Using an Environment Chain Model to Teach Inheritance in C#

Steven J. Ratering
Department of Computer Science
University of Wisconsin - Eau Claire
Eau Claire, WI 54701
raterisj@uwec.edu

Michael R. Wick
Department of Computer Science
University of Wisconsin - Eau Claire
Eau Claire, WI 54701
wickmr@uwec.edu

Abstract

Inheritance is a central topic in most introductory object-oriented programming courses and frequently the most challenging for students to comprehend. The difficulties students experience often center around the rather complicated and obscure dynamic method invocation algorithms used in languages like C++, Java, and C#. We apply a visual model for teaching inheritance that captures the essential aspects of these algorithms yet is easy for students to comprehend and apply.

1 Introduction

Inheritance is frequently listed as one of the central themes in object-oriented programming. As such, it is a necessary component of any introductory object-oriented programming course. However, teaching entry-level students the intricate details of how inheritance is achieved in a particular language is extremely difficult. We have found that entry-level computer science students are overwhelmed by the complexities of the complete dynamic method invocation algorithms that support inheritance. Further, we have found that it isn't necessary for a beginning computer science student to understand these algorithms in their entirety. Students in their first object-oriented programming course rarely devise code that requires a detailed knowledge of dynamic method invocation. Rather, beginning students require an easy-to-understand model of inheritance that correctly handles the class design issues that most frequently occur in an entry-level object-oriented programming course. The exact inheritance issues of interest vary depending on the implementation language. In C#, entry-level students should be given a sufficient understanding of inheritance in order to handle virtual vs. non-virtual method invocation, overrides, constructor chaining, destructor chaining, and dynamic typing.

We present an *environment chain model* of inheritance (motivated by the environment diagrams of [1]) that we believe represents an easy-to-understand model of inheritance appropriate for inclusion in an entry-level C# object-oriented programming course. It is not our claim that the environment chain model presented in this paper represents the actual implementation techniques used in C#, nor do we claim that our model accurately describes all of the details of the complete dynamic invocation algorithms. The model presented is simply a pedagogical tool that can be used to introduce inheritance topics to students without overwhelming them with all the subtle implementation details.

2 Background

Obviously, we are not the first authors to deal with the difficulties of teaching inheritance in an object oriented language. We have found that the approaches taken by most entry-level texts fall into three broad categories (when inheritance is treated at all). First, many authors teach inheritance through multiple examples (e.g., [2]). While examples are necessary, we believe the "case-based" approaches fail to provide students with the underlying principles on which to base their understanding. Second, some authors present objects using a merged memory space – showing instance variables and methods for all inherited classes in a single, flat structure (e.g., [3]). We believe the flat structure of these models makes it difficult for students to easily grasp important inheritance concepts like method overriding. Third, some authors use diagrams that show each object as a layered collection of memory, each layer built from each level in the hierarchy (e.g., [4]). We find this notation friendly and easy for students to grasp. However, the explanations for how dynamic method invocation occurs within these layers lack sufficient detail to account for the type of inheritance hierarchies that are typically found in an entry-level programming course. We have found that students become confused when they produce code that violates these overly simplistic invocation algorithms.

3 The Environment Chain Model

Our environment chain model provides a visual representation of objects that helps students understand inheritance and polymorphism. The model has four components: a diagram representing each object in the system, an algorithm for the construction of the diagrams, an algorithm for dynamic method invocation using the diagrams, and an algorithm for the destruction of the diagrams. The construction, destruction, and dynamic invocation algorithms are specific to the implementation language (in our case C#) and are provided in the next section. The diagram notation, however, is consistent across languages and is presented here.

In the environment chain model, each object is represented by a chained sequence of environments, one for each class in the object's hierarchy. Each node in the environment chain holds the variables and methods defined by the corresponding class. Further, each node points to the environment node of the parent class. For example, consider a system that defines three classes: Shape, Rectangle, and Square. In this design, Square is derived from Rectangle, which is derived from Shape. The following illustrates the three different environment chain diagrams for representing instances of Shape, Rectangle, and Square.

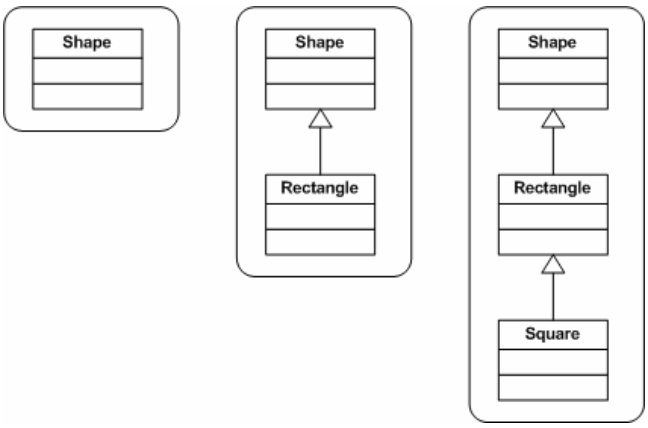

Figure 1: Environment Chain Diagram

It is important to realize that the environment chain model as presented does not handle every conceivable inheritance hierarchy. For example, multiple inheritance and interfaces are not modeled. The model also assumes that all methods are non-private and non-static within a class hierarchy. While these are certainly significant restrictions, they do not interfere with the goal of developing an easy-to-understand model of inheritance for use in entry-level object-oriented programming courses. When necessary and appropriate, the model can be extended to handle these restricted features.

4 The Environment Chain Model for C#

In C#, an environment chain is built whenever an object is created. Each environment chain is constructed node-by-node from top-to-bottom (i.e., from superclass-to-subclass) before any constructor is invoked. Students familiar with C++ may be surprised by this behavior, since in that language, a constructor is invoked prior to building the node at the next lower level.

C# Algorithm for Chain Construction

```
build the node for the base class
while there is a derived class {
  build an environment node for the derived class
  point the new node to the parent node
}

invoke the constructor for the base class
while there is a derived class {
  move down the chain one level
  invoke the constructor for this class
}
```

The destruction of a C# environment chain is essentially the reverse of the construction algorithm.

C# Algorithm for Chain Destruction

```
enter the chain at the bottom node
do {
  invoke the destructor
  move up the chain to the next node
} while the node exists

enter the chain at the bottom node
do {
  destroy the node
  move up the chain to the next node
} while the node exists
```

The algorithm for finding the method to invoke within an environment chain is more complicated. The basic idea is to enter the chain based on the apparent type of the expression used to invoke the method. From this point, the algorithm searches upward through the chain until a matching method signature is found. (This step can invoke non-trivial type casting. In our use of the environment chain model, we leave the details of the casting algorithm out and instead present simple examples for which the appropriate casts are apparent.) If the matching signature is not qualified with virtual or override, then the invoked method will be the one found in the current environment node. However, if the matching signature is qualified with virtual or override, we look for a lower node with that signature. If such a node is found and the method is qualified with override we tentatively select that method. We continue to look for a match further down the chain until we see a match not qualified by override or we reach the end of the chain. Then the last tentatively selected method will be the one invoked.

C# Dynamic Method Invocation Algorithm

```
enter the chain at the environment node of the
  apparent type of the expression

while not at the top node and a matching signature is not found{
    move up one environment node
}

if a matching signature is found {
  candidate ← def. of most specific matching signature in current node

  if candidate method is virtual or override {

    drop down one environment node
    while node exists and
          non-override of selected signature is not found in node {
      if override signature found {
        candidate ← def. of selected signature in current node
      }
      drop down one environment node
    }
  }

  invoke candidate method
} else {
  halt – error
}
```

4.1 Demonstration Example

The following code is used to illustrate the C# environment chain model. The code includes the definition of a Point class and a derived Dimension class.

Sample C# Code – Auxiliary Classes

```
class Point {
  public int x, y;
  public Point(int xp, int yp){
    x = xp;
    y = yp;
  }
}
class Dimension : Point {
  public Dimension(int xp, int yp) : base(xp, yp){}
}
```

The main hierarchy includes the definition of three classes: Shape, Rectangle, and Square (with the obvious inheritance structure). The signatures of the methods for each class should be self-explanatory. The bodies are simply stubs that output trace information for use in the explanation.

Sample C# -- Shape Hierarchy

```
class Shape {
  public Shape() {
    Console.WriteLine("Shape:" + ToString());}

  ~Shape() {
    Console.WriteLine("~Shape:" + ToString());}

  public virtual void Translate( Dimension d ) {
    Console.WriteLine("Shape:Translate");}

  public override string ToString() {
    return ("Shape"); }

  public virtual void Grow( int dx, int dy ) {
    Console.WriteLine("Shape:grow"); }
}

class Rectangle : Shape {
  public Rectangle() {
    Console.WriteLine("Rectangle:"+ ToString());}

  ~Rectangle() {
    Console.WriteLine("~Rectangle:"+ToString());}

  public void Translate( Point p ){
    Console.WriteLine("Rectangle:Translate"); }

  public override string ToString() {
    return ("Rectangle"); }

  public override void Grow( int dx, int dy ){
    Console.WriteLine("Rectangle:Grow"); }
}

class Square : Rectangle {
  private int n;  // for construction/destruction illustration

  public Square() {
    n = 3;  // arbitrary value for illustration
    Console.WriteLine("Square:" + ToString());}

  ~Square() {
    Console.WriteLine("~Square:" + ToString());}

  public override string ToString() {
    return ("Square " + n); }

  public virtual void Grow( int d ) {
    Console.WriteLine("Square:Grow");}

  public override void Translate(Dimension d) {
    Console.WriteLine("Square:Translate Dimension");}

  public override void Translate(Point p){
    Console.WriteLine("Square:Translage Point");}
}
```

The following Main() illustrates the use of the environment chain model in the explanation of instance construction, dynamic typing, and instance destruction in C#. Each specific portion of the example is discussed in its own subsequent subsection.

Sample C# -- Main Class

```
class TestShape {
  public static void Main(){
    Shape shape = new Square();
    Shape.Grow(10, 20);
    Rectangle square = (Rectangle)shape;
    square.Translate( new Dimension( 10, 20 ) );
  }
}
```

Output from Main Class

```
Shape:Square 0
Rectangle:Square 0
Square:Square 3

Rectangle:Grow

Square:Translate Point

~Square:Square 3
~Rectangle:Square 3
~Shape:Square 3
```

4.2 Construction Explained

Let's start with the construction portion of our example. Notice that each constructor outputs the result of the ToString() method. The main program allocates a single Square object but the output prints all three class names. A careful trace of the construction of the environment chain model clearly explains why and why they appear in the order shown.

Sample C# -- Construction Code Segment

```
Shape shape = new Square();
```

Output Segment

```
Shape:Square 0
Rectangle:Square 0
Square:Square 3
```

According to the construction algorithm, all three nodes are created before the constructors are invoked. The first constructor invoked is for Shape. This constructor invokes ToString(). Virtual methods have the virtual or override qualifier. If the qualifier is "virtual", then that method is the original. Shape's ToString is qualified with

"override" because it overrides Object's ToString(). Since ToString is virtual we search for another match farther down the chain. We find matches at the Rectangle and Square levels. Note that Square's constructor has not yet executed and we are about to execute its ToString() method. At this time n has the value 0 so ToString returns "Square 0". Similarly Rectangle's call to ToString() will also return "Square 0". Finally, when Square's constructor is invoked, n is initialized to 3, and Square's ToString() returns "Square 3". This simple example can be used to show the importance of avoiding virtual method calls in a constructor.

4.3 Dynamic Typing Explained

Next, let's examine a method invocation example involving dynamic typing. In this code segment, the main program invokes the Grow() method using a Shape reference. Notice that the signature used in the main program requires two integer variables.

Sample C# -- Dynamic Typing Code Segment 1

```
shape.Grow(10, 20);
```

Output Segment

```
Rectangle:Grow
```

The static type of the method invocation is Shape (since the method is invoked using a Shape reference). Therefore, the dynamic method invocation algorithm begins in the Shape node. A virtual method matching that signature is found in this node, causing the algorithm to search for a lower match. A match is found in Rectangle, but not in Square, so the Rectangle Grow() method is invoked. The Shape Grow() method can easily be made non-virtual and the students can quickly see that the Shape Grow() method would then be invoked instead.

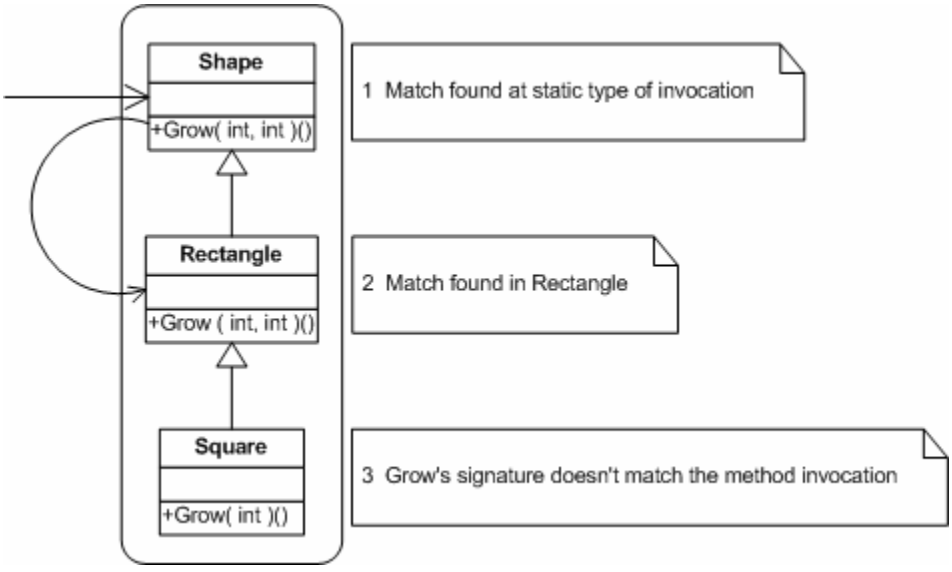

Figure 2: Method Invocation with Dynamic Typing

Another source of confusion for entry-level students regards dynamic method invocation when a parameter is a derived object. In the next segment of our example, Main() invokes the Translate() method using a Square reference passing an instance of Dimension as the actual parameter.

Sample C# -- Dynamic Typing Code Segment 2

```
Rectangle square = (Rectangle)shape;
square.Translate( new Dimension( 10, 20 ) );
```

Output Segment

```
Square:Translate Point
```

The dynamic method invocation algorithm enters the chain at the Rectangle node (Fig. 3). The signature for Translate(Point) is found and since Dimension derives from Point, we have a match, we will not search further for a better match. Since this Translate is virtual we look in Square and find a method with a matching signature. Because the signature was already determined, the Translate with the Point parameter is invoked.

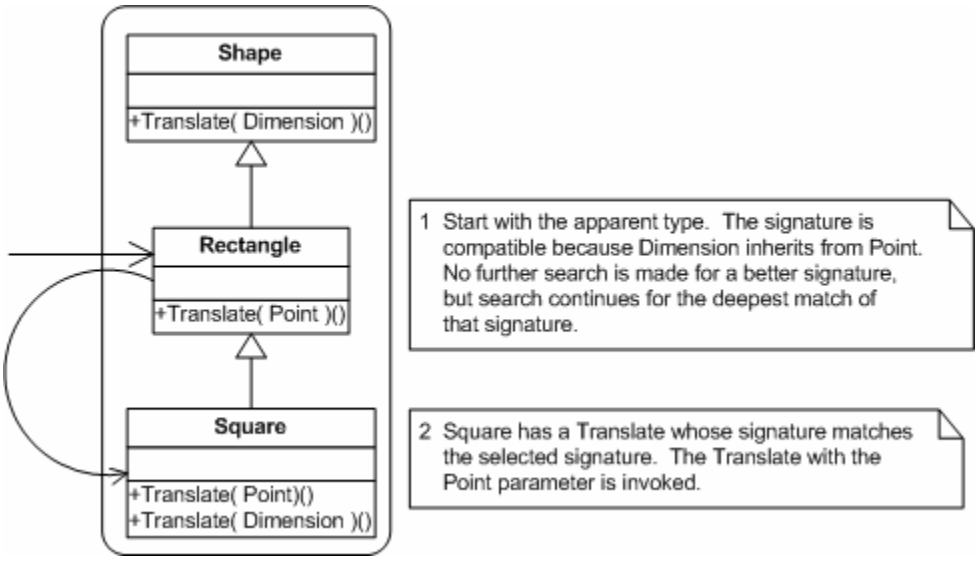

Figure 3: Method Invocation with inherited parameter types

This example illustrates to students the manner in which parameter inheritance and search are combined in dynamic method invocation in C#. Again, the environment chaining model makes it relatively easy to determine which methods are invoked and why they are invoked.

4.4 Destruction Explained

Output Segment for Destruction

```
~Square:Square 3
~Rectangle:Square 3
~Shape:Square 3
```

Finally, let's look at how the environment chain model can help explain instance destruction. Returning to our original example, notice that each destructor is defined to output the result of invoking the virtual ToString() method. When C# performs garbage collection, the three destructors will be invoked in order from Square to Shape. ToString is virtual so Square's ToString() is invoked each time. After the three destructors are finished, the three nodes of the environment chain model are removed. This shows why destructors, like constructors, should avoid virtual method calls. The problem could be more serious than accessing an instance variable of a derived class after the derived class has been destructed. Square's constructor could open a connection to a database or reserve some other resource and then Square's destructor would close the connection or free the other resource. The destructors of Rectangle or Shape might then invoke a virtual method that accessed a Square method that accessed that database or other resource.

5 Summary

Inheritance is an important topic in object-oriented programming. We have found that beginning students have a difficult time grasping the basic ideas of inheritance when presented with the full-blown dynamic method invocation techniques defined by the language designers. Further, we have found that the approaches taken by most entry-level texts lack enough substance to account for the behavior that a typical entry-level student will experience. We have developed a visual tool that allows students to quickly and easily learn the basic C# inheritance rules that account for most of the behavior they are likely to encounter. This model also gives a framework to point out the subtle but important differences between Java, C++, and C# (see [5]).

References

[1] H. Abelson, G. Sussman, and J. Sussman, **Structure and Interpretation of Computer Programs**, 2nd Edition, McGraw-Hill Publishing, (1996).

[2] H. Deitel, and P. Deitel. **C++ How To Program**, Prentice-Hall Publishing (1994).

[3] R. Johnsonbaugh and M. Kalin, **Object-Oriented Programming in C++**, 2nd Edition, Prentice-Hall Publishing, (2000).

[4] D. Kaura, **Object-Oriented Software Design and Construction with C++**, Prentice-Hall Publishing, (1998).

[5] S. Ratering, M. Wick, and D. Stevenson, Using an Environment Chain Model to Compare C++, Java, and C#, *In Preparation*.

[6] M. Wick, D. Stevenson, and A. Phillips, Using an Environment Chain Model to Teach Inheritance in C++, *Proceedings of the Thirty-third SIGCSE Technical Symposium on Computer Science Education* (2002), pp. 297-301.