

# The Java Generic Programming System

Chris Mueller  
Scott Jensen

## Introduction

In this paper, we explore how to implement support for generic programming in Java using a combination of the compile time type system, the runtime type system, and the reflection features of Java without using Java Generics. As a starting point, we specifically looked at what generic programming techniques could be exploited in Java without using Java Generics. In addition, while the Java Generics approach seems heavily weighted towards being able to use templates in a fashion that mimics C++, our approach was to instead determine what is actually required for generic programming to be implemented and then to determine the extent to which those requirements can or cannot be met using the current version of Java. To do this, we provide a definition of Generic Programming and use it to design and implement the generic programming system in Java. We used portions of both the Standard Template Library (STL) and the Boost Graph Library in C++ to demonstrate our approach.

In section 1 we first look at defining generic programming and what capabilities are actually needed to implement generic programming in Java. In section 2 we then describe the approach we used based on Java reflection, illustrate the capabilities of the approach based on the implementation of algorithms from both the Standard Template Library and the Boost Graph Library<sup>1</sup>

## 1. What Is Generic Programming

There is not one concrete definition of generic programming, but we define generic programming as a style of programming that allows algorithms to be implemented once and used over and over with arbitrary data types. Whereas object oriented programming empowered types by allowing them to take on sophisticated behaviors, generic programming empowers algorithms, allowing them to work on arbitrary data types, not just the ones the algorithm was designed to use.

In order to accomplish this, generic programming introduces a new way of looking at types. Rather than assigning a concrete type to a value, generic algorithms only care that a value is able to meet certain 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. In generic programming parlance, the requirements on a value are a *concept*. If a class type meets the requirements of a concept, then it *models* the concept. A concept can *refine* another concept by requiring all the features of the original concept plus additional requirements. In refining a concept, a concept can refine multiple concepts (similar to multiple inheritance in object oriented programming).

At a high level, the concept/model/refine relationship is analogous to the class/instance/inherit relationship in object-oriented programming. Concepts define what is required, while types that implement these requirements model the concept. Concepts refine concepts just as classes inherit from other classes. The important distinction is that concepts are not tied to a particular type, whereas classes define a new type that instance objects are tied to.

In addition to these basic components, Garcia, et al<sup>2</sup>, identify eight other “properties” of generic programming that they felt were necessary for a system to fully support generic programming. These are:

- Multi-type constraints
- Multiple constraints
- Associated type access
- Retroactive modeling
- Type aliases
- Separate compilation
- Implicit instantiation
- Concise syntax

While we agree with the intentions of most of these requirements, we feel that their definitions are too heavily tied to the evolution of Generic Programming in C++. As appropriate, we offer slightly different interpretations of these requirements.

One important definition for generic programming from Jazayeri, et al<sup>3</sup> imposes efficiency restrictions on any generic programming implementation. In short, they state that if there are multiple ways to create a physical instance of an algorithm, the generic system must always select the most efficient approach. This definition is very much derived from the C++ approach to generic programming, which focuses heavily on generating generic instances of algorithms at compile time. While this can lead to very efficient systems, it imposes restrictions on how code can be managed and reused, the most notable being that it precludes shared object files – all code must be continually built from scratch. For the scope of this paper, we have relaxed this requirement to enable the exploration of other ways to implement generic programming systems. The hope is that by exploring mixed compile and run time solutions, new insight will be gained into what generic programming should encompass and how best to achieve it.

One other restriction used in this study is the use of Object Oriented (OO) programming exclusively to define types. This is not meant as an endorsement of OO programming or an attempt to show how to do generic programming with OO. Rather, this is a necessary restriction imposed by using Java as the language for this study. Java is in spirit an Object Oriented language. However, to be familiar to C and C++ programmers and support fast operations on common types, Java includes a set of primitive types and operators that more or less matches those in C. However, Java drew a line in the sand between its OO system and its primitive type system: classes can use the primitive types and operators, but the primitive types and operators cannot be used with classes. To

perform any operations on classes, instance and class methods must be used. Since primitive types and operators are essentially syntactic sugar and can be implemented using classes and methods, this restriction is acceptable, though an abstraction penalty will generally be present when primitive types are used.

## Generic Programming in Java

One important goal in implementing generic programming in Java is to respect the spirit of the language and not make it conform to the way generic programming is implemented in other languages. Three notions essential to the practice of programming in Java are:

- Using OO techniques for system design
- Using compile time checks to catch obvious errors and allowing the runtime system to catch the rest
- Using reflection to create flexible system architectures

Using these as guiding principles, we have implemented a complete system for generic programming that meets the spirit of generic programming and addresses many of the issues found with Java Generics in Garcia.

### *Defining Concepts*

The heart of generic programming is defining concepts. In our system, concepts are defined using interfaces. For instance, consider a set of Iterator concepts:

```
Interface TrivialIterator {
    public Object next();
}

interface ForwardIterator extends TrivialIterator {
    public boolean hasNext();
}
```

This defines a TrivialIterator concept that allows movement through a list and a ForwardIterator concept that allows an algorithm to test for more elements before advancing. While somewhat contrived, this example demonstrates how to define and refine concepts.

### *Defining Algorithms*

If concepts are the heart of generic programming, algorithms are the soul. In our generic system, algorithms are implemented to work on specific concepts:

```
static Object find(ForwardIterator first, EqualityComparable value) {
    Object retVal = null;
    Object temp;
```

```

while(first.hasNext()) {
    temp = first.next();
    if(value.isEqual(temp)) {
        retVal = temp;
        break;
    }
}
return retVal;
}

```

This example simply iterates through the elements in the list starting at first until either all elements have been checked or a match was found. The EqualityComparable concept requires one operation: isEqual(Object b).

Up to now, what we have demonstrated could easily be construed as “An Introduction to OO in Java Using Interfaces”. However, using just the interfaces shown so far would preclude retroactive modeling in that types would not only need to model a concept, but they would need to implement the concept’s interface in order for the algorithms to compile and run. The following code snippet demonstrates how our system enables a fundamentally different style of programming in Java (i.e., Generic Programming):

```

Vector v = new Vector();
Integer i = new Integer(80), iFind = null;

for(int j=0;j<100;j++) {v.add(new Integer(j));} //fill in the vector

iFind = (Integer)Algorithms.find(v.iterator(), i); // Call find!

```

The classes Vector and Integer have both been around since the beginning of Java and do not implement or even know about either the ForwardIterator or EqualityComparable concepts used by find(). However, using our generic system, this code executes flawlessly. It is worth pointing out how this code meets our ‘respect Java’ requirements (and limitation). First off, it only uses objects in the generic aspects of the system. Second, it relies on the runtime type system to check the type of the returned value. And, as we will see, it uses reflection under the hood but still allows the compile time system to be used for maximum performance.

## 2. The Java Generic Programming System

The Java Generic Programming System is composed of a set of design patterns for developing and using generic libraries and a support class for concept checking. The system allows library writers to develop algorithms and concept frameworks that can be used by both new and existing code bases and supports different specialization mechanisms to help improve performance when necessary. From a user’s perspective, the generic algorithms are simply method calls. The user does not have to implement any extra code, but can use the same specialization techniques that library writers can use to increase performance.

### *The GenericProxy Class*

The key to the JGPS is the `GenericProxy` class. The `GenericProxy` class has one static method, `newInstance(Object obj, Class concept)` that checks an object against a concept. If the object models the concept, it returns an instance of a *new* class that proxies the method calls between the object and the generic algorithm. This new class is created at runtime using the `java.lang.reflect.Proxy` class, a custom `InvocationHandler`, and the interface that defines the concept.

For instance, in the example above, a Java `Iterator` was used as a `ForwardIterator`. Under the hood, the JGPS is invoked to map the java `Iterator` to the `ForwardIterator` concept

```
ForwardIterator gfirst = (ForwardIterator)
GenericProxy.newInstance(first, ForwardIterator.class);
```

Because the Java `Iterator` meets all the requirements of the `ForwardIterator` concept, this will succeed and return a proxied object. If it fails, a `ConceptModellingException` is thrown.

### *The Concept Checker Pattern*

In order to support generic algorithms, library writers must follow the Concept Checker pattern. The pattern requires that two methods be implemented for each generic method. The first is the actual method and the second is the *concept checking* method. The signature for the actual method contains the concepts as arguments. Using the `find` example, the actual signature is:

```
static Object find(ForwardIterator first, EqualityComparable value)
```

The body of this method is the same as in the earlier example. It simply implements a `find` algorithm. (Note that most of our examples use static methods. This is simply an implementation choice and is not required.)

The concept checking method uses a general signature based on the fact that all objects in Java have a common base class. Every parameter to the concept checking method must be an `Object`. The concept checking signature for `find` is:

```
static Object find(Object first, Object value)
```

The job of the concept checking method is to check each argument against the concept that is required for the argument and either perform a runtime cast to the correct interface or create a `GenericProxy` for the object. Once the arguments have been cast or proxied, the actual method is called.

Concept checking is accomplished using the `instanceof` operator and the `GenericProxy` via the following pattern, using the first argument from `find` as an example:

```

ForwardIterator gfirst;
if(first instanceof ForwardIterator) {
    gfirst = (ForwardIterator)first;
} else {
    gfirst = (ForwardIterator) GenericProxy.newInstance(first,
                                                         ForwardIterator.class);
}

```

If instanceof is true for an object, the runtime type cast is guaranteed to succeed. Additionally, newInstance is guaranteed to return an object of the concept type. In both cases, the usual requirement for catch ClassCastExceptions is relaxed by the Java compiler, which leads to simpler code. The g prefix is a convention that stands for 'generic'.

Once all the arguments have been checked, the actual method is called:

```
return find(gfirst, gvalue);
```

The general Concept Checker pattern is:

```

// Concept checking method
Object method(Object arg1, Object arg2, ...) {
    Concept1 garg1;
    Concept2 garg2;
    ...

    if(arg1 instanceof Concept1) {
        garg1 = (Concept1)arg1;
    } else {
        garg1 = (Concept1) GenericProxy.newInstance(arg1, Concept1.class);
    }

    if(arg2 instanceof Concept2) {
        garg2 = (Concept2)arg2;
    } else {
        garg2 = (Concept2) GenericProxy.newInstance(arg2, Concept2.class);
    }

    return method(garg1, garg2);
}

// Actual method
Object method(Concept1 arg1, Concept2 arg2) { return something; }

```

An important thing to notice about the Concept Checker pattern is that it lets the compiler choose which version to call. If all arguments implement the concept interfaces directly, then the actual method is called. If one or more arguments do not, then the concept checker is called. Within the concept checker, runtime casts are used to cast any arguments that directly implement the concept interfaces to concepts. This is a very cheap operation in Java. Only objects that do not directly implement the interfaces are proxied.

## Associated Types

Instead of requiring explicit associated types, we introduce two different kinds of associated types: *opaque* associated types and *associated concepts*. Opaque associated types are used when the algorithm does not need to know the type of an object, but rather just shuttle it around. Because all objects inherit from Object in Java, Object is sufficient for opaque types. Associated concepts are the concept requirements on objects that come from parameters but are not parameters themselves and have methods called on them in the algorithm. For instance, consider the Foo interface and callBar algorithm:

```
interface Foo { void bar(); } // Foo concept

static void callBar(ForwardIterator first) {
    Foo current;

    while(first.hasNext()) {
        current = (Foo)first.next();
        current.bar();
    }
}
```

The associated concept requirement of this algorithm is that the elements returned from first.next() model the Foo concept. Foo is a associated concept of ForwardIterator in the context of this algorithm, essentially defining a multi-type concept. As a minor detail, the syntax for doing GP using our system is not this clean and does require some support code.

## The Generic Proxy

The reflection API in Java allows for determining the class of an object and all of the interfaces implemented by a class as well as the methods, parameters, and return types of the methods in a class. However, the reflection API also includes the Proxy class which allows for the creation of a proxy based on an interface and a handler that will be called to invoke any method called on that proxy. We built a class called GenericProxy that uses this proxy feature to create a proxy for any object that implements the methods in a concept that it is expected to model. When a program needs to create a new proxy, it can call the newInstance method in GenericProxy with the object that should model the concept and the concept's class (which is available through the class method on the interface). The GenericProxy class has a private constructor that uses reflection to get the methods of the interface and matches them up with the corresponding methods and signatures in the object that models the concept. When a method is called on a proxy in an algorithm, it is called the same as if it was on an object of a class that directly implements the interface for a concept. However, the method call on the proxy will actually be rerouted to a call on the invoke method of the generic proxy with the method name and parameters. The invoke method in the generic proxy then calls the actual method. From the perspective of code written to implement an algorithm, there is no difference between using a proxy and using an object of a class that implements the concept's interface. This keeps the code that implements an algorithm clean and readable.

The first question that arises with this approach is whether there is a performance cost. Since invoking a method on the proxy requires a call to the invoke method on the generic proxy which must determine which method was called, there will be a performance cost. However, if performance is critical for a particular call, (such as if it is called often) then either the class could be changed to implement the interface, or a proxy for that specific type can be hand-coded. Following is an example of a proxy coded for the ForwardIterator concept used in the find algorithm, for a class called myIterator:

```
public class iterator_proxy implements ForwardIterator {
    myIterator proxyIter;

    public iterator_proxy(myIterator iter) {proxyIter = iter;}

    public Object next() {return(proxyIter.next());}

    public boolean hasNext() {return(proxyIter.hasNext());}
}
```

## Generic Programming Using JGPS

To test the effectiveness of this approach, we implemented two different sets of algorithms. The first is a collection of algorithms from the STL that covers most of the concept categories. The second is a subset of the BGL.

### Generic Collection Algorithms

The STL has four main categories of components: *containers*, *iterators*, *algorithms*, and *functors*. The Java Collection Classes provide a similar set of abstractions, but lack the genericity of the STL. The Java Collection Classes can only be used with each other and libraries designed to work with them. Rather than duplicate the STL, we used the design of the Collection Classes and provided generic algorithms that are analogous to their STL counterparts, but not direct copies. An obvious difference is the lack of a 'last' iterator argument on the algorithms. This is simply because the Collection Classes have a different mechanism for slicing lists, namely `AbstractList.subList()`.

To cover the breadth of components in the STL, we chose the *find*, *fill*, *forEach*, and *transform* algorithms. The find algorithm has already been demonstrated above.

## *fill*

The fill algorithm demonstrates a mutating algorithm. It introduces two new concepts:

```
interface Assignable { public void set(Object obj); }

interface AssignableForwardIterator
    extends ForwardIterator, Assignable { };
```

The assignable concept replaces the dereference/assign construct in the STL (`*iter = val`) and is backwards compatible with the Collection Class' `ListIterator` interface. The full code for the fill algorithm is:

```
static Object fill(AssignableForwardIterator first, Object value) {
    while(first.hasNext()) {
        first.next();
        first.set(value);
    }
    return first;
}
```

## *forEach*

`forEach` is a non-mutating algorithm that uses functors. Working with Method objects in Java is simple and well supported by the language, making it trivial to write algorithms that use functors. To get a reference to a method from a class, the `Class.getMethod()` method is used. `getMethod()` takes the name of the method and an array of Classes that represent the arguments for the method. If the class has a matching method, a reference to it is returned. For instance, consider `TestTransform.printObj(Object obj)` that prints the string representation of the object. The following code gets a reference to the method:

```
Method printObjMethod = TestTransform.class.getMethod("printObj",
    new Class[] { Object.class });
```

With the reference to the functor, the user can call the `forEach` algorithm using `ForwardIterator` and `Method` objects:

```
Vector c = new Vector(); // fill this in...
Algorithms.forEach(c.iterator(), printObjMethod);
```

The implementation of the `forEach` method uses reflection to invoke the functor on each element supplied by the iterator:

```
static ForwardIterator forEach(ForwardIterator first, Method ufunc) {
    return forEach(first, ufunc, null);
}

static ForwardIterator forEach(ForwardIterator first, Method ufunc,
    Class src) {
```



Now, we define three vectors, *a*, *b*, and *c*. Both the *b* and *c* vectors each contain 10 Integers. The elements in *b* contain their index value, i.e. *b* = {Integer(0), Integer(1), ...}. Vector *c* is the results array and the values in *c* are unspecified. The *a* vector contains one Integer element with the value 100.

The following calls to transform apply `sum` and `sum4` to the vectors and store the result in *c*:

```
Algorithms.transform(new Object[] { a.iterator(), b.iterator() },
                    c.listIterator(), sumMethod);

// c = {100, 101, 102, 103, 104, 105, 106, 107, 108, 109}

Algorithms.transform(new Object[] { a.iterator(), b.iterator(),
                                    a.iterator(), b.iterator() },
                    c.listIterator(), sumMethod4);

// c = {200, 202, 204, 206, 208, 210, 212, 214, 216, 218}
```

Because vector *a* contains only one element, the first call just adds the index value to 100. The second call uses vectors *a* and *b* again, but this time it adds each value twice, resulting in  $100 * 2 + \text{index} * 2$ .

## BGL Implementation

The Boost Graph Library is a C++ generic programming graph library that contains algorithms and data structures for traversing, searching and sorting graph structures. This library is an example of the capabilities of generic programming in C++ and was used by Garcia et al. in their comparison of six programming languages that implement generic programming. For purposes of illustrating the generic programming capabilities of the JGPS, we implemented the interfaces, data structures and algorithms needed to perform a breadth-first-search of a graph using generic programming in Java.

## Defining the Concepts

The first step in implementing the BFS is to define the concept interfaces. For the BFS, the graph needs to model both the vertex-list and incidence graph concepts. Both of these concepts extend the graph concept and are defined as follows:

```
public interface ConceptIncidenceGraph extends ConceptGraph {
    Object source(ConceptGraphEdge e);
    Object target(ConceptGraphEdge e);
    Object out_edges(Object v);
    int out_degree(Object v);
}

public interface ConceptVertexListGraph extends ConceptGraph {
    Object vertices();
}
```

```
}
```

The vertex-list concept extends the graph concept by adding a method for getting a forward iterator of the vertices in the graph. The forward iterator is a concept developed for the find algorithm that was then be reused in the graph library. The incidence graph adds methods for getting a forward iterator of the edges as well as being able to get the number of out edges for any vertex as well as the source or target vertex of any edge. However, although these methods need to return an object that *models* the ForwardIterator concept, the return type for these methods is Object. If the return type for the concept methods is other than Object, it would limit the possibility for retroactive modeling. Although the implementation of the graph concept could return an object that implements the ForwardIterator concept (and not just models the concept) this is not required and the BFS algorithm will always ensure that the return type models the ForwardIterator.

As shown above, the source and target methods of the incidence graph concept take a graph edge, which is another concept in the graph library. As we started programming the graph library, we needed to define a number of concepts, which led to our adopting a convention of starting all concept interface names with “concept” so that it was easier to identify the concept interfaces.

The VertexListIncidenceGraph concept shown below illustrates how easily concepts can build on multiple existing concepts, with this interface extending both the edge list and incidence graph concepts:

```
public interface ConceptVertexListAndIncidenceGraph extends  
ConceptVertexListGraph, ConceptIncidenceGraph{}
```

## Implementing the Breadth-First-Search Concept

To implement the BFS algorithm, we created a breadth-first-search class and a graph\_search class as done by Garcia et al. in their comparison involving Java Generics. As can be seen in the code below, the algorithm requires parameters that model the vertex list and incidence graph, the graph “visitor” concept, and a color map which is used to identify the completed nodes during the search. This algorithm calls a method named graph\_search which is not significantly different from not using generic programming since from the viewpoint of the graph\_search routine it is being passed parameters that implement the specific interfaces. The only difference required for generic programming using JGPS is that there is another implementation of the breadth-first-search routine that implements the concept checking pattern as follows:

```
public static void breadth_first_search(Object g, Object s, Object vis,  
Object color) throws ConceptModellingException {  
    ConceptVertexListAndIncidenceGraph gProxy = null;  
    ConceptVisitor visProxy = null;  
    ConceptReadWritePropertyMap colorProxy = null;  
  
    if(g instanceof ConceptVertexListAndIncidenceGraph) {
```

```

        gProxy = (ConceptVertexListAndIncidenceGraph) g;
    } else {
        gProxy = (ConceptVertexListAndIncidenceGraph)
GenericProxy.newInstance(g, ConceptVertexListAndIncidenceGraph.class);
    }

    if(vis instanceof ConceptVisitor) {
        visProxy = (ConceptVisitor) vis;
    } else {
        visProxy = (ConceptVisitor) GenericProxy.newInstance(vis,
ConceptVisitor.class);
    }

    if(color instanceof ConceptReadWritePropertyMap) {
        colorProxy = (ConceptReadWritePropertyMap) color;
    } else {
        colorProxy = (ConceptReadWritePropertyMap)
GenericProxy.newInstance(color, ConceptReadWritePropertyMap.class);
    }

    try {
        breadth_first_search(gProxy, s, visProxy, colorProxy);
    } catch(Exception e) {
        e.printStackTrace();

        System.err.println("Error executing breadth_first_search(): " +
e.toString());
    }
} //end of generic version of breadth_first_search

```

As can be seen from the code above, the concept checker version of BFS calls the actual version of BFS that uses the interfaces as parameters, so from the perspective of that version of the BFS algorithm and the graph\_search routine, the generic version is the same as if the parameters had actually implemented the necessary interfaces.

## Conclusion

As we have demonstrated, by taking advantage of standard features in Java, it is possible to implement support for Generic Programming. The following list addresses each feature presented in Garcia with respect to the JGPS:

- **Multi-type constraints** are handled by documentation using our system. We did not explore this topic for this study.
- **Multiple constraints** are expressed using extends when defining concept interfaces.
- **Associated types** are handled using opaque associated types and associated concepts.
- **Retroactive modeling** comes from using the GenericProxy

- **Type aliases** are not necessary. The use of the runtime system in conjunction with opaque associated types and associated concepts remove this as a requirement for Generic Programming.
- **Separate compilation** is fully supported.
- **Implicit instantiation** is not required since there is only one instance of each generic algorithm.
- **Concise syntax** Both the library writer and the end user can use very concise syntax to use and implement generic algorithms. The concept checker pattern cleanly separates the generic system from the implementation.
- **Performance** is not a specific feature, but is worth mentioning. The techniques used the JGPS do inflict a performance penalty on programs that use it. However, this is in large part because this is a prototype and its goal was to explore what is possible, not how to make it fast. By using other features in Java, including runtime bytecode generation and JNI, it is thought that most of the performance issues can be addressed.

The JGPS provides a refreshing alternative to the currently popular type system approach to Generic Programming. It shows how a strongly typed language with a powerful runtime system can support the basic features of Generic Programming. It also demonstrates how to explore a topic while taking advantage of the features of a language, much in the same way the STL and BGL used C++ to create a strong system for Generic Programming. It is hoped that this study has led to a better understanding of what generic programming can be and how a broader interpretation can lead to new insights.

---

<sup>1</sup> <http://www.boost.org/>

<sup>2</sup> Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock, “A Comparative Study of Language Support for Generic Programming” OOPSLA 2003, p. 115 – 134.

<sup>3</sup> M. Jazayeri, R. Loos, D. Musser, and A. Stepanov, Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, Schloss Dagstuhl, Germany, Apr. 1998.