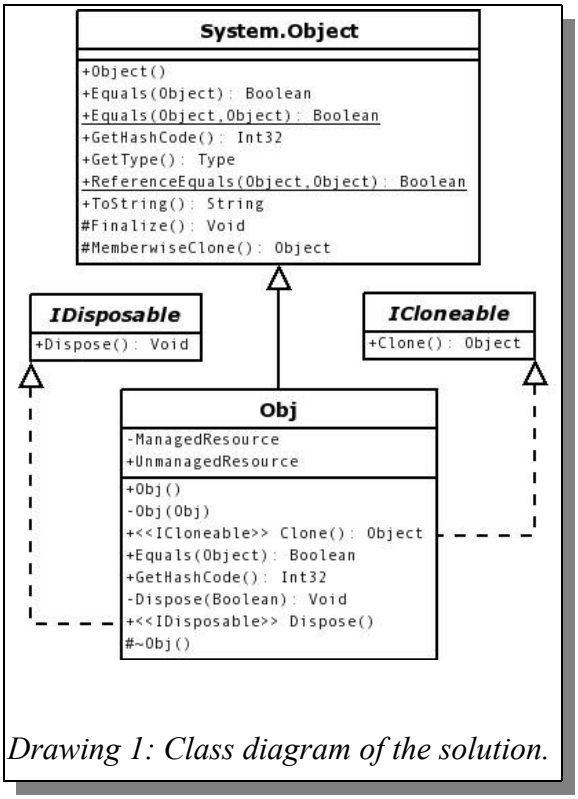
Curb your objects!
An Orthodox Form for C# Classes

Roberto Jimeno
<roberto.jimeno@gmail.com>
PhD first level student
Graduate Center, City University of New York
Computer Science Department
356 Fifth AvenueNew York, NY 10016

M.S. Jorge L. Ortega-Arjona
<jloa@fciencias.unam.mx>
PhD candidate
Department of Computer Science, University College London.
Currently at Facultad de Ciencias, Departamento de Matemáticas, Cubículo 023 . Circuito Exterior de Ciudad Universitaria UNAM. México D.F. C.P. 04510.



Drawing 1: Class diagram of the solution.

Abstract

The Orthodox Form for C# Classes (OFC#C) is an idiom proposed for the C# programming language which intends to provide its classes with a basic structure that assures a predictable behavior for creation, copy and destruction of instances.

When classes in C# are written it is desirable built them in such a way that their instances produce objects which behave in predictable ways.

Programmers tend to make lake mistakes when their

objects behave at run time in the same way the they expects them to behave at write/compile time.

Keywords

C# programming language, .NET, software pattern, idiom.

Introduction

The Orthodox Form for C# Classes (OFC#C) is an idiom proposed for the C# programming language which intends to provide its classes with a basic structure that assures a predictable behavior for creation, copy and destruction of instances. Such a basic structure is comparable C++'s Orthodox Canonical Form (OCF) proposed by James Coplien and Java's Canonical Object (CO) by Bill Venners.

This idiom's objective is to provide a general scheme to take as a base to write classes in C#. When classes in C# are created it is desirable to produce objects which behave in predictable ways. Classes built with this structure will provide their instances with a run-time behavior which is clearly defined at compile-time.

As a whole, all common behaviors on C# objects constitute a set of services which should be built into most classes. This idiom intends to answer the question: What do the C# programmer need to code (as a minimum) so that any instance's behavior handles the services common to all objects?

It is suggested to adopt this idiom on the proper practice of C# programming, or if a set of guidelines is
desirable for a programmers team in an organization, this idiom might be included as part of the guidelines. Virtually all classes written as part of a library, module, component or framework should be modeled after this idiom.

OFC#C's expression as a software pattern

Name Orthodox Form for C# Classes.

Related patterns Orthodox Canonical Form, Canonical Object, Dispose pattern, *using* pattern.

Context The OFC#C should be used during creation of any non-trivial class on a C# program, unless there is a specific reason for not using it.

Problem Frequently, default methods provided by the compiler or the runtime system cause objects to
behave in unpredictable, inconsistent or inefficient fashion.

Solution The programmer should provide implementations for certain methods of key importance so all objects behave in a predictable and consistent fashion under any conditions.

• *Solution structure* – Any non-trivial class requires to define (or override) at least the following members (shown grouped)

- *Creation group.*
  - Default constructor.
- *Copy and cloning group.*
  - Copy constructor and `Clone()` for reference types.
  - Implement `ICloneable` interface for producing deep copies.
- *Comparison group.*
  - Replace `Equals()`.
  - Replace `GetHashCode()`.
- *Destruction group.*
  - Implement `IDisposable` interface.
  - Define the class destructor.

Drawing 1 shows the class diagram of the solution. Class Obj is a simple example of a class which uses the OFC#C. By comparing classes Object and Obj on the diagram it is possible to observe which are all `System.Object`'s methods are overridden by Obj class. Information regarding interfaces, methods and their visibility is also shown.

• *Solution's dynamic* – It is convenient to classify solution's dynamics in two categories:

- *External dynamics:* When an object conforming to the OFC#C is created, its default constructor warrants initialization to a predictable and consistent state. Afterwards the object can be assigned or copied, always producing consistent instances in and predictable way. Orthodox objects (i.e. Instances of classes conforming with the OFC#C form) can be cloned using a copy mechanism to go along with the object's semantics, and also be compared according to that semantics by using

`Equals()`, besides they can be utilized in other useful ways. When convenient, the object can be stored persistently, so its life time transcends the life time of the process which created it. Finally, this objects are prepared to release their resources once they become redundant for the program.

• *Internal dynamics:* A destructor method of a C# class produces a protected method named `Finalize()` which overrides the `Finalize()` method inherited from `System.Object`. When the `IDisposable` interface is implemented, there is a public parameter-less method named `Dispose()`, which calls the protected method `Dispose(Boolean)` for the class. Also, the copy constructor (whose visibility is private) can be used by the public method `Clone()` to produce copies of objects. Likewise, it is frequent for the `Clone()` method to internally use the protected method `MemberwiseClone()` to perform shallow copies. Finally, the `Equals()` method can call other overridden `Equals()` methods from other classes to determine if the present object (`this`) is equal to other of the same type.

Example The class shown below complies with the OFC#C end is part of a real program (a program used to organize bibliographic references).

```
// This class sticks to the OFC#C and is
// part of a program which verifies
// bibliographic references on LyX/TeX/LaTeX
// documents.

using System;


// Has no managed resources, so does not
// implement IDisposable nor a destructor;
// nonetheless it adheres to the OFC#C.

class BibRef : ICloneable
{

protected readonly string sId; // Due to its
// being readonly it can only be modified
// from within a constructor.

//////////////////////////// Creation group.
// Default constructor.
BibRef()
{
      // The default constructor is private,
      // which impedes its use from outside
      // the class (reducing the chance of
      // errors).
}

// Builds the object from a
// character string.
public BibRef(string s)
{
      sId=s;
}

////////////// Copy and cloning group.
// Copy constructor.
BibRef(BibRef r)
{
      this.sId=r.sId;
}

// This method uses the copy
// constructor.
public Object Clone()
{
      return new BibRef(this);
}

///////////////// Comparison group.
public override Boolean
Equals(Object o)
{
      // If the object is null then
      // is different from this
      // object:
      if(o == null) return false;
      // If the type of the object
      // is different from this
      // object, then they are
      // different:
      if(this.GetType() !=
            o.GetType())
      return false;
      // Now a field by field
      // comparison is performed.
      // For this simplistic
      // example there is only one
      // field to check:
      if(this.sId !=
            ((BibRef)o).sId)
      return false;
      // The objects are equal:
      return true;
}

public override Int32 GetHashCode()
{
      // The algorithm uses the
      // readonly field sID.
      Int32 hc=0;
      for(Int32 i=0;
```

```
            i<sId.Length;
            i++)
      {
      hc+=(sId[i].GetHashCode()*(i+1));
      }
      return hc;
}

// Equals(), GetHashCode() and ToString()
// can also be called without explicit
// intervention from the programmer. (v.g.
// by WriteLine())
public override string ToString()
{
      return sId;
}

} // End of class BibRef
```

Due to space constraints the complete program is not shown here, but only a snippet instead.

Consequences

• Advantages: A class built with this features produce objects which are easier to control than those not conforming with this idiom. Therefore, objects designed in this way turn out to be flexible (simple to understand, use and change) and contribute for the code to be easy to modify, given the fact that they are readily adaptable in any way most objects are often adapted.

• Disadvantages: Having so many points to consider, the programmer ought to be careful in order to avoid making mistakes. A quite frequent error is to write a hash function (`GetHashCode()`) which is either simply wrong or at least slow.

References

[JIMR04] Jimeno, Roberto (2004). Una Forma Ortodoxa para las Clases en C#. UNAM. Mexico.

[ORTJ96] Ortega Arjona, Jorge Luis (1996). Estudio y evaluación de la programación orientada a objetos. UNAM. Mexico.

[RICJ02] Richter, Jeffrey (2002). Applied Microsoft .NET Framework Programming. Microsoft Press. USA.