JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 8, September-October 2004

Some Examples of Generics in Java 1.5 and C# 2.0

Dr. Richard Wiener, Editor-in-Chief, JOT**,** Associate Professor of Computer Science**,** University of Colorado at Colorado Springs

The forthcoming releases of Java JDK v1.5 and C# v2.0 support generic classes (classes with generic type parameters) and generic methods. Furthermore, each also supports constrained generic types.

Much has been written about generic types. In "A Comparative Study of Language Support for Generic Programming" by Garcia et al (http://www.osl.iu.edu/publications/pubs/2003/comparing_generic_programming03.pdf), details of generic types and programming in C++, Haskell, Standard ML, Eiffel, Java and C# are compared. Useful details and generic coding examples from the soon to be released Java JDK 1.5 (now in beta) are presented in the paper "Generics in the Java Programming Language" by Gilad Bracha. Some of the examples used in this column were inspired by this paper.

Generic types in Java and C# introduce more expressiveness at the source code level and move type checking from run-time to compile-time when inserting objects into generic collections. In the current pre-generic versions of Java and C#, genericity in collections is obtained through the backdoor of using the universal super type *Object* as a polymorphic placeholder for the actual reference type that defines the objects to be inserted into the collection. Other than programmer comments, there is little in the source code that reveals the type of object that the programmer intends to hold in the collection. Only by carefully reverse-engineering the source code does this become apparent if in fact only a single type is inserted into the non-generic collection. Generic collections in both Java and C# provide useful self-documentation in the source code in addition to strengthening compile-time type checking.

Java and C# use invariant generic typing in contrast with Eiffel which uses covariant typing. In Java and C# a List<String> is not a subtype of a List<Object>. These are considered separate stand-alone classes. In Eiffel, a List<B> would be considered a subtype of a List<A> if B conforms to A (B is a descendent of A or equal to A). Although covariant typing generally leads to more flexibility in the use of generics, it has been shown to allow code that is statically correct but fails at runtime (see "Type-Save

Cite this column as follows: Richard Wiener: "Some Examples of Generics in Java 1.5 and C# 2.0", in *Journal of Object Technology*, vol. 3, no. 8, September-October 2004, pp. 81-96.
http://www.jot.fm/issues/issue_2004_09/column8

Covariance: Competent Compilers Can Catch All Catcalls" by Mark Howard et al). Wildcards have been introduced into the generic lexicon of Java 1.5 to provide more flexibility in the absence of covariant typing.

The declaration *Collection<T> myCollection* declares *myCollection* as a collection with some unknown element type.

```
// Assume class Person has been defined elsewhere
Collection<?> myCollection = new ArrayList<Person>();
myCollection.add(new Object()); // COMPILE-TIME ERROR
```

The *add* command cannot be invoked on the *myCollection* object since the wildcard represents some unknown type. Since we do not know the type, we are not allowed to pass anything except the value null into *add*.

There is no direct equivalent to wildcards in C# generics but the same functionality can be achieved indirectly. We consider an example used in Gilad Bracha's "Generics in the Java Programming Language." Thanks go to Peter Sestoft (Professor in Information Technology at the Department of Mathematics and Physics, http://www.matfys.kvl.dk/ of the Royal Veterinary and Agricultural University, http://www.kvl.dk, in Denmark) for his thoughtful comments concerning the C# version of the implementation.

We wish to store and maintain a list of lists that contain elements that conform to an abstract class *Shape* (are of type *Shape* or a descendent of *Shape*). We present the Java solution first using wildcards and then show how, without wildcards, we can achieve the same functionality using generic C#.

Listing 1 – List of generic lists extending Shape in Java

```
public abstract class Shape {
    // Commands
    public abstract void draw(Canvas c);
}

public class Circle extends Shape {
    // Fields
    private int x, y, radius;

    // Constructor
    public Circle(int x, int y, int radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    public void draw(Canvas c) {
        System.out.println("In draw method of class Circle.  x = " +
                           x + " y = " + y + " radius = " + radius);
    }
}

public class Rectangle extends Shape {
```

```
    // Fields
    private int x, y, width, height;

    // Constructor
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    // Commands
    public void draw(Canvas c) {
        System.out.println("In draw method in class Rectangle. x = " +
          x + " y = " + y + " width = " + width + " height = " +
          height);
    }
}

public class RightTriangle extends Shape {
      // Fields
      private int x, y, baseTriangle, height;

      // Constructor
      public RightTriangle(int x, int y,
                int baseTriangle, int height) {
         this.x = x;
         this.y = y;
         this.baseTriangle = baseTriangle;
         this.height = height;
      }

      // Commands
      public void draw(Canvas c) {
         System.out.println(
            "In draw method in class RightTriangle. x = " + x +
            " y = " + y + " base = " + baseTriangle +
            " height = " + height);
      }
}

public class Canvas {

    // Commands
    public void draw(Shape s) {
        s.draw(this);
    }
}

import java.util.*;

public class Generics {

    /* A list of lists available in the global namespace of the system.
       Each list contains a list of some shape bounded by Shape
    */
```

```
    static List<List<? extends Shape>> history =
        new ArrayList<List<? extends Shape>>();

    /*  The generic parameter List<? extends Shape> represents
        all classes that are bounded by the supertype Shape.
    */
    public void drawShapeList(List<? extends Shape> shapes) {
        history.add(shapes);
        for (Shape s : shapes) {
            s.draw(new Canvas());
        }
    }

    public static void outputHistory() {
        System.out.println("\nIn outputHistory method.");
        int index = 0;
        for (List<? extends Shape> lst : history) {
            for (Shape shape : lst) {
                shape.draw(new Canvas());
            }
        }
    }

    public static void main(String [] args) {

        List<Circle> circleList = new ArrayList<Circle>();
        circleList.add(new Circle(3, 0, 5));
        circleList.add(new Circle(1, 4, 10));

        List<Rectangle> rectangleList = new ArrayList<Rectangle>();
        rectangleList.add(new Rectangle(0, 0, 10, 20));
        rectangleList.add(new Rectangle(3, 4, 20, 30));
        rectangleList.add(new Rectangle(20, 40, 1, 2));

        List<RightTriangle> triangleList = new
                            ArrayList<RightTriangle>();
         triangleList.add(new RightTriangle(0, 0, 10, 20));
         triangleList.add(new RightTriangle(3, 4, 20, 30));
         triangleList.add(new RightTriangle(20, 40, 1, 2));
         triangleList.add(new RightTriangle(200, 400, 11, 21));
         triangleList.add(new RightTriangle(78, 42, 17, 21));

        Generics app = new Generics();
        app.drawShapeList(circleList);
        app.drawShapeList(rectangleList);
        app.drawShapeList(triangleList);
        outputHistory();
    }
}
```

Program output

In draw method of class Circle. x = 3 y = 0 radius = 5
In draw method of class Circle. x = 1 y = 4 radius = 10
In draw method in class Rectangle. x = 0 y = 0 width = 10 height = 20

```
In draw method in class Rectangle. x = 3 y = 4 width = 20 height = 30
In draw method in class Rectangle. x = 20 y = 40 width = 1 height = 2
In draw method in class RightTriangle. x = 0 y = 0 base = 10 height = 20
In draw method in class RightTriangle. x = 3 y = 4 base = 20 height = 30
In draw method in class RightTriangle. x = 20 y = 40 base = 1 height = 2
In draw method in class RightTriangle. x = 200 y = 400 base = 11 height = 21
In draw method in class RightTriangle. x = 78 y = 42 base = 17 height = 21

In outputHistory method.
In draw method of class Circle. x = 3 y = 0 radius = 5
In draw method of class Circle. x = 1 y = 4 radius = 10
In draw method in class Rectangle. x = 0 y = 0 width = 10 height = 20
In draw method in class Rectangle. x = 3 y = 4 width = 20 height = 30
In draw method in class Rectangle. x = 20 y = 40 width = 1 height = 2
In draw method in class RightTriangle. x = 0 y = 0 base = 10 height = 20
In draw method in class RightTriangle. x = 3 y = 4 base = 20 height = 30
In draw method in class RightTriangle. x = 20 y = 40 base = 1 height = 2
In draw method in class RightTriangle. x = 200 y = 400 base = 11 height = 21
In draw method in class RightTriangle. x = 78 y = 42 base = 17 height = 21
```

Discussion of Listing 1

The declaration,

```
static List<List<? extends Shape>> history =
      new ArrayList<List<? extends Shape>>();
```

defines the static field history as an *ArrayList* containing a *List* of *Shape* objects where *Shape* is a polymorphic placeholder for any conforming sub-type (*Circle, Rectangle or RightTriangle*).

Method *drawShapeList* first adds a new *List*, *shapes*, to *history*. It then uses the new *for* loop construct (C#'s *foreach* loop) to iterate through the shapes and send each shape the *draw* command (which produces output to the console).

Finally, method *outputHistory* uses two nested for loops to iterate through the lists and then for each list to iterate through the shapes contained within that list.

We next examine how to achieve the same functionality in generic C#.

Listing 2 – List of generic lists extending Shape in C#

```
namespace Generics {

    public abstract class Shape {

        // Commands
      public abstract void Draw(Canvas c);
```

```
    }
}

namespace Generics {

    public class Circle : Shape {

        // Fields
        private int x, y, radius;

        // Constructor
        public Circle(int x, int y, int radius) {
            this.x = x;
            this.y = y;
            this.radius = radius;
        }

        public override void Draw(Canvas c) {
            System.Console.WriteLine(
               "In draw method of class Circle.  x = " +
                x + " y = " + y + " radius = " + radius);
        }
    }
}

namespace Generics {

    public class Rectangle : Shape {
        // Fields
        private int x, y, width, height;

        // Constructor
        public Rectangle(int x, int y, int width, int height) {
            this.x = x;
            this.y = y;
            this.width = width;
            this.height = height;
        }

      // Commands
      public override void Draw(Canvas c) {
            System.Console.WriteLine(
               "In draw method in class Rectangle. x = " + x +
               " y = " + y + " width = " + width + " height = " +
               height);
      }
   }
}

namespace Generics {

   public class RightTriangle : Shape {
      // Fields
        private int x, y, baseTriangle, height;
```

```
        // Constructor
        public RightTriangle(int x, int y, int baseTriangle,
                             int height) {
            this.x = x;
            this.y = y;
            this.baseTriangle = baseTriangle;
            this.height = height;
        }


      // Commands
      public override void Draw(Canvas c) {
         System.Console.WriteLine(
            "In draw method in class RightTriangle. x = " + x +
            " y = " + y + " base = " + baseTriangle +
            " height = " + height);
      }
   }
}

namespace Generics {

    public class Canvas {

        // Commands
        public void Draw(Shape s) {
            s.Draw(this);
        }
    }
}

using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;

namespace Generics {
   public class GenericApp {
      public static class Consts<E> where E : Shape {
         public static List<List<E>> history = new List<List<E>>();
         private static void AddShapeList<F>(List<F> shapes)
                             where F : E {
            List<E> lst = new List<E>();
            foreach (Shape s in shapes) {
               lst.Add((E) s);
            }
            history.Add((List<E>)lst);
         }

         public static void DrawShapeList<F>(List<F> shapes)
                               where F : E {
            AddShapeList((List<F>) shapes);
            foreach (Shape s in shapes) {
               s.Draw(new Canvas());
```

```
            }
        }

        public static void OutputHistory<F>() where F : E {
            Console.WriteLine("\nIn OutputHistory method");
            for (int index = 0; index < history.Count; index++) {
                List<E> list = history[index];
                foreach (Shape shape in list) {
                    shape.Draw(new Canvas());
                }
            }
        }
    }

    public static void Main() {

        List<Circle> circleList = new List<Circle>();
        circleList.Add(new Circle(3, 0, 5));
        circleList.Add(new Circle(1, 4, 10));

        List<Rectangle> rectangleList = new List<Rectangle>();
        rectangleList.Add(new Rectangle(0, 0, 10, 20));
        rectangleList.Add(new Rectangle(3, 4, 20, 30));
        rectangleList.Add(new Rectangle(20, 40, 1, 2));

        List<RightTriangle> triangleList = new List<RightTriangle>();
        triangleList.Add(new RightTriangle(0, 0, 10, 20));
        triangleList.Add(new RightTriangle(3, 4, 20, 30));
        triangleList.Add(new RightTriangle(20, 40, 1, 2));
        triangleList.Add(new RightTriangle(200, 400, 11, 21));
        triangleList.Add(new RightTriangle(78, 42, 17, 21));

        Consts<Shape>.DrawShapeList(circleList);
        Consts<Shape>.DrawShapeList(rectangleList);
        Consts<Shape>.DrawShapeList(triangleList);

        Consts<Shape>.OutputHistory<Shape>();
        System.Console.ReadLine();
    }
  }
}
```

Program Output

```
In draw method of class Circle. x = 3 y = 0 radius = 5
In draw method of class Circle. x = 1 y = 4 radius = 10
In draw method in class Rectangle. x = 0 y = 0 width = 10 height = 20
In draw method in class Rectangle. x = 3 y = 4 width = 20 height = 30
In draw method in class Rectangle. x = 20 y = 40 width = 1 height = 2
In draw method in class RightTriangle. x = 0 y = 0 base = 10 height = 20
In draw method in class RightTriangle. x = 3 y = 4 base = 20 height = 30
In draw method in class RightTriangle. x = 20 y = 40 base = 1 height = 2
In draw method in class RightTriangle. x = 200 y = 400 base = 11 height = 21
```

In draw method in class RightTriangle. x = 78 y = 42 base = 17 height = 21

In OutputHistory method
In draw method of class Circle. x = 3 y = 0 radius = 5
In draw method of class Circle. x = 1 y = 4 radius = 10
In draw method in class Rectangle. x = 0 y = 0 width = 10 height = 20
In draw method in class Rectangle. x = 3 y = 4 width = 20 height = 30
In draw method in class Rectangle. x = 20 y = 40 width = 1 height = 2
In draw method in class RightTriangle. x = 0 y = 0 base = 10 height = 20
In draw method in class RightTriangle. x = 3 y = 4 base = 20 height = 30
In draw method in class RightTriangle. x = 20 y = 40 base = 1 height = 2
In draw method in class RightTriangle. x = 200 y = 400 base = 11 height = 21
In draw method in class RightTriangle. x = 78 y = 42 base = 17 height = 21

Discussion of Listing 2

It is not possible in C# to declare *history* as a stand-alone field with a constrained generic parameter as was done in Java. It is necessary to embed the *history* field inside of a static generic class *Consts* that explicitly establishes the generic constraint through,

```
public static class Consts<E> where E : Shape {
   public static List<List<E>> history = new List<List<E>>();
   …
}
```

This class also contains the public methods *DrawShapeList* and *OutputHistory*. Each of these methods utilizes a generic parameter F that extends the constrained generic parameter E.

```
public static void DrawShapeList<F>(List<F> shapes) where F : E {…}

public static void OutputHistory<F>() where F : E {…}
```

The private static method *AddShapeList* requires two downcasts as well.

From the three method invocations,

```
Consts<Shape>.DrawShapeList(circleList);
Consts<Shape>.DrawShapeList(rectangleList);
Consts<Shape>.DrawShapeList(triangleList);
```

the *DrawShapeList* method is able to infer the parameter F (inner list type).

Clearly there is more complexity and subtlety associated with the generic C# implementation than the generic Java implementation.

Since generics in Java was designed to be totally compatible with the existing JVM (Java virtual machine), it offers no performance improvement over straight non-generic Java. One may in fact view Java generics as an extension to the allowable syntax of standard Java with the compiler translating generic code first to standard non-generic code. So the declaration,

Collection<Integer> is seen by the virtual machine as *Collection<Object>*.

Generics were designed into the .NET framework so *List<int>* (aka *List<Int32>*) does not translate to *List<Object>*. There is no wrapping and unwrapping overhead required when constructing collections of value types such as *int* in generic C#. C# uses code specialization (like C++) when implementing collections of value types and uses code sharing (like Java) when implementing collections of non-value types (ordinary reference types). A simple experiment was designed to enable performance comparisons to be undertaken so that the efficiency of C# generics could be compared with that of generic Java both for value types (primitive types in Java) and reference types. Listings 3 and 4 contain the code that was used in this simple experiment. In each experiment, four stacks of base-type integer (*int* in C# and *Integer* in Java) were constructed by pushing one million integer objects onto the first stack and then popping that stack while loading the contents of the first stack onto the second stack and repeating this process until the fourth and last of the stacks is loaded with a million objects and then the objects popped from this last stack. The same scenario was repeated only using *String* as the base type rather than integer objects. Since *String* is a reference type in both languages this second experiment allows us to see whether any significant differences exist between the performance on generic reference types versus value types.

Listing 3 – Benchmark for Java Generics

```
import java.util.*;

public class LinkedStack<T>  {

    // Fields
    private Node<T> top = null;
    private int numberElements = 0;

    // Commands
    public void push(T item) {
        Node<T> newNode = new Node<T> (item, top);
        top = newNode;
        numberElements++;
    }

    public void pop() {
        if (isEmpty())
            throw new NoSuchElementException("Stack is empty.");
        else {
            Node<T> oldNode = top;
            top = top.next;
            numberElements--;
```

```
            oldNode = null;
        }
    }

    // Queries
    public T top() {
        if (isEmpty())
            throw new NoSuchElementException("Stack is empty.");
        else
            return top.item;
    }


    public boolean isEmpty() {
        return top == null;
    }

    public int size() {
        return numberElements;
    }

    private class Node<T> {

        // Fields
        public T item;
        public Node<T> next;

        // Constructors
        public Node(T element, Node<T> link) {
            item = element;
            next = link;
        }
    }

    static public void main(String[] args) {
        LinkedStack<Long> intStack = new LinkedStack<Long>();
        TimeInterval time = new TimeInterval();
        time.startTiming();
        // Push 1,000,000 ints onto the stack
        for (int i = 0; i < 1000000; i++) {
            intStack.push((long) i);
        }

        LinkedStack<Long> intStack2 = new LinkedStack<Long>();
        // Get sum of ints on stack
        long sum = 0L;
        for (int i = 0; i < 1000000; i++) {
            sum += intStack.top();
            intStack2.push(intStack.top());
            intStack.pop();
        }
        LinkedStack<Long> intStack3 = new LinkedStack<Long>();
      for (int i = 0; i < 1000000; i++) {
         sum += intStack2.top();
         intStack3.push(intStack2.top());
         intStack2.pop();
```

```
        }
        LinkedStack<Long> intStack4 = new LinkedStack<Long>();
        for (int i = 0; i < 1000000; i++) {
           sum += intStack3.top();
           intStack4.push(intStack3.top());
           intStack3.pop();
        }
        for (int i = 0; i < 1000000; i++) {
           sum += intStack4.top();
           intStack4.pop();
        }
        time.endTiming();
        System.out.println("sum = " + sum);
          System.out.println("Elapsed   time   for   intStack   =   "   +
                               time.elapsedTime() + " seconds.");

          LinkedStack<String> strStack = new LinkedStack<String>();
          time = new TimeInterval();
          time.startTiming();
          // Push 1,000,000 ints onto the stack
          for (int i = 0; i < 1000000; i++) {
              strStack.push("ABCDEFG");
          }

          LinkedStack<String> strStack2 = new LinkedStack<String>();
          for (int i = 0; i < 1000000; i++) {
              strStack2.push(strStack.top());
              strStack.pop();
          }
          LinkedStack<String> strStack3 = new LinkedStack<String>();
          for (int i = 0; i < 1000000; i++) {
              strStack3.push(strStack2.top());
              strStack2.pop();
          }
          LinkedStack<String> strStack4 = new LinkedStack<String>();
          for (int i = 0; i < 1000000; i++) {
              strStack4.push(strStack3.top());
              strStack3.pop();
          }
          for (int i = 0; i < 1000000; i++) {
              strStack4.pop();
          }
          time.endTiming();
          System.out.println("sum = " + sum);
          System.out.println("Elapsed   time   for   strStack   =   "   +
                               time.elapsedTime() + " seconds.");

    }
}

public class TimeInterval {

    private long startTime, endTime;
    private long elapsedTimeInterval; // Time interval in milliseconds

    // Commands
```

```
    public void startTiming() {
        elapsedTimeInterval = 0;
        startTime = System.currentTimeMillis();
    }

    public void endTiming() {
        endTime = System.currentTimeMillis();
        elapsedTimeInterval = endTime - startTime;
    }

    // Queries
    public double elapsedTime() {
        return (double) elapsedTimeInterval / 1000.0;
    }
}
```

Program Output
sum = 1999998000000
Elapsed time for intStack = 2.11 seconds.
sum = 1999998000000
Elapsed time for strStack = 0.937 seconds.

Listing 4 - Benchmark for Java Generics

```
using System;
using System.Collections.Generic;
using System.Text;
using Timer;

namespace Stackbenchmark {

   public class StackBenchmarkApp {

      static public void Main() {
         LinkedStack<long> intStack = new LinkedStack<long>();
         Timing time = new Timing();
         time.StartTiming();
         // Push 1,000,000 ints onto the stack
         for (int i = 0; i < 1000000; i++) {
            intStack.Push((long)i);
         }

         LinkedStack<long> intStack2 = new LinkedStack<long>();
         // Get sum of ints on stack
         long sum = 0L;
         for (int i = 0; i < 1000000; i++) {
            sum += intStack.Top();
            intStack2.Push(intStack.Top());
            intStack.Pop();
         }
         LinkedStack<long> intStack3 = new LinkedStack<long>();
         for (int i = 0; i < 1000000; i++) {
            sum += intStack2.Top();
```

```
            intStack3.Push(intStack2.Top());
            intStack2.Pop();
        }
        LinkedStack<long> intStack4 = new LinkedStack<long>();
        for (int i = 0; i < 1000000; i++) {
            sum += intStack3.Top();
            intStack4.Push(intStack3.Top());
            intStack3.Pop();
        }
        for (int i = 0; i < 1000000; i++) {
            sum += intStack4.Top();
            intStack4.Pop();
        }

        time.EndTiming();
        Console.WriteLine("sum = " + sum);
        Console.WriteLine("Elapsed time for int stack = " +
                          time.ElapsedTime() + " seconds.");

        LinkedStack<String> strStack = new LinkedStack<String>();
        time = new Timing();
        time.StartTiming();
        // Push 1,000,000 ints onto the stack
        for (int i = 0; i < 1000000; i++) {
            strStack.Push("ABCDEFG");
        }

        LinkedStack<String> strStack2 = new LinkedStack<String>();
        for (int i = 0; i < 1000000; i++) {
            strStack2.Push(strStack.Top());
            strStack.Pop();
        }
        LinkedStack<String> strStack3 = new LinkedStack<String>();
        for (int i = 0; i < 1000000; i++) {
            strStack3.Push(strStack2.Top());
            strStack2.Pop();
        }
        LinkedStack<String> strStack4 = new LinkedStack<String>();
        for (int i = 0; i < 1000000; i++) {
            strStack4.Push(strStack3.Top());
            strStack3.Pop();
        }
        for (int i = 0; i < 1000000; i++) {
            strStack4.Pop();
        }
        time.EndTiming();
        Console.WriteLine("Elapsed time for str stack = " +
                          time.ElapsedTime() + " seconds.");
        Console.ReadLine();
    }
  }
}

using System;

namespace Timer {
```

```
    public class Timing {
        // Fields
        private long startTicks, endTicks;

        public void StartTiming() {
            DateTime t = DateTime.Now;
            startTicks = t.Ticks;
        }

        public void EndTiming() {
            DateTime t = DateTime.Now;
            endTicks = t.Ticks;
        }

        public double ElapsedTime() {
            return (endTicks - startTicks) / 10000000.0;
        }
    }
}
```

Program Output
sum = 1999998000000
Elapsed time for int stack = 0.921875 seconds.
Elapsed time for str stack = 0.78125 seconds.

Discussion of Listing 3 and 4

The experiments were conducted on the same computer. The ratio of execution times is interesting.

For integer types (primitive type in Java and value type in C#), generic Java was 2.29 slower than generic C#. That is significant. For reference types, generic Java was 1.20 times slower. That is not significant and is in part due to the overall efficiency of Java JIT compared with the C# JIT (other experiments have suggested that the C# JIT is slightly more efficient than the Java 1.5 JIT).

In conclusion, the design decision to use code specialization for value types in C# has paid off in performance benefits. The wildcard semantics in generic Java appear to provide a more straight-forward mechanism for expressing complex generic constructions than the equivalent C# implementation.

About the author

Richard Wiener is Associate Professor of Computer Science at the University of Colorado at Colorado Springs. He is also the Editor-in-Chief of JOT and former Editor-in-Chief of the Journal of Object Oriented Programming. In addition to University work, Dr. Wiener has authored or co-authored 21 books and works actively as a consultant and software contractor whenever the possibility arises.