Runtime Access Control in C# 3.0 Using Extension Methods*

Mihály Biczó, Krisztián Pócza, and Zoltán Porkoláb

Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages
Pázmány Péter sétány 1/C, H-1117 Budapest, Hungary.
mihaly.biczo@t-online.hu, kpocza@kpocza.net, gsd@elte.hu

Abstract. Encapsulation is one of the most important features of object-oriented programming. Reducing the interface where software components can communicate with each other increases software quality, security and decreases development cost. Compile time or runtime visibility and access control checking that support encapsulation is the key part of modern languages and runtime environments. They enforce responsibility separation, implementation and security policies. Most modern programming languages like C++, C# and Java do not have sophisticated access control mechanisms. They employ a subset or combination of public, private, protected, internal, and friend access modifiers. However, this is not true for the Eiffel programming language that defines sophisticated selective access control called selective export. In this paper first we describe the existing access control features of C++, C#, Java, Eiffel and other popular programming languages. After that we show an example where the current access control features of C# (and most object-oriented languages) are insufficient. We introduce a method level access control checking mechanism to C# 3.0 using extension methods. Our method is able to enforce Eiffel-like selective export in runtime. The implementation does not require the modification of the compiler and the caller, only the callee, and introduces minimal syntactic and performance overhead. It can be a practical solution for modular systems where runtime security is important.

1 Introduction

Even before the birth of the object-oriented programming paradigm [5], the necessity of introducing constraints in accessing different resources (members, functions, etc.) in a program has also emerged [7]. These constraints are usually referred to as access control mechanisms, which directly support the basic principles of object oriented programming such as data encapsulation and information hiding. Information hiding means that a module is capable of hiding its implementation behind its public interface that is exposed to clients.

In object-oriented environments the lowest level of modularity is the class. Consequently, existing access control mechanisms operate at the level of resources within a class, namely member variables and methods. Of course, these

* Supported by GVOP-3.2.2.-2004-07-0005/3.0

control mechanisms can be defined at higher (or possibly lower) granularity levels as well according to the notions that the underlying programming environment supports. Consequently, one can speak about package level, assembly level, or even object level access control. Besides class level strategies, these are the usual levels of access control that programming environments directly support. The question might arise: is this enough to handle most of the situations?

An instance of a class often has several interactions with other objects during its lifetime, and each interaction might belong to a different client. A different client could possibly mean two restrictions:

- During the design phase of the software static client scenarios should be established.
- For each client scenario the minimal interface of a class that can accomplish the desired behavior should be set in order to minimize coupling and security risks between different components.

As a consequence, it is important to be able to expose different client profiles and interfaces in each such interaction.

Different programming environments expose very different access control strategies, ranging from simple naming conventions (Python) to refined feature exports (Eiffel)[4]. Regardless of the chosen strategies, environments can be divided into two basic categories based on when and how the evaluation and enforcement of an access control rule happens. While for compiled, statically typed languages it is the compiler that is responsible for enforcing access control rules in compile time, for interpreted, dynamically typed languages this is usually done in runtime by the interpreter.

In this paper we are to compose the two categories on the .NET platform in a C# based environment [3]. The method that will be introduced has the power of Eiffel's feature export mechanism, however, access control checks will be done in runtime. Hopefully, this extends the dynamic nature of Eiffel's method, because theoretically access control can even be adjusted and refined in runtime. The tools shown include the extension methods of C# 3.0 [14], and also reflection.

In Section 2 an overview of existing programming environments and strategies will be summarized. In Section 3 a motivating example will be presented. In Section 4 the high level overview of our solution will be shown. In Section 5 we present a case study in order to show how the solution works for a real example. In Section 6 the performance measures will be published. Some related work can be found in Section 7. Section 8 will provide an overview of current limitations as well as possible extension points and future work.

2 Overview of Access Control Features in Different Programming Languages

In this section we briefly overview existing access control features of different general-purpose programming languages. Besides the analogous mechanisms pro-

vided by C++, Java, and C#, the more sophisticated approach of Eiffel's selective export and other alternatives – methods applied by Smalltalk and Python – will also be mentioned.

The C++ language [13] can be regarded as the ancestor of many modern programming languages like C#, D and Java, therefore we describe the access control features of the C++ language first. Access control constraints can be set at class member level (for both static and instance members). The default level of access is private, which means that the given resource can be accessed only within the given class and within objects of that class. This access level can be changed to public, when all access constraints would be abandoned. Consequently, a class is not visible to the "outside" world by default, but this behavior can be changed. In order to set access control on the members of a class in a class hierarchy, one may want to set protected access level. A protected member can be reached within the class and from derived classes. C++ exposes a special mechanism to control access to class members: it introduces the notion of friend methods and classes. Once a class has accepted friend methods or classes, these friends can access all private and protected members of that particular class (of course along with the public members). However, such a contract between two entities must be marked explicitly, and may lead to the violation of object-oriented principles, such as encapsulation.

As for access between base and derived classes, C++ has three kinds of inheritance: public, private and protected. The difference between the inheritance modes can be seen in Table 1.

Access modifier in the base Inheritance class mode	public	private	protected
public	public	private	protected
private	private	private	private
protected	protected	private	protected

Table 1. Inheritance modes and access modifiers in C++

Members that become public in the derived class can be accessed in the derived class and also from the outside world. Members that become protected can be accessed only in the derived class, while members that become private are hidden in the derived class and cannot be accessed from the outside world.

The Java programming language extends the above mentioned access control features of C++ in a sense, however it does not support different inheritance modes, only public inheritance. The extension is that access level can be set not only for class members, but also for classes as well. There are three class access levels: private (to be used for nested classes), public, and internal. The notion of an internal class contributes to packages in Java: an internal class or class member can be accessed only within the package it was defined in. At class member level, Java has four access modifiers: public, private, protected and package-private (default).

In C# [3], the access control mechanism is very similar to Java's implementation. Likewise Java, C# has also two levels of access control: class and class member level. A class can be public, private and internal (package-private in Java). Public classes are accessible by everybody; private classes can be accessed from the current namespace. Internal (default) classes behave in the same way as package-private classes in Java; they are accessible from the current assembly. At class member level C# has five different access modifiers: public, private, protected, internal and protected internal. Public, private, protected and internal members behave in the same way as in Java. Protected internal members behave as if they were protected and internal at the same time.

The Eiffel programming language [4] has a very different approach to access control than the previously described mechanisms. It is called selective export and allows features (methods) to be exposed to any named class. The default access level of a feature is public. However, an export clause can be defined for any feature which explicitly list classes that are allowed to access the underlying feature. As for inheritance, Eiffel features are always visible from derived classes, so – at least from this point of view – there is no private access level between base and derived classes like there is in C++, C# or Java. In order to clarify the scene, we are going to present an example.

```
class READER feature
    Read ( book : BOOK ) is ... end;
    book : BOOK;
feature { READER, BOOK }
    RentBook ( book : BOOK ) is ... end;
end -- READER

class BOOK feature
    reader : READER;
feature { BOOK, READER }
    SetReader ( reader : READER ) is ... end;
end -- BOOK
```

In both the `READER` and the `BOOK` classes we can find examples of selective export. In the `READER` class access to the `RentBook` method is restricted to the class of `READER` (the defining class) and to the class of `BOOK`. Similarly, in the `BOOK` class, access to the `SetReader` method is limited to the `BOOK` and `READER` classes. This way when associating a reader with a book instance, the link can be built from both directions: the reader can rent a book, or the book can be rented by a reader. In both cases a consistent state can be established. When using feature export, one can employ the `{ANY}` and `{NONE}` export lists after a feature clause. `{ANY}` is just another form of a fully public feature. On the other hand, `{NONE}` is the most restrictive form of a feature export. When it is applied to a feature, that feature will not be exported to any class, not even to the defining class. A feature with a `{NONE}` export list can be accessed from a given instance of a class, and not even from an other instance.

As opposed to the sophisticated solution of Eiffel, there are environments which have only two level of access control. One such example is the Smalltalk language, where all attributes (members) are protected, and all methods are public. Smalltalk does not support the notion of protected methods, if a method is intended to have protected access level, it should be placed in the so called – private protocol – of the defining class.

Another fine example of simple access control is the interpreted, dynamic Python programming language. Interestingly, instead of the traditional public-protected-private triple of access levels, Python provides only a naming mechanism called name mangling. If the name of the feature begins with an underscore, the Python interpreter will mangle its name. All that mechanism provides is that a feature cannot be mistakenly used, it does not explicitly prevent clients from using the underlying feature. Although at first this can be regarded as only a convention that does not provide the strength of the public-protected-private approach, even Stroustrup admits the weakness of existing access control methods in [13]: "The protection of private data relies on restriction of the use of the class member names. It can therefore be circumvented by addressing manipulation and explicit type conversion. But this, of course, is cheating. C++ protects against accident rather than deliberate circumvention (fraud)." The same applies for Python: you can circumvent the mechanism, but this has to be done explicitly.

Likewise Python, Ruby is also fully interpreted therefore access control is determined dynamically during runtime. Otherwise, the access control implementation of Ruby is very close to other popular object oriented languages (public, protected, private); so it is only interesting because the access control checking is done in runtime not in compilation time.

In the next section we will show a motivating example why very fine grained access control can be needed in everyday situations.

3 Motivating example

Consider the C# language example in the next code listing that shows a dummy `Car` class that has 5 different public methods for different purposes from getting the color to setting the owner of the car. The methods are not fully implemented only some dummy implementation is added to the methods to be able to compile it using the C# compiler.

The public methods are the following:

1. `Color GetColor()` is responsible for getting the color of the car.
2. `void AddOil(double amount)` that can be used to fill oil in the oil tank.
3. `double CheckOilLevel()` that returns the oil level in the oil tank.
4. `SetOwner(Person owner)` that sets the owner of the car.
5. `Person GetOwner()` that returns the owner of the car.

When we publish this class to the outside word everybody can reach and call all public methods of the class. It does make sense that everybody can get the

color of the car using the `GetColor` method. However only a car mechanic should be able to check the oil level (`CheckOilLevel`) and add extra oil (`AddOil`) if required. Furthermore, only offices that deal with the registration of cars should be able to call the `SetOwner` method, and only the registration office or a policeman should be able to check the owner of the car using the `GetOwner` method.

```
public class Car
{
    #region  Private fields
    //fields
    #endregion

    #region Misc methods
    public Color GetColor()
    {
        // real implementation goes here
        return default(Color);
    }
    #endregion

    #region Oil methods
    public void AddOil(double amount)
    {
        //real implementation goes here
    }

    public double CheckOil()
    {
        //real implementation goes here
        return default(double);
    }
    #endregion

    #region Owner methods
    public void SetOwner(Person owner)
    {
        //real implementation goes here
    }

    public Person GetOwner()
    {
        //real implementation goes here
        return default(Person);
    }
    #endregion
}
```

There are three ways to enforce the previously mentioned constrains on the `Car` class:

1. Use the built-in features of the programming language we are working with.
2. If the access control features of the current programming language is not eligible then switch to a programming language that has these features.
3. Implement some custom mechanism.

The built-in features of the C# programming language do not provide Eiffel-like selective export; therefore we cannot properly restrict the method access. We could change the access control modifiers of the methods that need special access control features (`CheckOilLevel`, `AddOil`, `SetOwner`, `GetOwner`) to internal and move the classes that implement the registration office and the car mechanic to the same assembly or component. However this solution is not adequate because the registration office still can check and add oil, and the car mechanic still can get and set the owner of the car.

We cannot often change the programming language or runtime environment of our application, so the second solution usually cannot be carried out.

The third solution is the hardest however there is no other way to achieve our aim in C#. When implementing such a solution it is important to leave the caller unchanged and change the callee to the less possible degree. It is also important when one wants torefactor an existing system.

In C# we have two ways to achieve this goal:

1. Use the built-in call interception mechanism of the .NET runtime and check the type of the caller before the call.
2. Use the new language features of C# 3.0.

The built-in call interception mechanism of the .NET runtime is very slow (some thousand calls per second) and the callee has to inherit from the class `ContextBoundObject`, therefore we decided to use C# 3.0 feature called extension methods. C# 3.0 will be presented to the public within a year, now a technology preview version is downloadable [14].

4 High Level Architectural, Component Overview

As we have mentioned previously, our ultimate goal is to extend standard access control strategies with the least possible intrusion using the latest technologies and achievements in the C# 3.0 language specification.

The tools we are going to employ include extension methods, attributes, and reflection.

Extension methods are language enhancements in the C# 3.0 language specification that mimic as if objects could be extended with new methods in runtime. Although they are regular static methods, and can be called as common static methods, they can also be called as if they were pure object methods.

Attributes are elements that allow for adding information to classes and methods in a declarative way. This declarative information is used for various purposes during runtime (in our case it will be used to control access to specific resources).

Attributes add metadata to assemblies, which in turn describe the assembly they belong to. Via a process known as reflection, a program's attributes can be retrieved from its assembly metadata. This concept allows us to extend the language by creating customized declarative syntax with attributes.

After introducing the basics of the tool we are going to apply, we present the high level overview of the framework. In the following figure, the structure can be seen in detail in a UML-like notation. The rectangles are the classes, and the arrows between them denote dependencies. Rectangles with a dashed border represent static classes.

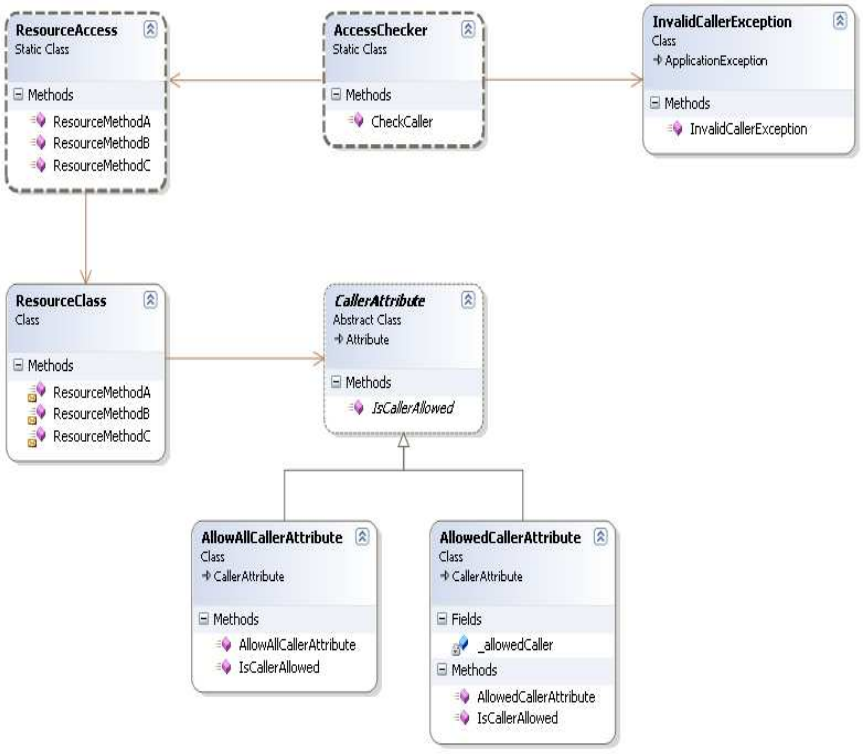

Fig. 1. High level overview

In the above scenario we are to control access on ResourceClass. As it can be seen this class has three methods, each of which should be accessed by a known

set of clients independently of each other. We have previously declared our most important concepts:

1. Allow only minor changes to the called resource and minimize efforts to fit existing structures to this framework
2. The caller (the client who uses the methods of ResourceClass) must not be altered in any way at source level
3. Check for unauthorized access attempts in runtime, and throw an exception if such an attempt is made

In the following we propose the necessary modifications for each point.

1. Requirement 1:
   (a) Methods that should be accessible through this framework should be set to internal access level, in other words should be confined to their defining assembly.
   (b) Each such method should be marked with attributes indicating which types can access the given function.
   (c) For each set of protected resources such as `ResourceClass` a static class should be defined in the same assembly through which resources can be accessed from outer assemblies as well. (In the architecture figure this class is `ResourceAccess`). In `ResourceAccess` one should define extension methods for each such method that is to be accessed through this framework. The body of the extension methods is a simple indirection to the protected method anticipated by a call to the `CheckCaller` method of `AccessChecker`.
2. Requirement 2:
   (a) The caller itself remains unchanged.
   (b) In the source file the namespace where the extension methods are defined has to be included with the using directive.
3. Requirement 3:
   (a) Using reflection, the `AccessChecker` gets the original method and the attributes that control access on it.
   (b) If an unauthorized access attempt is made, a typed exception will be thrown.

The framework is quite simple, but it has many advantages over existing techniques. First of all, it can coexist with legacy access control management techniques, the only place where traditional access control is modified is the called resource (methods should be declared internal), but the restriction is only assembly-level, therefore allowing other clients from the same assembly to access the protected resource in the usual way.

It is just as fine-grained as the selective export features of Eiffel, since for each method a set of types can be marked in a declarative manner using attributes. However, these set of attributes can be extended, all we have to do is to inherit from the `CallerAttribute` class, and implement the arbitrarily complex access checking behavior. This means that access control can be managed even through

a configuration file or web service and can be changed every time the application is started.

In the following section we give a detailed illustrating example how this framework works in practice.

5 Case Study

In this case study we will show the upgraded version of the example presented in Section 3 using the framework presented in Section 4. In the examples the `GetOwner` and `SetOwner` methods will be shown and their appropriate callers will be described. We will examine in detail what kind of changes have to be applied to the callee, what extra components have to be introduced and what (slight) changes are to be made to the caller.

Changes to the callee:

1. The first change is that the `Car` class, which is our 'protected' resource has to be moved to another assembly (Library in our example).
2. The access control modifier of methods we would like to extend with runtime access control features has to be changed from public to internal so that it would only be accessible from the current assembly.
3. The allowed callers have to be annotated using C# attributes. We use the `AllowedCaller` attribute which accepts the caller type name as a string parameter (i.e.: `NamespaceName.TypeName`). (Cross references are not allowed, so the assembly where the caller is implemented cannot be referenced. Therefore, we decided to describe the allowed types by name.)

Consider the upgraded version of the `Car` class:

```
public class Car
{
    //...

    [AllowedCaller("em.RegistrationOffice")]
    internal void SetOwner(Person owner)
    {
        //real implementation goes here
    }

    [AllowedCaller("em.RegistrationOffice")]
    [AllowedCaller("em.Policeman")]
    internal Person GetOwner()
    {
        //real implementation goes here
        return default(Person);
    }
```

```
    //...
}
```

A new accessor class (called `CarAccess` in our example) has to be introduced which contains the extension methods that are responsible for checking if the caller is allowed to call a particular method.

Requirements of the class are the following:

1. Has to be in the same assembly as the upgraded `Car` class.
2. It should contain the extension methods which have to share the same name with the appropriate method in the Car class it extends, should accept all parameters and return the return value of the appropriate method in the `Car` class.
3. Every extension method has to call the runtime access control checker method and call the method in the `Car` class that has the same name as itself.
4. It should be explicitly prohibited to inline the extension methods.

A section of the implementation of the CarAccess class can be seen in the following code fragment:

```
public static class CarAccess
{
     //...

    [MethodImpl(MethodImplOptions.NoInlining)]
    public static void SetOwner(this Car car, Person owner)
    {
        RACInfrastructure.CheckCaller();

        car.SetOwner(owner);
    }
    [MethodImpl(MethodImplOptions.NoInlining)]
    public static Person GetOwner(this Car car)
    {
        RACInfrastructure.CheckCaller();

        return car.GetOwner();
    }
    //...
}
```

The implementation of the `CheckCaller` method will be presented later. It checks if the direct caller is in the list of allowed callers specified by `AllowedCaller` attribute. If it is then it simply returns and allows the appropriate method to be called otherwise throws an `InvalidCaller` exception.

In the caller we have to ensure that methods that call the extension methods will not be inlined by the runtime. Other changes are not required because:

1. If the caller and the callee (`Car`) are in different assemblies then the `internal` methods of the callee cannot be reached. The public methods with the same name in `CarAccess` will be visible and called. The runtime access control checking is performed.
2. If the caller and the callee are in the same assemblies the method in `CarAccess` hide the methods in `Car` with the same name. This scenario works but not suggested because of encapsulation and code separation.

In the following listing we show the implementation of the `RegistrationOffice` class which is allowed to call only the `SetOwner` and `GetOwner` methods of the `Car` class:

```
public class RegistrationOffice
{
    [MethodImpl(MethodImplOptions.NoInlining)]
    public void DoTheRegistration(Car car, Person person)
    {
        car.SetOwner(person);
    }

    [MethodImpl(MethodImplOptions.NoInlining)]
    public Person CheckRegistration(Car car)
    {
        return car.GetOwner();
    }

    //an exception will be thrown because unauthorized
    [MethodImpl(MethodImplOptions.NoInlining)]
    public void AddOil(Car car)
    {
        car.AddOil(42);
    }
}
```

From the point of the caller it looks as if it would directly call the appropriate method of the `Car` class, but behind the scenes it calls the appropriate extension method defined in `CarAccess`.

The `CheckCaller` method called by the `CarAccess` class performs the following steps:

1. Queries the current call trace (the prohibition of method inlining for some methods is introduced because tiny methods are often inlined by the runtime an it would hide them in the call trace).
2. Queries the extension method from the call trace (the first slot in the call trace – e.g. `CarAccess.SetOwner`).
3. Queries the type of the caller of the extension method (the second slot in the call trace - e.g. `RegistrationOffice` or `Policeman`).

4. Using `Reflection`, it determines the method of the callee hidden by the extension method (from the name and the first parameter of the extension method - e.g. `Car.SetOwner`).
5. Using `Reflection`, it detects if the type name of the caller is the parameter of any of the `AllowedCaller` attributes of the method in the callee.
   (a) If true, then returns the `CheckCaller` method.
   (b) Else throws an `InvalidCaller` exception indicating an unauthorized caller.

6 Performance Analysis

We have implemented the Case Study presented in Section 5. In this section the performance of the solution will be measured. Furthermore a caching algorithm will be introduced and analyzed to increase performance.

1. Host machine with a 2.6 Ghz single core Pentium 4 processor running Virtual PC 2007 virtualization environment
2. In the virtual machine there is the March CTP version of Visual Studio " Orcas" [14] installed on a Windows Server 2003 guest Operating System

The performance test intended to measure the pure number of calls / sec. The test application executed together 100 000 valid calls to 4 different methods which took 8.39 seconds. The methods of the callee were empty to measure the pure performance of our solution.

```
int calls = 100000;
Car car = new Car();
Person p = new Person();
RegistrationOffice regOffice = new RegistrationOffice();
CarMechanic carMechanic = new CarMechanic();
DateTime start = DateTime.Now;

for (int i = 0; i < calls/4; i++)
{
    regOffice.DoTheRegistration(car, p);
    regOffice.CheckRegistration(car);
    carMechanic.CheckCar(car);
    carMechanic.AddOil(car);
}
TimeSpan timeDiff = DateTime.Now - start;
Console.WriteLine(string.Format("Time elapsed: {0}", timeDiff));
Console.WriteLine(string.Format("calls/sec: {0}",
                    calls / (timeDiff.TotalMilliseconds/1000)));
```

There are three kinds of static information that can be cached regarding the 5 steps CheckCaller method performs presented in the end of Section 5:

1. In step 4 one extension method calls one method from the callee, the extension method itself determines the method of the callee.
2. In step 5 the list of AllowedCaller attributes that belong to the method of the callee can be cached, because it is static information.
3. In step 5 if a call from a particular type to a particular method was allowed it will be always allowed because the access control information is determined by static attributes. (If the attributes would depend on dynamic information, this caching step could not be accomplished.)

Incorporating this caching mechanism into our solution we executed the same test case. Now it performed the same 100000 calls in 4.6 seconds which means 21707 calls/sec.

We also measured the number of invalid (unauthorized) method calls where an exception is thrown at every method call. 10000 calls were executed without using cache in 1.75 seconds which means an average of 5706 calls/sec, while the same 10000 calls where executed with using cache in 1.19 seconds which means 8391 calls/sec.

The comparison of the performance with and without caching can be seen in Table 2.

	Number of calls	Time taken (sec)	Calls/sec
Authorized calls without cache	100 000	8.39	11918
Authorized calls with cache	100 000	4.6	21707
Unauthorized calls without cache	10 000	1.75	5706
Unauthorized calls with cache	10 000	1.19	8391

Table 2. : Performance comparison

The caching algorithm almost doubled the number of valid method calls for a given period.

Access control checking mechanism can be disabled or enabled dynamically by setting a boolean flag to false or true. When disabled, the overhead of a single method call is two simple method calls (`CheckMethod`) and the extension method) and the evaluation of the value of the boolean flag. The computational overhead is negligible.

7 Related work

Encapsulation in different object-oriented languages and object-oriented software design are widely studied and compared by many authors [1, 5, 8]. In [12] Snyder defined that encapsulation is a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces. By incorporating encapsulation, design patterns in [2] try to loosen up the coupling between connected/independent modules.

It is also important to note that encapsulation is widely used in the industry according to the survey in [13]. The industry level language that has the most sophisticated access control features is the Eiffel programming language [4].

Scharli noted in [9] that some mainstream, industry level object-oriented languages offer a very limited approach of encapsulating methods or almost have no ways to implement encapsulation and access control (e.g. Smalltalk and Python). Moreover access control or visibility checking is performed in very early stage, which leads to inflexible and fragile code. Scharli also noted in [9, 10] that access rights are inseparable from classes (access rights are parts of the implementation of methods), client categories are fixed (users and heirs), and access rights are not customizable (encapsulation decision of the clients are impossible). To overcome these weaknesses, Scharli introduced the concept of Composable Encapsulation Policies and a model where operators and relations over encapsulation policies enable us to express encapsulation policy compositions.

To describe the relationship between a class and its subclasses and overcome the fragile base class problem a unique method was developed [6].

8 Summary

It is clear that access control strategies are of the same age as the object oriented programming paradigm itself. Therefore, it is no surprise that all mainstream languages provide mechanisms for controlling access to sensitive, protected resources.

Interestingly, in spite of the fact that programming languages have seen a continuous evolution, they failed to advance on the access control side, or at least not to that extent that the increasing complexity of systems would have demanded. We usually have the same old public-private-protected triplet as we had decades ago, occasionally completed with modifiers for larger modules.

During the design of Eiffel this situation has been taken into account and a sophisticated method of feature export has been established. In this paper we developed an easy-to-use framework for C# 3.0 that is based on the concept of Eiffel and requires no modifications to be made to either the language or the compiler. Our solution uses the latest technical offerings from C# 3.0 such as extension methods, and the new technologies are composed with existing and proven tools like attributes and reflection.

Furthermore, because of the flexibility of attributes, and because they are evaluated in runtime, our solution is not confined to type checking, it can be extended to handle arbitrarily complex access control strategies. This way the refined version of Eiffel feature exports can be ported to C# 3.0, which is the next release of the popular C# 3.0 language.

We have implemented a pilot system to test the performance of our framework, and concluded that the solution can be applied in production environments because the overhead it causes is acceptable in most situations. However, we have also shown a caching mechanism which can help to increase performance.

Although the pilot implementation served great to verify the concept we have established, there are a couple of limitations. We can currently restrict access to types specified by their names, and cannot combine restrictions. Additional work has to be done by programmers, which means extra efforts. Furthermore, the identity of the caller cannot be checked efficiently.

However, analyzing the limitations may lead us to many extension possibilities. Currently we plan to advance on how these attributes can be combined and composed, and whether it is possible to implement Composable Encapsulation Policies with this framework. It would also be interesting to generate the frame dynamically so that programmers would not need to hand-code the infrastructure.

References

1. W. Al-Ahmed. *Encapsulation in Object-Oriented Programming: Comparing and Evaluation.* In Workshop on Encapsulation and Access Rights in Object-Oriented Design and Programming, WEAR 2003.
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
3. Juval Lövy. *Programming .NET Components.* O'Reilly, 2003.
4. Bertrand Meyer. *Eiffel – the language.* Prentice-Hall, 1991.
5. Bertrand Meyer. *Object-oriented software construction.* Prentice-Hall, 1997.
6. L. Mikhajlov and E. Sekerinski. *A study of the fragile base class problem.* In Proceedings of ECOOP'98, No. 1445 in Lecture Notes in Computer Science, pages 355–383, 1998.
7. H. Morris Jr. *Protection in Programming Languages.* Communications of the ACM Volume 16, Issue 1. pages 15–21, 1973.
8. O. Nierstrasz. *A survey of object-oriented concepts.* In W. Kim and F. Lochovsky, editors, Object-Oriented Concepts, Databases and Applications, pages 3–21. ACM Press and Addison Wesley, Reading, Mass., 1989.
9. N. Scharli, A. P. Black, S. Ducasse, O. Nierstrasz, and Roel Wuyts. *Composable Encapsulation Policies.* Proceedings of European Conference on Object-Oriented Programming (ECOOP'04), LNCS 3086, Springer Verlag, June 2004, pages 26-50.
10. N. Scharli, A. P. Black, S. Ducasse. *Object-oriented Encapsulation for Dynamically Typed Languages.* Proceedings of 18th International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'04), pages 130-149. 2004.
11. M. Skoglund. *A Survey of the Usage of Encapsulation in Object-Oriented Programming.* In Workshop on Encapsulation and Access Rights in Object-Oriented Design and Programming, WEAR 2003.
12. A. Snyder. *Encapsulation and inheritance in object-oriented programming languages.* In Proceedings 1st International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'86), pages 38–45. ACM Press, 1986.
13. B. Stroustrup. *The C++ Programming Language.* 3rd ed., Addison-Wesley, 2004.
14. Visual Studio Code Name "Orcas" Related CTP Downloads. `http://msdn2.microsoft.com/en-us/vstudio/aa700831.aspx`