

On the Integration of Classboxes into C#

Markus Lumpe¹ and Jean-Guy Schneider²

¹ Department of Computer Science
Iowa State University
Ames, IA 50011, USA
lumpe@cs.iastate.edu

² Faculty of Information & Communication Technologies
Swinburne University of Technology
P.O. Box 218
Hawthorn, VIC 3122, AUSTRALIA
jschneider@swin.edu.au

Abstract. Classboxes are a new module system for object-oriented languages defining a packaging and scoping mechanism for controlling the visibility of isolated extensions to portions of class-based systems. Unlike object-oriented specialization, the class extension mechanisms supported by classboxes preserve the identity of extended classes and, therefore, all clients of extended classes can benefit from the applied extensions. In this paper, we present a language design and a corresponding implementation strategy for classboxes in C#. A particular challenge in incorporating classboxes into C# is to preserve the identity of extended classes as the .NET framework represents classes as metadata type declarations and access to classes by static links into metadata of the host assembly. However, the local refinement of an imported class results in a new metadata type declaration. In order to guarantee the identity of extended classes, new metadata type declarations have to be incorporated into the original metadata of imported classes. But this “re-wiring” has to occur in a manner that is consistent with the Common Language Infrastructure (CLI).

1 Introduction

Today, many real-world software systems are built using mainstream object-oriented techniques and languages. However, when using object-oriented technology, one often faces an *extensibility problem* that arises from the fact that mainstream object-oriented languages provide only limited support for modular addition of both *horizontal* and *vertical* extensions to classes. While in general it is always possible to add (vertically) new classes to a system, existing classes can only be extended with new, orthogonal behaviour (horizontally) in an often non-object-oriented style, that is, by breaking the object-oriented encapsulation property (e.g., the Visitor pattern [11]). Such extensions are awkward at best, and error-prone at worst. Furthermore, the inheritance relationships in mainstream object-oriented languages are not powerful enough to capture many useful forms of incremental modifications [3, 5, 6, 17].

To address this problem, several approaches have emerged that focus on tangible techniques for evolving object-oriented software systems that do not rely on standard

inheritance mechanisms [1, 5, 7, 12, 15, 17]. Of special interest is the concept of *classboxes* proposed by Bergel et al. [5], a new *module system* for object-oriented languages that defines a packaging and scoping mechanism for controlling the visibility of isolated extensions to portions of class-based systems. Classboxes define *explicitly named scopes* within which (i) classes, methods, and variables are defined, (ii) classes can be extended using the “traditional” operation of subclassing, and (iii) classes can be *imported* from other classboxes. More importantly, however, classboxes also support *local refinement* of imported classes by adding and/or modifying their features, without affecting the originating classbox. As such, classboxes offer a promising approach, as they provide support for extending existing classes both vertically and horizontally.

Classboxes have been implemented for both the Smalltalk [5] and the Java environments [4]. The Smalltalk implementations of classboxes mainly rely on a modified, “classbox-aware” virtual machine for dynamic class and method lookup and a reification of the method call stack, respectively. Central to the modified virtual machine is a graph search algorithm that implements the local rebinding of methods at runtime, an approach that cannot be easily mapped to languages whose runtime environment does not offer the same amount of flexibility. Classbox/J [4], on the other hand, is a prototype classbox implementation for Java that is based on preprocessor directives and reification of the method call stack. It uses an implementation scheme that may reveal extensions of classes to clients, which should not be able to see them (i.e., new class members are inserted into the original class without any additional visibility control [4, §6.1]).

But is it possible, to incorporate classboxes into an industrial-strength programming language without relying on a “classbox-aware” virtual machine or a preprocessor, respectively? To answer this question, we present a backward-compatible implementation strategy for classboxes in C# in this work. Our implementation shows that the metadata concept of the Common Language Infrastructure (CLI) [14] plays a crucial role in integrating class extensions seamlessly into the C# language without the need to modify the underlying runtime infrastructure. Furthermore, to facilitate code reuse and to assist in building families of classes that are subject to the same change, we introduce the notion of *reusable class extensions* to the classbox concept. Our results not only demonstrate the expressive power of classboxes in an industrial-strength programming language such as C#, but also illustrate how reusable class extensions substantially improve the specification of incremental modifications in classboxes.

The rest of this paper is organized as follows: in Section 2, we present an example to illustrate the concept of classboxes, followed by our model and implementation of classboxes in C# in Section 3. We conclude this paper in Section 4 with a summary of the presented work and outline future work in this area.

2 Applying Classboxes

In order to illustrate the concept of classboxes and the associated expressive power, consider the example given below that motivates the need to restrict the impact of a modified class.

A class `Point` implements the behaviour of two-dimensional points. It contains two private instance variables `x` and `y`, two public properties `X` and `Y` to *get/set* the corre-

sponding point coordinates, a public method `MoveBy` to move a point by a given offset (`dx,dy`), and a public method `MoveByXY` that doubles the values of the `x` and `y` coordinates by invoking `MoveBy` (using dynamic method lookup). The class `BoundedPoint` is a direct specialization of `Point` that ensures that the `y` coordinate of an instance never exceeds a given upper bound `yBound`. This bound is a constant in `BoundedPoint`, but this behaviour can be altered by overriding the property `Bound`, as show below.

In order to define a non-constant bound, we can specialize the class `BoundedPoint` by overriding the property `Bound` to return `X`. That is, the resulting class `LinearBoundedPoint` implements a behaviour guaranteeing that the value of the `y` coordinate is always smaller than the value of the `x` coordinate. As this specialization does not affect either the class `Point` nor the class `BoundedPoint`, any clients of these two classes will not be affected.

If we want to add color to our point-class hierarchy (i.e., adding a private instance variable `color` and a corresponding property `Color` to all classes), we can simply add the corresponding features to the class `Point`. As the additional behaviour is *orthogonal* to the existing behaviour, none of the clients of any of the point classes will be affected by this modification.

However, if we suddenly need to alter the behaviour of bounded points (i.e., restricting the `x` coordinate instead of the `y` coordinate), we cannot simply modify the class `BoundedPoint`, as such a modification would affect all clients of `BoundedPoint` and the (implicit) contract between `BoundedPoint` and `LinearBoundedPoint` would be broken. Hence, “traditional” subclassing fails to provide us with the required expressive power. What we need is an approach where we can *restrict* the modified behaviour of bounded points to well-encapsulated parts of our application, leaving all existing clients unaffected by this modification.

Classboxes provide us with a solution to this problem, as they provide a framework in which we can control both the scope and the impact of change [4,5,13]. In particular, classboxes exhibit the following main characteristics [5]:

- A classbox is an explicitly named unit of scoping in which classes (and their associated members) are defined. A class belongs to the classbox it is first *defined*, but it can be made visible to other classboxes by either *importing* or *extending* it.
- Any extensions to a class are only visible to the classbox in which they occur first and any classboxes that either explicitly or implicitly import the extended class. Hence, overriding a particular method of a class in a given classbox will have no effect in the originating classbox.
- Although class extensions are only locally visible, their embodied refinements extend to all collaborating classes within a given classbox, in particular to any subclasses that are either explicitly imported, extended, or implicitly imported.

In order to illustrate how classboxes address the problems associated with implementing the point class hierarchy illustrated above, consider the four classboxes depicted in Figure 1, namely *OriginalCB*, *LinearCB*, *ColorCB*, and *TraceCB*, respectively.

The classbox *OriginalCB* contains the class `Point` as well as the class `BoundedPoint` as a direct specialization of `Point`. The classbox *LinearCB* introduces the class `LinearBoundedPoint`, which, in order to define a non-constant bound, specializes

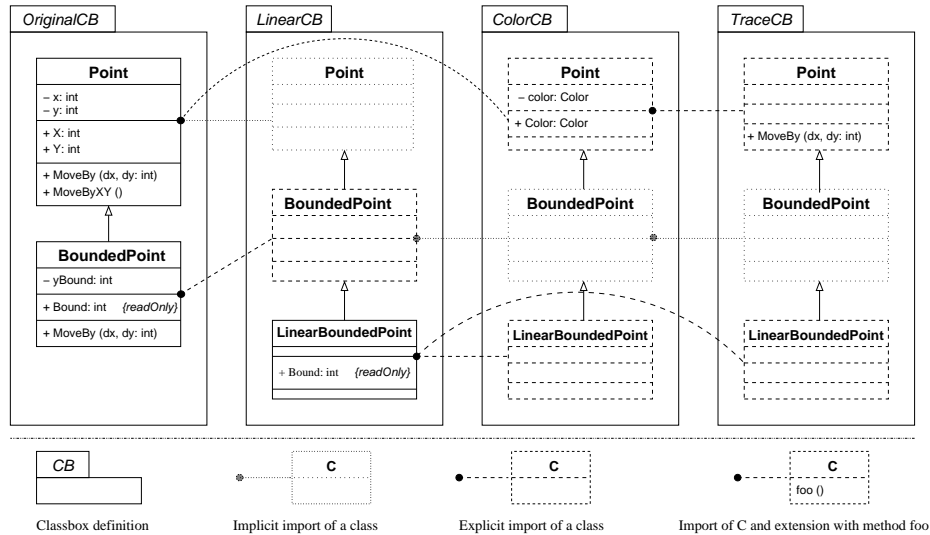


Fig. 1. Sample classboxes.

BoundedPoint by overriding the property Bound to return X. LinearBoundedPoint is defined as a subclass of BoundedPoint imported into LinearCB from OriginalCB. As a consequence, the class Point is implicitly imported also, making it visible as the direct superclass of BoundedPoint, but not accessible to clients of the classbox LinearCB.

The classbox ColorCB extends the class Point from OriginalCB by adding a private instance variable color and a corresponding property Color. As a consequence, all instances of Point in ColorCB as well as the instances of any of its subclasses possess this additional behaviour. Therefore, the class LinearBoundedPoint imported from LinearCB also possesses the color feature, but all classes in OriginalCB and LinearCB remain unaffected by this alteration. The reader should note that the explicit import of LinearBoundedPoint from LinearCB also triggers the implicit import of BoundedPoint from LinearCB.

Finally, the classbox TraceCB defines an extended version of the class Point from ColorCB by overriding the method MoveBy to add a tracing facility (i.e., each invocation of MoveBy is monitored by a console message).

3 Classboxes in C#

In previous work on formalizing classboxes [13], we have identified four unique classbox operations, namely (i) import of classes, (ii) introduction of subclasses, (iii) refinement of classes, and (iv) inclusion of new behaviour. The latter two operations are deduced from the original extend operator [5] by revising the notion of extending classes. That is, the refinement operator should exhibit a behaviour similar to C#'s new modifier [8, §17.2.2], which can be used to hide any super-class method by declaring a new method with the same signature in a subclass, whereas the inclusion operator roughly

corresponds to *mixin application* [6] in which the extended class takes the role of an *abstract subclass*.

In addition, both *refinement* and *inclusion* are modeled in a way that enforces a separation of the incremental modification defined by these operations from the underlying extension mechanism [13]. As a result, incremental modifications become reusable software artifacts. Within a classbox, this allows for an approach in which we can compose these software artifacts with multiple imported classes simultaneously.

In the following, we outline the design and implementation of classboxes in C#. As a proof-of-concept, we have implemented our classbox model in standard C# by means of source code transformations, refactoring [16], and locally updating metadata type declarations. While our ultimate goal is to define an approach in which class extensions can be compiled into existing classes without source code access, using refactoring has produced valuable insights into the compilation process.

3.1 Classboxes as C# Namespaces

In our model, classboxes are represented by C# *namespaces*. A namespace in C# is a compile-time construct that defines a scope to organize source code and *globally-unique* types [8]. All namespaces have implicitly public access. Moreover, namespaces are open-ended and can be divided into separate compilation units, which then all contribute to the same declaration space [8, §16.2]. However, once a type has been defined, it can only be extended by means of “traditional” subclassing, a technique that does not suffice to express the incremental modifications used, for example, in the classboxes *ColorCB* and *TraceCB*, respectively. Therefore, we refine the notion of namespace to be an *open* scoping mechanism that enables the local redefinition of type declarations. However, to retain complete backward compatibility with the extant C# language, a corresponding compilation scheme for our extended C# language has to target the standard CLI.

At the technical level, C# is compliant with the *Common Language Infrastructure* (CLI), a specification that provides a basis for a *virtual execution system* insulating programs from the underlying operating system [14]. In CLI, new types are introduced via *read-only* metadata type declarations.³ Moreover, metadata contains information to locate and load classes, lay out instances in memory, resolve method invocations, and enforce security constraints. Consider, for example, the classbox *OriginalCB*: it defines two types *Point* and *BoundedPoint*, respectively, both defined within the namespace *OriginalCB*. The two types are represented by *TypeDef* metadata tokens as shown in Figure 2. *TypeDef* tokens encode the name of a type, its declaration namespace, the super type (index into a *TypeDef* or *TypeRef* table), and a list of the type’s members. In addition, *OriginalCB*’s metadata contains a *TypeRef* token that encodes an index into the *TypeDef* table of the assembly defining *System.Object*. This token is required, since *Point*’s super type is *System.Object*, the root of every reference type in C#.

A particularly difficult problem in representing classboxes in the .NET framework is induced by *fully qualified class names*, as they are encoded as pointers into *metadata*

³ The reader should note that without loss of generality, we will only consider *TypeDef* and *TypeRef* tokens in this work. These tokens normally utilize other metadata tokens to correctly establish references to the defining assemblies.

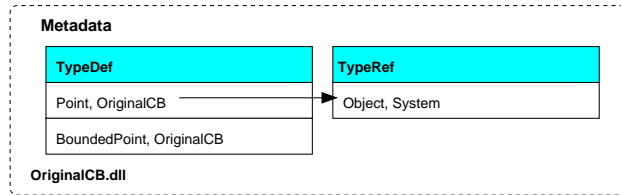


Fig. 2. *OriginalCB*'s metadata.

associated with every assembly and, therefore, hard-wire classes with their corresponding direct parent-classes. However, the key provision for providing support for classboxes in C# is that we need to be able to locally *update* metadata type declarations. This requirement arises from the fact that the local refinement of an imported class results in a new metadata type declaration. To restore the identity of extended classes, this new metadata type declaration needs to be incorporated into the original metadata of imported classes.

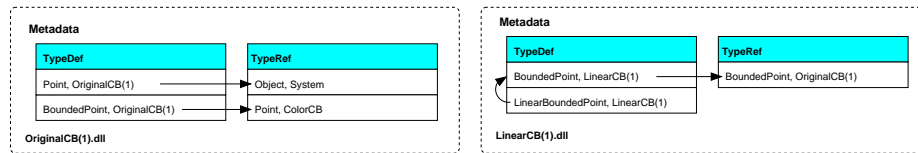


Fig. 3. Updated metadata of *OriginalCB(1).dll* and *LinearCB(1).dll*.

Consider again the classbox *ColorCB* that defines an extended version of the class `Point` imported from *OriginalCB*. Compiling this code yields a new assembly, called `ColorCB.dll`, that contains the classes `Person` and `LinearBoundedPoint` (explicitly imported from *LinearCB*) and their corresponding metadata (c.f., Figure 4). However, the inheritance chain between both classes and the implicitly imported class `BoundedPoint` is not preserved.

In order to link the extended class `Point` with the implicitly imported class `BoundedPoint` in *ColorCB*, we need to *update* `BoundedPoint`'s metadata type declaration in the assemblies of both *OriginalCB* and *LinearCB*, respectively. More precisely, we need to create new versions (marked (1)) of their corresponding assemblies, in which the super-class type declaration of `BoundedPoint` refers to `ColorCB.Point` (see Figure 3). To add `ColorCB.Point` in *OriginalCB* we need to extend the `TypeRef` table with a new token for `ColorCB.Point` and rewire `BoundedPoint`'s super type index to the newly added token. In *LinearCB*, we need to "patch" the original `Point` `TypeRef` token in order to connect it to `ColorCB.Point`. Adding a new token does not work, since this would break the link between `Point`'s `TypeRef` token and its applications in the member lists of `BoundedPoint`. This process results in a *physical* structure consisting of three assemblies for the *logical* structure defined by the classbox *ColorCB*.

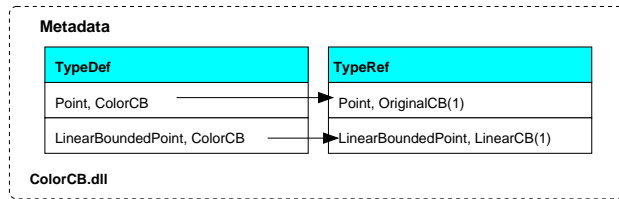


Fig. 4. Metadata of *ColorCB.dll*.

Classes in .NET assemblies are represented as metadata type tokens and references to classes are encoded as pointers to these metadata type tokens. To change the link structure in metadata, we need to create new assemblies, since .NET assemblies do not allow these links to be changed dynamically. Altering the metadata structure, as shown in Figure 3, guarantees that the local refinement of the imported class *Point* in classbox *ColorCB* (the resulting metadata is shown in Figure 4) does not impact the originating classbox *OriginalCB* and its clients.

This approach works for all *managed* assemblies [14]. Unfortunately, some namespaces in the .NET framework are only available in *native* code (e.g., the *System* namespace) and classes defined in those namespaces cannot be extended due to the lack of metadata.

3.2 Import and Subclassing

Both, import and subclassing do not require any language changes. The available language abstractions suffice to specify these operations. However, the import operation may require a local update of the metadata type declaration associated with an imported class, as outlined in Section 3.1.

Consider the classbox *OriginalCB* defining the classes *Point* and *BoundedPoint*, which are in an inheritance relationship. The corresponding implementation of *OriginalCB* is shown in Listing 1. Compiling *OriginalCB* yields a dynamic link library (DLL) or *assembly* called *OriginalCB.dll*. This library defines two public types *OriginalCB.Point* and *OriginalCB.BoundedPoint*, both recorded as type declarations in *OriginalCB*'s metadata.

The classbox *LinearCB* can be defined similarly in standard C#, as shown in Listing 2. *LinearCB* explicitly imports *BoundedPoint* from *OriginalCB*. We use the *alias* form of the *using* directive to specify class imports [8, §16.4.1]. An alias for a type is a user-defined name that is only available within the namespace body introducing it. However, to enable a local rebinding of features of explicitly imported classes, we create an "empty" subclass with the same name for each imported class in the importing classbox. This empty subclass does not define any new functionality, but it enables clients of the importing classbox to use the explicitly imported class as it had been defined in the importing classbox. This technique corresponds to the approach proposed by Bergel et al. [5], in which importing a class into a classbox is the same as extending this class with an empty set of methods. The newly created subclass for an imported

```

namespace OriginalCB {
public class Point {
private int x, y;

public Point( int ix, int iy ) { x = ix; y = iy; }

public int X { get{ return x; } set{ x = value; } }
public int Y { get{ return y; } set{ y = value; } }

public virtual void MoveBy( int dx, int dy ) { X += dx; Y +=dy; }
public virtual void MoveByXY() { this.MoveBy( X, Y ); }
}

public class BoundedPoint : Point {
private int yBound;

public BoundedPoint( int ix, int iy, int ibound ) :
base( ix, iy ) { yBound = ibound; }

public virtual int Bound { get{ return yBound; } }

public override void MoveBy( int dx, int dy )
{ if ( Y + dy < Bound ) base.MoveBy( dx, dy ); }
}
}

```

Listing 1. The namespace OriginalCB.

class is linked to its original image by a type reference (i.e., a `TypeRef` metadata token) in the importing classbox (c.f. Figure 4). Therefore, clients of the classbox *LinearCB* can access the imported class `BoundedPoint` as it had been originally defined in classbox *LinearCB*, even though `BoundedPoint`'s behaviour is actually being defined by the class `OriginalCB.BoundedPoint` hosted by `OriginalCB.dll`.

```

namespace LinearCB {
using BoundedPoint = OriginalCB.BoundedPoint;

public class LinearBoundedPoint : BoundedPoint {
public LinearBoundedPoint( int ix, int iy, int ibound ) :
base ( ix, iy, ibound ) { }

public override int Bound { get{ return X; } }
}
}

```

Listing 2. The namespace LinearCB.

In order to compile *LinearCB*, we need to add `OriginalCB.dll` as a *reference* to look up the metadata associated with the class `OriginalCB.BoundedPoint`. The result is the assembly `LinearCB.dll` that defines one public type `LinearCB.LinearBoundedPoint`. The reader should note, however, that due to the implicit import of `OriginalCB.Point` and the explicit import of `OriginalCB.BoundedPoint`, clients must have access to both `OriginalCB.dll` and `LinearCB.dll` in order to use the classbox *LinearCB*.

3.3 Reusable Class Extensions

The original classbox concept defines the extension of classes as an operation that works like import, except that the imported class is instantaneously altered in order to add and/or change its features [5]. More precisely, a particular local refinement is exclusively associated with the imported class. Thus, if we want to apply the same refinement to another class, then we need to *duplicate* its corresponding specification. However, it is more desirable to separate the local refinement from the import of classes to facilitate code reuse and to assist in structuring the specification of incremental modifications to classes within a classbox [17, 18].

Therefore, we propose the introduction of a new linguistic facility, called *explicit class extension*, that is based on the concept of *mixins* [6]. However, while mixins are *class-to-class functions* [10], we define explicit class extensions as *open mixins*, which are parameterized over a “composition” mechanism. The purpose of the composition mechanism is to determine how a given extension is to be merged in an imported class. In this work, we shall consider two composition mechanisms: *refinement* and *inclusion* [13]. The syntax for *extension-declarations* is given below:

```
extension-declaration:  
    extension identifier class-body ;opt
```

An *extension-declaration* consists of the keyword **extension**, an *identifier* that names the extension, a *class-body* [8, §17.1.3], followed by an optional semicolon.

At present, extensions are a pure mechanism to create incremental modifications. Thus, extensions can only be type-checked after they have been applied to a class and this class is *linearized* (or “flattened”) [6, 18]. We plan, however, to add *required* interfaces [15, 18] to extensions in future work to address this shortcoming. In this work, we shall assume that extensions are always well-typed.

The linearization process involves two steps: (i) *closing* the extensions using a corresponding extension composition mechanism, and (ii) *applying* the closed extensions to the imported class. Assume, for example, the extensions Δ_1 and Δ_2 , the extension composition mechanisms W_1 and W_2 , and the imported class C . Then the extended imported class C' is defined as

$$C' = \Delta_2(W_2) \oplus (\Delta_1(W_1) \oplus C)$$

where \oplus is a non-commutative *class-to-class* mixin composition operator [6], and both $\Delta_1(W_1)$ and $\Delta_2(W_2)$ denote extensions closed by their corresponding extension composition mechanism W_1 and W_2 , respectively. However, if conflicts arise due to composing an imported class with extensions that provide identical named members, these conflicts have to be resolved manually.

The classboxes *ColorCB* and *TraceCB* each import the class **Point** and define an extension to it. *ColorCB* defines the extension **Color**, as shown in Listing 3. This extension defines a **Color** property and an associated private instance variable **color**. In order to define the **Color** extension, we also need to import the namespace **System.Drawing** that provides the definition for the type **Color**. In addition, the reader should note that every public member in an extension is *virtual* by default. However, it is the extension

composition mechanism that determines the actual required modifier (e.g., `new` in case of refinement, and `override` in case of inclusion).

```
using System.Drawing;

extension Color
{
    private Color color;

    public Color Color { get{ return color; } set{ color = value; } }
}
```

Listing 3. The extension `Color`.

Similarly, *TraceCB* defines the extension `TraceMoveBy`, which is shown in Listing 4. This extension captures a specialization of the method `MoveBy` that (i) prints a message to the console and (ii) transfers the control to the *original* base method `MoveBy`. Hence, the extension `TraceMoveBy` requires that the class it is eventually applied to defines at least a public virtual method `MoveBy`. This property is verified when `TraceMoveBy` is composed with a given imported class.

```
extension TraceMoveBy
{
    public void MoveBy( int dx, int dy )
    { Console.WriteLine( "MoveBy: {0}, {1}", new object[] { dx, dy } );
      base.MoveBy( dx, dy ); }
}
```

Listing 4. The extension `TraceMoveBy`.

3.4 Refinement and Inclusion

Refinement and *inclusion* both consist of three elements: (i) an imported class, (ii) a corresponding extension composition mechanism, and (iii) some explicit class extensions. To facilitate the specification of the corresponding classbox operations, we propose a linguistic facility that combines these elements in a single language construct – the *using-extension-directive*. The syntax for the *using-extension-directive* is given below:

```
using-extension-directive:
    using identifier = type-name extension-application

extension-application:
    includesopt refinementsopt

includes:
    include type-list ;

refinements:
    append type-list ;
```

The *using-extension-directive* introduces an identifier that serves as an alias for a type within the immediately enclosing namespace body. The *using-extension-directive* works like the *using-alias-directive* [8, §16.4.1], except that a non-empty *extension-application* specification is required. The *extension-application* may contain *includes*, *refinements*, or both.⁴ *Includes* and *refinements* are processed in the order they are specified. However, the linearization process requires that all members occurring in the extensions have pairwise distinct names.

```
namespace ColorCB {
    using System.Drawing;
    using Point = Original.Point append Color;
    using LinearBoundedPoint = LinearCB.LinearBoundedPoint;

    extension Color
    {
        private Color color;

        public Color Color { get{ return color; } set{ color = value; } }
    }
}
```

Listing 5. The classbox *ColorCB*.

To illustrate the use of the *using-extension-directive*, consider the implementation of the classbox *ColorCB*, as shown in Listing 5. *ColorCB* explicitly imports the classes *Point* and *LinearBoundedPoint*. Furthermore, it defines the extension *Color*, which is applied to the imported class *Point* with the specification

```
using Point = Original.Point append Color;
```

where **append** *Color* defines a *refinement* of the class *Point*.

The *refinement* operation defines an information hiding protocol that, when applied to a concrete class, renders the features of the extensions invisible to the class' behaviour. Hence, *refinement* yields a membrane for a class that permits calls originating from extensions, but prevents the class' behaviour to see the extensions.

To implement this behaviour, we mark all public members defined by a *refinement* extension with the **new** modifier [8, §17.2.2] to shield them from the imported class. Secondly, the resulting specifications are then used to construct a subclass of the imported class. When using subclassing to incorporate extensions into a class, former clients generally have to be modified as well in order to benefit from the refinements [9]. However, by “re-wiring” the metadata type declarations, as illustrated below, this problem disappears, because the identity of imported classes is restored. To illustrate this process, consider Listing 6, which shows the corresponding source code of the class *Point* in *ColorCB*.

⁴ *Type-list* is like the base class specification [8, §17.1.2], except that the elements are extension type names. We follow the scheme applied in the Mono compiler in which *type-list* is used to denote a list of both class and interface type names.

```

// Color(append)  $\oplus$  Original.Point
public class Point : OriginalCB.Point {
    private Color color;

    public new virtual Color Color { get{ return color; } set{ color=value; } }

    // required constructor(s)
    public Point( int ix, int iy ) : base( ix, iy ) { }
}

```

Listing 6. Transformation of class ColorCB.Point.

The class Point in *ColorCB* behaves like Original.Point, except that all clients of class ColorCB.Point now have access to the property Color. Moreover, since Color is declared new in *ColorCB*, existing clients of Original.Point remain unaffected by this change. However, to compile *ColorCB*, we need to *rewire* the metadata inheritance chain of all implicitly or explicitly imported classes in *ColorCB*. In particular, we need to link LinearCB.BoundedPoint with ColorCB.Point and ColorCB.LinearBoundedPoint with the updated LinearCB.BoundedPoint (see Figure 3). Therefore, we have to create new versions of OriginalCB.dll and LinearCB.dll, say OriginalCB(1).dll and LinearCB(1).dll, respectively, in which the metadata type definition for class Point refers to ColorCB.Point. Using OriginalCB(1).dll and LinearCB(1).dll as references, we build ColorCB.dll that implements the desired functionality of the classbox *ColorCB*.

```

namespace ColorCBX {
    using System.Drawing;

    public class Point : OriginalCB.Point {
        private Color color;

        public new virtual Color Color
            { get { return color; } set { color = value; } }

        public Point( int ix, int iy ) : base( ix, iy ) { }
    }

    public class BoundedPoint : Point
    { /* Point members */ }

    public class LinearBoundedPoint : BoundedPoint
    { /* BoundedPoint members */ }
}

```

Listing 7. Implementation of classbox ColorCB by refactoring.

The effect of compiling the classbox *ColorCB* in this way results in an assembly, whose functionality is equivalent to the code extract illustrated in Listing 7. Here, source code refactoring [16] is used to construct the namespace ColorCBX, which defines a functionality that corresponds to the one provided by the classbox *ColorCB*.

The classbox *TraceCB* (see Listing 8) defines an *inclusion* extension to the class *Point* and an explicit import of the class *LinearBoundedPoint* from the classbox *LinearCB*. Inclusion enables *down calls* to class extensions. Thus, unlike refinement, inclusion extensions may be visible throughout the extended class, as they can override existing members. However, the visibility of this effect is confined to the defining classbox and its clients; the originating classbox remains unaffected.

```

namespace TraceCB {
    using Point = ColorCB.Point include TraceMoveBy;
    using LinearBoundedPoint = LinearCB.LinearBoundedPoint;

    extension TraceMoveBy
    {
        public void MoveBy( int dx, int dy )
        { Console.WriteLine( "MoveBy: {0}, {1}", new object[] { dx, dy } );
          base.MoveBy( dx, dy ); }
    }
}

```

Listing 8. The classbox *TraceCB*.

We implement *inclusion* by marking all public members with *override* (or *virtual* if no member with the same signature exists in the imported class). The transformed extensions are then again used to construct a subclass of the imported class (e.g., class *Point*, as shown in Listing 9).

```

// TraceMoveBy(include) ⊕ ColorCB.Point
public class Point : ColorCB.Point {
    public override void MoveBy( int dx, int dy )
    {
        Console.WriteLine( "MoveBy: {0}, {1}", new object[] { dx, dy } );
        base.MoveBy( dx, dy );
    }

    // required constructor(s)
    public Point( int ix, int iy ) : base( ix, iy ) { }
}

```

Listing 9. Transformation of class *TraceCB.Point*.

Once more, in order to compile *TraceCB*, we need to rewire the inheritance chain originating from *TraceCB.Point*. Therefore, we need to create new versions (marked (2)) of the assemblies for the classboxes *OriginalCB*, *LinearCB*, and *ColorCB*, respectively. Please note that the new assemblies *OriginalCB(2).dll*, *LinearCB(2).dll*, and *ColorCB(2).dll* are all derived from the versions that were created to compile the classbox *ColorCB*. Hence, we only need to patch the corresponding metadata type tokens to link *BoundedPoint*'s superclass type to *TraceCB.Point*.

4 Conclusions and Future Work

Classboxes provide a feasible solution to the problem of controlling the visibility of change in object-oriented systems without breaking existing applications, as they strictly limit both its scope and impact to clients of the extending classbox. Consequently, classboxes can significantly reduce the risk for introducing design and implementation anomalies due to the need to adapt a software system to changing requirements [5].

In this paper, we have presented an approach to augment C#, an industrial-strength programming language, with the classbox concept. In our model, classboxes are represented by C# *namespaces* and the corresponding operations are defined using a small extension to the C# language. The integration of this extension is achieved by means of source code transformations, refactoring, and locally updating the metadata type declarations in assemblies. This approach allows us to preserve the identity of extended classes in classboxes, resulting in a seamless integration of classboxes into the .NET framework.

Although omitted due to the lack of space, our implementation of classboxes guarantees that different versions of a class can *co-exist* in the same classbox. This is achieved by representing the different versions of a class by unique metadata type tokens and appropriately linking the corresponding superclass types. Furthermore, due to the fact that class extensions are linked into the classes at *compile-time*, our approach does not add any runtime overhead in order to perform method calls. This contrasts with the existing Smalltalk implementations where, compared to “normal” method lookup, a 25% to 60% runtime overhead is added in the classbox-aware virtual machine [5].

In order to facilitate code reuse and to assist in building families of classes that are subject to the same change, we have also added the notion of *explicit class extensions* to the classbox concept. In contrast to the approach taken by Bergel and Ducasse [3] that combines classboxes with *traits*, in our model explicit class extensions are mixin-like code abstractions that combine both behaviour and state. Furthermore, explicit class extensions can be composed with imported classes using a refined C# *using-alias-directive*, giving us the flexibility to specify how and in which order specific extensions should be integrated into a given class. Explicit class extensions allow us to explicitly separate the local refinement from the import of a class and, as a consequence, substantially improve the specification of incremental modifications to classes within a given classbox.

In our extended classbox model, both refinement and inclusion extension provide a linguistic means for *separation of concerns*. However, in contrast to aspects that can be defined in a separate specification unit [12], these class extensions are *explicitly* applied to the imported classes and are *confined* to the declaration space within which they occur. The aspect-oriented programming model, on the other hand, employs a much looser coupling between extensions (i.e., aspects) and the locations of their application, denoted by *advises*. This can result in *surprises*, since (possibly unseen) code is executed in response to a method invocation simply because the method’s signature matches a corresponding *advise*, usually specified elsewhere [7].

To address this problem, Clifton [7] recently proposed the “MAO discipline” that encompasses both a design discipline and language features for modular reasoning in

aspect-oriented programs. In MAO, we distinguish between two categories of aspects: *assistants* and their associated *concern maps* that can change the behaviour of modules to which they apply, and *spectators* that do not affect other modules, as they only *view* methods. Both categories, even though not completely, correspond to our model of explicit class extensions. Inclusion can be characterized as an instance of an assistant, whereas refinement corresponds to a spectator.

In this work, we have focused on a seamless integration of classboxes into the C# language and have explored the concept of *explicit class extensions* to further enhance separation of concerns. Besides the definition of an extended C# compiler in which class extensions can be compiled into existing classes without source code access, there are a number of open questions that need to be addressed in future work.

In our model, both classes and explicit class extensions specify their “own” state whereas the trait model by Bergel and Ducasse [3] only allows the explicit extension of behaviour, but not state. Hence, in future work, we plan to investigate how to decouple behaviour and state by introducing *state-only* abstractions as first-class entities into the model. This would allow us to express classes and explicit extensions as compositions of explicit state, behaviour, and compositions thereof. As a consequence, both traits and Scala-style class composition [15] could be seamlessly integrated into the classbox model. A natural extension of such a model would then be to allow classboxes to import (and possibly extend) any of the abovementioned first-class entities. The implications of such an extension to the classbox model in the .NET framework is however not yet fully understood.

Our current model lacks the notion of *required* interfaces [15, 18] for explicit class extensions. As a consequence, explicit class extension can solely be type-checked in the context of an application to a class, but not as standalone entities. Hence, future work will need to address this issue. Similarly, we plan to investigate the implications and applicability of having classboxes as first-class values as well as parameterized (i.e., generic) classboxes.

Finally, an area where further investigations are needed is *method overloading*. Whereas we do not need to worry about this in languages such as Smalltalk, both, Java and C# allow developers to overload methods. In such a context, a correct implementation of a classbox model needs to be able to determine which of the overloaded methods is refined by an (explicit) extension. This will very probably require some form of additional type annotation to method names. A similar problem arises when instance variables with restricted visibility are *shadowed* by extensions, both issues that need to be addressed in future work.

Acknowledgement. The authors thank Alexandre Bergel and the anonymous reviewers for their valuable comments and discussions.

References

1. Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam – Designing a Java Extension with Mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, September 2003.

2. Alexandre Bergel. *Classboxes — Controlling Visibility of Class Extensions*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics, November 2005.
3. Alexandre Bergel and Stéphane Ducasse. Supporting Unanticipated Changes with Traits and Classboxes. In *Proceedings of Net.ObjectDays (NODE'05)*, pages 61–75, Erfurt, Germany, September 2005.
4. Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proceedings OOPSLA '05*, pages 177–189, San Diego, USA, October 2005. ACM Press.
5. Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling Visibility of Class Extensions. *Journal of Computer Languages, Systems & Structures*, 31(3–4):107–126, May 2005.
6. Gilad Bracha and William Cook. Mixin-based Inheritance. In Norman Meyrowitz, editor, *Proceedings OOPSLA/ECOOP '90*, volume 25 of *ACM SIGPLAN Notices*, pages 303–311, October 1990.
7. Curtis Clifton. *A Design Discipline and Language Features for Modular Reasoning in Aspect-oriented Programs*. PhD thesis, Iowa State University, Department of Computer Science, July 2005.
8. European Computer Machinery Association. *Standard ECMA-334: C# Language Specification*, third edition, June 2005.
9. Robert Bruce Findler and Matthew Flatt. Modular Object-Oriented Programming with Units and Mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34 of *ACM SIGPLAN Notices*, pages 94–104. ACM Press, 1998.
10. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *Proceedings POPL '98*, pages 171–183, San Diego, January 1998. ACM Press.
11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.
12. Grégor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP 2001*, LNCS 2072, pages 327–355, Budapest, Hungary, June 2001. Springer.
13. Markus Lumpe and Jean-Guy Schneider. Classboxes – An Experiment in Modeling Compositional Abstractions using Explicit Contexts. In Mike Barnett, Steve Edwards, Dimitra Giannakopoulou, Gary T. Leavens, and Natasha Sharygina, editors, *Proceedings of ESEC '05 Workshop on Specification and Verification of Component-Based Systems (SAVCBS '05)*, pages 47–54, Lisbon, Portugal, September 2005.
14. James S. Miller and Susann Ragsdale. *The Common Language Infrastructure Annotated Standard*. Microsoft .NET Development Series. Addison-Wesley, 2003.
15. Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Scinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausanne, School of Computer and Communication Sciences, 2004.
16. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1992.
17. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable Units of Behavior. In Luca Cardelli, editor, *Proceedings ECOOP 2003*, LNCS 2743, pages 248–274, Darmstadt, Germany, July 2003. Springer.
18. Charles Smith and Sophia Sophia Drossopoulou. Chai: Traits for Java-Like Languages. In Andrew P. Black, editor, *Proceedings ECOOP 2005*, LNCS 3586, pages 453–478, Glasgow, Scotland, July 2005. Springer.