# Implementing design patterns in Java using C# 3.0 concepts

Judith Bishop
Dept Computer Science
University of Pretoria
Pretoria, South Africa
jbishop@cs.up.ac.za

Andrich van Wyk
Dept Computer Science
University of Pretoria
Pretoria, South Africa
avanwyk@cs.up.ac.za

## ABSTRACT

In the object-oriented language world, it is often the case that general purpose languages such as Java and C# are considered as very similar. From a distance, their good and bad points are glossed over, and there is very little hard evidence as to whether particular features, or groups of features really matter in terms of readability, writeability, maintainability, and most of all, efficiency. A valid corpus for evaluating these differences is the set of classic design patterns. We have implemented these afresh in C# 3.0, and then translated the implementations back into Java. This paper presents the results of this investigation into the language features that C# 3.0 has included, that Java 5.0 does not have, and that are relevant to the implementation of design patterns. These absent features were covered by the implementation of generic Delegate and Property classes, as well as boilerplate classes for the provision of generic yield-based Iterators and LINQ queries. The evidence suggests that Java has not up until now been stretched in terms of best practice for design patterns, and that the lessons learnt from C# can render improvements in these implementations.

## Categories and Subject Descriptors

D.3.2 [**Language classifications**] Object-oriented languages, D.3.3 [**Programming Languages**]: Language Contructs and Features: Patterns

**General Terms:** Algorithms, Languages

**Keywords**: Java 1.5, C# 3.0, delegates, properties, iterators, LINQ, language translation, language comparison

## 1. INTRODUCTION

What's in a language? Benjamin Whorf wrote in the 1950s that "Language shapes the way we think, and determines what we can think about." [11] Expounding on this theme, Alford [1] said that Whorf had two hypotheses:

> *"Structural differences between language systems will, in general, be paralleled by non-linguistic cognitive differences, of an unspecified sort, in the native speakers of the language.*
>
> *The structure of anyone's native language strongly influences or fully determines the worldview he will acquire as he learns the language." [1]*

Both of these statements have profound implications for computer science when applied to programming languages. Both home in on the notion of a native language, which in computer terms is one's first language. Whorf is saying that the first language one learns has an influence on one's "worldview". In computing terms, this could be interpreted as an inability to see and grasp concepts not present in the first language. A popular quote from the great 20th century philosopher, Ludwig Wittgenstein, is in line with this view: "The limits of my language mean the limits of my world. All I know is what I have words for."

The last sentence refers to words, or *vocabulary*. A programming language (in a given version) has a fixed syntax and semantics but it does not have a finite vocabulary: all languages are extensible by the addition of new libraries that introduce new namespaces and hence new "words" to the language. However, a programmer can miss them, or consciously ignore them, staying within the bounds of an earlier taught vocabulary. Whorf's emphasis on the *structure* of a language is more serious. He is saying that cognition will be influenced, and certainly will be different, depending on the structures in one's first language. Thus, if a programmer moves on to a second language, he is less likely to be able to absorb or use concepts that were not present originally.

For both natural languages and programming languages this statement is a generalization. There will be many individuals who through a lifetime or a career of thirty years or more, will be fully fluent and productive to a high level in several languages. In the case of natural languages, there are people who speak three or four fluently; in the case of programming languages, a professor who teaches the subject would be expected to be or have been thoroughly conversant with up to twenty. However, the vast majority of the population will conform to Whorf's hypothesis and will have a tendency to see the world through the structures and vocabulary that they learnt first.

It is the purpose of this paper to present an experiment in the technical possibilities of extending one language (Java 1.5) according to the structures of another (C# 3.0). The driving force behind this investigation is to debunk the commonly held belief that "Java and C# are the same" or if not the same, then of the same structure (to use Whorf's term). We hold that this is not the case and that C# 3.0 has significant features that make it a more advanced and powerful medium for programming. Nevertheless,


*Conference'04*, Month 1–2, 2004, City, State, Country.

since Java is so entrenched in industry and academia, we look at how some of the concepts that are present in C# could be provided in Java.

## 1.1 Methodology

In the object-oriented language world, it is often the case that general purpose languages such as Java and C# are considered as very similar. From a distance, their good and bad points are glossed over, and there is very little hard evidence as to whether particular features, or groups of features really matter in terms of readability, writeability, maintainability, and most of all, efficiency. A valid corpus for evaluating these differences is the set of classic design patterns.

> *Design patterns encapsulate common ways of using language features together.*

The original 23 design patterns were implemented in C++ and Smalltalk in 1995 [5] and have over the years been implemented also in VB, Java and now C#. A well-known C# implementation that is available online is that from the DoFactory [4]. DoFactory is a commercial organization that sells frameworks of patterns in C# and Visual Basic. They are widely consulted. The implementations come in three versions – structural, real-world and .NET optimized. Since the programs have not been updated since 2006, they do not include any features new to C# 3.0.

We implemented all the patterns afresh in C# 3.0 [3] and then translated the implementations back into Java. In so doing, certain features stood out as lacking. This paper presents the results of this investigation into the language features that C# 3.0 has included, that Java 5.0 does not have, and that are relevant to the implementation of design patterns. Three of these are properties, iterators and delegates. We present the related code and assess how the Java implementations could be improved by using a extended "vocabulary" in the "key of C#" as it were.

The benefits of such a translation are:

- Java programmers have an extended vocabulary with which to work.
- Java programmers will have practice with features that might be added to the language later.
- Improved mobility of developers in both directions.

## 1.2 Previous work

As languages evolve, there is a need and a tendency to compare them. The comparisons tend to consist of lists of features that appear or do not appear in each language. However, since we are concentrating on C# 3.0 and Java 1.5 (both post 2005 languages) the literature in this regard is rather thin.

One of the earliest to come out for Java and C# was by Obasanjo in 2001 [9]. His thorough comparison stands as the standard work on the subject. It was updated in 2007 for C# 3.0. In the 2007 conclusion to that work, the author states:

> *"As predicted in the original conclusion to this paper, a number of features have become common across both C# and Java since 2001. ... However after years of convergence it seems that C# and Java are about to go in radically different directions. ... the differences between C# and Java will become more stark over the next few years in contrast to the feature convergence that has been happening over the past few years." [9]*

Indeed, it is this divergence that we wish to highlight and in some ways overcome.

Another recent work is [7] which presents the resulys of an experiment to convert Java code to C#. The paper's focus is on automated language transformers, but it does have interesting tables in the appendix that map Java features to C# equivalents, where they are not identical. Microsoft itself provides a similar translation tool JLCA [6] that we have used with success in other work.

Translations in the other direction (C# to Java) do not seem to exist. This is a novel sideline of this study.

# 2. Languages and patterns

## 2.1 C#

C# 1.0 was announced by Microsoft with the first .NET release in 2000. It made significant advances over Java, which had been seriously in use for three years by then. The second column in Table 1 summarises the novel features of C# 1.0 as compared to Java. C# 2.0 added five important features in 2005, especially generics, which had been available in some implementations for two years. C# 3.0, finalized in 2006, focused on features that would bring the language closer to the needs of databases and has a distinct functional feel about it [8].

## 2.2 Java

The development of the Java language has been far less dramatic. With the Java virtual machine (JVM) on which Java runs being circulated worldwide by numerous players (for example in all browsers), Sun Microsystems was severely constrained as far as changes to the language and its supporting bytecode were concerned. Although the Java Platform forged ahead with a constant array of new APIs and updates to existing ones, the language did not receive a significant boost until 2004. The third column in Table 1 shows the six language improvements that were introduced then. The bottom row of the table highlights what could be called language-related APIs, those for reflection, collections and iterators. The energy and excitement surrounding Java has been in the development of its APIs which have stretched into every corner of computing.

## 2.3 Design patterns

It is still open territory as to whether, and how, these new language features should be used in implementing design patterns. In books and writings on web sites the pull of custom is very strong. Because implementations of the patterns were originally given in C++ and Smalltalk, which have their own particular object-oriented styles, the translations into other languages have not always been completely satisfactory. It is a challenge to make the most of a language, while at the same time retaining the link with the design pattern and its terminology.

Although design patterns do not force a certain way of coding, a look at the expository examples in most Java or C# books will show little deviation from the C++ style of the 1990s. It would seem that the promise of language features making patterns easier to implement has been slow to realize. The features are there now, and it is a question of showing how they can be used, and in assessing their efficiency. Not all the features listed in Table 1 are directly relevant for patterns, but a surprising number are.

| *Java 1.2 1998* | *C# 1.0 2002* | *Java 5.0 2004* | *C# 2.0 2005* | *C# 3.0 2007* |
|---|---|---|---|---|
| inner classes | structs<br>properties<br>foreach loop<br>autoboxing<br>delegates and events<br>indexers<br>operator overloading<br>enumerated types with IO<br>in, out and ref parameters<br>formatted output | generics<br>autoboxing and unboxing<br>enhanced for loop<br>typesafe enumerations<br>varargs<br>static import | generics<br>anonymous methods<br>iterators<br>partial types<br>nullable types<br>generic delegates | implicit typing<br>anonymous types<br>object and array initializers<br>extension methods,<br>lambda expressions<br>query expressions (LINQ) |
| *API*<br>Reflection<br>Collections framework | *API*<br>Reflection | *API*<br>Iterable | | standard generic delegates |

**Table 1. Timeline of the development of Java and C# language features**

## 2.4 Design patterns and language features

The implementations in Bishop [3] had the specific aim of exploring new language features. They come in two versions, known as the Theory code and Example code. The Theory code is similar in length and intent to the Structural versions from DoFactory, and presents a minimalist version of each pattern, in which the essential elements can be seen in stark relief. The Examples add flesh to the pattern, and in many cases use more or slightly different features as a result.

In [2] we itemized those pattern implementations in DoFactory's NETOptimized and/or Bishop's Example sets that use advanced C# features. All of the pattern implementations made use of normal OOPS features such as inheritance, overriding, composition, access modifiers and namespaces, object and array initializers. From the list presented there, we can extract the following non-Java features:

1. Auto-properties are used in most patterns that have data classes.
2. Yield-based Iterators appear in the Iterator pattern.
3. Delegates appear in five patterns – Adapter, Command, Mediator, Chain and Observer.
4. Query expressions are used in the Iterator pattern.

We were able to implement all of these in Java by means of wrapper classes, but we had to make use of the most uptodate Java features, as added to Java 1.5 (see Table 1). These are

1. Generics – including the rarely used class <?> construct for unspecified classes, to be used along with reflection
2. Enhanced for loop – mirroring C#'s foreach and a cleaner Iterable interface.
3. Typesafe enumerations
4. Varargs for generalizing methods over variable number of parameters with the new syntax …

Based on web searching for these constructs, it would seem that they have not been widely incorporated into the Java programmer's repetoire yet. This paper therefore has the additional purpose of exposing their usefulness both in design patterns and in general.

# 3. The experimental program

We translated all the programs in [3] to Java, and they worked. We then started on three of the theory programs illustrating the features above: Mediator, Iterator and Prototype. For Mediator, we defined a Delegate class that enabled us to mimic the original C# design and implementation. For Iterator, we implemented a customized Iterator class with the new Java Iterable interface. For the Prototype, we wrote a simple Property class.

We then embarked on a more challenging example, that of querying a tree via LINQ based structures. LINQ is a set of extensions to the .NET Framework that encompasses language-integrated query, set, and transform operations. It extends C# and Visual Basic with native language syntax for queries and provides class libraries to take advantage of these capabilities. The objective was to maintain as much of the "syntactic sugar" provided by C# for LINQ. The premise is that this syntactic sugar is "good sugar", making programs leaner and cleaner, and therefore more maintainable and more easy to develop.

To support LINQ-like syntax in Java, we went through numerous iterations of our new Delegate, Iterator and Property classes, making them more powerful and more suited to general use. The driver program for the experiment is shown in the Appendix, alongside the full program in C#. In the remainder of this section, we discuss the three classes that support LINQ in Java, referring to their use in the Java drive in the Appendix. Each is introduced with one its own design pattern first.

## 3.1 The Delegate class and LINQ

The delegate facility in C# allows the creation of types and objects of them that will have methods as values. At a low level of detail they are similar to C++ function pointers, but Java does not have them at all. Not only are delegates used in several of our C# 3.0 design pattern implementations, they are also the underlying basis for the LINQ mechanism. Referring to the C# program in the Appendix, we can see the following LINQ query for a subset of the family data set up in the object initializer earlier:

```
var selection = from p in family
   where p.Birth > 1980
   orderby p.Name
   select p;
}
```

Leaving aside the sorting, this excerpt can be translated as a C# 3.0 lambda expression:

```
var selection2 = family.
    Where (p => p.Birth > 1980);
```

Which in itself is syntactic sugar for an anonymous delegate as in:

```
var selection3 = family.
   Where (delegate (Person p) {
       return p.Birth > 1980;});
```

In this usage, Where is a heavily disguised method declared on any collection class. Its definition is

```
//C# 3.0
public static IEnumerable<T> Where<T>(
    this IEnumerable<T> source,
    Func<T, boolean> predicate) {
    foreach (var item in source) {
      if (predicate(item)) {
         yield return item;
      }
    }
}
```

Why does Where, as defined, take two parameters when Where, as called above, takes only one? The reason is that Where is an *extension method* to any generic enumerable type (virtually anything). As such, it specifies its type in its first parameter, but does not require it when called. Extension methods are defined separately to the class in C# 3.0. The second parameter is a *built-in generic delegate* that returns the type of the last parameter (in this case boolean) and accepts as many parameters as are listed before that, in this case only one of type T.

Given this explanation of the C# 3.0 query syntax, we created a Delegate class in Java 1.5, making heavy use of reflection (Figure 1). The Delegate constructor takes the object in which the delgate is based, as its first parameter, then the method name, then the type of any number of parameters for the method. These are specified using Java's new *varargs* feature. The method is looked up and stored in the field name, m.

```
// Java 1.5
import java.lang.reflect.*;

 class DelegateCreationException
   extends RuntimeException {};
 class DelegateInvocationException
   extends RuntimeException {};

 public class Delegate<T> {
   Method m;
   Object self;
   public Delegate(Object obj, String name,
            Class<?>... argTypes) {
     self = obj;
     try {
       m = obj.getClass().
           getDeclaredMethod(name, argTypes);
     } catch (NoSuchMethodException e) {
         throw new DelegateCreationException();
    }
 }

 public T invoke(Object... args) {
   try {
     T result = (T)m.invoke(self, args);
       return result;
     } catch (IllegalAccessException e) {
         throw new DelegateInvocationException();
     } catch (InvocationTargetException e) {
         throw new DelegateInvocationException();
     }
   }
 }
```

**Figure 1 Generic Delegate class in Java**

The declaration of an anonymous delegate object in Java using this class is reassuringly close to the original C# 3.0 (taken from the Appendix):

```
// Java
Iterable<Person> selection = family.where(
      new Delegate<Boolean>
         (this,"after1980",Person.class));
```

The where method in Java would be declared alongside the iterator for this selection, as described in the next section.

Another use of delegates is in the Mediator pattern, which is shown in Figure 2.

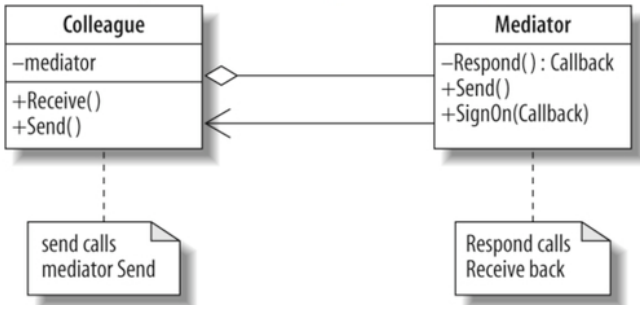


**Figure 2 Mediator pattern**

The Mediator pattern enables objects to communicate according to a certain protocol without knowing each other's identities. These are given to the Mediator, which handles the traffic. In our C# 3.0 version, the Callback is a delegate. Callbacks use the feature of delegates that enables a chain of methods to be stored, so that when the delagte object is called, all of the subscribed methods are activated. This is possible also using the Java Delegate class, simply by declaring respond (the Callback object's name) as a list of such delegates.

## 3.2 Iterators

The Iterator design pattern is one that lends itself most to language assistance. Many languages in the past three decades have investigated and perfected iterator mechanisms. The diagram

in Figure 3 shows our design of a C# 3.0 Iterator pattern. It relies on the foreach loop, on a collection that has an enumerator implemented for it (as with a list or array) or on a customized one (as for a tree). Finally, the yield-return mechanism interacts with the foreach to supply items in a lazy fashion. In the program in the Appendix, the C# 3.0 LINQ query is not executed until the foreach starts. Then each time the loop interfaces with the corresponding iterator, it will yield a value of the correct type back.

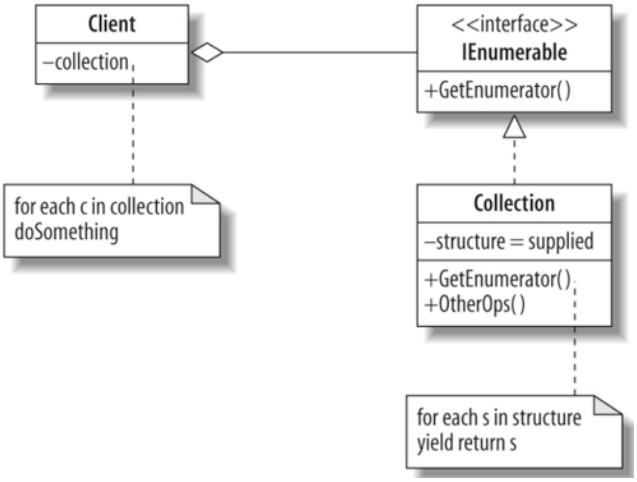


**Figure 3 Iterator pattern**

Unfortunately, the iterator mechanism in Java is not the same. In Java, the foreach loop connects to an appropriate Iterator that has a Next method. This is called by the loop in a simply way for each iteration of the foreach.

Most examples of iterators in Java 1.5 and C# 3.0 only go as far as lists, for which built in Next methods and yields exist. The program in the Appendix deliberately employs a generic Tree class, using a generic Node class. The C# loop to process the whole tree includes a call to Preorder (it could just as well have been any other order appropriate for a tree). In the Tree class we find the code in Figure 4.

```
//C# 3.0
public IEnumerable <T> Preorder {
  get {return ScanPreorder (root);}
}

// Enumerator with T as Person
   private IEnumerable <T>
     ScanPreorder (Node <T> root) {
     yield return root.Data;
     if (root.Left !=null)
       foreach (T p in ScanPreorder (root.Left))
           yield return p;
       if (root.Right !=null)
        foreach (T p in ScanPreorder (root.Right))
           yield return p;
     }
   }
```

**Figure 4 C# 3.0 Yield-return mechanism with foreach**

The yield statement implements a coroutine mechanism, allowing the foreach in the calling routine to continue after each item has been supplied.

Since Java does not have such a mechanism, all iterators for foreach loops must implement the next, hasNext and remove methods, and the calling is in one way only. For complex structures such as trees, the next method also has to employ backup support to implement recursion so that a return can be made after each item. Nevertheless, we can mimic the behaviour of the yield-return by using sensible identifier names, for example having a yield method.

```
// Java
public class PreorderIterator
             implements Iterator <T> {
  protected Stack<Node<T>> traversalStack;

  PreorderIterator (Node <T> root) {
    traversalStack = new Stack<Node<T>>();
    traversalStack.push(root);
  }

  //Standard Java Iterator methods
  public boolean hasNext() {
    return !traversalStack.empty();
  }

  public T next () {
    return yield().data.get();
  }

  public void remove() {
    throw new UnsupportedOperationException(
              "Method not implemented");
  }

  protected Node<T> yield() {
    Node<T> node = traversalStack.pop();
    if (node.right.get() != null)
      traversalStack.push(node.right.get());
    if (node.left.get() != null)
      traversalStack.push(node.left.get());
    return node;
 }
}
```

**Figure 5 Generic yield-based Iterator in Java**

Comparing Figure 4 and Figure 5, we see that the Java is not very much more complex than the C#. C# can safely use recursion, because the coroutine mechanism will pass control back to the foreach, and when the next item is needed, the recursive stack is still correct. Java has to emulate this process.

## 3.3 The Property Class

We end up by taking a simple pattern that has a direct translation from C# 3.0 to Java – but has a surprising twist in the tail. The Prototype pattern, illustrated in Figure 6, creates new objects by cloning one of a few stored prototypes. The list of prototypes is maintained in a dictionary data structure. The point that the pattern illustrates is that there is a need for a deep copy process when the prototype is a data structure, rather than a single level object. Thus the familiar clone method needs to be extended by serializing to disk and back again to achieve the deep copy. Fortunately, both languages provide serialization and the sequence of steps to activate it is well known.

To test the Prototype pattern, one needs to set up a class with several fields, such as in Figure 7.

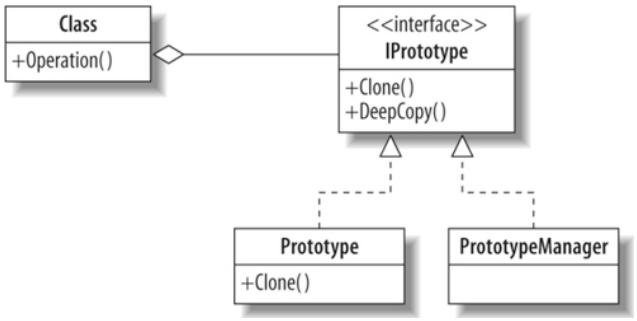


**Figure 6 Prototype Pattern**

DeeperData is a secondary class that will mean that Prototype objects will need serialization when they are copied. Country is a field that gets a value in the constructor and can be seen by objects of the class, but cannot be overwritten. This is indicated by the omission of the special word set in the property.

```
//C# 3.0
class Prototype : IPrototype <Prototype>  {

    public string Country {get;}
    public string Capital {get; set;}
    public DeeperData Language {get; set;}

    public Prototype (string country,
          string capital, string language) {
      Country = country;
      Capital = capital;
      Language = new DeeperData(language);
    }

    public override string ToString() {
      return Country+"\t\t"+Capital+
          "\t\t->"+Language;
    }
  }
```

**Figure 7 Prototype class in C# 3.0**

As would be normal in C#3.0 programming, the Prototype class uses automatic properties[1]. Country is one such. It hides a private field of the given type and provides get and set access to it. Either of the properties can be omitted. These automatic properties are a considerable improvement over the C# 1.0 version that required the declaration of the private variable as well, as in:

```
// C# 1.0
private string country;
public  string Country {
  get {return country;}
  put {country = value;};
```

Compare this to the Java equivalent:

```
// Java 1.5
private String country;
public String getCountry() {
  return country;
}
public void setCountry(String c) {
   country = c;
}
```

[1] The term properties is more often used to refer to the fields of a GUI component in Java. Technically these are the same as the fields we describe, but having already been declared in a Java API, GUI properties would not be susceptible to this approach.

Apart from the lines-of-code metric that one line in C# 3.0 becomes six in Java 1.5 (for every field) there are inherent insecurities in the Java version. The connection between the accessor methods and the variable is a convention only. It is a common error to cut and paste this code for additional fields and then not to update them completely. The difference in usage is also significant:

```
//C#
c.Prototype = "China";

//Java 1.5
c.setCountry("China");
```

The intention of the C# syntax is to render private fields accessible in a variable-like syntax, but still to retain full control over what can be altered. We therefore implemented a generic Property class that would give Java programmers a simpler way to define fields. The class is given in Figure 8. The class encapsulates a private field of type T and, right at the bottom, provides get and set methods to it.

```
// Java
import java.io.Serializable;
public class Property <T>
             implements Serializable {
//Property class J Bishop and A van Wyk, May 2008
// Mimics C3.0's automatic properties

  public static enum Kind
                {GETANDSET, GETONLY, SETONLY};
  private T x;
  private Kind propertyKind = Kind.GETANDSET;

  public Property () {
    x = null;
  }
  public Property (Property<T> copy) {
    x = copy.get();
  }
  public Property (T value) {
    x = value;
  }
  public Property (T value, Kind p) {
    x = value;
    propertyKind = p;
  }

  public T get() {
    if (propertyKind != Kind.SETONLY)
      return x;
    else {
      System.out.println(
                "Invalid get property access");
      System.exit(0); return null;
    }
  }
  public void set (T value) {
    if (propertyKind != Kind.GETONLY)
       x = value;
    else {
       System.out.println(
         "Invalid set property access");
       System.exit(0);
    }
   }
 }
```

**Figure 8 Generic Property class in Java**

In order to provide for different combinations of get and set accessors, there is enum field on the value constructor. Within the get and set methods, this field is queried each time. This is not efficient, and we are investigating better ways, such as conditional compilation.

In the program in the Appendix, there are two declarations of Properties in the Person class. As objects, these are instantiated in the constructor and are used in the toString method. However, all is not so simple.

Going back to the Prototype pattern, its implementation in Java follows that of C# of the Prototype class except that it has to implement the clone method for its simple data, now represented as Property objects. Thus we have:

```
// Java 1.5
public Prototype clone() {
   Prototype copy = new Prototype();
   copy.country = new
     Property<String>(this.country);
   copy.capital = new
     Property<String>(this.capital);
   copy.language = new
     Property<DeeperData>(this.language);
   return copy;
}
```

In the C# 3.0 versions of the abstract class IPrototype, use is made of the built-in generic MemberwiseClone method which sorts all this out. Note that in all implementations, deep copy – for a data structure – must still be done using serialization.

## 4. Evaluation

The preceding discussion has described in detail an experiment in using C# 3.0 implementations of design patterns as models for Java implementations. In so doing, we have

- implemented three generic classes – Delegate, Property,
- provided a boilerplate for complex Yield-based iterators in Java, and
- emulated the LINQ queries.

Each of the classes has been tested in more than one Pattern program, and as described in Section 3.3, the classes had to be upgraded under emerging requirements. The classes themselves are small, and they render the implementation of programs in Java 1.5 remarkably similar to those in C# 3.0.

It is important to realize that it is the latest versions of the languages that we are employing: neither of the sets of patterns will compile on older versions.

### 4.1 Other work

As mentioned in Section 1.2 there has not been much work reported on stretching Java 1.5. An older and more complex attempt at a Delegate class exists at [13]. We could not find a prior attempt to mimic Properties. Iterators in Java are well explored, but we did not find an example of yield-based iteration.

## 5. Conclusions

At the start of this paper, we introduced the notion of language shaping the way we thing. If one starts out in programming with what is evidently a higher-level language, then moving to a lower one presents challenges. This experiment shows that those challenges can be overcome by mimicking the features that are lacking using classes. This requires careful programming, and, in the case of Java, we had to stretch it to its limits. From the point of view of obtaining a user program that looked in Java the same as in C#, the experiment could be deemed a success.

Future work involves testing more of our Java patterns, and looking for more avenues to improve Java's power.

## REFERENCES


[1] Alford, Danny K H, Demise of the Whorf hypothesis, *Phoenix: New Directions in the Study of Man*, Vol. IV, Nos. 1 and 2, 1980, found at http://www.enformy.com/dma-dwh.htm, accessed on 5 May 2008

[2] Bishop, J, and Horspool, R. N., On the Efficiency of Design Patterns Implemented in C# 3.0, TOOLS Europe 2008, to appear.

[3] Bishop, J., **C# 3.0 Design Patterns**. O'Reilly Media, Sebastapol, CA, 2008

[4] Data and Object Factory, Design Pattern Framework: C# Edition. http://www.dofactory.com/Default.aspx 2006, accessed on 5 May 2008

[5] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. **Design Patterns: Elements of Reusable Object-Oriented Software**. Boston, MA, Addison-Wesley (1995).

[6] Java Language Conversion Assistant [Online]. Available: http://msdn.microsoft.com/en-za/aa718346(en-us).aspx/ accessed on 5 May 2008

[7] M. El-Ramly, R. Eltayeb, H.A. Alla, An Experiment in Automatic Conversion of Legacy Java Programs to C#, IEEE International Conference on Computer Systems and Applications, pp. 1037-1045, 2006

[8] Microsoft Corporation: C# 3.0 Reference Documentation, http://msdn2.microsoft.com/vcsharp

[9] Obasanjo, D., A comparison of Microsoft's C# programming language to Sun Microsystem's Java Programming Language, http://www.25hoursaday.com/CsharpVsJava.html, 2007, accessed on 5 May 2008

[10] Steven D. Fraser, James Gosling , Anders Hejlsberg, Ole Lehrmann Madsen, Bertrand Meyer, Guy Steele, Celebrating 40 years of language evolution: simula 67 to the present and beyond, Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications, 0Pages: 1021 – 1023, 2007

[11] Whorf, Benjamin (John Carroll, Editor) (1956). Language, Thought, and Reality: Selected Writings of Benjamin Lee Whorf. MIT Press.

[12] Wikipedia: Java version history, http://en.wikipedia.org/wiki/Java_version_history, accessed on 5 May 2008

[13] Winston, A. Strongly-typed Java delegates, 2005, at http://weblogs.java.net/blog/alexwinston/archive/2005/04/strongly_types_1.html, accessed on 5 May 2008

## APPENDIX – Sample program in Java 1.5 and C# 3.0

```
//Iterator Pattern Example, J Bishop, Sept 2007
//Illustrates the use of iterators and
//LINQ based queries on a tree structure
//Uses the classes Console, Delegate,
//Property, Node and Tree
//Java 1.5 with A van Wyk May 2008

class Person {
  public Property <String> name;
  public Property <Integer> birth;

  public Person (String name, Integer birth) {
    this.name = new Property<String>
                (name,Property.Kind.GETONLY);
    this.birth = new Property<Integer>(birth);
  }

 public String toString () {
   return ("["+name.get()+", "+birth.get()+"]");
 }
}

public class IteratorFamilyTree {

  public boolean after1980 (Person p) {
      return (p.birth.get() > 1980);
  }

   public static void main(String [] args) {
     Tree <Person> family =
       new Tree <Person> ( new Node <Person>
         (new Person ("Tom", 1950),
            new Node <Person>
           (new Person ("Peter",1976),
              new Node <Person>
            (new Person ("Sarah", 2000), null,
                  new Node <Person>
            (new Person ("James", 2002), null,
               null) ),//no more siblings James
            new Node <Person>
            (new Person ("Robert", 1978), null,
                  new Node <Person>
            (new Person ("Mark", 1980),
                  new Node <Person>
            (new Person("Carrie",2005),
               null,null),
            null)// no more siblings Mark
            )),
         null) // no siblings Tom
      );

   Console.WriteLine("Full family");
   for (Person p : family)
      Console.Write(p+"  ");
   Console.WriteLine("\n");

   Iterable<Person> selection = family.where(
      new Delegate<Boolean>
         (this,"after1980",Person.class));

   Console.WriteLine("Full family");
   for (Person p : family)
      Console.Write(p+"  ");
   Console.WriteLine("\n");

   }
 }
```

```
//Iterator Pattern Example J Bishop Sept 2007
//Illustrates the use of LINQ with iterators
//on a tree structure
//C# 3.0

using System;
using System.Collections.Generic;
using System.Linq;

class Person {
  public Person() {}
  public string Name {get; set;}
  public int Birth {get; set;}

  public Person (string name, int birth) {
    Name = name;
    Birth = birth;
  }

  public override string ToString () {
    return ("["+Name+", "+Birth+"]");
  }
}

class IteratorPattern {

  static void Main() {
       var family =
         new Tree <Person> ( new Node <Person>
         (new Person ("Tom", 1950),
            new Node <Person>
           (new Person ("Peter",1976),
              new Node <Person>
            (new Person ("Sarah", 2000), null,
                  new Node <Person>
            (new Person ("James", 2002), null,
               null) ),//no more siblings James
            new Node <Person>
            (new Person ("Robert", 1978), null,
                  new Node <Person>
            (new Person ("Mark", 1980),
                  new Node <Person>
            (new Person("Carrie",2005),
               null,null),
            null)// no more siblings Mark
            )),
         null) // no siblings Tom
      );

    Console.WriteLine("Full family");
    foreach (Person p in family.Preorder)
      Console.Write(p+"  ");
    Console.WriteLine("\n");

    var selection = from p in family
      where p.Birth > 1980
       orderby p.Name
       select p;
    }

    Console.WriteLine(
      "Born after 1980 in alpha order");
    foreach (Person p in selection)
      Console.Write(p+"   ");
    Console.WriteLine("\n");
  }
}
```