Ada 95, this study focuses on Ada 95.

One can easily understand how such a powerful language as Ada could have suffered along the years facing an irresistible challenge from C and C++. Moreover, when Ada 95 gave stronger signs of possible challenge to C++ by supporting object-oriented programming, it was the time for Java. However, when it comes to genericity, Ada led and the others are following with less efficient and even less safe features. The reasons behind that could be summarized in the following three points:

- Generics, or templates, were considered essential in C++ for the design of proper container classes, as stated explicitly in Stroustrup (1994). So, the need for a sufficiently general facility for defining container classes necessitated the design of template, but it was not considered to be a defining feature for the language or a contribution to the modern software design needs as is the case with object-oriented features, although templates then proved that it could provide unexpected great functionalities or even a sublanguage, such as *template metaprogramming*.
- To support a flexible version of generic programming, a language needs to be less restricted than a mere object-oriented programming language. Having this in mind, one can guess that Java creators were trying to avoid

genericity by first introducing container classes, which hold only handles to objects, using *wrapper classes* and casting before they have finally extended the language with generics.

- Genericity in Ada represents one of the defining features of the language, by which the ability to code for reuse in Ada is one of its strongest claims. According to Tang (1992), Ada is carefully designed to meet the software needs of the 1990s. From the beginning, Ada seemed to be designed to play a fundamental role in shaping the designers' way of thinking, although it did not provide support for object-oriented programming. This is because the value of all object-oriented design principles, such as inheritance and polymorphism, was not well-known when Ada was designed. However, as an object-based programming language, generic subprograms and generic packages were able to take objects, data types, and other subprograms as parameters.

The aim of this thesis work is not to show that one language is superior to the others, they are each different and stand on their own merits; however, due to the design nature of some languages, it might be inevitable that some language would seem to support more powerful version of generic programming.

For each individual language, each of the next three sections provides the necessary information about the essential language features required for the construction of generic components. Using a single simple example implemented in each of the target languages, section 5 describes the methodology and terminology of generic programming; also the language features identified by Garcia et al. (2003) as being necessary for the development of high quality generic libraries are to be described as well. Section 6 discuses and evaluates some more genericity related issues. Finally, section 7 provides the concluding remarks.

2. ADA GENERIC UNITS

Ada is a general-purpose computer programming language developed by the United States Department of Defense. It embodies many modern software development principles, and is carefully designed to meet the software needs of the 1990s. (Tang, 1992.)

Generic units are templates from which several analogous subprograms and packages can be produced without duplication of efforts (Cohen, 1996). This section will provide details about constructing and instantiating generic units, and about the different categories of generic formal parameters.

2.1 A Simple Generic Package

First, we consider a package for stacks holding up to 10 integers. The package declaration might read as follows (usually, such a package is made up of two parts: the declaration and the body. See example 2.2):

```
package  Stack_Package  is
    type  Stack_Type  is private;
    procedure Push (Item: in Integer; Stack: in out Stack_Type);
    procedure Pop (Item: out Integer; Stack: in out Stack_Type);
private
    type Item_List_Type is array (1 .. 10) of Integer;
    type Stack_Type is
        record
            Top          : Integer range 0 .. 10 := 0;
            Elements  : Item_List_Type;
        end record;
end Stack_Package;
```

Example 2.1 (Slightly adapted from Cohen, 1996, p. 28).

So, to create a similar package for stacks that can hold a number of items of some other data type, one only needs to change the type of the *Item* parameter

of *Push* and *Pop*, the definition of the type *Item_List_Type*, and maybe the number 10, which is the stack size, to be as needed. Then, it is clearly a duplication of efforts. Instead, a generic package could allow the programmer to turn the algorithm into a general-purpose template for such packages so that the algorithm would be reused with arbitrary types and arbitrary integer values for the stack size. The generic package declaration and body might read as follows:

```
generic
    Stack_Size: in Integer;
    type  Item_Type  is private;
package  Generic_Stack_Package  is
    type  Stack_Type  is private;
    procedure Push (Item: in Item_Type; Stack: in out Stack_Type);
    procedure Pop (Item: out Item_Type; Stack: in out Stack_Type);
private
   type Item_List_Type is array (1 .. Stack_Size) of Item_Type;
   type Stack_Type is
         record
            Top          : Integer range 0 .. Stack_Size := 0;
            Elements  : Item_List_Type;
        end record;
end Generic_Stack_Package;

package body Generic_Stack_Package  is
    procedure Push (Item: in Item_Type; Stack: in out Stack_Type) is
    begin
        Stack.Top := Stack.Top +1;
        Stack.Elements (Stack.Top) := Item;
    end Push;
    procedure Pop (Item: out Item_Type; Stack: in out Stack_Type) is
    begin
        Item := Stack.Elements (Stack.Top);
        Stack.Top := Stack.Top - 1;
    end Pop;
end Generic_Stack_Package;
```

Example 2.2 (Cohen, 1996, p. 34, 35).

The word **generic** in the first line indicates that *Generic_Stack_Package* is not really a package, but a *template* for a package. The next two lines declare *Stack_Size* and *Item_Type* to be *generic formal parameters.* An *instance* of a generic package is a copy of the template with each occurrence of a given generic formal parameter replaced by specific value, variable, type, subprogram, or package. In the above example, the generic formal parameters *Stack_Size* and *Item_Type* stand respectively for an integer value and a type. (Cohen, 1996.)

```
package Character_Stack_Package is
    new Generic_ Stack_Package
        (Stack_Size => 255, Item_Type => Character);
```

Example 2.3 (Cohen, 1996, p. 35).

For stacks holding up to 255 items of type *Character*, the package *Character_Stack_Package* in the above example is declared as an instance of the generic package *Generic_Stack_Package*. According to some Ada compilers, such as GNAT, this instantiation would result in a non-generic compilable version of the generic package with appropriate substitutions of the formals by the actuals. That is, the instantiation would result in a package whose declaration has no generic declaration header, and whose declaration and body are named *Character_Stack_Package* in place of the generic package name, using the number 255 in place of *Stack_Size*, and the type *Character* in place of *Item_Type*.

In whatever case, whether the semantics permit what is basically a compile-time macro expansion or not, the name *Character_Stack_Package* is the static name to be used to access the visible package components. That is, a procedure call such as *Character_Stack_Package.Push(Ch, S)* can be made, or a type such as *Character_Stack_Package.Stack_Type* can be assigned to a variable. As stated above, a generic package is not really a package, but a template for a package. Therefore, it is not allowed outside a generic package to refer to an entity such as *Generic_Stack_Package.Stack_Type*.

In addition to generic packages, Ada allows the definition of individual generic subprograms. Generic packages and generic subprograms may be separately compiled. A separately compiled generic package or subprogram may be instantiated by any compilation unit naming the generic unit in a context clause. (ibid.) The following example is a main subprogram that considers *Generic_Stack_Package* as a separately compiled package:

```
with Ada.Integer_Text_IO, Ada.Text_IO, Generic_Stack_Package;
procedure Reverse_Integers is
    package Integer_Stack_Package  is
        new Generic_Stack_Package
            (Stack_Size => 10, Item_Type => Interger);
    S :  Integer_Stack_Package.Stack_Type;
    X: Integer;
begin
    for I in 1 .. 10 loop
        Ada.Integer_ Text_IO.Get (X);
        Integer_Stack_Package.Push (X, S);
    end loop;
    for I in 1 .. 10 loop
        Integer_Stack_Package.Pop (X, S);
        Ada.Integer_ Text_IO.Put (X);
        Ada. Text_IO.New_Line;
    end loop;
end Reverse_Integers;
```

Example 2.4 (Cohen, 1996, p. 36).

As is the case with ordinary packages, a generic package is allowed to be made up of only a package declaration without having to coexist with a package body. This is only permitted when the package declaration contains types, subtypes[1], objects, or named-number declarations, but no subprogram declarations. As for generic subprograms, unlike ordinary subprograms, a generic subprogram must consist of both a declaration and body. That is, an

[1] The term subtype in Ada indicates a subset of the elements of a primitive data type. That is, it does not have the same meaning as in object-oriented terminology.

ordinary subprogram declaration in addition to the generic formal parameters must be put together in order to form a generic subprogram declaration, which is not only required to hold the generic formal parameters, but also to facilitate the ability of separating a generic subprogram declaration and its body from each other in different compilation units, and more importantly to apply the required consistency checks to generic instantiations. The generic subprogram might read as follows:

```
-- Generic subprogram declaration:
generic
    Variable              : in out Integer;
    Limit, Reset_Value    : in Integer;
procedure Reset_Integer_Template;  -- this declaration is a must.
-- Generic subprogram body:
procedure Reset_Integer_Template is
begin
   if Variable > Limit then
      Variable := Reset_Value;
   end if;
end Reset_Integer_Template;
```

Example 2.5 (Cohen, 1996, p. 684).

2.2 Generic Formal Parameters

There are four categories of generic formal parameters:

- *Generic formal objects* represent values and variables.
- *Generic formal types* represent subtypes.
- *Generic formal subprograms* represent procedures and functions.
- *Generic formal packages* represent instances of other generic packages.

This subsection provides details about the differences within these categories and the way each of them can be declared within the generic declaration.

2.2.1 Generic Formal Objects

Constant and variable values can be represented through two sorts of generic formal objects. First, *generic formal constants* that represent fixed values. Second, *generic formal variables* that represent variables external to the generic unit; the values assigned to these variables can be changed after the instantiation, so each call on an instance may assign different values to the generic formal variables.

The syntax of a general formal constant declaration is:
identifier,…, identifier: ***in*** *subtype-name;*

The syntax of a general formal variable declaration is:
identifier,…, identifier: ***in out*** *subtype-name;*

Example 2.5 illustrates both kinds of generic formal objects, so let us assume that the following declarations are contained in some declarative part:

```
A : Integer;
B : Integer := 10;
C : Integer := 0;
procedure Reset_A is
    new Reset_Integer_Template
        (Variable => A, Limit => B, Reset_Value => C);
```

Example 2.6 (Cohen, 1996, p. 684).

Because *Limit* and *Reset_Value* are generic formal constants, the values 10 and 0 are always used, even if the values of *B* and *C* change after the instantiation. Because *Variable* is a generic formal variable, it is always the latest value of *A* that is examined and replaced. (Cohen, 1996.)

2.2.2 Generic Formal Types

Generic formal types represent subtypes. When a generic formal type is declared to belong to a certain kind of type, it must be used within the generic

unit as a type of that kind (Cohen, 1996). The corresponding generic actual parameter in an instantiation must be some subtype that can be used in at least the way that a type of that kind can be used (ibid.). Identifiers declared as generic formal types can be used throughout the respective generic units as ordinary subtype names.

Generic formal parameters for numeric types. There are five kinds of generic formal parameters that correspond to numeric types:

1. *Generic formal signed integer type.* It is declared as follows:

   type *identifier* **is range <>;**

   The angle brackets here, and later on in this section, stand for the unconstrained information that is expected to be in their place in the declaration of the corresponding generic actual subtype.

2. *Generic formal unsigned (modular) integer type.* It is declared as follows:

   type *identifier* **is mod <>;**

   The angle brackets stand for the modulus.

3. *Generic formal floating-point type.* It is declared as follows:

   type *identifier* **is digits <>;**

   The angle brackets stand for the number of digits of precision.

4. *Generic formal ordinary fixed-point type.* It is declared as follows:

   type *identifier* **is delta <>;**

   The angle brackets stand for the size of the delta. Range is not needed in the declaration of the formal.

5. *Generic formal decimal fixed-point type.* It is declared as follows:

   type *identifier* **is delta <> digits <>;**

   The angle brackets stand for the size of the delta and the number of digits respectively.

Generic formal discrete types. They can simply be declared as follows:

type *identifier* **is (<>);**

This looks like an enumeration-type declaration with the list of enumeration literals left unspecified (ibid.). Accordingly, the corresponding actual parameter could be a subtype of either an enumeration type or an integer type.

Generic formal array types. The obvious beneficial or practical use of such formal parameters is for the writing of general-purpose array-manipulation algorithms (e.g., sorting) using generic program units. There are two kinds of generic formal array types:

1. *Unconstrained generic formal array type.* It is declared as follows:

   type *identifier* **is**
   array (*subtype-name* **range <>** , … , *subtype-name* **range <>**)
   of *subtype-name*;

2. *Constrained generic formal array type.* It is declared as follows:

   type *identifier* **is**
   array (*subtype-name* , … , *subtype-name*) **of** *subtype-name*;

   No discrete ranges allowed here to specify the index bounds. They should be subtype names.

The subtype names used to specify the index subtypes and the component subtype may themselves be generic formal types declared earlier in the same generic unit. However, any subtype name visible at the point of the generic declaration may be used in the generic-formal-array-type declaration, including names of predefined types and subtypes. In an instantiation, a generic actual parameter corresponding to a generic formal array type must be an array subtype S that matches the generic formal array type in the following senses:

- If the generic formal array type is unconstrained, S must be unconstrained; If the generic formal array type is constrained, S must be constrained.

- S must have the same number of dimensions as the generic formal array type.
- In each dimension, the index subtype of S must be the same as the index subtype of the generic formal array type.
- The component *sub*type of S must be the same as the component subtype of the generic formal array type. (ibid.)

Generic formal access types. There are five kinds of generic formal access types. The first three of the following are called access-to-object types. For these three types, the subtype pointed to by the generic actual subtype must be the same as that pointed to by the generic formal parameter. That is, the two subtypes must *statically match.* (ibid.)

1. *Generic formal pool-specific access-to-variable type.* It is declared as follows:

   type *identifier* **is access** *subtype-name;*

   The values of pool-specific access types can only point to dynamically allocated variables.

2. *Generic formal general access-to-variable type.* It is declared as follows:

   type *identifier* **is access all** *subtype-name;*

   The values of general access-to-variable types can only point to variables, which are either dynamically allocated or declared.

3. *Generic formal general access-to-constant type.* It is declared as follows:

   type *identifier* **is access constant** *subtype-name;*

   The values of general access-to-constant types can point to variables or constants, which are either dynamically allocated or declared.

4. *Generic formal access-to-procedure type.* It is declared as follows:

   type *identifier* **is**
   access procedure
   [(parameter-specification ; … ; parameter-specification)];

   Square brackets indicate that the in-between are optional.

5. *Generic formal access-to-function type.* It is declared as follows:

```
type identifier is
    access function
        [ ( parameter-specification ; … ; parameter-specification ) ]
        return subtype-name;
```

Values of the types declared above in 4 and 5 can be generated within the generic template by applying 'Access attributes to subprograms with matching parameter types, parameter modes, and (in the case of functions) result types; and the subprograms pointed to by these values can be called in a manner consistent with those types and modes. The generic actual parameter corresponding to a generic formal access-to-procedure type can be any access-to-procedure type with the same parameter types and modes, and the generic actual parameter corresponding to a generic formal access-to-function type can be any access-to-function type with the same parameter and result types. Only the parameter and result *types* have to match, not the subtypes. (ibid.)

Generic formal private types. The simplest form of a generic-formal-private-type declaration is:

```
type identifier is private;
```

This has the same form as a private-type declaration in a package declaration, but a different meaning. It follows from the principle that a generic formal type may only be used within its generic unit as a private type. The only operations available for private types are certain operations available for all types (e.g., the ability to call subprograms with parameters or results of that type), plus assignment and predefined comparison for equality. These operations are available for numeric types, enumeration types, access types, and certain array types, record types, and private types; it follows from the principle that any type for which these operations are available may be used as a generic actual parameter. Thus generic formal private types are quite general. (ibid.)

Certain private types, array types, and record types (e.g., private types that are declared with the word ***limited***, array types that have limited components, and record types that are limited for either of these reasons) are not allowed as generic actual parameters that correspond to a generic-formal-private-type. A generic-formal-limited-private-type provides more freedom in the correspondence. Its declaration takes the form:

type *identifier* **is limited private**;

Since assignment and comparison for equality are not available for limited types, the freedom in the correspondence necessitates restrictions on the use of the formals within the generic unit.

There are special forms of generic formal private types for tagged types:

type *identifier* **is tagged private**;

type *identifier* **is tagged limited private**;

Within the generic template, these generic formal types can be used in the same ways, respectively, as private and limited private tagged types. In particular, they can be extended. When the word ***limited*** appears in the generic-formal-type declaration, the corresponding generic actual parameter can be any definite subtype of a concrete tagged type. When the word ***limited*** does not appear, the corresponding generic actual parameter can be any definite subtype of a nonlimited concrete tagged type. (ibid.)

2.2.3 Generic Formal Subprograms

This subsection, as well as the following one, will describe the Ada techniques for constrained genericity. These techniques basically rely on the idea of encapsulating a type, or a list of functionally dependent types, with the constraints on them into one entity which is the generic declaration part of the program unit. The constraints are therefore treated as generic formal parameters. Syntactically these parameters will be subprogram declarations or

instances of generic packages preceded by the word ***with***, as in example 5.2. So, as the ***with*** clause is allowed within the generic declaration to declare generic formal subprograms or generic formal packages, the ***use*** clause is also allowed within the generic declaration so that the visible components of the constraining subprograms and packages can be expressed concisely.

A generic formal subprogram may be either a *generic formal procedure* or a *generic formal function*. In an instantiation, the generic actual parameter corresponding to the generic formal subprogram may be the name of any subprogram that has the same parameter-and-result-type profile and the same parameter modes, such as ***in***, ***out***, and ***in out***. The subprogram name used as a generic actual parameter may be an overloaded name, as is the case with the ***better*** function in example 5.2. However, for the instantiation to be legal, there must be only one version with a matching parameter-and-result-type profile. It is this version that is used as the generic actual parameter. (Cohen, 1996.)

In example 5.2, the subprogram name ***better*** is omitted from the list of actual parameters to the generic instantiation of the package ***Comparable***. This is because the name of the actual subprogram matches the name of the corresponding formal subprogram. This ability to use default actual subprograms with matching names and types is obtained by specifying ***is <>*** in the declaration of the formal generic subprogram (Meyer 1986).

2.2.4 Generic Formal packages

A formal package parameter matches any instance of the specified generic package (Barnes, 1995). The generic declaration of the function ***Go*** in example 5.2 illustrates the semantics of declaring such a parameter.

Generic formal packages are appropriate in two different circumstances. In the first circumstance, the generic is defining additional operations, or a new

abstraction, in terms of some preexisting abstraction defined by some preexisting generic. This kind of "layering" of functionality can be extremely cumbersome if all of the types and operations defined by the preexisting generic must be imported into the new generic. The generic formal package provides a direct way to import all of the types and operations defined in an instance of the preexisting generic. In other words, generic formal packages allow generics to be parameterized by other generics, which allows for safer and simpler composition of generic abstractions. In particular it allows for one generic to easily extend the abstraction provided by another generic, without requiring the programmer to enumerate all the operations of the first in the formal part of the second. (ibid.)

A second circumstance where a generic formal package is appropriate is when the same abstraction is implemented in several different ways. A generic package can be used to define a signature, and then a given implementation for the signature is established by instantiating the signature. Once the signature is defined, a generic formal package for this signature can be used in a generic formal part as a short-hand for a type and a set of operations. (ibid.) Example 5.2 illustrates the definition of such a signature as a generic empty package called ***Comparable.***

3. C++ TEMPLATES

C++, developed by Bjarne Stroustrup of the AT&T Bell Laboratories, is a successor of the C language. It enhances C by supporting object-oriented programming. Almost all features of C are still available in C++. Because of the popularity of C in software industry, this extension gave C++ the required success. (Tang, 1992.)

According to Stroustrup (1995), C++ directly supports a variety of programming styles. In this, C++ deliberately differs from languages designed to support a single way of writing programs. The support for multiple styles is one of its major strengths. Key programming styles supported by C++ include: traditional C–style, concrete classes, abstract classes, traditional class hierarchies, abstract classes and class hierarchies, and generic programming. (ibid.)

Given classes and class hierarchies, we can elegantly and efficiently represent individual concepts and also represent concepts that relate to each other in a hierarchical manner. However, some common and important concepts are neither independent of each other nor hierarchically organized. For example, the notions ‘‘vector of integers’’ and ‘‘vector of complex numbers’’ are related through the common concept of a vector and differ in the type of the vector elements (only). Such abstractions are best represented through parameterization. For example, the *vector* should be parameterized by the element type. (Stroustrup, 1998.)

C++ provides parameterization by type through the notion of a template. It was a crucial design criterion that templates should be flexible and efficient enough to be used to define fundamental containers with severe efficiency constraints. In particular, the aim was to be able to provide a *vector* template class that did not impose run-time or space overheads compared to a built-in array. (ibid.)

3.1 A Simple Class Template

A class template specifies how individual classes can be constructed much like the way a class specifies how individual objects can be constructed (Stroustrup, 1994). Here is a declaration of a vector class template:

```
template<class T> class Vector {
        T* v;
        int sz;
public:
        Vector (int);
        T& operator[] (int);
        T& elem (int i)  { return v[i]; }
        // …
};
```

Example 3.1 (Stroustrup, 1994, p. 341).

The ***template<class T>*** prefix[1] specifies that a template is being declared and that an argument ***T*** denoting a type will be used in the declaration. After its introduction, ***T*** is used exactly like other type names within the scope of the template declaration. (ibid.) Vectors can then be used like this:

```
vector<int> v1(20);
vector<complex> v2(30);
```

Class templates are no harder to use than classes. When the full name of an instance of a class template is considered too long, abbreviations can be introduced using ***typedef***. (ibid.)

```
typedef vector<int> IntVec;      // make IntVec a synonym for vector<int>.
IntVec v3(10);                   // v1 above and v3 are of the same type.
```

The process of generating a class declaration from a template class and a template argument is often called *template instantiation* (Stroustrup, 1997). Anything preceded by ***template<…>*** means the compiler will not allocate storage for it at that point, but will instead wait until it is told to (by a template

[1] The ***typename*** keyword can be used instead of ***class*** when declaring template parameters. It is now even preferred to ***class***.

instantiation), and that somewhere in the compiler and linker there is a mechanism for removing multiple definitions of an identical template (Eckel, 1995). That is, given a template definition and a use of that template, it is the implementation's job to generate correct code. From a class template and a set of template arguments, the compiler needs to generate the definition of a class and the definitions of those of its member functions that were used. The generated classes are called *specializations.* Generated specializations and specializations explicitly written by the programmer are referred to as *generated specializations* and *explicit specializations,* respectively. (Stroustrup, 1997.) Explicit or user-defined specialization is described in subsection 3.5.

The way members of a template class are declared and defined does not differ from that of members of a non-template class. Correspondingly, the bodies of the member functions are not necessarily required to be defined within their template classes. If such template members are to be defined outside their classes, they need to be declared explicitly as templates. For example:

template<class T> T& Vector<T> :: operator[] (int i) { /…*/ }*

So, in such a non-inline member function definition, the compiler needs to see a template declaration in front of the member function. Also, the class name needs to be specified with the template argument type:***Vector<T>***.

3.2 Template Parameters

As might be expected, a template can take values and objects, not only types. In addition, it can use a template as a template parameter. In this subsection, templates with several parameters and templates as template parameters are to be described.

3.2.1 A Template with Several Parameters

The following example illustrates how a template can take two parameters, where the first type parameter is used in the definition of the subsequent parameter. As a rule, objects such as ***def_val*** cannot be defined beforehand.

```
template<class T, T def_val> class Cont { /*…*/ };
```

Example 3.2 (Stroustrup, 1997, p. 331).

Here is another example:

```
template<class T, int i> class Buffer {
    T v[i];
    int sz ;
public:
    Buffer() : sz (i) { }
    // ...
};
Buffer<char, 127> cbuf ;
Buffer<Record, 8> rbuf ;
```

Example 3.3 (Stroustrup, 1997, p. 332).

Simple and constrained containers such as ***Buffer*** can be important where run-time efficiency and compactness are paramount (thus preventing the use of a more general ***string*** or ***vector***). Passing a size as a template argument allows ***Buffer***'s implementer to avoid free store use. (Stroustrup, 1997)

A template argument can be a constant expression, the address of an object or function with external linkage, or a non-overloaded pointer to member. A pointer used as a template argument must be of the form &***of***, where ***of*** is the name of an object or a function, or of the form ***f***, where ***f*** is the name of a function. A pointer to member must be of the form &***X : : of***, where ***of*** is the name of an member. In particular, a string literal is *not* acceptable as a template argument. (ibid.)

3.2.2 Templates as Template Parameters

On certain occasions, templates have indeed a beneficial and practical use as template arguments. For example:

```
template<class T, template<class> class C> class Xrefd {
    C<T> mems;
    C<T*> refs;
    // ...
};
Xrefd<Entry, vector> x1; //store cross references for Entries in a vector
Xrefd<Record, map> x1;  //store cross references for Records in a map
```

Example 3.4 (Stroustrup, 1997, p. 855).

To use a template as a template parameter, we specify its required arguments. The template parameters of the template parameter need to be known in order to use the template parameter. The point of using a template as a template parameter is usually that we want to instantiate it with a variety of argument types (such as ***T*** and ***T**** in the previous example). That is, we want to express the member declarations of a template in terms of another template, but we want that other template to be a parameter so that it can be specified by users. The common case in which a template needs a container to hold elements of its own argument type is often better handled by passing the container type. (Stroustrup, 1997.)

3.3 Template Class Relationships

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.*** It is logically unacceptable to treat a list of ***Manager***s as a list of ***Employee***s, because otherwise users could add ***Employee***s who are not ***Manager***s to the list. As far as the C++ language rules are concerned, there is no relationship between two classes generated

from a single class template; and therefore, ***List<Manager>*** guarantees that the members of the list are all ***Manager***s, so that users can safely and efficiently apply ***Manager***-specific operations on the members of the list (Stroustrup, 1994).

A class template is usefully understood as a specification of how particular types are to be created. In other words, the template implementation is a mechanism that generates types when needed based on the user's specification. Consequently, a class template is sometimes called a *type generator*. (Stroustrup, 1997.) That is, according to the following example, ***set<Shape*>*** and ***set<Circle*>*** are two different types. There cannot be any *default* relationship between classes generated from the same template (Stroustrup, 1994). The following example 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 3.5 (Stroustrup, 1997, p. 348, 349).

This will not compile because there is no built-in conversion from ***set<Circle*>&*** to ***set<Shape*>&***. Nor should there be. (Stroustrup, 1997.)

3.4 Function Templates

Templates were added to C++ for the same reason that generics were added to several other languages: to provide a means for developing type safe containers. Greater emphasis was placed on clean and consistent design than restriction and policy. 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. (Garcia et al., 2003.)

According to Stroustrup (1994), function templates were introduced partly because member functions were clearly needed for class templates and partly because the template concept seemed incomplete without them. Naturally, there were also quite a few textbook examples, such as ***sort()*** functions. Andrew Koenig and Alex Stepanov were the main contributors of examples requiring function templates. Sorting an array was considered the most basic example (see example 3.6 below). (ibid.)

Function templates proved extremely useful in their own right, and they also proved essential for supporting class templates when non-member functions are preferred over member functions (ibid.).

Regarding template instantiation, the compiler treats function templates as well as it does with class templates. That is, from a template function definition and the use of that template function, the compiler will generate a function(s) with the correct code. Different from class templates, whose parameters are never deduced, the compiler can deduce type and non-type arguments from the function calls; the compiler handles this with reference to the function argument list that determines the set of template arguments. Furthermore, only

class templates can be template arguments; function templates neither are types nor considered as those non-type objects that are allowed as template arguments. Lastly, we cannot overload a class template name; specialization is the means by which we can provide alternative implementations for a template. Overloading is a facility for functions only. Function template overloading is described in subsection 3.5.3.

```
// declaration of a template function:
template<class T> void sort (vector<T>&);

 void f(vector<int>& vi, vector<string>& vs)
 {
        sort (vi) ;        / / sort(vector<int>& v);
        sort (vs) ;        / / sort(vector<String>& v);
}

// definition of a template function:
template<class T> void sort (vector<T>& v)
/*
        Sort the elements into increasing order

        Algorithm: bubble sort (inefficient and obvious)
*/
{
        unsigned int n = v.size();

        for (int i=0; i<n-1; i++)
            for (int j=n-1; i<j; j--)
                if (v[j] < v[j-1]) {   // swap v[j] and v[j-1]
                    T temp = v[j];
                    v[j] = v[j-1];
                  v[j-1] = temp;
                 }
}
```

Example 3.6 (Stroustrup, 1994, p. 348).

3.5 User-defined Specialization

As a matter of fact, specialization had to be available in C++ in order to prevent critical code enlargements that might be experienced with macro expansion and primitive instantiation mechanisms. Specialization preserves the highest run-time performance that could be experienced with implementation behaviors that replicate the code at each call site, and at the same time avoids such code-bloating behaviors. The compiler only checks the syntax of template definitions without allocating any storage; then at the points of template instantiations, it starts to generate code. What is known as template-generated code is also called generated specializations, relying on the fact that the code will not be replicated when a template is used with the same template arguments.

C++ allows the programmer to provide specialized implementations, or to write explicitly alternative definitions of a template for specific arguments, in which case the specialization is called *explicit specialization*, *user-defined specialization*, or simply, *user specialization*. This is particularly important when it is necessary to have an implementation that differs from the default, or to give an error whenever the template arguments are not expected, given the fact that template arguments are not constrained in any way. A perfectly predictable build process is essential to some users (Stroustrup, 1997).

The following couple of subsections explain, with reference to examples from Stroustrup (ibid.), how containers such as ***Vector*s** 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. It might read as follows:

```
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 3.7 (Stroustrup, 1997, p. 341).

3.5.1 Complete Specialization[1]

Relying on 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 3.8 (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.) In this manner, ***Vector<void*> v;*** is an instance with a definition of its own.

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

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.*

3.5.2 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 3.9 (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 ***Vector<void*>*** implemented exclusively through derivation and inline expansion. (Stroustrup, 1997.)

3.5.3 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 3.10 (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. JAVA GENERICS

Java is an object-oriented programming language developed primarily by James Gosling and colleagues at Sun Microsystems. The language, initially called **Oak** (named after the oak trees outside Gosling's office), was intended to replace C++, although the feature set better resembles that of Objective C. As part of the *Green Project*, Java was developed in 1991. It was published in 1994. (wikipedia.org)

Before JDK 1.5 or "Tiger", 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). Therefore, since primitive data types (e.g., ***int***) are not references, items of these types cannot be shoved into a collection. Instead, all primitive data types have their corresponding *wrapper* classes (e.g., ***Integer***) in the Java standard library. 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. This could be clarified using an example from Bracha (2004) as follows:

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

Example 4.1 (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.) However, it is possible that someone has stored something other than an ***Integer*** in this ***List***, in which case the code above would throw a ***ClassCastException*** (Goets, 2004).

According to Goets (ibid.), to eliminate the casts from one's code and, at the same time, gain an additional layer of type checking that would prevent someone from storing keys or values of the wrong type in a collection, JDK 1.5 introduces generics as a new extension to Java. It permits a type or method to function on objects of different types with compile-time type safety. Generics provide additional compile-time type safety to the Collections Framework and get rid of the tedious unpleasant work of casting. That is, programmers would actually be able to express their intent, and mark a list as being restricted to contain a particular data type (Bracha, 2004). 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 4.2 (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.)

It is clear from the above example that the *wrapper* classes (e.g., ***Integer***) of the Java standard library are still in service. This is not only because collections, after they are generified, still can only hold handles to objects of type ***Object***, but in a general sense, because Java generics are designed to use only classes

and interfaces as type arguments. That is, the technique is to restrict generics to take only objects that are, in fact, inherited from the type ***Object***. For certain, this technique provided Java genericity with one of its defining characteristics, relative to C++, which is the single compilation of the generic type declaration. Any generic type declaration will be compiled only once and finally, since the compiler treats any generic unit as if it were parameterized by the type ***Object***.

4.1 Benefits of Java Generics[1]

The addition of generics to the Java language is a major enhancement. Not only were there major changes to the language, type system, and compiler to support generic types, but the class libraries were overhauled so that many important classes, such as the Collections framework, have been made generic. This enables a number of benefits:

- **Type safety.** The primary goal of generics is to increase the type safety of Java programs. By knowing the type bounds of a variable that is defined using a generic type, the compiler can verify type assumptions to a much greater degree. Without generics, these assumptions exist only in the programmer's head (or in a code comment). A popular technique in Java programs is to define collections whose elements or keys are of a common type, such as "list of String" or "map from String to String." By capturing that additional type information in a variable's declaration, generics enable the compiler to enforce those additional type constraints. Type errors can now be caught at compile time, rather than showing up as ClassCastExceptions at runtime. Moving type checking from runtime to compile-time helps programmers to find errors more easily and improves their programs' reliability.

[1] This subsection is entirely cited from Goets (2004) with a few changes.

- **Elimination of casts.** A side benefit of generics is that many type casts can be eliminated from the source code. This makes code more readable and reduces the chance of error. Although the reduced need for casting reduces the verbosity of code that uses generic classes, declaring variables of generic types involves a corresponding increase in verbosity. Comparing the above two code examples 4.1 and 4.2, the use of a variable of generic type only once in a simple program does not result in a net savings in verbosity. But the savings start to add up for larger programs that use a variable of generic type many times.
- **Potential performance gains.** Generics create the possibility for greater optimization. In the initial implementation of generics, the compiler inserts the same casts into the generated bytecode that the programmer would have specified without generics. But the fact that more type information is available to the compiler allows for the possibility of optimizations in future versions of the JVM. (Goets, 2004.)

Because of the way generics are implemented, (almost) no JVM or classfile changes were required for the support of generic types. All of the work is done in the compiler, which generates code similar to what programmers would write without generics (complete with casts), only with greater confidence in its type safety. (ibid.)

4.2 Simple Java Generics

Many of the best examples of generic types come from the Collections framework, because generics let programmers specify type constraints on the elements stored in collections (Goets, 2004). Here is a small excerpt from the definitions of the interface ***List*** and ***Iterator*** in package ***java.util*** (Bracha, 2004):

```
public interface List<E> {
  void add(E x);
  Iterator<E> Iterator();
}
public interface Iterator<E> {
  E next();
  boolean hasNext();
}
```

Example 4.3 (Bracha, 2004, p. 3)

According to Bracha (ibid.), 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 interfaces ***List*** and ***Iterator.*** Then, throughout the generic declaration, the type parameter can be used almost in any place in which class names could be used. Each of the ***List*** and ***Iterator*** interfaces are parameterized by one type ***E***. Methods that would (without generics) accept or return ***Object*** now use ***E*** in their signatures instead, indicating additional typing constraints underlying the specification of ***List*** and ***Iterator*** (Goets, 2004). Normally, the values of the type parameters, or the *actual type parameters*, should be specified at the time of declaring or instantiating objects of a generic type:

```
List<Integer> myIntList = new LinkedList<Integer>();
```

Example 4.4

In this statement, the type parameter had to be specified a couple of times. First, with the declaration of the type of the variable ***myInList***; and secondly, to parameterize the ***LinkedList*** class so that an instance of the correct type can be instantiated. The instantiation of a generic in Java is never actually expanded at compile-time like in this example:

```
public interface IntegerList {
  void add(Integer x);
  Iterator<Integer> Iterator();
}
```

Example 4.5 (Bracha, 2004, p. 3)

There are not multiple copies of the code: not in source, not in binary, not on disk and not in memory. This is very different from a C++ template; 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. (Bracha, 2004.)

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.) For instance, whenever the compiler finds a variable of type ***List<Integer>***, ***myIntList*** according to example 4.4, it regards ***E*** as constrained to ***Integer***; as a result, it recognizes the formal parameter ***x*** of the method ***myIntList.add()*** as of type ***Integer***. That is, the compiler will ascertain that any value passed to ***List.add()*** on the variable ***myIntList*** should be an ***Integer***.

As mentioned above, the technique of restricting generics to take only classes and interfaces as type arguments led to Java's capability to perform a generic type declaration's single compilation. However, this will 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 4.6 (Garcia et al., 2003, p. 128).

Generics interact synergistically with several of the other new language features in JDK 1.5, including the enhanced ***for*** loop (sometimes called the foreach or for/in loop), enumerations, and autoboxing (Goets, 2004). 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, solves the problem illustrated in example 4.6 so that the syntax can make as if it permits primitive data types as actual type parameters, and therefore the code would be simpler as in the following version of example 4.6:

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

Example 4.7 (Garcia et al., 2003, p. 128).

Any class, except an exception type, an enumeration, or an anonymous inner class, can have type parameters (Goets, 2004).

4.3 Generic Methods

Methods can also be made generic, whether or not the class in which they are defined is generic. With reference to example 4.3, Generic classes enforce type constraints across multiple method signatures. In ***List<E>*** and ***Iterator<E>***, the type parameter ***E*** appears in the signatures for ***add()*** and ***next()***, respectively. When a variable of type ***List<E>*** is created, a type constraint is asserted across methods. The values to be passed to ***add()*** will be the same type as those returned by ***next()***. Similarly, a generic method is generally declared to assert a type constraint across multiple arguments to the method. (Goets, 2004.) For example, depending on the ***boolean*** value of the first argument to the ***ifThenElse()*** method in the following code, it will return either the second or third argument (ibid.):

```
public <T> T ifThenElse(boolean b, T first, T second) {
  return b ? first : second;
}
```

Example 4.8 (Goets, 2004).

ifThenElse() can be called without explicitly telling the compiler what value of T is required. The compiler does not need to be told explicitly what value T will have; it only knows that they must all be the same. The compiler allows a call,

such as ***String s = ifThenElse(b, "a", "b");*** because the compiler can use type inference to infer that substituting String for T satisfies all type constraints. Similarly, this is allowed ***Integer i = ifThenElse(b, new Integer(1), new Integer(2));*** However, the compiler doesn't allow ***String s = ifThenElse(b, "pi", new Float(3.14));*** because no type will satisfy the required type constraints. (ibid.)

Why is a generic method used instead of adding the type T to the class definition? There are (at least) two cases in which this makes sense:

- When the generic method is static, in which case class type parameters cannot be used.
- When the type constraints on T really are local to the method, which means that there is no constraint that the *same* type T be used in *another* method signature of the same class. Making the type parameter for a generic method local to the method simplifies the signature of the enclosing class. (ibid.)

4.4 Generic Types are not Covariant

A common source of confusion with generic types is to assume, for example, that ***Collection<Object>*** is the supertype of all kinds of collections. This is actually not true. What applies to C++ concerning this issue applies exactly to Java. See subsection 3.3.

4.5 Subtype-based Constraints

On how a particular programming language (Eiffel) uses a balanced combination of genericity and inheritance, Meyer (1986) states that the Ada-like constrained genericity was avoided because it would be redundant with the inheritance mechanism. To provide the equivalent of a constrained formal generic parameter, we declare a special class whose features correspond to the constraints (i.e., the ***with*** subprograms in Ada terminology), and declare any

corresponding actual parameters as descendants of this class (ibid.). This is exactly what applies to Java; the above-mentioned special constraining class is best represented in Java through an interface definition, as is the case with the interface ***Comparable*** in examples 4.10 and 5.1; the types ***Apple*** and ***Orange*** in example 5.1 represent the actual parameters that are descendants of the constraining class.

In addition, the Java compiler will not permit a code such as the following:

```
class pick {
  static<T> T go(T a, T b) {
     if (a.better(b)) return a; else return b;
  }
}
```

Example 4.9

The above example will not compile because ***T*** does not stand for an arbitrary type, as it does in Ada and C++; it rather stands for type ***Object***. Such an example would compile only if ***better()*** is one of the ***Object*** methods (e.g., ***equals()***). Accordingly, the way to make such a code compile is by defining an interface that encapsulates the signature of the ***better*** method. The following example will compile:

```
interface Comparable<T> {
   boolean better(T x);
}

class pick {
   static<T extends Comparable<T>> T go(T a, T b) {
      if (a.better(b)) return a; else return b;
   }
}
```

Example 4.10 (Garcia et al., 2003, p. 118).

In this example, ***extends*** does not mean inheritance. The ***extends*** keyword is here used to define the bounds of a type parameter. We can then supply a class

type that ***implements*** the bound. In other words, ***extends*** in conjunction with type parameter bounds does not strictly mean inheritance, but it also includes the ***implements***-relationship that exists between classes and interfaces (see example 5.1). In conjunction with type parameter bounds, the ***extends*** keyword refers to an even broader notion of subtyping. It includes relationships that cannot be expressed in terms of ***extends*** or ***implements*** as we know them from non-generic Java. (Langer, 2005.)

Clearly, the subtype-based constraining mechanism proves inevitable because of the way type parameterization is handled in Java. However, this technique is somehow representing an obstacle to effective generic programming in Java, as explained later on in this study, and seems to be the base reason from which most criticisms about Java's version of generic programming initiate.

4.6 Wildcards with Methods' Formal Parameters

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 4.11 (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.

Bounded Wildcards

As said in subsection 3.3, ***List<Manager>*** is not a ***List<Employee>***. We should consider first the following example:

```
public void viewAllNames(List<Employee> employees) {
        for (Person i: employees){
                System.out.println (i.full_name);
} }
```

Example 4.*12*

We cannot call ***viewAllNames()*** 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 ***viewAllNames()*** can accept lists of any subtype of ***Employee.*** That is, it is possible now to call ***viewAllNames()*** on a ***List<Manager>.*** Furthermore, ***List<Employee>*** can also be replaced with ***List<? super Employee>*** so that ***viewAllNames()*** can accept lists of any supertype of ***Employee***; and therefore, it is possible to call ***viewAllNames()*** on a ***List<Person>***, for example. We know that ***?*** symbolizes an unknown type, which is in fact a subtype (supertype) of ***Employee***. We say that ***Employee*** is the upper bound (lower bound) of the wildcard. The unknown type could be ***Employee*** itself, or some subtype (supertype). Since we do not know what type it is, we do not know if it is a supertype (subtype) of ***Manager*** (***Person***); it might or might not be such a supertype (subtype), so it is not safe to add a ***Manager*** (***Person***) to a ***List<? extends Employee>*** (***List<? super Employee>***). (Bracha, 2004.)

The wildcard instantiation (e.g., ***List<? extends Employee>***) is a supertype of the concrete instantiation on a subtype of the upper bound (e.g., ***List<Manager>***). At the same time, a wildcard instantiation with an upper bound is supertype of all generic subtypes that are instantiated on the same upper bound wildcard. For instance, ***List<? extends Employee>*** is supertype of ***LinkedList<? extends Employee>*** and ***ArrayList<? extends Employee>***. The upper bound wildcard instantiation is also supertype of other upper bound wildcard instantiation with an upper bound that is a subtype of the own upper bound. For instance, ***List<? extends Employee>*** is supertype of ***List<? extends Manager>*** and ***List<? extends Trainer>***, because ***Manager*** and ***Trainer*** are subtypes of ***Employee***. (Langer, 2005.)

In the same way, the wildcard instantiation (e.g., ***List<? super Employee>***) is a supertype of the concrete instantiation on a supertype of the lower bound (e.g., ***List<Person>***). At the same time, a wildcard instantiation with a lower bound is supertype of parameterized subtypes that are instantiated on the same lower bound wildcard. For instance, ***List<? super Employee>*** is supertype of ***LinkedList<? super Employee>*** and ***ArrayList<? super Employee>***. The lower bound wildcard instantiation is also supertype of other lower bound wildcard instantiation with a lower bound that is a supertype of the own lower bound. For instance, ***List<? super Employee>*** is supertype of ***List<? super Person>***, because ***Person*** is a supertype of ***Employee***. The idea is that if the lower bound of one wildcard is a subtype of the lower bound of another wildcard then the type family with the subtype bound includes the type family with the supertype bound. If one family of types (e.g., ***? super Employee***) includes the other (e.g., ***? super Person***) then the wildcard instantiation on the larger family (e.g., ***List<? super Employee>***) is supertype of the wildcard instantiation of the included family (e.g., ***List<? super Person>***). (ibid.)

5. GENERIC PROGRAMMING

Like most new ideas, generic programming actually has a long history. Some of the early research papers on generic programming are nearly 25 years old, and the first experimental generic libraries were written in Ada[1] and Scheme. The first example of generic programming to become important outside of research groups was the STL, the C++ Standard Template Library. The Standard Template Library, designed by Alexander Stepanov and Meng Lee, was accepted in 1994 as part of the C++ standard library. (Austern, 1999.)

From an interview conducted by Russo (2001) with Alexander Stepanov, the following quote summarizes the goals of generic programming:
"Generic programming is a programming method that is based in finding the most abstract representations of efficient algorithms. That is, you start with an algorithm and find the most general set of requirements that allows it to perform and to perform efficiently. The amazing thing is that many different algorithms need the same set of requirements and there are multiple implementations of these requirements. The analogous fact in mathematics is that many different theorems depend on the same set of axioms and there are many different models of the same axioms. Abstraction works! Generic programming assumes that there are some fundamental laws that govern the behavior of software components and that it is possible to design interoperable modules based on these laws. It is also possible to use the laws to guide our software design. STL is an example of generic programming. C++ is a language in which I was able to produce a convincing example ".

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

[1] Musser & Stepanov (1987, 1989).

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.)

Another more inclusive definition by Musser and Stepanov, stated in 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*

So, the notion of abstraction is fundamental to generic programming: generic algorithms are specified in terms of abstract properties of types, not in terms of particular types (Siek & Lumsdaine, 2004). Thus, generic algorithms must be polymorphic (Garcia et al., 2003). According to Garcia et al. (ibid), the terminology to be used subsequently is of Alexander Stepanov and Matthew Austern.

A ***concept*** is the formalization of an abstraction as a set of requirements on a type (or several types). These requirements may be semantic as well as syntactic. A concept may incorporate the requirements of another concept, in which case the first concept is said to ***refine*** the second. A type (or list of types) that meets the requirements of a concept are said to ***model*** the concept. Concepts are used to specify interfaces to generic algorithms by ***constraining*** the type parameters of an algorithm. A generic algorithm may only be used with type arguments that model its constraining concepts. (Siek & Lumsdaine, 2004.) 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, algorithms defined with reference to concepts should be implemented in a way that allows them to function using multiple types. In languages such as C++ and Java, calling a template function or generic method is not different from calling a non-template function or non-generic method. That is, type parameters can be deduced without requiring explicit syntax for instantiation (Garcia et al., 2003). Thus, a concept could be thought of as a contract; as long as this contract is kept, one value can take certain type(s); and therefore, generic algorithms (functions or static methods) are fundamentally concerned with the capability of an argument (or 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 (Mueller & Jensen, 2004).

In the next three subsections, one simple example will be implemented in each of the target languages to show how the methodology of generic programming can be applied in these languages. Other supplemental language features that

contribute in the development of high quality generic libraries are to be described subsequently in this section.

5.1 Java

The means to approximate ***concepts*** in Java is ***interfaces***. A Java interface is perfectly what is required to formalize an abstraction as a set of requirements on types. Example 5.1 shows how the concept ***comparable*** could be realized as an ***interface*** and how ***Apple*** and ***Orange***[1] could **model** the ***comparable*** concept.

```
interface Comparable<T> {
  boolean better(T x);
}

class pick {
  static <T extends Comparable<T>>
  T go(T a, T b) {          // modified[2]
    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;
}

class Orange implements Comparable<Orange> {
  Orange(String s) { name = s; }
  public boolean better(Orange x)
    { return x.name.compareTo(name) > 0;}
  String name;
}
```

[1] The class type *Orange* in example 5.1 was not included in the original example by Garcia et al. (2003). It was implemented in C++ for some other example.

[2] 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.

```
public class Main {
   public static void main(String[] args) {

      Apple a1 = new Apple(3),
            a2 = new Apple(5);

      Orange o1 = new Orange("Miller"),
               o2 = new Orange("Portokalos");

      Apple a3 = pick.go(a1, a2);
      Orange o3 = pick.go(o1, o2);
   }
}
```

Example 5.1 (Slightly adapted from Garcia et al., 2003, p. 118).

The ***Generic algorithm*** in the above example is realized as a ***static method***. 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* (subsection 5.4.7), while the second alternative requires the more verbose explicit specification of type arguments (Garcia et al. 2003). In example 5.1, the generic algorithm is realized as a static parameterized method, called ***go,*** in a non-parameterized class, called ***pick***.

Subtyping is utilized to ***constrain*** the type parameters of the generic algorithm so that the generic method can only be called with arguments of types that model the constraining concept ***comparable***. In example 5.1, although it is enough to use only either of the two class types (***Apple*** and ***Orange***) to demonstrate the modeling relationship, they were both used to demonstrate *implicit instantiation*. In a coming C++ example, it would be enough to use type ***int*** to demonstrate the above-mentioned notions as well as *implicit instantiation*. In Java, as explained earlier, we must use *wrapper* classes as type parameters instead of primitive types. However, *wrapper* classes are not qualified to support subtype base constraining.

5.2 Ada

Ada 95 allows generic packages to be used as generic formal parameters. The Rationale, Barnes (1995), shows one use—also known as ***signature***—of this possibility: characteristics of an abstraction are grouped using the formal parameters of a generic empty package. ***Signatures*** are the natural way to express ***concepts***. (Duret-Lutz, 2001.)

```
-- define signature for Comparable
generic
    type T is private;
    with function better( x, y : T ) return Boolean is <>;
package Comparable is end;

-- generic declaration for the generic algorithm
with Comparable;
generic
    with package Pick is new Comparable (<>);
    use Pick;
function Go (a, b : T) return T;

-- generic algorithm
function Go (a, b : T) return T is
begin
    if better(a, b) then
        return a; else return b;
    end if;
end Go;

with Comparable;
package Comparable_Instances is

package Pick_Int is new Comparable(Integer, ">");

type Apple is
    record
        rating : Integer;
    end record;
function better(a, b: Apple) return Boolean;
package Pick_Apple is new Comparable(Apple);

end Comparable_Instances;
```

```
package body Comparable_Instances is

function better(a, b: Apple) return Boolean is
begin     return b.rating < a.rating;     end better;

end Comparable_Instances;

with Go ,Comparable_Instances;
use Comparable_Instances;
procedure Main is

i: Integer:= 0;
j: Integer:= 2;

a1: Apple:= (rating => 3);
a2: Apple:= (rating => 5);

function Better_Int is new Go (Pick_Int);
k: Integer:= Better_Int(i, j);

function Better_Apple is new Go (Pick_Apple);
a3: Apple:= Better_Apple(a1, a2);

begin
   …
end Main;
```

Example 5.2

In the above example, ***Pick_Apple*** and ***Pick_Int*** are explicit instances of the ***Comparable*** concept. ***Pick_Int*** is parameterized by the operator ″**>**″ as the generic actual function parameter to be used with type ***Integer***. ***Pick_Apple*** is intended to select the generic actual function ***better*** according to the type argument being used, which is ***Apple***, since this function is supposed to overload as in example 5.7. Thus, the type (or list of types) to be used as a type argument in an explicit instantiation of a concept is said to model explicitly the concept. ***Pick_Apple*** and ***Pick_Int*** are then passed as the ***constraining*** actual parameters of the generic algorithm ***Go***, by instantiating ***Go*** explicitly as ***Better_Apple*** and ***Better_Int***. This way, the type ***T*** is said to be equipped with the right ***better*** function.

5.3 C++

One of the interesting things about generic programming in C++ is that the STL documentation puts into use the term ***concept*** to indicate a set of requirements on types, while concepts are not represented by the syntax. Certainly because the STL as well as the methodology used were developed by Alexander Stepanov. So, C++ has no explicit features to support the notion of concepts. Because 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. (2003) 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 5.3 (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 5.4 (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&)
```

5.4 Supplemental Language Features

In addition to the primary aspects of generic programming, i.e., generic algorithms, 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 5.1 shows the level of support for these features in each of the target languages; the C++ and Java columns are of Garcia et al. (ibid.), while the third column is a finding of this thesis work.

5.4.1 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. (Garcia et al., 2003.) 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	Ada
Multi-type concepts	-	○	●
Multiple constraints	-	●	●
Associated type access	●	◉	●
Retroactive modeling	-	○	●
Type aliases	●	○	●
Separate compilation	○	●	●
Implicit instantiation	●	●	○
Concise syntax	●	◉	○

Table 5.1: The level of support for important properties for generic programming in C++, Java, and Ada.

● 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.)

The way Ada handles constrained genericity (by allowing subprograms and packages to be used as generic formal parameters so that constraints on type parameters can be declared in the generic definition) exhibits the quality required for a list of types to model simultaneously the constraining concept, or for supplying the list of types that models the constraining concept with the right operations simultaneously.

Likewise, multiple types can model simultaneously the documented constraining concept in C++, even with greater flexibility and reduced code, but less security, as there is no separate type checking for templates, and therefore there is no mechanism to constrain type parameters.

5.4.2 Multiple Constraints

Indicates whether more than one constraint can be placed on a type parameter (Garcia et al., 2003). As mentioned above, C++ has no mechanism to constrain

type parameters. This is because C++ does not provide separate type checking for templates. Before a template class or template function is used with specific arguments, the compiler need not to compile the template code. Presumably, templates define constraints implicitly, i.e., during the compilation, while the compiler places the code inline. Only while an instantiation is being compiled, the values of the generic parameters can be verified to be valid or not.

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 5.5.

In Ada, the way to specify that a type is equipped with the right subprograms or packages is to list the required constraining units within the generic declaration as *formal generic parameters* using the ***with*** clause. Any number of ***with*** clauses can be used within the generic declaration to specify constraints.

5.4.3 Associated Type Access

Rates the ease in which types can be mapped to other types within the context of a generic function (Garcia et al., 2003). 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. (Järvi et al., 2003.)

```
#include <vector>
using namespace std;
namespace traits {
        template <class T> struct arr_traits{};
        template <class T>
        struct arr_traits<T*> {
                typedef T element_type;
        };
        template <class T>
        struct arr_traits<vector<T> > {
                typedef T element_type;
        };
}
template <class E>
typename traits::arr_traits<E>::element_type*
copy_portion_to_new_array(E s, int start, int end) {
        typedef typename
                traits::arr_traits<E>::element_type M;
        M* array;
        for( int i = 0; i <= end-start; i++ )
                array[i] = s[start+i];
        return array;
}
int main(int, char*[]) {
        char* str="literal";
        vector<int> vec;
        for(int i=0; i<5; i++)
                vec.push_back(i);
        char* sub_str = copy_portion_to_new_array(str,1,4);
                // sub_str should contain "iter"
        int*  sub_vec = copy_portion_to_new_array(vec,1,3);
                // sub_vec should contain {1,2,3}
}
```

Example 5.6

[1] Introduced in Myers (1995).

The example above shows how an associated type could be referred to in the signature and body of the generic function. The return type of this function and the type of the local variable ***array*** are pointers of the type member of the generic type parameter. ***arr_traits*** is used to access this type member (or element type) so that the types ***traits::arr_traits<char*>::element_type*** and ***traits::arr_traits<vector<int> >::element_type*** resolve to ***char*** and ***int*** respectively.

According to Siek & Lumsdaine (2004), the use of template specialization inside of generic function relies on template parameters being transparent: the actual type must be known to find the matching templates specialization. Thus, the traits class idiom is not compatible with languages that have separate type checking and opaque parameters (parameters that do not delay the checking of dependent expressions, as normal template parameters do). (ibid.)

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.)

It is natural in Ada to encapsulate several functionally dependent types and constraints on them into the generic declaration of any program unit, or into a single concept abstraction using a generic empty package.

5.4.4 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.)

To establish a modeling relationship in Ada, a concrete type can be linked to the concept it models by passing it to the constraining generic package or signature at the point of explicitly instantiating this package. That is, modeling relationships are not supposed to be established in Ada at the point of type definition; and therefore, a new modeling relationship can easily be added after a type has been defined. Accordingly, the package ***Comparable_Instances*** of the example below can replace the corresponding package in example 5.2 so that the explicit instances of the ***Comparable*** concept, ***Pick_Apple*** and ***Pick_Int***, will be parameterized by the appropriate overloaded function ***better*** that matches the type argument being used, ***Apple*** or ***Integer***. This is clearly done independent of data type definitions.

```
with Comparable;
package Comparable_Instances is

function better(i, j: Integer) return Boolean;
package Pick_Int is new Comparable(Integer);

type Apple is
    record
        rating : Integer;
    end record;
function better(a, b: Apple) return Boolean;
package Pick_Apple is new Comparable(Apple);

end Comparable_Instances;

package body Comparable_Instances is

function better(i, j: Integer) return Boolean is
begin return j < i; end better;

function better(a, b: Apple) return Boolean is
begin return b.rating < a.rating; end better;

end Comparable_Instances;
```

Example 5.7

5.4.5 Type Aliases

Indicates whether a mechanism for creating shorter names for types is provided. Type aliasing is a simple but crucial feature for managing long type expressions commonly encountered in generic programming. With type aliasing, a short name could have been given to the type in the following example and thus reduce clutter in the code. Also, repeating the same type increases the probability of errors: changes to one copy of a type must be consistently applied to other copies. In addition to avoiding repetition of long type names, type aliases are useful for abstracting the actual types without losing static type accuracy. (Garcia et al., 2003.)

```
dijkstra_visitor<G,
  mutable_queue<Vertex,
    indirect_cmp<Vertex, Distance, DistanceMap,
      DistanceCompare> >,
  WeightMap, PredecessorMap, DistanceMap,
  DistanceCombine, DistanceCompare, Vertex, Edge,
  Distance> bfs_vis = new dijkstra_visitor<G,
    Mutable_queue<Vertex,
      Indirect_cmp<Vertex, Distance, DistanceMap,
        DistanceCompare> >,
      WeightMap, PredecessorMap, DistanceMap,
      DistanceCombine, DistanceCompare, Vertex, Edge,
      Distance>();
```

Example 5.8 (Garcia et al., 2003, p. 132).

The example above is taken from the Generic C# version of the graph library that was developed for the study of Garcia et al. (ibid.). It should be exactly identical to the Java version. ***bfs_vis*** is a reference to a ***dijkstra_visitor*** which is the type of the visitor object used in the Dijkstra algorithm. As in this example, type arguments are often instantiations of other parameterized components; such types can be overlong, resulting in cluttered and unreadable code (ibid.).

Java does not support type aliases, such as the ***typedef*** statement in C++. Ada is also a strongly typed language like Java, and maybe the strongest ever. Therefore, it does not allow type renaming. However, there is some other way to achieve the same effect; a subtype declaration without a constraint can be used to establish a new name for an existing subtype (Cohen, 1996). More importantly, it is still possible to constrain the parent type is some way. Furthermore, the ***use*** clause could also allow a more concise way of referring to certain types.

5.4.6 Separate Compilation

Indicates whether generic units 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++. (Garcia et al., 2003.)

Ada provides separate compilation for generic components, as well. An Ada compiler such as GNAT always handles generic instantiations by means of "macro expansion" in order to produce a non-generic compilable version of the code, as is the case with C++. However, for consistency and dependency checks, it is still necessary to compile generic units.

According to Cohen (1996), generic instantiation in Ada seems to be a form of "macro expansion"; however, this model is not quite accurate. A generic unit is actually compiled, either separately or as part of an enclosing compilation unit, before it is used in an instantiation. The declaration of a generic formal parameter describes very specific properties. These properties are used to check the internal consistency of the generic declaration and generic body when they are compiled, and the consistency checks are as demanding as for non-generic units. These same specific properties allow consistency checks to be applied to generic instantiations. After compiling a generic declaration, it is possible to compile the generic body and generic instantiations independently, in any order. If the generic body and an instantiation are each consistent with the declaration of the generic parameters, the body of the resulting instance is guaranteed to be legal. (ibid.)

5.4.7 Implicit Instantiation

Indicates that type parameters can be deduced without requiring explicit syntax for instantiation (Garcia et al., 2003). 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.

In Ada, before a subprogram can be used it should be instantiated explicitly, as in the following example:

```
-- Generic subprogram declaration:
generic
    type T is private;
procedure Do_Something (X, Y, Z : in out T);
-- Generic instantiation:
procedure Something is new Do_Something (T => Integer);
Something(X => X1, Y => Y1, Z => Z1);
```

Example 5.9

So, the type *Integer* is supplied explicitly for *T*, or the actual is supplied for the formal through an explicit instantiation statement. Then it is possible to call a procedure such as *Something* using integer values or variables of type *Integer*.

5.4.8 Concise Syntax

Indicates whether the syntax required to compose layers of generic components is independent of the scale of composition (Garcia et al., 2003). Referring to the examples in subsections 5.1, 5.2, and 5.3, the degree of supporting concise syntax in each language is obvious. Clearly, C++ is fully capable of supporting generic programming concisely, while Ada necessitates lengthy and verbose coding; Java is in between.

Because of the object oriented based design of the language, Java's version of generic programming requires a somewhat verbose syntax. An example of a newly introduced feature for Java generics, which contributes to a more concise syntax, is *Wildcards*. Furthermore, utilizing the Autoboxing/Unboxing feature in JDK 1.5 makes Java seem as if it allows built-in types as type arguments.

Ada is known to be a language of verbose syntactic constructions. Verbose syntaxes, in fact, provide more readable and reliable code that is, in certain cases, more flexible in use. However, in relation to certain statements, the corresponding concise syntaxes exist. For example, it is possible to use either positional notation or named notation to specify the actual parameters to subprograms and generic units, or to specify aggregates to arrays, extensions, and records. Named notation is verbose, but it provides flexibility in order for named parameter associations and named aggregates to be listed in any order regardless of the order in which the formals are specified.

In Ada we can use mathematical notations like “+ “, “-”, “*”, “/”, “>” and “e” to define operators for a user-defined data type. This capability, called operator overloading, allows computations involving a user-defined data type to be expressed as concisely and naturally as computations of predefined data types. (Tang, 1992.) This also applies to C++.

Furthermore, the efficient use of package ***use*** clauses and renaming declarations as well as the wise distribution of separately compiled modules could provide a more concise way to avoid the use of long expanded names.

So, to a limited extent, Ada could provide concise syntaxes; however, it is in fact verbose. The obvious reason behind such verbosity is that Ada is a language that enhances security through some mechanisms, such as explicit instantiation, static typing, constrained genericity, constrained subtyping, separate

compilation, etc. Verbosity in Ada is basically meant for the reliability, readability, maintainability, and extensibility of safety-critical systems.

6. RELATED ISSUES

This section describes four issues. The first is related to a limitation in the C++ compiler other than that of type checking, although it is a normal consequence of it. Secondly, the limitations of the type erasure mechanism in Java. The third subsection is about subtyping, and how it could interact with generics. Finally, constraints on generic arguments will be described further.

6.1 Templates Link-time Problem

C++ and Java do not demand explicit instantiation operations similar to that of utilizing the Ada's ***new*** keyword. Generic instantiation in C++ and Java is not directly expressed; it takes place automatically by the event of referencing a class, or calling a function or method. In C++, the compiler generates no multiple definitions for the same template, if it is used several times with the same type. Tang (1992) states: now suppose a program consists of multiple compilation units, and a template is invoked in different units with the same actual parameters. Then multiple definitions of the same function (or class) will be generated, and the compilation will fail at link time (ibid.). That is, the compiler by itself is not able to know whether the function or variable definition for a specific template is satisfied by code generated in another object module (Itzkowitz & Foltan, 1998). The C++ Standard provides facilities for the user to specify where a template entity should be instantiated (ibid.). When the user explicitly specifies template instantiation, the user then becomes responsible for ensuring that there is only one instantiation of the template function or static data member per application; this responsibility can necessitate a considerable amount of work (ibid.).

6.2 The Type Erasure Problem in Java

Cabana et al. (2004) demonstrates some problems with Java generics. These problems emerge basically from the use of the type erasure mechanism that replaces all type parameters with their bounds, or ***Object*** if the type parameter is not bounded. Moreover, when the compiler finds a parameterized type, i.e. an instantiation of a generic type, then it removes the type arguments; for instance, the types ***List<String>***, ***Set<Long>***, and ***Map<String, ?>*** are translated to ***List*** , ***Set*** and ***Map*** respectively (Langer, 2005). This technique allows violation of the Java type system and turns a type safe language into an unsafe one (Cabana et al., 2004).

Concisely, the following example will be type checked successfully, but will cause a ***ClassCastException*** at run-time.

```
import java.util.*;
public class Main {
    public static void main(String [] args) {
        List<Integer> list = new ArrayList<Integer>();
        howCome(list);
        Integer i = list.get(0);
    }
    public static void howCome(List l)
    {
        l.add("How come?");
        }
     }
```

Example 6.1

The actual parameter in the call ***howCome(list)*** is of type ***List<Integer>***. However, because of the type erasure, the compiler treats the actual parameter as of type ***List***. Therefore, it is possible to add instances of type ***String*** to the list ***list*** which is of type ***List<Integer>***.

For the same reason, overloading is not working as well. An example of a compilation error would read as follows:

name clash: myMethod(java.util.List<java.lang.String>) and myMethod(java.util.List<java.lang.Integer>) have the same erasure.

6.3 Subtyping

The term subtype in Ada indicates a built-in type with additional constraint; C++ and Java do not provide such a facility that allows specifying a subset of the elements of a primitive data type. It is only possible to implement additional constraints on classes in C++ and Java. Primitive data types in Java represent the only major facility that is not relying on object orientedness, but they cannot be used as type parameters. Rather, *wrapper* classes can substitute them as type parameters, as a consequence of the object oriented based design of the language. In spite of that, they are declared ***final*** so they are prevented from being extended or subtyped.

Since template type parameters in C++ do have run-time type representations of their own, it is possible to inherit from a type parameter. For example:

template<class Type> class subType : public Type { ... };

Example 6.2

This way the parameter ***Type*** is the class from which ***subType*** inherits. By other words, ***subType*** is a generic subclass that is parameterized by its superclass. Such a subclass is called a *mixin*. Bracha and Cooke (1990) demonstrated delayed inheritance using type parameters, calling the resulting components *mixins* or *abstract subclasses*. Mixin-based inheritance is a very useful alternative to both single and multiple inheritance.

According to Smaragdakis and Batory (1998), mixins represent a mechanism for specifying classes that will eventually inherit from a superclass. This superclass,

however, is not specified at the site of the mixin's definition. Thus a single mixin can be instantiated with different superclasses yielding widely varying classes. This property of mixins makes them appropriate for defining uniform incremental extensions for a multitude of classes. When the mixin is instantiated with one of these classes as a superclass, it produces a class incremented with the additional behavior. Mixins can easily be implemented using parameterized inheritance. In this case, a mixin is a parameterized class with the parameter becoming its superclass. (ibid.)

In Java, a type parameter cannot have a run-time type representation of its own. All type parameters are replaced by their bounds, or ***Object*** if the type parameter is unbounded. Consequently, there is no point to deriving from a type parameter, because we would be deriving from its bound, not from the type that the type parameter stands for. In addition, the actual type argument can be a final class or an enum type, from which we must not derive anyway. (Langer, 2005.)

Ada 95 can provide mixin inheritance using tagged type extension (single inheritance) and generic units. The generic template defines the mixin. The type supplied as generic actual parameter determines the parent. (Barnes, 1995.)
Thus we can write:

```
generic
    type S is abstract tagged private;
package P is
    type T is abstract new S with private;
    -- operations on T
private
    type T is abstract new S with
        record
            -- additional components
        end record;
end P;
```

Example 6.3 (Barnes, 1995, p. 138).

Where the body provides the operations and the specification exports the extended type. We can then use an instantiation of ***P*** to add the operations of ***T*** to any existing tagged type and the resulting type will of course still be in the class of the type passed as actual parameter. (ibid.)

Another interesting interaction between templates and subtyping in C++ is that of Coplien (1995), and is called *Curiously Recurring Template Patterns* (CRTPs). According to Rising (2000), a CRTP refers to a class that is derived from a base class instantiated from a template. The derived class is passed as a parameter to the template instantiation. This pattern captures a circular dependency using inheritance in one direction and templates in the other. In its simplest form, A CRTP reads as follows:

```
template<class Type> class baseType { ... };

class subType : public baseType<subType> { ... };
```

Example 6.4

Obviously, this is not a parameterized inheritance, as is sometimes mistakenly thought. A CRTP does not demand ***subType*** to be parameterized.

6.4 Constraints on Generic Arguments

In general, the constraining concept on a generic algorithm's type parameter should be modeled by a type argument to that algorithm. Each of the three languages in this study employs a distinct mechanism for creating modeling relationships. While Java utilizes subtyping at the point of type definition, Ada demands autonomous explicit instance declarations that have no conjunctions with the process of type definition. On the contrary, C++ provides no mechanism for setting up such modeling relationships. C++ brings templates and function overloading into harmonious union in order to enable generic programming.

In languages that are designed to facilitate only object orientedness, subtyping is the natural way to establish modeling relationships and to constrain type parameters, and this is the case with Java. As far as we are concerned in this study, subtype-based modeling and constraining make generic programming in Java suffer from at least two downsides: multiple types cannot be simultaneously constrained, and new modeling relationships cannot be added after a type has been defined. So, building abstractions using object-oriented techniques restricts generics to satisfy limited and specific needs.

In such a language whose highest priority is to enhance security as Ada, it is natural to have mechanisms such as explicit instantiation and constrained genericity. But the fact is that modeling a multi-type concept as well as adjusting modeling relationships have been made easy and straightforward due to the way constrained genericity is handled in Ada, by encapsulating several functionally dependent types and constraints on them into the generic declaration. Using the ***with*** clause, the names and signatures of the required functions and operations for the parameters can be listed within the generic declaration. This way, the compiler can separately type check the instantiations as well as the implementations.

A constraining technique akin to that of Ada had been thought of by Stroustrup and proposed by the C++ community:

```
        // The operations =, ==, <, and <=
        // must be defined for an argument type T
template <
      class T {
          T& operator = (const T&);
          int operator == (const T&, const T&);
          int operator <= (const T&, const T&);
          int operator < (const T&, const T&);  };
 > Class vector {  …  };
```

Example 6.5 (Stroustrup, 1994, p. 343).

According to Stroustrup (1994), some people thought that better code could be generated if template arguments were constrained, and it would be easier to read and understand parameterized types when the full set of operations on a type parameter is specified. Stroustrup does not believe that, because such lists would often be long enough to be unreadable and a higher number of templates would be needed for many applications. He admits that he initially underestimated the importance of constraints in readability and early error detection, but still sticks to the idea that constraining template parameters weakens the expressive power of templates. If one could agree with him that requiring the user to provide the set of operations on a type parameter decreases the flexibility of the parameterization facility without easing the implementation, it is hard to agree that this decreases the flexibility without increasing the safety of the facility. However, since we are able to provide specialized implementations, and to write explicitly alternative definitions for specific data types, safety can be assured somehow. 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).

7. CONCLUSION

In this paper, the support for generic programming in Ada 95, C++, and Java 1.5 has been examined; strengths and weaknesses have been identified; the qualification of the key language features engaged with generics to enable generic programming has been determined. To that extent, it can be ascertained that each of these languages provides a completely different approach to generic programming, and that the paradigm of generic programming demands flexible and consistent language designs in order to facilitate powerful versions of that paradigm.

It is not easy to compare programming languages neutrally. To some extent, it is a matter of subjective concerns based on language orientations, preferred programming style, programming culture, and so on. Yet we cannot claim superiority of one language over the other. Instead, we can focus the argument on one specific area so that our claims would be pragmatic; and this is what has been done in this study. It has been made clear that Java generics are satisfying limited and specific needs due to the restrictions imposed by the language design, which facilitates only object oriented techniques. With the exception of some considerable capabilities that are available in C++ because of the macro-like nature of templates (e.g., template metaprogramming), Ada 95 and C++ are of almost similar power. Apparently, Ada's form of parameterization is safer and more powerful than that of C++. Nevertheless, we cannot ignore the flexibility of the C++'s template facility, which is very much affected by the overall conciseness and consistency of the language.

REFERENCES

Austern, M. 1999 Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley.

Barnes, J. 1995. Ada 95 Rationale: The Language - The Standard Libraries. Lecture Notes in Computer Science, Vol. 1247, Springer-Verlag, Berlin.

Bracha, G. & Cook, W. 1990. Mixin-Based Inheritance. In Proceedings of the ECOOP/OOPSLA' 90, ACM, New York, 303 – 311.

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: 1.8.2005]

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: 1.8.2005]

Bracha, G. 2004. Generics in the Java Programming Language. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf. [Checked: 1.8.2005]

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

Cabana, B., Alagić, S. & Faulkner, J. 2004. Parametric polymorphism for Java: is there any hope in sight? In ACM SIGPLAN Notices, Vol. 39, Issue 12, ACM, New York, 22 – 31.

Cohen, N. H. 1996. Ada As A Second Language. 2nd Edition, McGraw-Hill.

Coplien, J. 1995. Curiously Recurring Template Patterns. In C++ Report, Vol. 7, Issue 2, SIGS Publications, Inc., New York, 24 – 27.

Duret-Lutz, A. 2001. Expression Templates in Ada. In Proceedings of the Ada-Europe '01, LNCS 2034, Springer-Verlag, Berlin, Heidelberg, 191 – 202.

Eckel, B. 1995. Thinking in C++. Prentice-Hall, Englewood Cliffs, New Jersey.

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.

GNAT User's Guide. 2002. The GNU Ada 95 Compiler, GNAT Version 3.15p, Ada Core Technologies, Inc.

Goets, B. 2004. Introduction to generic types in JDK 5.0. In developerWorks, IBM's resource for developers.
http://www-106.ibm.com/developerworks/edu/j-dw-java-generics-i.html.
[Checked: 1.8.2005]

Itzkowitz, A. & Foltan, L. 1998. Automatic Template Instantiation in DIGITAL C++. In COMPAQ's DIGITAL Technical Journal, Vol. 10, No. 1, 22 – 31.
http://research.compaq.com/wrl/DECarchives/DTJ/DTJT00/index.html.
[Checked: 1.8.2005]

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.

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

Langer, A. 2005. Java Generics FAQs. Angelika Langer - Training & Consulting. http://www.langer.camelot.de/GenericsFAQ/JavaGenericsFAQ.html. [Checked: 1.8.2005]

Meyer, B. 1986. Genericity versus Inheritance. In Proceedings of the OOPSLA '86, ACM, New York, 391 – 405.

Mueller, C. & Jensen, S. 2004. The Java Generic Programming System. http://www.osl.iu.edu/~chemuell/classes/b629/GenericJava.pdf. [Checked: 1.8.2005]

Musser, D. & Stepanov, A. 1987. A library of generic algorithms in Ada. In Proceedings of the SIGAda '87, ACM, New York, 216-225.

Musser, D. & Stepanov, A. 1988. Generic Programming. In Proceedings of the ISSAC '88, LNCS 358, Springer-Verlag, Berlin, Heidelberg, 13-25.

Musser, D. & Stepanov, A. 1989. The Ada Generic Library: Linear List processing packages. Springer-Verlag, Berlin, Heidelberg.

Musser, D. 2003. Generic Programming. http://www.cs.rpi.edu/~musser/gp/. [Checked: 1.8.2005]

Myers, N. 1995. Traits: a new and useful template technique. In C++ Report. http://www.cantrip.org/traits.html. [Checked: 1.8.2005]

Rising, L. 2000. The Pattern Almanac. Addison-Wesley.

Russo, G. 2001. An Interview with A. Stepanov. Edizioni Infomedia srl., Italy. http://www.stlport.org/resources/StepanovUSA.html. [Checked: 1.8.2005]

Seidewitz, E. 1994. Genericity versus Inheritance Reconsidered: Self-reference Using Generics. In Proceedings of the OOPSLA '94, ACM, New York, 153 – 163.

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

Siek, J. & Lumsdaine, A. 2004. Modular Generics. In Concepts: a Linguistic Foundation of Generic Programming, Adobe Tech Summit, San Jose, CA., Adobe Systems.

Smaragdakis, Y. & Batory, D. 1998. Implementing Layered Designs with Mixin Layers**.** In Proceedings of the ECOOP' 98, LNCS 1445, Springer-Verlag, London, UK, 550 – 570.

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

Stroustrup, B. 1995. Why C++ is not just an ObjectOriented Programming Language. In Addendum to the Proceedings of the OOPSLA '95, ACM, New York, 1 – 13.

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

Stroustrup, B. 1998. An Overview of the C++ Programming language. From The Handbook of Object Technology (Editor: Saba Zamir). CRC Press, Florida.

Tang, L. S. 1992. A Comparison of Ada and C++. In Proceedings of the TRI-Ada '92, ACM, New York, 338 – 349.