A Solution to Mixed-Type Comparisons in C# .NET

Robert Dollinger
rdolling@uwsp.edu
Mathematics and Computing Department
University of Wisconsin Stevens Point
Stevens Point, WI 54481, USA

Abstract

Overriding Equals() in order to provide meaningful semantics to object comparisons, turns out to be quite a challenging task, especially when involving objects at different levels of a class hierarchy. One need to reconcile the requirements of the *equals contract* with the legitimate expectations of programmers of being able to meaningfully compare objects of different types. Langer&Kreft (Langer, 2002b) provided an implementation of equality checks for Java class hierarchies where they use a recursive navigation method that performs the non-trivial task of navigating up and down in the inheritance tree in order to make sure that objects on different levels or even on different branches of the inheritance tree are compared correctly. In this paper we first present a generalized implementation of the navigation method by using reflection and late binding techniques available in C# .NET. In this implementation navigation is still using recursion very much like the one in (Langer, 2002b). A non-recursive version of the navigation method is also given; this later version is more efficient and easier to understand. The generalized implementation of the navigation method is class independent and, as a result, one can factor it out to the hierarchy's root class. If it would be implemented in the very top class of the .NET hierarchy, the Object class, this would simply make mixed-type equality comparisons generally available by requiring classes to implement some sort of field comparing method thus defining the specific equality semantics, instead of struggling to override the Equals() method.

Keywords: Equals contract, mixed-type comparisons, reflection, late binding

1. INTRODUCTION

There are three basic things one needs to take care of whenever a new class is defined in C#: override the ToString() method, override the Equals() method, and override the GetHashCode() method. Overriding ToString() is easy, and almost everybody seems to be comfortable with this task, as it most often reduces to provide a string representation of the content of an object. Implementing GetHashCode() can be challenging and there are some good articles (most of them in the Java literature where the problems are very similar) dealing with this topic (see for example (Davis, 2000b)). With Equals(), things seem to be simple and straightforward, but they are not. Most programmers tend to consider the implementation of Equals() a trivial task, overlooking many of the subtle issues that are involved. Simply comparing the content of two objects to see if they are equal is just not enough because objects are in most cases part of a hierarchy. This results in incorrect implementations of the equality comparisons with hard to predict implications over the code using them. This is partly due to the fact that object equality comparison is a very basic operation and is silently used in so many places (e.g. collections management). Also, most of the textbooks fail to correctly address the problem of Equals() and provide many erroneous and incomplete implementations for it.

There are many articles dealing with the problem of equality comparisons (Davis, 2000a)(Langer, 2002a, 2002b) (Schaefer, 2004). These mostly end up providing recommendations of how to implement the equality test in some given limited contexts, rather then giving a comprehensive and "correct" solution. The ultimate solution, if there is one, is still not available. In our paper, we deal with the equality comparison problem in a more extended context, that is, under the assumption that objects of different classes in the same hierarchy can and should be compared for equality, providing meaningful and consistent results. Mixed type comparisons of objects have been first dealt with by Langer&Kreft (Langer, 2002a, 2002b). They provide a correct Java solution to the problem in the context of a precisely defined semantics. Their solution requires that each class joining the hierarchy of comparable objects implements two methods. The first one solves the specific task of locally comparing the fields of two objects. The second one is a recursive navigation method that performs the non-trivial task of navigating up and down the inheritance tree in order to make sure that objects on different levels or even on different branches of the inheritance tree are compared correctly. Their navigation method is class specific and each class of the hierarchy has to implement its own navigation method. However, from a conceptual point of view, the navigation method has nothing to do with any particular class. Furthermore, the burden of implementing the navigation method each time one wants to add a new class to the hierarchy makes this approach less appealing in practice.

In this paper we present two versions of a generalized implementation of the navigation method by using reflection and late binding techniques available in C# .NET. The first version of the navigation method is still using recursion very much like the one in (Langer, 2002b). The second version of the navigation method we propose is non-recursive, thus is more efficient and easier to understand. The generalized navigation method is class independent and can be factored out from the classes in the hierarchy and implemented only in the root class. All classes will inherit and use this method without any change. The implementation is provided in C#. NET, but a Java implementation is equally conceivable.

The remainder of this paper is organized as follows: in section 2, we introduce the *equals contract,* which specifies the correctness criteria for all implementations of equality comparisons, and deal with the pitfalls of those implementations that ignore this contract. In section 3, we bring arguments that mixed typed comparisons are unavoidable and we also give a possible semantics for such kind of comparisons. Previous approaches and their limitations are presented briefly in section 4. Section 5 provides the details of implementing mixed type comparisons in a class independent way. The key techniques for developing a general navigation method are reflection and late binding. Section 6 provides a simple, iterative version of the navigation method. Section 7 contains conclusions and some ideas of further developments.

2. CHALENGES OF "Equals"

The Equals() method is intended for content based comparison of objects. For this reason, it is common for new classes to override Equals() in order to capture class specific semantics, usually based on the values of data members. The minimal requirements users have to satisfy when overriding Equals() are specified in what is known both in the Java and .NET documentations as the *equals contract*. According to this contract, the equals methods we are expected to implement are supposed to behave like any equivalence relation among the objects we compare. This means that Equals() should be: *Reflexive* (i.e. x.Equals(x) returns **true.**), *Symmetric* (i.e. x.Equals(y) returns the same value as y.Equals(x).), and *Transitive* (i.e. if (x.Equals(y) && y.Equals(z)) returns **true** then x.Equals(z) returns **true**.). In addition to these, the *equals contract* specifies that successive calls of x.Equals(y) return the same value as long as the objects referenced by x and y are not modified, and x.Equals(null) always returns false.

Failure to comply with any of the rules produces incorrect implementations of Equals(), no matter how "correct" they may seem, and results in subtle bugs which are hard to fix.

Pitfalls in Implementing Equals()

In spite of most expectations and in spite of the apparent simplicity of the rules in the *equals contract*, providing correct implementations of Equals() is far from simple. This has caused quite a bit of disagreement in the programming community and, triggered a fair amount of debate. (Davis, 2000a)(Langer, 2002a, 2002b)(Schaefer, 2004) Everything is fine and looks simple as long as we deal with objects that belong to the same class. The first problems show up in the presence of inheritance.
To exemplify let us take a class A and its subclass B. If a is an instance of class A and b is an instance of class B having all common fields set to equal values, then in most usual implementations the expression a.Equals(b)will evaluate to true, while b.Equals(a) would evaluate to false, thus violating the symmetry rule of the *equals contract*. The reason is that, in the first case we are using the Equals() from class A, which will succeed because it compares objects a and b as instances of class A. This is called a *slice comparison,* since only the slice of object b that is inherited from class A gets compared (e.g. a person object compared with a student object as person may evaluate to true if both represent the same person). The second test, b.Equals(a) will usually attempt to compare the two objects as instances of class B and if so it will fail (comparing a person and a student as if there would be two students simply does not work!) . In order to fix this problem, let us agree to compare objects as class A objects whenever one of them is an instance of class A, which means we will ignore fields defined in class B. For this we only need to add an additional test as the first line of Equals() in class B:

```
//test after reversing operands
if(other is A) return other.Equals(this);
```

With this adjustment, now both expressions b.Equals(a) and a.Equals(b) evaluate to true.
With such a modification the implementation of Equals() is symmetric, but is still incorrect. As we already mentioned, the *equals contract* is not easy to comply with and, as we are going to show, we may still violate the transitivity rule. To illustrate this let us take three objects a, b1 and b2 with the following properties:

- a is an instance of class A;
- b1 and b2 are instances of class B;
- a, b1 and b2 have the same values for their common fields (which are the fields of class A);
- there is at least one field defined in class B for which b1 and b2 have different values.

Based on the above properties, it's easy to see that object b1 and b2 are not equal, and the expression b1.Equals(b2) will correctly evaluate to false. On the other hand, the expressions b1.Equals(a) and a.Equals(b2) will both evaluate to true because they perform slice comparisons comparing object as instances of class A. Now, by transitivity we should have b1.Equals(b2) evaluate to true as if objects b1 and b2 would be equal, which contradicts our previous assertion (e.g. this may correspond to the case of the same person being enrolled as student at two different universities; it will be the same person, but with two different student IDs and GPAs – equal as persons, but still different as students).
The point of this entire discussion is that after several so called "fixes", we still do not have a correct implementation of Equals(). Fixing the transitivity problem is not easy, and it involves trade-offs on which there is still much disagreement in the programmers' community. In fact, let us observe that even our solution for the symmetry problem is incomplete. It works, indeed, if one of the objects is the ancestor of the other one, but what if they are instances of classes located on different branches of the class hierarchy?

3. MIXED TYPE COMPARISONS

All the problems illustrated in section 2 would disappear if we eliminate the possibility of comparing objects of type A with objects of type B that is, if we forbid *mixed type comparisons*. By allowing equality comparisons only between objects that are precisely of the same type, both the symmetry and transitivity problems disappear, and it becomes relatively easy to provide implementations of Equals() which, as such, are correct by the terms of the *equals contract*. This is the approach most people would take in their applications for various reasons, from ignorance to assuming mixed types will not be compared to each other, and in most cases ends up causing problems that are subtle enough and show up late in the development cycle making their fix costly and painful.

Mixed Type Comparisons as the Norm

Most of the articles about equality comparisons end up with the easy approach of disallowing mixed type comparisons in order to avoid the conceptual problems that come with the development of a more general, yet correct, solution. Some would permit mixed type comparisons in some particular cases, while alternative approaches are proposed for some other situations. All these end up in a mixture of partial or limited solutions, and/or sets of recommendations of when and how to implement equality tests in various circumstances. This is far from any useful and comprehensive solution of the problem at hand.

The first argument in favor of allowing mixed type comparisons comes from Langer&Kreft (Langer, 2002b). While considering same type comparisons as the recommended way of implementing equality tests, they advocate for the use of mixed typed comparisons in some limited cases. The decision of which way to compare objects would be based on the semantics of the classes they instantiate. According to their examples, it makes sense to use mixed type comparison in a hierarchy where Student and Employee are subclasses of Person, since it would allow comparing a student to an employee to see if they represent the same person in a polymorphic collection of Person objects. On the other hand, it would make little or no sense to compare an apple to a pear in a hierarchy in which Apple and Pear are subclasses of Fruit.

Clearly, one cannot ignore or totally avoid mixed type comparisons: this is far too limiting and in contradiction with the diversity of the real world we are trying to model in our applications. On the other hand, using one or other type of comparability (same type or mixed type) based on circumstances like the semantics of the classes seems to be causing more problems than it is solving. On what basis would someone decide what kind of comparability to use for a given application or a given set of classes? Could this change over the development stages of the application? Could we use both kinds of comparability types in the same application? If yes, how would these interact or coexist?

Given these complications, why wouldn't we go beyond the hesitant position of using both same type and mixed type comparisons? If we adopt mixed typed comparisons as the norm, same type comparisons would be just a particular case of it, and the entire issue of object comparisons suddenly gets simplified. In fact, it becomes a purely technical problem of if and how it can be done right. The semantic argument is always debatable and not very productive in this case. After all, why would not apples and pears be comparable in terms of, let's say, the amount of vitamins brought into our body when consumed?

We advocate in favor of the idea that considering mixed type comparisons as the norm is realistic. That is, programmers should have some tool allowing them to compare objects from different parts of an inheritance tree. By providing a class independent implementation for the Equals() method, we show that mixed type comparisons can be technically supported in a way that is acceptable for programmers. The idea is to separate the inherent functionality of navigating across the inheritance tree from the local operations of comparing member data on a field by field basis. The navigation function is class independent and generic (application independent!), while the field comparisons capture the specific semantics associated with a given class(hierarchy) as required by a particular application. One of the key issues in the success of such an approach is to agree upon an acceptable semantics of mixed type comparisons.

Possible Semantics for Mixed Type Comparisons

Let us observe that a necessary condition for two objects to be equal is to have all their common fields equal. Second, the transitivity rule is violated if we repeatedly compare objects of different classes by ignoring their subclass specific fields. In the example of section 2, we compared object b1 with a, and object a with b2 by testing only the common fields in these objects, and ignoring the possible differences between b1 and b2 coming from *fields defined in class B*. This caused the violation of the transitivity rule.

The problem can be solved if we add the additional requirement that subclass objects have default values for all non-common fields. That is, in order to have object b equal with object a, they must have equal values for their common fields and object b must have default values for all other fields. These default values *are the same* for all objects of a given class. So, if b1.Equals(a) and a.Equals(b2) are true that means b1 and b2 have the same values not only for their fields which are common with object a, but they also have the same default values for the rest of their fields, which means that b1.Equals(b2) is true, and thus transitivity of Equals() holds.

This semantics is consistent with the *equals contract* and uniformly applies on all objects of a class hierarchy. For example, given two objects on different branches of the class hierarchy, like student and employee, we would have equality if they have equal values for all their Person fields and default values for whatever fields are defined in the Student and Employee classes. As a special case, two objects with all their fields set to their default values will be equal even if they have no fields in common, that is they have totally different descriptions. This result may seem quite counterintuitive, but is perfectly consistent with the *equals contract*. Semantically, this may be interpreted like "if two objects contain no relevant information (i.e. all defaults) then they are the same".

Alternative Semantics for Mixed Type Comparisons

Although the above semantics seems to solve some problems and supports an implementation which is in accordance with the *equals contract* it has a limited applicability and proves to be too restrictive in some cases. For example in the case of the Student and Employee classes, one would be able to identify that a student and an employee represent the same person only if the objects have default values for all their fields except the ones defined by the common Person class. Problems like this can be addressed at the local level of the field compare functionality, where the specifics of each class and application are captured, and should not affect the idea of comparing mixed types. One can imagine using a whole variety of alternative local semantics; e.g. one could take into consideration only some attributes, considered as relevant for the equality test, or use attributed programming techniques in order to determine when a given field would be compared or not.

4. PREVIOUS APPROACHES

Based on the semantics defined above, Langer&Kreft (Langer, 2002b) propose an approach that allows mixed type comparisons of objects in a class hierarchy. Their solution relies on two methods each class in the hierarchy is required to implement: a method for comparing fields, and a navigation method that makes sure that objects on different levels or even on different branches of the inheritance tree are compared correctly. The equality comparison method is implemented only once in the root class of the hierarchy. They provide a Java implementation of the entire solution, which turns out to be compliant with the *equals contract*.

One can observe the trade-off Langer&Kreft propose in their approach. User classes are no longer required to implement the challenging equality comparison method. Instead they are required to implement two other methods: a fields comparing method and a recursive navigation method.

The field comparing method is class specific, and its implementation in each class is a natural requirement.

The situation is different with the navigation method. Its implementation is not straightforward and can be almost as difficult as implementing the equality comparisons. Understanding the mechanics of how this method works is quite a challenging task. This is hardly something that any user would like to deal with whenever defining a new class. Unfortunately, this makes the proposed solution quite unappealing, if not impossible to use in practice.

5. MIXED TYPE COMPARISONS, RECURSIVE APPROACH

Given the difficulties to deal with the navigation method, we propose a generalized and more practical solution of the mixed type comparison problem. The general layout of our approach follows the one proposed by Langer&Kreft, except that we provide a generalized and class independent implementation of the navigation method. We achieved this by using reflection and late binding techniques. Our implementation is in C#, but it can be easily translated to Java as well. The result is that the navigation method can be factored out to the top of the class hierarchy. There will be only one implementation of this method, and it will be inherited by all classes. Consequently, the trade-off for the user classes is a much more practical one: a new class joining the hierarchy is required to implement only the field comparison method. Both Equals() and the navigation method are implemented in the root class of the hierarchy.

Implementation of Equals()

For a given hierarchy of classes there is one single implementation of the Equals() method, and that is located in the top of the hierarchy, the RootClass. Figure 1 shows the implementation of Equals() in the RootClass.

The code is quite straightforward and class independent in spite of the fact that it may be called from each class in the hierarchy. After a couple of routine tests included for the sake of efficiency, the call is made to the navigation method, called Navigate().

```
public class RootClass
{
   public override bool Equals(Object other)
   {
     if(other==this) return true;       //same object
     if(other==null) return false;      //nothing equals null
     if(!(other is RootClass))          //incompatible types
                      return false;
     return Navigate(this.GetType(),
                      (RootClass)other,false);
   }
}
```

Fig. 1. Implementation of Equals() in the RootClass

The CompareFields Method()

The method that deals with the field comparisons, called CompareFields(), is class specific and it is the only one that needs to be implemented in each class. This method is the materialization of the local semantics chosen for the classes at hand as required by the current application. It compares the slice of relevant fields defined in the current class. For the semantics defined in section 3 this method compares the fields defined in the current class with the corresponding fields of the object, called other, given as parameter. This is actually happening only when the currently defined fields are common with other, which means that other needs to be an instance of this class or of one of its subclasses. When this is not the case, one would check that the currently defined fields are set to their default values. In general, the implementation of this method is straightforward for classes having only value types as fields and one can follow the pattern given in the sample implementations from (Langer, 2002b). If our classes have complex types as their fields then all we need to do is to recursively call the CompareFields() methods of these types.
Note that the entire semantics of object comparisons is captured locally in the CompareFields() method of each class. By changing the implementation of the CompareFields() method users can adopt different comparison semantics for their classes; e.g. like relaxing the condition of having default values for all non-common fields. The only requirement is to keep their semantics consistent with the *equals contract*.

The Navigation Method()

Since both Equals() and CompareFields() turn out to be fairly simple, one can expect that most of the functionality involved in the mixed type comparison operations is concentrated in the Navigate() method. As its name may suggest, this method will navigate the inheritance tree in order to check all fields of the compared objects. Indeed, all versions of the CompareFields() method perform strictly local operations that involve only the fields defined by the current class. In order to have the inherited fields compared as well, one will need to perform some kind of navigation across the inheritance tree in order to call the corresponding versions of the CompareFields() method. This is exactly the task of the Navigate() method. In order to understand how this method works, let us take as an example the sample hierarchy depicted in Figure 2.

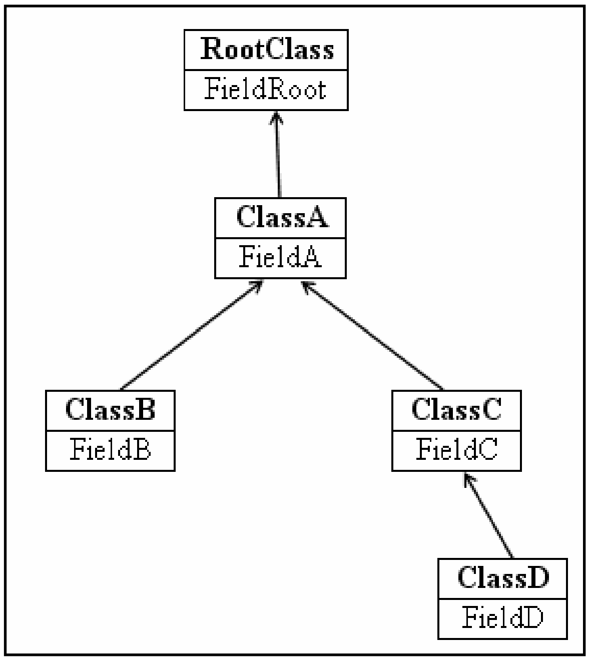

Fig. 2. Sample Class Hierarchy

The sample class hierarchy is composed of the RootClass on top of the hierarchy and classes: ClassA, ClassB, ClassC and ClassD as depicted in the figure. For simplicity, let us assume that each class defines exactly one field of its own: FieldRoot is defined by the RootClass, ClassA defines FieldA and so on. As a result, an object b, instance of ClassB, will have 3 fields: FieldRoot and FieldA as inherited from RootClass and ClassA and, FieldB defined in ClassB. Similarly, an object d instance of ClassD, will have fields: FieldRoot, FieldA, FieldC and FieldD.
Based on the semantics defined in section 3, two objects will be equal if they have equal values for their common fields and all their subclass specific fields are set to default values. For example, objects b and d, instances of ClassB and ClassD, are equal if and only if their FieldRoot and FieldA are equal and the specific fields: FieldB in b and, FieldC and FieldD in d are set to their default values.
Let us observe that the most general case of equality comparison is when the two objects are instances of classes located on different branches at different depths in the class hierarchy. All other are particular cases of this one. So, keeping things simple, let us use the example in Figure 2 and analyze what is to be done when comparing objects b and d, instances of ClassB and ClassD. During the navigation across the class hierarchy, there are three main tasks that need to be solved:

- check for the default values of the subclass specific fields of object b – this requires navigation on the left side branch from ClassB to ClassA.
- check for the default values of the subclass specific fields of object d – this requires navigation on the right side branch from ClassD to ClassA.
- check for equality of the common fields - this requires navigation on the common branch from ClassA to RootClass.

Navigation will be done from the subclass levels to the upper level classes in the direction indicated by the arrows in Figure 2, by simply calling the Navigate() method of the base class. At each step of the navigation process, the corresponding CompareFields() method is called in order to check the fields defined at the current level.

Implementing the Navigation Method()

One of the challenges during the navigation process across the left or right side branch is to detect where each branch ends. This is the lowest level class that is also common for both branches, which in our example is ClassA. This class has the property that both objects b and d are instances of ClassA, i.e. b is ClassA and d is ClassA evaluate both to true. So, when navigating on the left side branch from ClassB to ClassA, one would stop when the other object (d in our example) is an instance of the current class. Using class specific tests of the form b is ClassA or d is ClassA makes the navigation method itself to be class specific, which means that a specialized version of the Navigate() method would be required for each class. This is where *reflection* comes in providing the functionality needed to generalize the navigation method by avoiding direct references to the class names. First, one can use the GetType() method in order to get the type of a class or the class type of a given object. In our case the Equals() method (see Figure 1) will get the type of the current object, i.e. this and pass it as the first parameter of the Navigate() method:

```
return Navigate(this.GetType(),
                          (RootClass)other,true);
```

while the navigation method is designed to receive as first parameter a Type object which will always be set to the type of the current class:

```
public bool Navigate(Type typeOfThis,
        RootClass other, bool reverseOrder)
```

The second useful functionality is that Type objects come with a method called IsInstanceOfType(), which is the dynamic counterpart of the is operator. This means that given the variable typeOfThis that holds the type of the current class, the expression typeOfThis.IsInstanceOfType(other) will tell us when the other object is an instance of the current class that is, when we are at the end of the branch. Another problem with the navigation method is to make sure that all the required branches are processed and that they are processed one single time. The technique proposed by Langer&Kreft (Langer, 2002b) is to use a flag variable to control the navigation process. This is exactly what the third parameter of Navigate() does. Initially, the navigation method is called with parameter reverseOrder set to false. When reaching the common class for the first time, the first two parameters are reversed and reverseOrder is set to true. In our example, this means that the current class type becomes ClassB while parameter other will be object b. This sets the right values for navigation along the right side branch. When returning from the right side branch, navigation simply continues on the common branch up to the RootClass level. If this level is reached, the entire comparison process successfully terminates and the two compared objects are equal. The implementation of Navigate() as a recursive method is given in Figure 3.

The last challenge in the development of a class independent navigation method is to compare the fields defined at the current class level, by issuing a call to the right version of the CompareFields() method. One may expect that some kind of cast operation by the current level class type may provide the binding to the right method. Unfortunately, this is not the case, since cast operations are meant to alter compile time binding. What we need here is to dynamically bind our call to the version of the CompareFields() method that corresponds to the current class level. That is, at each navigation step, a different version of the CompareFields() method needs to be called. This kind of functionality can be achieved by using *late binding* techniques.
In C# there are two ways one can use to dynamically invoke a method: by using the Invoke() method, or by using the more general InvokeMember() method.
When using the Invoke() method, a required preliminary step is to get the method information corresponding to the method we want to call dynamically, that is CompareFields(). This information is contained in a MethodInfo object which is returned by the GetMethod() method of a type object. In our case, the expression:

```
public bool Navigate(Type typeOfThis,
                 RootClass other, bool reverseOrder)
{
 if(typeOfThis.IsInstanceOfType(other)&&
      !reverseOrder)
         // reverse order
         return Navigate(other.GetType(),this,true);
 // compare my fields
 if(!(bool)typeOfThis.InvokeMember("CompareFields",
                 BindingFlags.InvokeMethod
                 | BindingFlags.Default,null,this,
                 new Object[] {other})) return false;
 //succesfully done when at RootClass
 if(typeOfThis==Type.GetType("RootClass"))
                                 return true;
 //navigate to upper level
 return Navigate(typeOfThis.BaseType,
                      other,reverseOrder);
}
```

Fig. 3. Recursive version of the Navigate() Method

typeOfThis.GetMethod("CompareFields")

returns a MethodInfo object describing the CompareFields() method for the current level class type represented by the variable typeOfThis. The MethodInfo object is the one that provides access to the Invoke() method through which the right version of Compare-Fields() is called. The complete code for the dynamic call is shown bellow:

```
MethodInfo compareFieldsMethod=
typeOfThis.GetMethod("CompareFields");
if(!(bool)compareFieldsMethod.Invoke( this,
                new Object[] {other}))
        return false;
```

The first parameter of Invoke() is the object making the dynamic call, while the second one is an array of objects representing the list of parameters with which the dynamic method is called.
The equivalent dynamic call sequence using InvokeMember() is:

```
if(!(bool)typeOfThis.InvokeMember(
        "CompareFields",
        BindingFlags.InvokeMethod |
        BindingFlags.Default,null,this,
        new Object[] {other})) return false;
```

Details about InvokeMember() and its parameters can be found in (Liberty, 2003) and (Troelsen, 2003).

6 ITERATIVE NAVIGATION

Things become even simpler after a closer look at the functionality of the navigation method. Even in the most complex case all that needs to be done is to navigate along three branches of the class hierarchy and since there is only one single navigation method implemented in the root class there is really no need for recursion in its implementation. The iterative version is given in Figure 4.

Let us notice that now there is one single parameter passed to the Navigate() method. Passing the type of the current object is not needed anymore, and the reverseOrder boolean flag can be discarded as well. Obviously, the call of Navigate() in the Equals()

```
public bool Navigate(RootClass other)
{
  //process this branch
  Type typeOfThis=this.GetType();
  while(!typeOfThis.IsInstanceOfType(other))
  {
     if(!(bool)typeOfThis.InvokeMember(
           "CompareFields",
           BindingFlags.InvokeMethod
           | BindingFlags.Default,null,this,
           new Object[] {other})) return false;
     typeOfThis=typeOfThis.BaseType;
   }
  //process other branch
  Type typeOfOther=other.GetType();
  while(!typeOfOther.IsInstanceOfType(this))
   {
     if(!(bool)typeOfOther.InvokeMember(
           "CompareFields",
           BindingFlags.InvokeMethod
           | BindingFlags.Default,null,other,
           new Object[] {this})) return false;
     typeOfOther=typeOfOther.BaseType;
   }
  //process common trunk up to RootClass
  if(!(bool)typeOfThis.InvokeMember(
           "CompareFields",
           BindingFlags.InvokeMethod
           | BindingFlags.Default,null,this,
           new Object[] {other})) return false;
  while(typeOfThis!=Type.GetType("RootClass"))
   {
     typeOfThis=typeOfThis.BaseType;
     if(!(bool)typeOfThis.InvokeMember(
           "CompareFields",
           BindingFlags.InvokeMethod
           | BindingFlags.Default,null,this,
           new Object[] {other})) return false;
   }
  return true;
}
```

Fig. 4. Iterative version of the Navigate() Method

method will change accordingly.

The method consists of three blocks of code. The code in the first block performs the navigation from the class type of object this until its closest common ancestor with object other. The second block is similar, just that it navigates on the branch of object other, while the last block navigates along the common branch up until the RootClass.

7. CONCLUSIONS AND FURTHER DEVELOPMENTS

Equals() is intended to capture the semantics of content based equality comparisons of objects as opposed to object identity implemented by the == operator. This would make Equals() a class specific method; thus overriding it is expected to be an every day routine. However, correctly implementing Equals() turns out to be a challenging task, and everyday routines are not supposed to be challenging. The correctness criteria for equality comparisons are given by the *equals contract*, and are not easy to comply with. On the other hand, programmers would expect a solution that is both simple and free of artificial limitations. Programmers should not find a difficult challenge in implementing such a basic functionality like object equality and should not be limited to compare only objects of the same type. We provide an approach to the equality comparison problem which is a generalization of the solution provided by Langer&Kreft in (Langer, 2002b) and is able to reconcile the requirements of the *equals contract* with the legitimate expectation of programmers. This means that mixed type comparisons of object are allowed without limitations, while programmers are expected to implement a fairly straightforward CompareFields() method instead of having to override Equals(). The key techniques used are based on reflection and late binding, which allow class independent navigation of the inheritance tree.
There are several directions for further development of the work presented in this paper. The functionality of mixed type comparisons could be made generally available in a user transparent manner. All it would take is to implement both Equals() and Navigate() methods at the level of the Object class. Providing these as standard system level functionality would leave programmers only with the requirement of implementing the CompareFields() method as their own local concept of object equality. This requirement could be enforced by having classes to implement an adequate interface defining the CompareFields() method. Given its simple structure, an even more convenient approach could be to automatically generate the code of the CompareFields() method based on a list of user designated fields, considered as relevant for the equality tests, along with their default values.
As a final thought, we would like to emphasize the idea that it may be beneficial to have more elaborate content-based default functionality implemented at system level both for Equals() and GetHashCode(). The two are closely related and a navigation technique similar to the one presented here could be used in the computation of objects' hash codes. As with field comparisons, users will only have to designate which fields would be used in the generation of the hash code value.

REFERENCES

Davis, M. (2000a) Durable Java: Liberte, Egalite, Fraternite, Java Report, January 2000, URL: http://www.macchiato.com/columns/Durable5.html

Davis, M. (2000b) Durable Java: Hashing and Cloning, Java Report, April 2000, URL: http://www.macchiato.com/columns/Durable6.html

Langer, A., Kreft, K. (2002a) Secrets of equals() – Part 1, Not all implementations of equals() are equal, Java Solutions, April 2002, URL: http://www.langer.camelot.de/Articles/JavaSolutions/SecretsOfEquals/Equals-1.html

Langer, A., Kreft, K. (2002b) Secrets of equals() – Part 2, How to implement a correct slice comparison in Java, JavaSolution, August 2002, URL: http://www.langer.camelot.de/Articles/JavaSolutions/SecretsOfEquals/Equals-2.html

Liberty, J. (2003) Programming C#, Third Edition, O'Reilly

Schaefer, A. (2004) All equals() are not born equal, Java.net, October 2004, URL: http://weblogs.java.net/blog/schaefa/archive/2004/10/all_equals_are.html

Troelsen, A. (2003) C# and the .NET Platform, Second Edition, Apress