C# Generics: Leverage code reuse without sacrificing type safety

By Bob Hauser

Recursion Software, Inc.
November 2006

RECURSION SOFTWARE, Inc.

Software for Software Developers™

C# Generics: Leverage code reuse without sacrificing type safety

By Bob Hauser

Introduction

Code reuse is an oft-touted benefit of modern object oriented programming. Developers frequently encounter situations where reusing code from previous projects would greatly increase their efficiency. Unfortunately, code isn't easily transferable from project to project, especially if the code contains application-specific logic that is tightly coupled to the desired functionality. With the advent of generic support in the C# language, appearing in the .NET Framework 2.0, developers have new leverage for writing code that can be reused without introducing additional errors, a concept known as type safety.

In this paper, we will discuss potential pitfalls that can occur when writing reusable code and introduce the concept of type safety. We will then review an example using C# generics to demonstrate effective code reuse and its benefits.

The Zen of Strong Typing

Strong typing means different things to different people. The Zen of this term is that a program should not operate on a particular data type as if it were some other type of data. Enforcing this kind of program behavior is what we call type safety. Type safety features usually introduce syntax into the programming language and can be enforced at program build time, run time or both. Type safety prevents such dangerous errors as having a program write a value of “0” into a memory location that it considers to hold an integer when another part of the program considers the same memory location to be part of some string data. Detecting type errors with type safety features of programming languages aids development in two related ways. First, it removes or at least reduces mismatched type errors from the built program. Second, it moves this class of error detection closer to the point of the mistake, which improves productivity. Traditionally, any type mismatch error that first appeared at run time incurred an extra cost by requiring the developer to switch to a different environment, the run time debugger, and debug the problem while trying to recall the thought process back when originally writing the code.

Type safety at build time and/or run time prevents code from manipulating data of an incorrect type. As part of its type safe approach, C# detects type mismatch errors at build time, in the compiler, and at run time, in the Common Language Runtime (CLR). The following is a type mismatch error that C# will catch at build time.

```
Object a = new Object();
String b = a;
```

The second line will generate a compiler error because a plain Object cannot be used as a String while maintaining type safety. Unfortunately, build time type checking is ineffective if explicit casts are used. A cast allows the programmer to circumvent the type checking at build time. For programming languages without run time type checking this can result in invalid operations on data types having unpredictable consequences, such as manipulating a fragment of a String data type as if it were an int. C# provides type safety checks at run time as well as build time. Here is a type mismatch error that will not be caught until the program is run.

```
Object a = new Object();
    String b = (String)a;
```

The program will build without errors. However, when running the program, it will generate an InvalidCastException on the second line because the Object referred to by ‘a’ cannot be converted to the type String.

This type mismatch error is blatant but serves to remind us to avoid explicit casting since it negates the type safety provided at build time. Relegating these errors to program run time incurs additional productivity cost. In fact, now that we have the flavor of the type checking performed at build time, languages providing type safety features should not lead us into situations that require us to turn off type safety by introducing such casts.

Reusing Code without Generics

Let’s review an example of code reuse without the benefits of generics. Evan, a programmer hired to write software for the local bakery, has a need to manipulate strings. The implementation of a minimal collection is shown in Listing 1 (see appendix). The collection is a singly linked list called List1. Since Evan is only interested in storing String type data at this point, he builds the collection to fit the need, that is, to store String data. Bold text in Listing 1 shows the locations where the collection is specific to the type String.

Now Evan can use the collection as follows.

```
List1 aList = new List1();
aList.Add("a");
aList.Add("b");
for (String item = aList.Remove();
   item != List1.NO_ITEM;
   item = aList.Remove())
{
   // Do something with item
}
```

List1 is type safe, meaning it accepts and returns String types and this type safety is enforced at build time.

Eventually, Evan needs to track the products waiting for the oven, items in the oven, and goods under the display counter. This is an opportunity for reusing the previously programmed collection, List1. The problem is that now he no longer needs to collect Strings, but rather BakeItems. Evan could copy the source code to form a new collection by replacing String with BakeItem, but this approach has dire consequences. If an error is found later then both copies of the source code need to be fixed. In addition, the baker has indicated that Evan will also need to track waiting customers, inventory shipments, and more. Writing separate collection code for each data type is clearly not desirable.

Evan decides to rewrite the collection with the most general available type, Object. This results in a List2 class where the only difference is the replacement of all uses of the data type String with Object, noted by bold text. Below is a fragment of List2.

```
public class List2
{
   public static Object NO_ITEM = default(Object);
   internal class ListNode
   {
      public ListNode next = null;
      public Object obj = NO_ITEM;
   }
   // …

   public virtual void Add(Object obj)
   // …
```

```
public virtual Object Remove()
// …
```

Deprecating List1 in favor of List2 requires refactoring the earlier string collecting code.

```
List2 aList = new List2();
aList.Add("a");
aList.Add("b");
for (Object item = aList.Remove();
   item != List2.NO_ITEM;
   item = aList.Remove())
{
   String theValue = (String)item;
   // Do something with theValue
}
```

The advantage of this rewrite is that we can use List2 for any type of thing we want to collect. For primitive types, C# will perform a conversion to an appropriate object in a process called *boxing*. This allows us to transparently handle primitive types as if they were objects. Therefore, we could use List2 with int as follows.

```
aList.Add(1);
aList.Add(2);
for (Object item = aList.Remove();
   item != List2.NO_ITEM;
   item = aList.Remove())
{
   int theValue = (int)item;
   // Do something with theValue
     }
```

For Evan, List2 seems well prepared for String, int, BakeItem, Customer, or anything else he might need. However, note that a cast is now necessary to convert from the most general type, Object, to the actual type, String or int in the code fragments. Recall that explicit casting is just the situation we want to avoid so we don't negate the type checking performed at build time. For instance, during the development effort, Evan accidentally added an instance of a data type called BakeItem to a List2 collection meant to only contain Customer elements. The error will appear as an InvalidCastException at run time, with no indication of problems at build time. This weakens the original intent of having multiple different homogenous collections in a way that allows any individual collection to accept unintended data types. Even with the explicit casts that negate build time type checking, this approach to reusing code is superior to the alternative of writing code for a multitude of type specific collections.

This code reuse versus type safety tension was the problematic state of affairs with C# 1.1. With the addition of generics to C# 2.0 there is now a better solution at hand.

Reusing Code with Generics

By using generics Evan can get the benefit of type safety at build time without needing to duplicate code. Let's see how with a code fragment showing the major changes needed. The full code for generic List3<T> is found in Listing 2 (see appendix).

```
public class List3<T>
{
   internal static T NO_ITEM = default(T);
   internal class ListNode
   {
      public ListNode next = null;
      public T obj = NO_ITEM;
```

```
    }
    // …

    public virtual void Add(T obj)
    // …

    public virtual T Remove()
    // …
```

With the syntax <T> immediately after the class name, we indicate a type parameter, T, which stands in place of some actual type that is specified at each location where a new List3 is used. In fact, we won't call this class List3 but rather List3<T>. Within the class itself the type parameter T is used as if it were a real type (like our old String or Object types). The C# build time and run time will conspire to use this generic class, List3<T>, as a template that can be used to generate as many "regular" classes as are needed for each actual type that a program specifies to take the place of the type parameter T.

Here's how Evan can refactor the string manipulation code to use the generic class List3<T>.

```
List3<String> aList = new List3<String>();
aList.Add("a");
aList.Add("b");
for (Object item = aList.Remove();
   item != List3<String>.NO_ITEM;
   item = aList.Remove())
{
   // Do something with item
}
```

Note that where we want to use List3<T> we must specify what the type parameter really is, in this case String. The C# run time will create a class for List3<String>, unless it has already done so. You don't need explicit casting because the Remove() method for List3<String> returns a String. At build time we have the advantage of full type checking that prevents entering the wrong type of object into the collection.

Where the same code could operate identically on many types, using generics elegantly solves the tension of how to write reusable code while maintaining type safety.

Back to the Bakery

Back at the bakery, Evan uses the generic List3<T> with Strings, Customer objects, and BakeItem objects.

```
List3<BakeItem> waitingForOven = new List3<BakeItem>();
List3<BakeItem> inOven = new List3<BakeItem>();
List3<BakeItem> onDisplay = new List3<BakeItem>();
```

With three collections the software is modeling the process as bakery items are queued for the oven, cooked, and then put on display. To facilitate some cooking experiments, the baker asks Evan to track how many of a particular bakery item were in the oven at the same time. Each bakery item should maintain the maximum and minimum number of items in the oven during its cook time.

First, Evan creates an IWatermark interface and implements that interface in the BakeItem class. The IWatermark interface is as follows and the BakeItem class that implements this interface is shown in Listing 3 (see appendix).

```
public interface IWatermark
{
        int HighMark { get; set; }
        int LowMark { get; set; }
    }
```

The bakery items that are in the oven are in a List3<BakeItem> collection in a variable called inOven. If this collection knew that it held items that implement the IWatermark interface then it could invoke the required instrumentation during the additions and removals of items in the collection.

Having looked at the documentation for C# generics, Evan considers placing a constraint on a generic parameter type. Evan considers turning List3<T> into an intrusive container that makes use of the properties available via IWatermark. It would look like this.

```
public class List3<T> where T : IWatermark
```

However, this would require implementing IWatermark on all of the classes List3<T> contains, including Customer types and String types. Also this would prevent List3<T> from using primitive types. A more viable approach could be to alter List3<T> to internally test if the type that it holds implements IWatermark and if so then to record the necessary statistics. We could make use of C#'s is operator within the implementation of List3<T> to dynamically test type compatibility for any T with IWatermark as follows.

```
public virtual void Add(T obj)
{
   if (obj is IWatermark) {
      // Do stuff with IWatermark methods
        }
```

Unfortunately, this approach is undesirable. It adds useless code in the general case where List3<T> is used with a type that does not implement IWatermark. Another pitfall of this approach is that any object that implements IWatermark will be instrumented in any List3<T> that it is placed in, not just the collection that is being used to represent the oven. Finally, this strategy hides the intrusive nature of List3<T> and thus can hide type mismatch errors.

Leaving the List3<T> as general as possible, Evan implements List4<T> with a constraint that it requires anything placed within it to implement IWatermark. See Listing 4 (see appendix) for the full watermarking collection. Here is the first line.

```
public class List4<T> : List3<T> where T : IWatermark
```

A generic class can be used as a base class so Evan chose to specialize List4<T> from List3<T> but added a constraint that List4<T> requires all items placed within it to implement IWatermark. The generic class List4<T> is explicit about its intrusive nature on the objects it holds and C#'s type safety provisions come into play at build time and run time to prevent it from operating on inappropriate types. Here is the modified code snippet and a program to simulate some oven activity is shown in Listing 5 (see appendix).

```
List3<BakeItem> waitingForOven = new List3<BakeItem>();
List4<BakeItem> inOven = new List4<BakeItem>();
     List3<BakeItem> onDisplay = new List3<BakeItem>();
```

Delving Deeper

Through this scenario, we introduced generic classes and how to place a constraint on a generic type parameter.

C# allows for generic classes, structs, interfaces, delegates, static methods, and instance methods. Any of these can be parameterized based on type and we are not limited to a single type parameter. For example, we could have a generic class Several<T, U, V> where each type parameter could be the same or unrelated.

```
Several<int, int, String> a = new Several<int, int, String>();
```

You can use generic methods to define algorithms that are clearly separated from the data type upon which they operate. A generic method specifies type parameters following the method name.

```
public U DoSomething<U, V>(U u, V v)
```

Both parameters and return type can be made generic and the type parameters need not be the same as type parameters on the class. The C# compiler can perform type inferencing on method parameters so that, for the above method, instead of coding DoSomething<String, int>("name", 2) you can simply type DoSomething("name", 2). The compiler infers the generic type parameters from the data types of the method parameters. Properties and indexers do not allow for introducing new type parameters but they can use any type parameters of the class that contains them. Generic methods support constraints on their type parameters like classes.

You can apply constraints to each type parameter separately and, for a type parameter T, the constraint begins with the syntax, "where T :". The list continues as comma separated items beginning with an optional base class followed by any interfaces. Following these, you can list a default constructor constraint using "new()", which requires the type parameter to support the public default constructor. Finally, you may add a reference or value constraint with a class or struct keyword. The class Test<T> shows multiple constraints.

```
class Test<T> where T : MyBase, IComparable, IMyInterface, new()
```

Use constraints with caution since they reduce generality and can prevent code reusability.

Operators, such as the == operator, cannot be a constraint. In List3<T> we commented out the code:

```
//if (obj == NO_ITEM)
//    throw (new Exception("Cannot hold the default() value."));
```

This code fragment is not permitted because the type parameter T cannot be guaranteed to have the == operator available. Our original design of the list collection is using default(T) as the value to return if the list is empty. For reference types, default(T) resolves to a null and produces zero-filled contents for value types. Since we cannot constrain type T to require the == operator, the compiler will not allow its use and we remove it. However this code disallowed entering the default(T) value into our List3<T> collection and we now have introduced ambiguity. When Remove() returns the default(T) value, how will the code using the collection distinguish whether the list is empty, or that the default value has actually been removed?

When storing only reference types such as String and Object, it was reasonable for our collection to reject storing null values. However, if the fully generic List3<T> is used as List3<int> it makes little sense to disallow adding the value '0', which is default(int), to the collection. Although we do not undertake it here, the right approach to this problem is to redesign the collection to be able to hold any value, including default(T), and to use another mechanism to indicate that the list is empty.

In the boxing process mentioned previously involves taking the primitive and creating a simple object for it from heap memory. When we use generics with primitives, no such process is necessary and we get a modest performance gain for doing things in the more elegant way. The performance gain is negligible in typical programs doing network and file operations, but for programs that spend the bulk of their time looping through collections of primitive types the gain is dramatic.

The type information about generics, the number of type parameters and their constraints, is fully supported by the C# 2.0 byte code and run time. This provides type safety for reflection even when using third party assemblies containing generics.

Generic Collections

In our illustration we used only a minimal generic collection. Due to their utility, collections are often the first classes to become generic. C# 2.0 provides the following collection classes in the System.Collections.Generic namespace.

Dictionary<K,V> - A key and value collection

LinkedList<T> - A linked list

List<T> - A list (implemented on an array)

Queue<T> - A queue

SortedDictionary<K,V> - A key and value sorted collection

SortedList<T> - A sorted linked list

Stack<T> - A stack

There is also System.ComponentMode.BindingList<T>, which can fire state change events. C# defines several generic interfaces including IEnumerable<T>, which enables all collections to support enumeration and C#'s foreach keyword. This is how it works on a LinkedList.

```
LinkedList<int> a = new LinkedList<int>();
a.AddLast(1);
a.AddLast(2);
foreach (int item in a)
{
   // Do something with item
      }
```

Both System.Collections.Generic.List<T> and System.Array have many useful generic methods, many designed to use four generic delegates in the System namespace.

```
public delegate void Action<T>(T t);
public delegate int Comparison<T>(T x, T y);
public delegate U Converter<T, U>(T from);
public delegate bool Predicate<T>(T t);
```

Combining generic delegates with utility methods enables powerful processing of collections. Unfortunately, collections other than List<T> and Array lack equivalent generic utility methods but there are third party generic libraries, such as Recursion Software's C# Toolkit, that support C# generics and provide extensive additional collections, algorithms, functions and predicates.

Conclusion

We have presented common problems that can occur when writing reusable code and have identified potential productivity costs associated with discovering casting errors at program run time. Next, we showed how C# 2.0 provides generics to elegantly address this problem. With generics, you can employ the type safety features of C# without preventing effective code reuse where the data type varies.

As programming techniques continue to evolve, we expect that programming with generics will become standard coding practice and their reusability will free developers focus on higher development objectives.

Appendix

Listing 1 – A minimal collection class for String types

```
public class List1
{
   public static String NO_ITEM = default(String);
   internal class ListNode
   {
      public ListNode next = null;
      public String obj = NO_ITEM;
   }
   internal ListNode head = null;
   internal ListNode tail = null;
   internal int length = 0;

   public List1() {}

   public virtual void Add(String obj)
   {
      if (obj == NO_ITEM)
         throw (new Exception("Cannot hold the default() value."));

      ListNode node = new ListNode();
      node.obj = obj;

      if (++length == 1)
         head = node;
      else
         tail.next = node;

      tail = node;
   }

   public virtual String Remove()
   {
      if (length > 0)
      {
         ListNode node = head;
         head = head.next;
         --length;
         return node.obj;
      }
      return NO_ITEM;
   }
}
```

Listing 2 – A minimal generic collection class for any type

```
   public class List3<T>
   {
      internal static T NO_ITEM = default(T);
      internal class ListNode
      {
         public ListNode next = null;
         public T obj = NO_ITEM;
      }
```

```
        internal ListNode head = null;
        internal ListNode tail = null;
        internal int length = 0;

        public List3() {}

        public virtual void Add(T obj)
        {
            //if (obj == NO_ITEM)
            //    throw (new Exception("Cannot hold the default() value."));

            ListNode node = new ListNode();
            node.obj = obj;

            if (++length == 1)
                head = node;
            else
                tail.next = node;

            tail = node;
        }

        public virtual T Remove()
        {
            if (length > 0)
            {
                ListNode node = head;
                head = head.next;
                --length;
                return node.obj;
            }
            return NO_ITEM;
        }
    }
```

Listing 3 – A class for representing bakery items.

```
    public class BakeItem : IWatermark
    {
        private int id;

        public BakeItem(int i)
        {
            id = i;
        }

        public int Id
        {
            get { return id; }
            set { id = value; }
        }

        // Implement IWatermark
        private int highMark = default(int);
        private int lowMark = default(int);

        public int HighMark
        {
            get { return highMark; }
            set { highMark = value; }
        }

        public int LowMark
        {
            get { return lowMark; }
            set { lowMark = value; }
        }
    }
```

Listing 4 – A special collection to record watermarks in each item collected.

```
    public class List4<T> : List3<T> where T : IWatermark
    {
        public override void Add(T obj)
        {
            base.Add(obj);
            obj.LowMark = length;
            for (ListNode node = head; node != null; node = node.next)
            {
                // Since T is constrained we can use the HighMark property here
                if (node.obj.HighMark < length)
                {
                    node.obj.HighMark = length;
                }
            }
        }

        public override T Remove()
        {
            T result = base.Remove();
            for (ListNode node = head; node != null; node = node.next)
            {
                // Since T is constrained we can use the LowMark property here
                if (node.obj.LowMark > length)
                {
                    node.obj.LowMark = length;
                }
            }
            return result;
        }
    }
```

Listing 5 – A method simulating some activity at the bakery

```
    public class Example5
    {
        public static void example5()
        {
            List3<BakeItem> waitingForOven = new List3<BakeItem>();
            List4<BakeItem> inOven = new List4<BakeItem>();
            List3<BakeItem> onDisplay = new List3<BakeItem>();

            // Simulate some oven activity
            int id = 1;
            for (int i = 0; i < 2; i++)
            {
                for (int j = 0; j < 5; j++) // Add 5 items to the oven
                {
                    inOven.Add(new BakeItem(id++));
                }
                for (int k = 0; k < 2; k++) // Remove 2 items
                {
                    BakeItem anItem = inOven.Remove();
                    Trace.WriteLine("Item: " + anItem.Id + " has cooked.");
                    Trace.WriteLine("HighMark: " + anItem.HighMark + " LowMark: " + anItem.LowMark);
                }
            }
            // Empty the oven
            for (BakeItem item = inOven.Remove();
                item != List4<BakeItem>.NO_ITEM;
                item = inOven.Remove())
            {
                Trace.WriteLine("Item: " + item.Id + " has cooked.");
                Trace.WriteLine("HighMark: " + item.HighMark + " LowMark: " + item.LowMark);
            }
        }
    }
```

Additional Resources

An Introduction to C# Generics. Juval Lowy.
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/csharp_generics.asp

Generics. http://msdn.microsoft.com/vcsharp/2005/overview/language/generics/

Generics in C#, Java, and C++ - A Conversation with Anders Hejlsberg, Part VII
by Bill Venners with Bruce Eckel. http://www.artima.com/intv/generics.html

An Extended Comparative Study of Language Support for Generic Programming -
http://www.osl.iu.edu/publications/prints/2005/garcia05:_extended_comparing05.pdf

Design and Implementation of Generics for the .NET Common Language Runtime. Andrew Kennedy and Don Syne. http://research.microsoft.com/projects/clrgen/generics.pdf

Standard ECMA-334. C# Language Specification. 4th edition. June 2006. http://www.ecma-international.org/publications/standards/Ecma-334.htm

About The Author

Robert R. Hauser has a Masters degree in Computer Science specializing in Artificial Intelligence with more than 15 years of development experience in C, C++, C#, and Java, building software for networking, filesystems, middleware, and natural language processing. He works at Recursion Software, which is focused on providing the next generation application platform.

Their agent-based Voyager Edge product gives software engineers maximum flexibility to easily develop dynamic, intelligent, mobile and decentralized applications in their software language(s) of choice, on the devices, virtual machines and operating systems they need to target, while leveraging all of their existing code.
Recursion Software also provides extensive toolkit libraries for C++, Java and C# .NET to further aid in the development of next generation applications.