Fine-Granularity Access Control for Java

M.Sc. dissertation
Submitted by Shay Cohen

Under the supervision of Yishai A. Feldman (IBM)

September, 2009.
Abstract

Eiffel language contains a fine-granularity access control model which Java does not. This study describes the semantics to such a fine-granularity access control in Java and the Jafar framework which enforces them. A custom Java Export annotation is used to define the set of exported classes by using strings and wildcards. Java APT uses the Jafar framework to create AspectJ aspects that enforce the fine-granularity access control. The enforcement is done by AspectJ compile-time declarations which deny compilation when an access is violated, and by using before declarations that assert run-time access is not violated by using reflection.
# Table of Contents

Abstract ........................................................................................................................................... ii  
Table of Contents ......................................................................................................................... iii  
List of Figures ............................................................................................................................... iv  
1. Introduction ............................................................................................................................... 1  
   1.1. Java Access-Level Model and Previous Work ................................................................. 2  
   1.2. A Proposed Fine Grained Access to Java ......................................................................... 7  
   1.3. Annotations and the Annotation Processing Tool ........................................................... 8  
   1.4. Aspect Oriented Programming ......................................................................................... 8  
   1.5. AspectJ ............................................................................................................................ 9  
2. Adding Fine Grained Access to Java ....................................................................................... 12  
   2.1. Semantics of the Fine-Grained Access Level ................................................................. 12  
   2.2. Fine-Granularity Access Control Implementation ........................................................ 15  
      2.2.1 Processing Architecture ........................................................................................... 15  
      2.2.2 Jafar Generated Code ............................................................................................... 17  
      2.2.3 Program Dependent Aspects .................................................................................. 18  
      2.2.4 Program Independent Aspects ................................................................................ 24  
      2.2.5 APT processing ....................................................................................................... 28  
3. Testing ....................................................................................................................................... 32  
   3.1 Runtime Tests ..................................................................................................................... 32  
   3.2 Compile-time Tests ............................................................................................................ 33  
   3.3 File Generation Tests ........................................................................................................ 34  
4. Future Work ............................................................................................................................... 35  
5. Conclusions and Related Work ............................................................................................... 36  
Bibliography .................................................................................................................................. 38
List of Figures

Figure 1.1 - Implementation of Shape in Eiffel ................................................................. 3
Figure 1.2 - Implementation of Canvas in Eiffel ................................................................. 3
Figure 1.3 - Implementation of Shape in C++ ........................................................................ 4
Figure 1.4 - Implementation of Canvas in C++ ....................................................................... 4
Figure 1.5 - Java LinkedList implementation ......................................................................... 5
Figure 1.6 - LinkedList proposed Eiffel implementation ....................................................... 6
Figure 1.7 - Proposed fine-grained implementation for Shape ............................................. 7
Figure 1.8 - Proposed fine-grained implementation for Canvas ........................................... 7
Figure 1.9 - Export annotation ............................................................................................. 8
Figure 1.10 - Compilation error example 1 .......................................................................... 9
Figure 1.11 - Compilation error example 2 .......................................................................... 9
Figure 1.14 - Compilation error example 3 ...................................................................... 10
Figure 1.15 - Using before statement to raise exception ..................................................... 10
Figure 1.16 - Using AspectJ target designator .................................................................. 11
Figure 1.17 - Using AspectJ thisEnclosingJoinpointStaticPart variable ......................... 11
Figure 2.1 - Processing Architecture ................................................................................... 16
Figure 2.2 - Class shape with Export declaration ............................................................... 18
Figure 2.3 - A restriction example ....................................................................................... 19
Figure 2.4 - Class Rectangle ............................................................................................... 20
Figure 2.5 - Generated compile-time declaration for class Rectangle .................................. 20
Figure 2.6 - Generated compile-time declaration that restricts an overriding method .......... 21
Figure 2.7 - An annotation restriction example ................................................................... 22
Figure 2.8 - Class UniversalRectangle ............................................................................. 23
Figure 2.9 - Generated compile time deceleration pattern ................................................. 23
Figure 2.10 - Illegal method reflection restriction 1 ............................................................ 24
Figure 2.11 - Illegal method reflection restriction 2 ............................................................ 25
Figure 2.12 - Method validateInvokeCall implementation ............................................... 26
Figure 2.13 - Double reflection call example ..................................................................... 27
Figure 2.14 - Illegal double reflection restriction ............................................................... 28
Figure 2.15 - Class Line ...................................................................................................... 28
Figure 2.16 - Class ColoredRectangle ............................................................................... 29
Figure 2.17 - Demonstration of AspectJ objects usage ....................................................... 30
Figure 3.1 - Testing element calls using reflection example ............................................. 32
Figure 3.2 - Testing correctness of specific methods example .......................................... 33
1. Introduction

A basic part of the Object Oriented model is the access modifiers each object element has. Java supports four access-level modifiers for methods, fields and constructors which will be referred to henceforth as java elements; from the least restrictive to the most restrictive they are: public, protected, package-level, and private. While the Java design may answer some of the developer needs, it is too coarse for some scenarios.

The reason Java modifiers lack the flexibility required by all applications is that they can't be specific. For example the implementation of a linked list data structure in Java's core classes consists of LinkedList and LinkedItem classes, LinkedItem can be created only by LinkedList but it cannot explicitly express that fact, rather it has to use some more general modifier or to hide LinkedItem from any later use by using textual nesting.

Consider the following scenario in the Java language, in which we have Canvas and Shape objects. A Shape object holds floats x and y representing its location inside the Canvas and introduces the setLocation method to access this values, the Shape object cannot be located in or deviate from the Canvas area, which is determined by the Canvas object width and height. Instead of changing a Shape location by accessing the Shape object setLocation method, the Canvas object introduces the move method that receives as an arguments the Shape to move and the new x and y values, the method verifies the new Shape location does not deviate from the Canvas area prior to calling the setLocation method. Therefore it is required that only the Canvas object will be allowed to access the setLocation method, such a requirement might prove difficult to implement in Java but it has been answered by different languages.

The Eiffel language [1] defines a different access-level model; in particular Eiffel allows the definition of custom levels, where each method or field (collectively called features) can be exported to a given set of classes and therefore resolving our Canvas and Shape implementation problem by restricting any class other than Canvas to access class Shape setLocation method.

The Java access-level model could be enhanced by implementing the Eiffel custom level paradigm, where a finer-granularity access is permissible. The prime requirement from such finer-granularity access is to follow the Java accessing paradigm and in no situation to break the current model but rather to extend it. But as in Java accessing-model, each defined custom level needs to be enforced. Without enforcement, such a defined custom level will be no different than plain documentation.

The proposed custom level will improve program modularity, by reducing the need to define new packages in order to let classes grant each other special privileges by using the package-level modifier. Instead the packages will be created to modulate the program, which will enhance our software design. Moreover the readability of the code will improved when each element will be declared with the classes allowed to access it, rather
than using a package-level or public modifier which will be too permissive and confuse a developer using this code.

The fine-grained access level can also improve security [2], as it enables role-based security, in which different clients are provided with different interfaces. In role-based security, an object is validated for a set of operations only when it is instantiated, removing the need to check the validity of every operation before it is executed. This also prevents the system from calling prohibited operations which could result in a runtime exception after running a job for several days. An example for such a system is the assigning of different interfaces for different user types. Before an interface is assigned, the user is confirmed to be privileged for that set of operations. From that moment on, this user is known to be privileged to execute every operation that interface has published.

To improve the security of Java by using a fine-grained access level, we first need to assert that the introduced fine-grained level cannot be broken. Asserting this has two aspects: the first is to make sure the environment used to enforce restrictions is not over passed and the second is too make sure the restrictions themselves cannot be broken by a loop hole of a scenario which is not answered.

The Jafar (Java Augmentation by Fine-granularity Access Restriction) framework presented here provides the semantics to a fine-grained access-level in Java, and attempts to answer design problems and implementation security issues. The framework objective is to provide users, the possibility to program in Java with a fine-grained access-level, which will improve software design and maintain program security.

1.1. Java Access-Level Model and Previous Work

One of the ways Java can try to solve the Canvas and Shape problem is to group the two in a unique package and use a package-level modifier for the setLocation method. While preventing other packages from changing the x and y values, this does not prevent classes of the same package from calling the setLocation method of Shape and therefore the problem will be only partially solved. Moreover this solution cannot be accepted, since a package can't be developed by a single vendor when Java does not grantee that such a requirement and new "unrelated" classes could be introduced into that same package.

Eiffel lacks the Java package-level that allows us to group different classes, and instead the conceptual relations between classes are defined by the fine-grained access-level. Moreover this fine-grained access-level is argued to be an architectural benefit, as it eliminates the need for a higher modular structure to group classes, resulting in a smaller, less complex language [1].

In Eiffel, an implementation to the Canvas and Shape problem will look as follows:
Figure 1.1 - Implementation of Shape in Eiffel

```
class SHAPE

feature
  x, y: REAL;

feature {CANVAS}
  setLocation(newX, newY: REAL) is
    do
      x := newX;
      y := newY;
    end
end
```

Figure 1.2 - Implementation of Canvas in Eiffel

```
class CANVAS

feature
  width, height: REAL;

  move(movedShape: SHAPE; newX, newY: REAL) is
    do
      -- enforce data integrity here.
      movedShape.setLocation(newX, newY);
    end
end
```

As shown, each feature declares the classes it is being exported to - in our case the feature `setLocation` is exported to the `CANVAS` class and all its subclasses. C++ has a similar mechanism where a class or one of its members can be declared as a `friend`; a class declared as a friend can access the friend member or class of the declaring class [3]. The C++ friend mechanism in contrary to Eiffel selective level access does not allow descendents of the friend class to access the friend member or class, which imposes a problem in the shown example since different types of `Canvas` items can exist and for each of them `Canvas` will be subclassed. In C++ the `Shape` declaration implementation will be:
Figure 1.3 - Implementation of Shape in C++

**Shape.h**

```cpp
class Shape {
    friend void Canvas::setLocation(float newX, float newY);

private:
    float x, y;

public:
    float getX();
    float getY();
};
```

**Shape.cpp**

```cpp
float Shape::getX() {
    return x;
}

float Shape::getY() {
    return y;
}

void Shape::setLocation(float newX, float newY) {
    x = newX;
    y = newY;
}
```

Figure 1.4 - Implementation of Canvas in C++

**Canvas.h**

```cpp
class Canvas
{
public:
    void move(Shape movedShape, float newX, float newY);
};
```

**Canvas.cpp**

```cpp
class Canvas {
    void move(Square movedSquare, float newX, float newY) {
        // enforce data integrity here.
        movedSquare.setLocation(newX, newY);
    }
}
```

Another difference between the languages is that C++ allows the declaration of an entire class as a friend. A declaration of the class as a friend is not supported by Eiffel and has two disadvantages; the first is that such accessing includes private members, which allow changing the object internal state from outside definition of the class. The second is that
classes declared as friends have access to everything in the other class rather than being restricted to the members of interest to them.

One of the methods of Java and C++ to compensate for the lack of fine-grained access export mechanism is to use nesting classes [2]. Nesting classes defines an intimate relationship between classes [4], and when the class is defined as private it allows access to the nested class only via objects of the outer class. Nested classes reserve namespaces and allow the helper classes to reside beside the classes that define them. Java's LinkedList defines class Entry as a nested class, as can be seen in the following example taken from Java's core source code:

```
Figure 1.5 - Java LinkedList implementation
public class LinkedList extends AbstractSequentialList
    implements List, Cloneable, java.io.Serializable
{
    private transient Entry header = new Entry(null, null, null);
    /**
     * Constructs an empty list.
     */
    public LinkedList() {
        header.next = header.previous = header;
    }

    private static class Entry {
        Object element;
        Entry next;
        Entry previous;

        Entry(Object element, Entry next, Entry previous) {
            this.element = element;
            this.next = next;
            this.previous = previous;
        }
    }

    //...
}
```

The shown scheme ensures that Entry cannot be instantiated from outside the LinkedList structure, which is highly beneficial in some scenarios. Although this scheme has several advantages it is also documented as a design solution to ensure that a certain class is the only one that instantiates a class that its creation is to be restricted [5].

In Eiffel, we achieve the same goal of hiding objects within a surrounding structure, by using logical nesting instead of textual nesting [2], which is an inferior mechanism because it enforces the location of the code to be related to the class access. In Eiffel, the LinkedList class will use two separated classes with appropriate access controls, as proposed in the following Eiffel implementation:
Figure 1.6 - LinkedList proposed Eiffel implementation

class LINKED_LIST[T]

feature {NONE}
    header: ENTRY[T]

feature {ANY}

... end

class ENTRY[T]

create {LINKED_LIST}
    make

feature {NONE}

... do

    make is

    ... end

... end

feature {LINKED_LIST}
    next, previous: ENTRY[T]
    element: T

end

The code presented tries to introduce an Eiffel implementation of the Java code; in practice the Eiffel core libraries define a LINKED_LIST that uses class LINKABLE which can be used by more than one class, such that class LINKABLE can be used by both classes CELL and CHAIN, which class LINKED_LIST inherits. Reusing the LINKABLE class in other classes prevents duplicate code, and thus it can be argued that lacking the fine-grained access level mechanism might lead to duplicate code. Duplicate code in a program is discouraged as it is harder to maintain since a new operation added to the class, is needed to be added to all duplicated classes. Adding operation to several classes predisposing the program to bugs because a bug fixed in one class will not necessarily be fixed in all the duplicated classes.

The Java textual nesting mechanism can be sometimes too restrictive [2]. A need might arise to define a new Cursor class that can iterate though the links independent of class LinkedList. If class Entry is defined as nested private class, a Cursor class will not be able to access it, meaning that only class LinkedList will be able to supply an iterator for class Entry and therefore this textual nesting prevents a certain design to be implemented.
To conclude, unlike Java and C++ which provide a limited number of access modifiers, Eiffel provides the programmer with an option to define an unlimited number of access types, allowing various classes-relationships. C++ provides the friend mechanism while Java can use packages and textual nesting together with the access modifiers which result in several mechanisms used instead of one.

1.2. A Proposed Fine Grained Access to Java

A fine grained access-level mechanism for Java will have similar characteristics to the Eiffel access-level, though it can't be identical. The fine grained access-level for Java will have to follow Java's guidelines and concepts rather than copying Eiffel's model. The basic guideline of the proposed access-model will be to keep Java's current access-model and to extend it to contain a fine grained access-level. The proposed syntax will look as follows:

**Figure 1.7 - Proposed fine-grained implementation for Shape**

```java
public class Shape {
    private float x, y;

    public float getX() {
        return x;
    }

    public float getY() {
        return y;
    }

    @Export("Canvas")
    public void setLocation(float newX, float newY) {
        x = newX;
        y = newY;
    }
}
```

**Figure 1.8 - Proposed fine-grained implementation for Canvas**

```java
public class Canvas {
    public void move(movedShape Shape, float newX, float newY) {
        // enforce data integrity here.
        movedShape.setLocation(newX, newY);
    }
}
```

This proposed implementation is an addition to the Java language by introducing a new `Export` annotation; the annotation will be pre-processed before each compilation to create restrictions in an aspect oriented addition to Java called AspectJ.

Since the created restrictions are only additional to the Java language and its base rules still apply, creating a restriction with a private modifier makes no sense because it is
more restrictive than any Export statement. Instead a low access-level Java modifier is needed, such that a compilation of the code will be allowed by the Java compiler regardless of any Export declaration.

The Export annotation receives a string argument, which expresses the set of classes for which access is allowed. The Export annotations will be parsed by using Java's Annotation Processing Tool. The restrictions, limiting the element access, will be expressed by constructing query like expressions in AspectJ, which will generate compile time errors for restricted calls in the program code.

### 1.3. Annotations and the Annotation Processing Tool

As of version 1.5 of the JDK, the Java language supports general purpose annotations which, similarly to Javadoc tags, are additional metadata on written Java code. As such, they do not directly affect the program's semantics, but rather the program's metadata. Unlike Javadoc tags, annotations can be compiled into the class files and can be extracted using reflection.

The following example shows how to add a new annotation named Export, which will be applicable to method, field and constructor declarations and will be processed by the JVM so retrieval by reflection will be possible.

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.FIELD, ElementType.CONSTRUCTOR})
public @interface Export {
    String value();
}
```

APT is the Java Annotation Processing Tool, the APT was provided by JDK version 1.5 and integrated into the javac compiler in version 1.6. Using the APT is done by implementing an interface called AnnotationProcessor. The APT allows examining the annotations of program elements, creating in the process messages regarding errors and warning in the analyzed class files. But most importantly the APT enables us to create and add new files to the program, including text files and Java source files.

### 1.4. Aspect Oriented Programming

Aspect Oriented Programming, also known as AOP, introduces a new methodology in software engineering; the idea behind AOP is to allow the definition of crosscutting elements that encapsulate a concern in one place. The place where those crosscutting elements are stated is called an aspect, and in it you can write advice which is additional behavior that will be applied to the different parts in the code, known as join points.
The program developer writes a query-like statement called a *pointcut* to define exactly where the crosscutting concerns in the program are, and the process of applying the advice into the matched set of concerns defined in the pointcut is called *weaving*. Some languages like Java have several AOP extensions and each can have different syntax and different set of operations that can be woven at the join points.

### 1.5. AspectJ

AspectJ [6, 7] is one of the AOP extensions to the Java language; it has a broad set of supported operations, of which some are essential for the implementation of the fine-grained access level. Earlier versions of AspectJ could only allow weaving of aspects on the code (source or binary) in compile-time by adding pieces of byte code to the compiled classes at the specified join points in the program. Current versions also support load-time weaving by providing a *weaving agent* or providing a *weaving class loader*. Load-time weaving enables the weaving of classes that were not woven and are loaded at runtime. This characteristic is important for some applications, and is essential for maintaining the security requirements in the presented fine-grained access-level, such that its constraints will not be broken.

One of the additions of AspectJ to the Java language is the option to add *compile-time declarations*; such declarations can define compile-time errors in specified pointcuts. The following example shows how to add the compilation error "Shape.setLocation(float, float) cannot be called" in all places where the method signature `setLocation(float, float)` of class Shape is called. The error will be applied to both static and non static implementations of the method that takes exactly two floats as arguments and will not raise an error to an overloading method with a different signature.

**Figure 1.10 - Compilation error example 1**

```
1 declare error : call(void Shape.setLocation(float, float))
2  : "Shape.setLocation(float, float) cannot be called";
```

In the last example, if class Rectangle inherits from class Shape, then calling the method `setLocation(float, float)` (if possible) by using as a target an instance of class Rectangle will also raise an error. Writing a pointcut that will not raise an error when using instance of class Rectangle as a target, can be done by using the "+" and "!" operators, as shown in the following example:

**Figure 1.11 - Compilation error example 2**

```
1 declare error : call(void Shape.setLocation(float, float)) &&
2  !call(void (Shape+ && !Shape).setLocation(float, float))
3  : "Shape setLocation(float, float) cannot be called";
```

The last example uses the "!" operator to define a call that will not be included in the pointcut. In the second line of Figure 1.11 it excludes from the expression in the first line
all the calls to `setLocation(float, float)` from the descendants of `Shape`. This is done by using "Shape+" which includes all Shape descendants including itself and then excluding itself by using "!Shape", which leaves us only with the proper descendants of Shape.

The AspectJ language allows expressing finer crosscutting than the one presented. For example, it can exclude certain classes from raising the exception if the call was made from that class body. The `within` key word allows us to include the exception to classes the call was made from, or to exclude them in case we use "!" operator for negation. As defined by the third line in Figure 1.12:

**Figure 1.12 - Compilation error example 3**

```java
1 declare error : call(void Shape.setLocation(float, float)) &&
2   !call(void (Shape+ && !Shape).setLocation(float, float)) &&
3   !(within(Canvas))
4 : "Shape.setLocation(float, float) cannot be called";
```

A different type of declaration is the `before` statement which allows to execute specified advice before the execution or call of a pointcut. The before statement carries, apart from the obvious, a decisive feature, if you throw an exception in the before advice, the captured operation won't execute. For example the following code will raise an exception, whenever the method `invoke` of the class `Method` is called:

**Figure 1.13 - Using before statement to raise exception**

```java
1 before () throws IllegalAccessException:
2   call(Object java.lang.reflect.Method.invoke(..))
3   { 
4     throw new IllegalAccessException("Cannot run any overload" 
5     + " of method Method.invoke");
6   }
```

Although the `Method.invoke` has no overloads, the double dot - "(..)" is used, which states that all overloads are captured in the join point. In this case the double dot was used simply as an abbreviation eliminating the need to specify the exact method signature. We can also note that similarly to standard Java, if the before statement throws an exception, it must state the exception type in the statement declaration.

When adding advice to a captured join point, in many cases the code needs to know information about the context on which it was executed. Such context can be retrieved by using the pointcut to reference the caller object or the called element, known to be the target. When we use the `target` designator we also ensure that captured join points `target` is of the type stated in the declaration, as can be seen below.
Another way to retrieve the context can be done by using special variables (thisJoinPoint, thisJoinPointStaticPart and thisEnclosingJoinpointStaticPart) that carry information about the captured join points. The following code demonstrates how the caller class is retrieved by using the thisEnclosingJoinpointStaticPart variable.

```java
Figure 1.15 - Using AspectJ thisEnclosingJoinpointStaticPart variable
1  before (java.lang.reflect.Method method)
2       throws  IllegalAccessException:
3       call(Object java.lang.reflect.Method.invoke(..)) &&
4       target(method)
5      {
6           System.out.println("Called method name : " + method.getName());
7           System.out.println("Calling class name : " +
8                thisEnclosingJoinPointStaticPart.getSignature()
9                .getDeclaringType());
10      }
```
2. Adding Fine Grained Access to Java

When approaching the issue of adding a fine grained access to Java, the first thing to do will be to define the new extension semantics. While fine grained access semantics exist for Eiffel, the semantics for Java will have to be Java oriented and follow Java's concepts rather than Eiffel's. After introducing the fine grained access to Java semantics, Jafar will be introduced. Jafar is a framework that implements those semantics by using Java's APT to generate AspectJ declarations that enforce access to Java's elements. AspectJ compile-time declarations are generated to prevent successful compilation when a restricted accessing is done while run-time declarations are generated to enforce restricted access in the program run-time by using reflection.

2.1. Semantics of the Fine-Grained Access Level

The fine-grained access level exposes the Java elements to a limited set of classes; it could be written in conjunction with public, protected and package-level modifiers. Since the fine-grained access level is only an addition to Java language it can only narrow the set of matched classes defined by Java modifiers and obviously will allow access to a subset of the classes allowed by those modifiers. Adding a fine-grained access level to a private modifier will have no practical use and is therefore discouraged.

As already explained, to define a set of classes that a Java element is limited to, a new custom annotation was added called Export. Like every annotation it is prefixed with a "@" sign and is declared before the Java element, in our case it is restricted to field, method and constructor.

The Export annotation receives a single string value that defines the classes allowed to access the current element. The string can contain an unlimited number of classes or sets of classes which are separated by commas. Regardless of the classes allowed to access the current element, the element can be called from anywhere inside that same class, as allowed by the private modifier. This behavior follows Java's concept and keeps Java's private modifier as the most restrictive.

<table>
<thead>
<tr>
<th>Class A Fine-Grained Access Level Notation</th>
<th>Matched Classes</th>
</tr>
</thead>
</table>
| @Export("B, C")
public void foo() | Method foo can be called from anywhere inside class B or class C. |
| @Export("B")
protected void foo() | Method foo can be called from anywhere inside class B which inherits from class A or has the same package as class A. |
| @Export("B")
void foo() | Method foo can be called from anywhere inside class B which is defined within the same package as class A. |
The fine-grained set expressed could define the set of classes allowed to access the Java element but program modularity is better when we limit to the minimum the number of modules a certain module is allowed to communicate with [1]. Therefore it is better to define also the set of packages allowed to access the Java element. The following table demonstrates how exporting to packages is expressed.

Table 2.2 - Fine-grained definitions with packages

<table>
<thead>
<tr>
<th>Class A Fine-Grained Access Level Notation</th>
<th>Matched Classes</th>
</tr>
</thead>
</table>
| @Export("jafar.B")
  public void foo() | Method foo can be called from class B in the jafar package. |
| @Export("jafar.tests.B, C")
  public void foo() | Method foo can be called from class B in the jafar.tests package or class C which is defined within the same package as class A. |

One possible option when writing a fine-grained set will be to enforce that the defined set exists, meaning each class allowed to access the element exists and compiles. This option enables the compilation to confirm that the fine-grained set was defined correctly without programmer errors, but it comes at a great cost, in such an implementation it will not be feasible to add the fine-grained access level if the corresponding class is still to be written, furthermore it opposes the Java current implementation in which you can add a public or protected modifier before an element definition, such a definition also permits future classes to use the defined element. Following this way of thinking dictates that any implementation must support the definition of unwritten class sets.

When writing a fine-grained access-level we do not wish to be confined to a fixed set, but rather use some notation to define an infinite set of classes that will match the group properties of that set. A good solution to this problem will be to use similar notation to the one used in AspectJ, this notation is a pattern based, which allows the definition of an unfixed set of classes. The pattern notation uses wildcards to express the classes a certain element is exported to. The three permissible wildcards are as follows:

- Asterisk 
  "*", denotes any sequence of characters, but does not match the package (or inner-type) separator ".".
- Double period 
  ".
  , denotes any sequence of characters that starts and ends with the package (or inner-type) separator ".".
- The plus sign 
  "+", denotes any subclass of a given type.
Table 2.3 - Fine-grained definitions with wildcards

<table>
<thead>
<tr>
<th>Class A Fine-Grained Access Level Notation</th>
<th>Matched Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@Export(&quot;B+&quot;)</code></td>
<td>Method <code>foo</code> can be called from anywhere inside a class whose name starts with <code>B</code> and is defined within the same package as class <code>A</code>.</td>
</tr>
<tr>
<td><code>public void foo()</code></td>
<td></td>
</tr>
<tr>
<td><code>@Export(&quot;B+&quot;)</code></td>
<td>Method <code>foo</code> can be called from anywhere inside a class <code>B</code> which is defined within the same package as class <code>A</code> or a class that inherits directly or indirectly from type <code>B</code>.</td>
</tr>
<tr>
<td><code>public void foo()</code></td>
<td></td>
</tr>
<tr>
<td><code>@Export(&quot;*B+&quot;)</code></td>
<td>Method <code>foo</code> can be called from anywhere inside a class that is suffixed with <code>B</code> and defined within the same package as class <code>A</code> or a class that inherits or implements directly or indirectly from a type that is suffixed with <code>B</code>.</td>
</tr>
<tr>
<td><code>public void foo()</code></td>
<td></td>
</tr>
<tr>
<td><code>@Export(&quot;jafar..B&quot;)</code></td>
<td>Method <code>foo</code> can be called from class <code>B</code> in the <code>jafar</code> package or any of direct or indirect subpackages.</td>
</tr>
<tr>
<td><code>public void foo()</code></td>
<td></td>
</tr>
</tbody>
</table>

Note that once a class and its descendants are allowed to access a method you cannot restrict inherited classes from accessing that method. For example if `B` inherits `A` and `C` inherits `B`, you cannot export to `A+` which allows `B` to access but restrict access to `C`.

An export can be directed to a class designated with a certain annotation - custom or not. The annotation must have a runtime retention designation.

Table 2.4 - Usage with Java annotations

<table>
<thead>
<tr>
<th>Class A Fine-Grained Access Level Notation</th>
<th>Matched Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@Export(&quot;@Server&quot;)</code></td>
<td>Method <code>foo</code> can be called from anywhere inside a class marked with the <code>@Server</code> annotation which was defined within the same package as class <code>A</code>.</td>
</tr>
<tr>
<td><code>public void foo()</code></td>
<td></td>
</tr>
<tr>
<td><code>@Export(&quot;jafar.B, @test.Server&quot;)</code></td>
<td>Method <code>foo</code> can be called from class <code>B</code> in the <code>jafar</code> package or from anywhere inside a class marked with the <code>@Server</code> annotation which was defined in package <code>test</code>.</td>
</tr>
<tr>
<td><code>public void foo()</code></td>
<td></td>
</tr>
</tbody>
</table>

The presented semantics allows significant flexibility to pinpoint exactly the set of classes allowed to access our field/method/constructor element.

Table 2.5 - Flexibility of fine-grained definitions

<table>
<thead>
<tr>
<th>Class A Fine-Grained Access Level Notation</th>
<th>Matched Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@Export(&quot;jafar.tests.*&quot;)</code></td>
<td>Method <code>foo</code> can be called from anywhere inside a class in the <code>jafar.tests</code> package.</td>
</tr>
<tr>
<td><code>public void foo()</code></td>
<td></td>
</tr>
<tr>
<td><code>@Export(&quot;jafar..*&quot;)</code></td>
<td>Method <code>foo</code> can be called from anywhere inside a class in the <code>jafar</code> package or any of direct or indirect subpackages.</td>
</tr>
<tr>
<td><code>public void foo()</code></td>
<td></td>
</tr>
</tbody>
</table>
2.2. Fine-Granularity Access Control Implementation

2.2.1 Processing Architecture

Before delving into the implementation, we need to understand how the processing of a project that contains a fine-grained access level is done as shown in Figure 2.1. We will assume for simplicity that our project contains four files Canvas.java, Shape.java, Rectangle.java and Square.java. Only Shape contains an export declaration. Rectangle inherits Shape while Square inherits Rectangle.

The project is first processed by using the APT. To process correctly, the APT needs the project source code, the jafar.jar package and the AspectJ aspectjweaver.jar package. The APT will process all the Export annotations in the source files and will produce AspectJ files that reflect the project Export declarations, it will also manipulate AspectJ objects to assert a certain condition is maintained as will be explained in section 2.2.5. In our case, four files will be created, $Shape_Export.aj, $Rectangle_Export.aj, $Square_Export.aj and $Export.aj. $Export.aj is the file containing the general purpose aspect. A file named $Canvas_Export.aj will not be created since Canvas doesn’t contain or override a class with an Export declaration. On the other hand Rectangle and Square inherit from a class that contains Export declarations and therefore corresponding aspects files will be generated.

In the second stage the AspectJ compiler called AJC will create our project bytecode files Canvas.class, Shape.class, Rectangle.class and Square.class. The aspects created by the APT have been woven into the classes, and they will be needed later in case a load time weaving is requested.

The compiled project can be run on a standard JVM that complies with the code version, but to run it we will need the Jafar runtime package jafarRuntime.jar which is a subset of the jafar.jar package and contains only the runtime code needed.
Figure 2.1 - Processing Architecture
AJC compiles the classes and adds the necessary bytecode segments in compiled classes, these bytecode segments will enforce the Export restrictions, such that the program can be run on a standard Java JVM. But in many situations we are forced to use classes that were not compiled together, either by using third party libraries or when classes are being loaded on demand when the program runs (which implies they were not compiled with the program). In such cases, we can still retain the restrictions by using the compiled Jafar aspects kept in the project to weave the restrictions to the new classes. For that the following is needed:

- If the classes are not being loaded in runtime, we can simply recompile them using AspectJ. We can even recompile class files and JAR files by a process called Post-compile weaving.
- In Java 5 JVMTI the javaagent:pathto\aspectjweaver.jar option can be specified to the JVM [8].
- By running the AspectJ executable aj.exe or aj5.exe (for version 5 or later) instead of the standard java.exe.
- By defining a custom class loader, which instantiates a weaver and weaves classes after they are loaded and before defining them in the JVM.

Retaining the restrictions is being done by a process which is called load-time weaving. In this process the classes are being woven when the class loader loads them to the runtime environment, and by weaving them it applies the advice in the compiled aspect classes which we kept in our program.

### 2.2.2 Jafar Generated Code

The Jafar framework's objective is to enforce the semantics of the fine-grained access level in the program. For this purpose Jafar generates AspectJ declarations from the Export statements in the program. There are two types of code generated by Jafar:

1. Program-dependent code – the code is generated for each written export statement. Each constraint will have one or more compile-time declarations generated. The declarations will be placed in a file named $X_Export.aj where X is the original class name.
2. Program-independent code – the code is generated for each program regardless of program-specific fine-grained access-level declarations. These constraints are fixed and are intended to restrict reflection. They will be checked in runtime while the application executes. The code of this constraints will be placed in a file named $Export.aj.

One of the reasons to generate a different aspect file for each java file is to allow easy inspection of the code. But more importantly it is to support incremental building. In incremental build only the Java file being compiled will be generated with a new or updated aspect file. This is also the reasoning behind putting all the statically generated code in the same file.
2.2.3 Program Dependent Aspects

The dynamically generated code is created according to the Export tags added to the code by the user. A new aspect file is created for each class file. For each Export tag of a single element, a single or multiple compile-time declaration is generated depending on the class subclasses. Let's review our class Shape, but this time with two introduced methods and an updated Export, before explaining the generated compile-time declaration.

Figure 2.2 - Class shape with Export declaration

```java
package drawing2D;

import jafar.runtime.Export;

public abstract class Shape {
    private boolean enabled;
    protected float x, y;

    public boolean getEnabled() {
        return enabled;
    }

    public float getX() {
        return x;
    }

    public float getY() {
        return y;
    }

    @Export("drawing2D.Canvas")
    public void setEnabled(boolean enabled) {
        this.enabled = enabled;
    }

    @Export("drawing2D.Canvas")
    public void setLocation(float newX, float newY) {
        x = newX;
        y = newY;
    }

    @Export("drawing2D.MonoscaleCanvasPrinter")
    public abstract void print();

    @Export("@Painter")
    public abstract void paint();
}
```

Although the example shows only method cases, fields and constructors will have similar declarations created for them. The example shows a restriction to call the method setEnabled in class Shape of package drawing2D only from class Canvas of package drawing2D. The following compile-time declaration will be generated for this declaration.
Figure 2.3 - A restriction example

```java
/**
 * Adding a declaration of error for exporting method [setEnabled]
 * in class [drawing2D.Shape].
 */

declare error :
call (void drawing2D.Shape.setEnabled(boolean)) &&
!call (void (drawing2D.Shape+ &&!drawing2D.Shape)
.setEnabled(boolean)) &&
!(within(drawing2D.Shape+) || within(drawing2D.Canvas)) :
"Access-level violation for void " +
drawing2D.Shape.setEnabled(boolean);
```

One of the advantages of the generated aspect is that it contains compile-time declarations, so in case access is violated an error will be raised on compilation, like any other Java compiler error.

In the example we do not use the static keyword in the declaration, therefore the signature will include both static and non-static methods. But it does not matter if the method setEnabled was declared as static or not, since only one method setEnabled with a single parameter boolean can exist in class Shape (static or non-static), and the signature will match only our intended method.

From the sixth line of the compile-time declaration in Figure 2.3 we can see that a compilation error will be raised on each call to the setEnabled method with the same signature pattern excluding several scenarios, calling to setEnabled in a subclass of drawing2D.Shape, calling from inside drawing2D.Shape or one of its subclasses or calling from the exported class drawing2D.Canvas.

The seventh and eighth lines in the compile-time declaration in Figure 2.3 makes a crucial limitation on the pointcut of the declaration; it limits the pointcut to the class where the method setEnabled introduced, in our example, the class drawing2D.Shape. The following class is a subclass of drawing2D.Shape and is not constrained by the above compile-time declaration:
package drawing2D;

import jafar.runtime.Export;

public class Rectangle extends Shape {

    @Export("drawing2D.Canvas, drawing2D.Grid")
    public void setLocation(float newX, float newY) {
        super.setLocation(newX, newY);
    }

    @Export("drawing2D.MonoscaleCanvasPrinter")
    public void print() {
        // printing...
    }
}

As can be seen from the code above (Figure 2.4), class Rectangle does not redefine the method setEnabled which was introduced in class Shape. The generated compile-time declaration of setEnabled in the Shape aspect does not restrict any access to setEnabled made by a Rectangle object. So for class Rectangle we will have to generate a different declaration with a different pointcut. The generated declaration will be as follows:

**Figure 2.5 - Generated compile-time declaration for class Rectangle**

```
01 /**
02 * Adding a declaration of error for exporting method [setEnabled]
03 * in class [drawing2D.Rectangle].
04 */
05 declare error :
06    call(void drawing2D.Rectangle.setEnabled(boolean)) &&
07    !call(void (drawing2D.Rectangle+ &&!drawing2D.Rectangle)
08      .setEnabled(boolean)) &&
09    !(within(drawing2D.Shape+) || within(drawing2D.Canvas)) :
10   "Access-level violation for void " +
11   "drawing2D.Rectangle.setEnabled(boolean)";
```

The presented compile-time declaration will be located in the same aspect as any other declarations of class Rectangle. The declaration's pointcut allows calls to the setEnabled method by using an instance of class Rectangle. The differences between the two compile-time declarations presented so far are minimal and are marked in bold in the declaration above. Both declarations create the same restrictions, but while the first declaration restricts the calls to setEnabled only in class Shape as stated in line seven of Figure 2.3 (drawing2D.Shape+ &&!drawing2D.Shape), the second restricts the calls to setEnabled only in class Rectangle as stated in line seven of Figure 2.5 (drawing2D.Rectangle+ &&!drawing2D.Rectangle). Unlike a method, a constructor is not inherited, and therefore a Compile-time declaration will not be created in the aspect associated with one of its subclasses.

The immediate question raised is why do we need to create two declarations (or more) for the same method? The reason for creating two or more declarations can be explained
using the example of method setLocation. In this example, the Export declaration in class Rectangle is different from the one in class Shape. In fact, class Rectangle allows both drawing2D.Canvas and drawing2D.Grid to call the method while class Shape allows only calls from drawing2D.Canvas. Apparently class drawing2D.Grid uses class Rectangle internally and therefore needs to access the setLocation method as well. By writing a compile-time declaration that restricts access to a class method, but does not restrict the overriding methods, it is possible to widen an overriding method visibility in a new compile-time declaration. The generated compile-time declaration that restricts the overriding method setLocation in class Rectangle, will be as follows:

Figure 2.6 - Generated compile-time declaration that restricts an overriding method

```java
/**
 * Adding a declaration of error for exporting method [setLocation]
 * in class [drawing2D.Rectangle]
 */
decclare error :
call(void drawing2D.Rectangle.
.setLocation(float, float)) &&
!call(void (drawing2D.Rectangle+ &&!drawing2D.Rectangle)
.setLocation(float, float)) &&
!(within(drawing2D.Shape+) || within(drawing2D.Canvas))
|| within(drawing2D.Grid)) :
"Access-level violation for void " +
"drawing2D.Rectangle.
.setLocation(float, float)"
```

In this scenario the Java concept that allows widening the visibility of an element as we go down the object hierarchy is followed. In Java, if a method is declared protected in a certain class, it can be overridden with a public modifier in an extending class. The setLocation example demonstrates the same scenario, implemented in the fine-grained access level. On class Shape method setLocation is exported to class drawing2D.Canvas, but on class Rectangle the method exported classes are drawing2D.Canvas and drawing2D.Grid, and therefore the visibility is widened.

It can be proved helpful to widen the export declaration of a method, but it is also possible to allow the developer of the caller class to decide whether his class answer the right criteria to use that class. If it does he can mark his class with a corresponding annotation which the java element is exported to. The method paint of class Shape in Figure 2.2 can be called by every class annotated as a @Painter, the generated compile time declaration will be as follows:
Figure 2.7 - An annotation restriction example

```java
/**
 * Adding a declaration of error for exporting method [paint]
 * in class [drawing2D.Shape].
 */
declare error :
call(void drawing2D.Shape.paint()) &&
!call(void (drawing2D.Shape+ &&!drawing2D.Shape)
.paint()) &&
!(within(drawing2D.Shape+) || within(@Painter *)) :
"Access-level violation for void " +
"drawing2D.Shape.paint();"
```

In all of the generated compile-time declarations examples (Figure 2.3, Figure 2.5, Figure 2.6 and Figure 2.7) line nine contains the expression representing the classes the element is exposed to. However, line nine also adds the expression !(within(drawing2D.Shape+). This expression is needed to allow the class that first introduced the element and all of its subclasses to access the element as well. What this practically means is that every element exposed in the fine-grained access level can also be accessed in a similar way to the protected modifier.

The element exposed in the expression !(within(drawing2D.Shape+) has similar access to the protected modifier because we follow Java concepts. These include:

1. An element of a class must be assignable and called by the class.
2. A fine-granularity access element must allow its subclasses to access it, because by definition, a method or a constructor can be overridden by a subclass which can widen the visibility and change its implementation or call the super directive. It's true a variable cannot be overridden, and therefore this argument does not hold on him, but there is no real motivation to exempt him and create a different behavior just for him.
3. A fine-granularity access element should not allow every class in the package to access it, since in many cases this access should be restricted. Moreover, one of the reasons to introduce the fine-grained access-level was to separate between the need to put a class in a certain package, from the need to restrict the access to this class. In any case, the class's package can be added explicitly by using the Export declaration.

These principles lead us to the conclusion that a fine-granularity access element must have access similar to the protected modifier, since it must allow its subclasses to access it.

If we examine the generated compile-time declarations that are created in the Shape and Rectangle associated aspect files (Figure 2.3 and Figure 2.5 respectively), and look for the differences in the Rectangle associated aspect file, we will be able to see that the name of class Shape was replaced with class Rectangle on all occurrences except line nine in Figure 2.3. In fact line nine is the same on both declarations, enabling method setEnabled to be called from class Shape, even if it is being called from an instance of class Rectangle. This behavior follows the Java language which allows such a call.

22
Since Jafar supports the widening of fine-granularity access elements, one of the points needed to be answered, is the case in which a method with an Export declaration, is being overridden and no Export declaration is written for the overriding method. If such a case is allowed, the access to the element will be widened so that every class will be allowed to access it, similarly to declaring an element with the public modifier.

For such access widening no compile-time declaration will be generated, simply because compile-time declarations restrict class access from other classes, while in this case we do not wish to impose any restriction whatsoever. As explained earlier, the restrictions made by the class parent on the override element are generated for that class alone, to allow future widening and therefore will not restrict this element. The following example shows this scenario, where the method setLocation has no Export declaration:

```java
package drawing2D;

public class UniversalRectangle extends Rectangle {
    public void setLocation(float newX, float newY) {
        super.setLocation(newX, newY);
    }
}
```

As shown, the compile time generated decelerations have a pattern, consisting of fixed code that stays exactly the same for each declaration and versatile code that depends on the export declaration and the element being exported. The pattern for the method element can be shown below:

```java
/** * Adding a declaration of error for exporting method [methodName] * in class [className]. */
declare error : call (returns className.methodName(pTypes)) &&
    !call (returns (className+ && !className).methodName(pTypes)) &&
    !(within (methodDeclarationTopLevelClassName+)) ||
    within (exportSet*) ||
    within (annotatedExportSet*)
: "Access-level violation for returns className.methodName(pTypes);"
```

The variables methodName, className and the returns type refer respectively to the method name, class name and return value of the method the Export declaration that was stated. The pTypes are the method parameter types which are part of the method signature and the methodDeclarationTopLevelClassName is the class name of the class that first introduced the Export declaration for this method signature in the class hierarchy. The exportSet is a single set the method is exported to, as defined in the Export declaration. The annotatedExportSet is a single set of annotations the method is exported to, as defined in the Export declaration. In case there are several sets the method is exported to, the underlined part of the pattern will repeat itself for each declared set.
2.2.4 Program Independent Aspects

The program independent code generated consists of four types of constraints:

1. "Illegal Method reflection usage"
2. "Illegal Constructor reflection usage"
3. "Illegal Field reflection usage"
4. "Illegal double reflection"

The presented constraints are applied only in runtime and therefore do not force the programmer, as he writes the program, to maintain the defined access-level. But while the constraints are useless in the aspect of software engineering, they are essential to maintain the program security.

The purpose of the program independent code generated is to close the security loophole of calling restricted elements by using reflection. Without it, the created restrictions are mere recommendations rather than constraints. This code truly ensures that an element will not be called without the proper permission, thus maintaining program access-level security. This, while satisfying the requirement to follow Java principles as well as maintain application security.

Since the element being used by reflection can be decided in runtime, the constraint must be enforced in runtime as well. Therefore, all four constraints generate overhead whenever the program uses reflection. This overhead is the validation that the element can be used without breaking the fine-grained access level restrictions.

The first three constraints prohibit the program from calling an exported element from a restricted class by using reflection. The exact pointcut and advice for each of the three, though distinct, it is very similar. A proposed crosscutting concern for the "Illegal Method reflection usage" can be as follows:

**Figure 2.10 - Illegal method reflection restriction 1**

```java
/**
 * Illegal method reflection restriction.
 */
before(Object caller, Method method)
  throws IllegalAccessException:
    call (Object java.lang.reflect.Method.invoke(..)) &&
    this(caller) && target(method)
{
  ValidationUtils.validateInvokeCall(caller, method);
}
```

In the method ValidationUtils.validateInvokeCall a runtime exception is raised whenever the code violates the fine-grained access-level restrictions. This behavior is similar to Java, where calling a method by reflection can only be validated in runtime. It also implies that, for each method call, validation is done by using reflection, adding some inevitable overhead. Unfortunately, this overhead is worsened by the fact that the proposed pointcut traps only calls from objects and not static calls. In a static call we
don’t have a caller and therefore a different pointcut is needed. This pointcut will have to use a special runtime variable like thisEnclosingJoinPointStaticPart which forces yet a bigger overhead over the concern (but still less than using non static information) [6]. This constraint can be expressed as follows:

Figure 2.11 - Illegal method reflection restriction 2

```java
/**
 * Illegal method reflection restriction.
 */
before(Method method)
    throws IllegalAccessException:
    call (Object java.lang.reflect.Method.invoke(..)) && target (method)
{
    Class<?> callerClass =
        thisEnclosingJoinPointStaticPart.
            getSignature().getDeclaringType();
    ValidationUtils.validateInvokeCall(callerClass, method);
}
```

The validation inside the validateInvokeCall method is done by extracting the Export declaration from the target method at run-time and asserting that callerClass can call it. The validateInvokeCall method makes use of the PointcutParser provided in AspectJ to parse the Export declaration. To use the PointcutParser the sections separated by a comma in the Export declaration are separated and wrapped in a within AspectJ statement. The PointcutParser parses each constructed within statement, and returns a PointcutExpression instance. The PointcutExpression instance returned is used to evaluate whether the expression matched joinpoints in the given callerClass. The method implementation can be seen in Figure 2.12.

The hasExportedAnnotation checks that if the class has an allowed annotation no error will be raised. This check will return a valid answer only if the annotation has a runtime designation, other false will be returned since the callerClass annotation will not be returned by a reflection in runtime.

The introduced PointcutParser and PointcutExpression are part of the org.aspectj.weaver.tools package, provided by the aspectjweaver.jar which is part of AspectJ. They are the same classes that AspectJ itself uses to parse and evaluate when the AJC is compiling code or when performing load time weaving. Using them in our process to parse and evaluate the code has its pros and cons.

By using AspectJ core tools, a highly validated code is gained, saving valuable code writing time. This also ensures that future fixes and features will be automatically integrated in Jafar, including more advanced syntax for the expression of sets of classes.
public static void validateInvokeCall(Class callerClass, Method method)
throws IllegalAccessException {
    Export export = method.getAnnotation(Export.class);

    if (export == null) {
        return;
    }

    Collection<String> exportedTo = ExportUtils.getExportedTo(export.value());

    // preparing the PointcutParser.
    PointcutParser pointcutParser = PointcutParser.getPointcutParserSupportingAllPrimitivesAndUsingContextClassLoaderForResolution();
    Properties properties = new Properties();
    properties.setProperty("invalidAbsoluteTypeName", "ignore");
    pointcutParser.setLintProperties(properties);

    boolean matchedJoinPoint = false;
    for (String s: exportedTo) {
        // Supporting annotations Export.
        if (s.startsWith("@")) {
            if (hasExportedAnnotation(callerClass, s)) return;
            continue;
        }

        // checking if the caller is allowed by our export expression.
        s = getClassWithPackageName(callerClass, s);
        PointcutExpression pointcutExpression = pointcutParser.parsePointcutExpression("within(" + s + ")");
        boolean matchedJoinPoint = pointcutExpression.couldMatchJoinPointsInType(callerClass);
        if (matchedJoinPoint) return;
    }

    // if the caller is not allowed then throwing an exception.
    String methodName = method.getName();
    String className = method.getDeclaringClass().getName();
    String returnName = method.getReturnType().getName();
    String parametersNames = ExportUtils.getParametersTypes(method.getParameterTypes());

    throw new IllegalAccessException(
            "Access-level violation for " + returnName + " " + className + "." + methodName + "(" + parametersNames + ")");
}

Nevertheless, there are some disadvantages to use the AspectJ classes; they need to be
included in the project used libraries in addition to the Jafar runtime package. The
AspectJ classes can also be changed in the future, forcing the Jafar framework to be
changed as well to accommodate the changes, assuming that it will still be possible to use
them. In addition, use of the AspectJ parser ties Jafar semantics to AspectJ semantics,
preventing changing the fine-grained access-level semantics should it be needed.
The purpose of the fourth constraint is to enforce that a restriction is not broken by using double reflection, meaning the reflection methods will be called by a reflection, as the example below shows:

Figure 2.13 - Double reflection call example

```java
public void doubleReflectionCall() {

    // Creating a new ExportExample instance.
    ExportExample e1 = new ExportExample();
    Class<? extends ExportExample> e2 = e1.getClass();

    try {
        Method exportToA = e2.getMethod("exportToA");
        Class<? extends Method> method = exportToA.getClass();

        // Creating the signature argument for the invoke method.
        Class<?>[] invokeArgs =
                new Class[]{Object.class, Object[].class};

        // Creating an instance of the invoke Method.
        // The invokeArgs argument represent the needed method
        // signature.
        Method invokeM = method.getMethod("invoke", invokeArgs);

        // Calling the "invoke" Method instance invokeM
        // with the invoke method - hence the double reflection.
        // The exportToA argument is given as the method that
        // will be called eventually by invokeM.
        invokeM.invoke(exportToA,
                new Object[] {e1, new Object[] {}});
    } catch(Exception e) {
        
    }
}
```

As shown above, the method invoke is called by reflection as well, creating a double reflection. In such a case the callerClass will hook to the Method class and will not be able to hook the caller of the Method class. Therefore, it will not be possible to ensure a class is allowed to call the specified element. A reasonable solution to this scenario is to prevent any such double reflection calls. Restricting double reflection will block even legitimate reflection calls, which are valid in Java, but do not have any practical use and are therefore discouraged in any case, as will be explained later.

Restricting double reflection is very similar to the method restriction; the main difference is in the method ValidationUtils.validateDoubleReflectionCall that validates the method. This method validates that the calling method is not the invoke method of class Method, a set type method of class Field or a newInstance of class Constructor. This concern will be written as follows:
**Illegal double reflection restriction**

```java
/**
 * Illegal double reflection restriction.
 */
before (Method method)
throws IllegalAccessException :
call(Object java.lang.reflect.Method.invoke(..)) &&
target
{ValidationUtils.validateDoubleReflectionCall(method);
}
```

It is important to emphasize that blocking legitimate double reflection calls was not part of our aim but rather a byproduct of missing runtime information. It has the disadvantage that previous applications working with Jafar might raise an error in runtime if they have a double reflection call, therefore checking they do not exists might prove to be a beneficial practice.

While blocking legitimate double reflection calls has a disadvantage, it is unlikely that this will be a significant factor for most given programs. In fact, though it can be used, there is no reasonable motivation or advantage in using double reflection. Moreover, this restriction is used only to enforce security requirements, not to create software design advantages.

### 2.2.5 APT processing

The main purpose of the APT is to generate AspectJ aspects that restrict accessing of Java elements. However, it also fulfills the important part of asserting that the Export annotations created by the user are valid.

Consider the following class:

**Figure 2.15 - Class Line**

```java
public class Line extends Shape {
    @Export("drawing2D.LinesCanvas") // illegal export.
    public boolean getEnabled() {
        return super.getEnabled();
    }

    @Export("drawing2D.LinesCanvas") // illegal export.
    public void setLocation(float newX, float newY) {
        super.setLocation(newX, newY);
    }
}
```

While Java allows widening the visibility of an element, it restricts its lowering. Therefore, an element in class Line which is inherited from class Shape cannot lower the element’s visibility; in such a case an error will be raised claiming that a class lowers an inherited element visibility.
In class Line the method getEnabled lowers the visibility of the method getEnabled in class Shape, since the method in class Shape did not have an Export declaration and is now restricted to class drawing2D.LinesCanvas. The method setLocation had Export to drawing2D.Canvas in class Shape and now has Export declaration to drawing2D.LinesCanvas, resulting in lowered visibility as well. An element in Line lowers the visibility in Shape in two possibilities:

1. The Shape element exports to a class that the Line element does not.
2. The Shape element does not export at all while the Line element does export to some set of classes.

One could argue that setLocation in Line does not lower the visibility of the method in Shape, because drawing2D.Canvas can be implicitly added to the Export in class Line by allowing previous declarations to be added to the current class. The Jafar implementation could have supported either way that was decided. It was decided not be allowed since Java principals are followed, requiring the programmer to restate the modifier of an element in every subclass as well. So in each Export declaration, the previous one must be restated as well.

Restating in each class the set of allowed classes in the Export declaration can be tedious, but it is also convenient when trying to discern the classes a certain method can be accessed by. In Eiffel where a method does not need to restate its accessibility a tool was provided to view the accessibility of class methods.

Another issue to be regarded when writing an Export declaration is whether the class in the export needs to exist before the Export declaration is written for such a declaration to compile correctly. It is only logical that if writing of patterns is allowed to express an unclosed set of Export classes, then classes that do not exist can be written in the Export declaration. And therefore when writing an Export declaration, no validation is made on the classes stated in the Export. This raises another issue; consider the following class:

```java
package drawing2D;

import jafar.runtime.Export;

public class ColoredRectangle extends Rectangle {

    // Color properties.
    @Export("CanvasPrinter+, MonoscaleCanvasPrinter")
    public void print() {
        // printing...
    }
}
```

The presented class above, adds color properties to class Rectangle. In this class the exporting to class CanvasPrinter and all of its decedents, marked as @Export("CanvasPrinter+"), could theoretically contain every explicit exported class...
name, like class `MonoscaleCanvasPrinter`, if it inherits from class `CanvasPrinter`. But as shown in our example of classes `Rectangle` and `ColoredRectangle`, method `print` has to declare `@Export("CanvasPrinter+, MonoscaleCanvasPrinter")` and not suffice with `@Export("CanvasPrinter+")` or `@Export("@CanvasPrinter")` in a similar example. Otherwise an error will be shown, stating we lower the visibility. This is done because it is already stated that the existence of classes will not be validated. There could not be an exception in this case, such as to follow the inheritance of class `CanvasPrinter` to make sure `MonoscaleCanvasPrinter` is one of its direct or indirect decedents. So in such cases the user is forced to export class `MonoscaleCanvasPrinter` explicitly.

The process of asserting that we do not lower the visibility is not done by using AspectJ aspects, but rather the APT. The APT allows generating compile-time errors while processing the files; we use this mechanism to assert that no lowering of visibility is done. For that the method `isExportAnnotationsValid` is called on each method with an `Export` declaration, this method goes up the class hierarchy and looks for methods with the same signature. If a method is found, it must have an `Export` declaration or an error will be raised stating that "Cannot reduce the visibility of the inherited method from " + typeDeclaration where `typeDeclaration` is the parent type. If the found method has an `Export` declaration, the method `ProcessingUtils.contains` is called to assert that current checked method `Export` declaration contains the set of classes that exists in the parent `Export` declaration, i.e. is a superset of any `Export` declarations it inherits.

The method `ProcessingUtils.contains` itself manipulates AspectJ objects such as `PatternParser`, `Pointcut` and `NamePattern`. Therefore the AspectJ package `aspectjweaver.jar` is needed for the APT processing as shown in section 2.2.1. The method has several overloads and calls other auxiliary methods, so instead of presenting it, the following code will demonstrate how AspectJ objects can be used:

```
Figure 2.17 - Demonstration of AspectJ objects usage
string matcher = "drawing2D.*";
string matched = "drawing2D.Canvas";

// Creating matcher NamePattern.
PatternParser parser1 = new PatternParser("(within(" + matcher + "))");
Pointcut pointcut1 = parser1.parsePointcut();
WithinPointcut withinPointcut1 = (WithinPointcut)pointcut1;
WildTypePattern wildTypePattern1 = (WildTypePattern)withinPointcut1.getTypePattern();
NamePattern[] namePatterns1 = wildTypePattern1.getNamePatterns();
bool result = matches(namePatterns1, matched);
```

The value of `result` will be true since `matcher` contains `matched`. The pros and cons for using AspectJ objects were already debated at section 2.2.4.
The reason the APT was used instead of the AspectJ that was used so far to assert no element visibility is lowered is because that AspectJ cannot perform the necessary validation on compile-time. AspectJ lacks the ability to inspect a java element annotation in the pointcut of a compile-time declaration. Instead it can inspect a java element annotation in the advice of a declaration, but only run-time declarations have a capability to add an advice. Throwing an error at run-time instead of compile-time misses the whole point of raising the error and does not follow Java's current behavior of preventing compilation when an element visibility is lowered.

Using the APT to only assert that no element visibility is lowered raises the question of why not use the APT instead of AspectJ to compile-time declaration? The answer is that the APT allows analyzing code annotations, but not the code inside the methods to determine if a restricted call is done. Moreover, even if the restrictions could have been added upon compilation, the problem of loading compiled classes that were not compiled with the rest of the program at run-time will remain. With AspectJ this can be solved by using load-time weaving as explained in section 2.2.1.

So if the APT is used to assert that no element visibility is lowered, compiled classes loaded at run-time will not be checked. If a class loaded at runtime, inherits one of the program classes and it contains an Export declaration, then the APT should have been used to process the classes otherwise no aspects will be generate as well. On the other hand if the class is a 3rd party class that was not compiled with the rest of the program, it cannot inherit one of the program classes and therefore cannot lower its visibility. So, even though the classes were compiled separately, as long as they were compiled with the APT (if they had Export annotations) there will be no violation.

As previously explained, the method isExportAnnotationsValid goes up the hierarchy of a class by calling itself recursively to assert no element visibility is lowered. The method that generates program dependent aspects also needs to go up the hierarchy of a class in order to define the generated compile-time declaration pointcut as explained in section 2.2.3. So when the APT processes the program it needs to recognize class parent types. However, it does not need to recognize class subclasses when processing. This is due to the design of the generated compile-time declaration that enforces a restriction only on the current class. Different implementations could have enforced the restriction on the class subclasses as well by using a simpler pointcut, but will have demanded a class to recognize its subclasses. It was required that a class will not recognize its subclasses to keep the APT mechanism similar to Java compiler and to support incremental builds.

Requiring a class to know its parent types also means that if the parent type Export declaration was changed or removed, the class may no longer be valid. This is because the element visibility was lowered or the restriction is no longer in place. Thus, whenever a type is changed, its subtype declarations also need to be validated. This is not unlike the current behavior of Java complier and thus it can be concluded that the APT fine grained access-level processing corresponds with Java's compilation mechanism and therefore can also support incremental builds.
3. Testing

Adding the fine-grained access level layer needed to be very precise, as every subtle change in the generated aspects could affect the correctness of the fine-grained access level declaration. The tests we made can be categorized into 3 types:

1. Runtime tests
2. Compile-time tests
3. Files generation tests

3.1 Runtime Tests

In order to do run-time tests, a class that implements Junit was created. Junit is a Java framework for conducting tests. On this framework a sample of Export statements were checked. This included various test of element calls using reflection. It was checked that the expected errors were raised while others errors were not. An example for such tests can be seen below:

Figure 3.1 - Testing element calls using reflection example

```
A a;
a = new A();
Class<? extends A> al = a.getClass();

// An IllegalAccessException should be raised
// (caused by the generated aspect).
try {
    Method foo = al.getMethod("foo", int.class);
    foo.invoke(a, new Object[] {new Integer(5)});
    fail("Expected IllegalAccessException");
} catch(IllegalAccessException iae) {
    assertTrue(iae.getMessage().startsWith("Access-level violation for"));
}
```

As shown, the test checks that an error of type IllegalAccessException is raised, and then further checks that the error contains the correct message prefix. Other tests also checked that no unnecessary errors are raised.

Another type of runtime tests were tests that checked the correctness of specific methods, the most important of those being the boolean contains(Collection<String> matchers, Collection<String> matched) method that returns if a certain collection is the subset of the other. An example of these tests can be seen bellow:
Figure 3.2 - Testing correctness of specific methods example

```java
@Test
public final void testProcessingUtilsContains() {
    Collection<String> c1, c2;
    c1 = ExportUtils.getExportedTo("C*, D");
    c2 = ExportUtils.getExportedTo("C, D");
    assertTrue("Patterns need to be contained",
                ProcessingUtils.contains(c1, c2));

    c1 = ExportUtils.getExportedTo("c.i.b.*");
    c2 = ExportUtils.getExportedTo("c.i.b.C, c.i.b.D");
    assertTrue("Patterns need to be contained",
                ProcessingUtils.contains(c1, c2));

    c1 = ExportUtils.getExportedTo("c.i.b.*+");
    c2 = ExportUtils.getExportedTo("c.i.b.C, c.i.b.D");
    assertTrue("Patterns need to be contained",
                ProcessingUtils.contains(c1, c2));

    c1 = ExportUtils.getExportedTo("c.i.b.C+");
    c2 = ExportUtils.getExportedTo("c.i.b.C, c.i.b.D");
    assertTrue("Patterns does not need to be contained",
                !ProcessingUtils.contains(c1, c2));
}
```

The method `contains` and its use context was introduced in section 2.2.5. The method itself manipulates AspectJ objects such as `PatternParser`, `Pointcut`, `NamePattern`, etc. to check if a certain string representing classes contains the other. It therefore needed to be checked for the expected behavior.

It also should be added that since the tests introduced here test the behavior of the AspectJ objects used in the fine-granularity access-control implementation, they can and should test the behavior of new versions of AspectJ. Doing so will assert they behave similarly to the current version, asserting that the fine-granularity access-control implementation works as well with that version.

### 3.2 Compile-time Tests

As with the runtime tests, it was necessary to assure that the compile-time declarations had the correct effect. For this purpose classes (A, B, C etc.) were created with hierarchy containing all elements types and different types of `Export` declarations. For each of those classes two satellite classes were created. One was a control group (`AControlGroup`) and the other the test group (`ATestGroup`). Those classes were almost the same except for their name, and contained various call tests, to their main group class (A, B, C etc.). The idea was that all `Export` declarations were directed only to the control group classes and therefore the control group should have no problems using the class elements, while the test group will raise a compile time error when calling the main class elements.
Making the classes was important, but it was also important to test that only the correct errors were raised for the test group and no errors at all are raised for the control group. Since each class contained many call tests, compiling and looking for the right errors was not desirable. Instead a process that verifies the correctness was required. For that purpose the AspectJ compiler class (org.aspectj.tools.ajc.Main) was used to compile the classes and then the messages generated by the compiler were matched against a set of expected messages that contained the errors that needed to be raised. Whenever an unexpected error was raised, or an expected error was not raised, the Junit test failed.

### 3.3 File Generation Tests

Another test applied was to check the correctness of the generated files. This test's prime goal was to make sure that during the changes in the Jafar code the generated files will be logically the same, assuming that the code change was not required. The idea was to take the generated aspect files and to compare them to files known to be correct.

The problem in such a comparison was to ignore non-logical changes (such as changes in indentation, spaces, remarks etc). To make such a comparison, AspectJ ASTParser (org.aspectj.org.eclipse.jdt.core.dom.astParser) was used. by comparing two files with the parser it was possible to compare between them using AjASTMatcher, that makes a comparison between two AST trees. Also the AjNaiveASTFlattener (org.aspectj.org.eclipse.jdt.core.dom.AjNaiveASTFlattener) was used to create a text file from an aspect and then compare between two text strings to assert they are the same.
4. Future Work

The framework presented does not fully answer the need to enforce the fine-grained access level, because it lacks the ability to enforce the running of the framework. A client of a certain code can easily run it with a plain Java runtime environment without an AspectJ agent to apply the restrictions, allowing code loaded at runtime to overpass that restrictions. As such it cannot be used for applications in which program security is mandatory.

A possible method to comply with such security demands will be to create a custom Java class loader that will guaranty the presence of an AspectJ agent before the loading of the classes or that will weave the classes in runtime. After integrating such a class loader, the whole application can be digitally signed, to prevent any option of modifying the code or removing of the class loader.

Such a class loader wasn't included it in the framework, but a class loader that weave code at runtime was proposed and demonstrated in a work by Martin Lippert [9, 10] to solve a similar problem. In that work it was suggested as a solution to enable the usage of AspectJ in Eclipse [11] custom plugins. Martin Lippert [12] also proposes a solution to the problem of runtime loading of classes and aspects containing restrictions generated by Jafar. In such case the aspects must be loaded at runtime and applied to a running application, the shown AJEER gives this case a solution.

The Jafar framework security problem can be solved in other ways rather than using a class loader, moreover the security problem is not limited to this study, but rather is a more general issue of enforcing an application to run under AspectJ agent. Because of that, the problem is out of the scope of the work and thus is left open. It can only be assumed that when the use of AspectJ will expand, such a need will be answered in the language itself or by an external method.

As the framework allows the developer to add fine grained access-levels, it will be most beneficial if the developer could also use analyzing tools that will recommend on elements and the access-level they should enforce. For example the framework can be used with DOMO [13] which analyses program code, to recommend on the access-level needed, add the needed Export declarations and then enforce them by using Jafar.
5. Conclusions and Related Work

Previous work has tried to create a mechanism for fine-grained access control in Java. M. Papa et al. designed a secured mechanism for specifying various access control policies for Java packages [14]. The mechanism uses a ticket that is passed to authorize access. The mechanism supports only package access control and allows a package to define which packages can access it, but does not allow wildcards.

Another approach to creating a fine-grained access control in Java was introduced by G. Kniesel et al. [15, 16]. Their approach answers the problems caused by aliasing to an object encapsulation by introducing the readonly and mutable modifiers. The new modifiers enable better access control over the data an object contains but it does not answer visibility of such objects as the presented Jafar framework does.

Trying to answer similar problems, J. Vitek et al. [15] proposed confined types for Java as an aid for writing secure code. They argue that code outside of the domain should never be allowed to manipulate confined objects directly. So their proposal is to extend Java with two modifiers, one for classes (confined) and one for methods (anon). This approach is different from Jafar's - where Jafar will limit an access to classes, confined types will limit manipulation of objects.

The aliasing problem presented in [15-17] is also discussed in [18-24]. While different attempts to solve the problem used different approaches, the presented framework Jafar can also answer some aspects of the problem by limiting an access only to a set of permitted classes. Limiting the access does not solve the aliasing problem, but in cases where Export is done to a finite set of classes and these classes are guaranteed not to wrongly manipulate the data, the problem can be contained.

N. Schärli et al. [26] claimed that the Eiffel solution is very limited since the designer has to take an up-front decision on the clients that will have access. Clients that are not known when the class is written cannot be taken into account and so can never have access. Instead they introduce encapsulation policies which are separate and independent from the implementation of a class and allow the designer to specify an arbitrary number of independent encapsulation policies for a given class. Their solution does not answer any security and program complexity issues that can be raised by allowing multiple encapsulation policies entities.

M. Shomrat and A. Yehudai[27] explored the possibility of using AspectJ programming language for the enforcement of design models. They tried to enforce a policy to control possible congestion of the server due to large volume of requests and a policy to make a kernel independent of the rest of the system in an Intensive Care Unit example. They don't address the security problems created by such enforcement in AspectJ and show that its current form is only partially adequate for design enforcement.
S. Herrmann [28, 29] motivates this work by claiming that "Java’s access modifiers are unfortunate bundles of export scopes". He introduces Object Teams, a programming model aiming at improving support for modularity. Object Teams uses family Polymorphism [30], where each Team defines a self contained world. Object interactions within a Team are defined solely in terms of methods of roles and the enclosing Team. Classes contained in a Team are called role classes. Role classes can be bound to classes outside the Team (“base classes”) using the keyword playedBy. In Object Teams a group of role instances can be seen as a horizontal slice of a system, which is protected against unwanted interaction with other groups (Teams) of the same type. Flexibility is gained by vertical integration denoted by the playedBy construct binding a role class to a base class.

As shown in this paper, the proposed implementation differs from Eiffel in several ways. In Eiffel, an exported feature is automatically exported to all of its descendants as a consequence of the idea that all features are exposed to all decedents. In addition, classes cannot create their own instances and use features that were restricted only to other classes; this differs from our presented fine-grained access level that follows the Java principles.

The friend mechanism in C++ also differs in several ways from our implementation; the mechanism gives the option to declare a class as a friend of another, allowing the friend class to access even private members. In our case a class cannot be exported, but rather its elements are exported. Another difference is the call of an element from a subclass of a friend. In the example scenario, class C inherits from class B, and class B is a friend of class A, but class C cannot use the declared friend members in class A. This case differs from our implementation that leaves the decision in the programmer’s hands by providing the "+" syntax to allow it.

The introduced Java access-level also defines a new option that wasn’t shown in C++ or Eiffel, which is the ability to export to a package, and even more, to a general set of packages and classes, that may have been implemented or perhaps will be introduced later by defining a flexible mechanism similar to AOP mechanism to express their names.

The Java language is widely used. Nevertheless its access control model lacks the fine-granularity defined by Eiffel. By introducing a tool to enforce a finer defined accessing, users can easily add new access-levels which improve coding modularity. The practical use of such access-level syntax by users can allow the incorporation of this syntax as an integrated part of Java or similar languages.
Bibliography


