Declassification: Transforming Java Programs to Remove Intermediate Classes

Bernadette Power
Computer and Networking Department, Carlow Institute of Technology, Kilkenny Road, Carlow, Ireland
bernadette.power@itcarlow.ie

G. W. Hamilton
School of Computer Applications, Dublin City University, Glasnevin, Dublin 9, Ireland
hamilton@compapp.dcu.ie

Abstract

This paper presents an optimisation technique which automatically *inlines* certain classes within their enclosing class. Inlining a class involves inserting the fields and methods of this class into the body of its enclosing class. The enclosing class is the class which declared an instance of the class. The declaration of the inlined class can then be removed from the program. This technique transforms Java programs into an equivalent form, which may be less readable, but is more efficient. The results of the empirical study showed that few classes were found suitable for inlining and that the declassification was not overly successful when optimizing the test programs. One of the advantages of declassification is that it does not result in code bloating. It is thought that further extensions to the declassification technique and an intrinsically object-oriented set of test programs could greatly improve it's effectiveness.

1. Introduction

Computer programming has undergone phenomenal growth in recent years. Software applications are increasingly being written in object-oriented languages like Java [1] and C++ [7], because they offer simple, uniform, abstract programming models. This programming model provides the benefits of increased flexibility, maintainability and extendibility. Object-oriented programming encourages the use of small methods and objects [4]. This style of programming introduces much overhead as each method call results in a dynamic dispatch and each field access becomes a pointer dereference to the heap allocated object.

Consequently, the run-time performance of object-oriented languages like Java is behind the most popular languages today, even with just-in-time compilation technology. There is an obvious need for more aggressive optimising techniques for the Java language.

Object-oriented programs are difficult to reason about and optimise because of the use of inheritance and dynamically bound method calls. A number of optimisation techniques have been researched and developed to improve the performance of object-oriented programs. For example [5, 6, 13, 14, 15]. The declassification technique is a novel approach to improving the performance of Java programs. Its inception was motivated by the success of the higher-order deforestation algorithm proposed in [8]. This algorithm can eliminate intermediate data structures from higher-order functional programs. By removing the intermediate data structures, the performance of the program should be improved and the heap space required by the program reduced.

A detailed discussion of the usage counting analysis algorithm and the proposed transformation is presented in an earlier paper [10]. This paper extends this work, by presenting the transformation technique used and also analyzing and illustrating the results of an empirical study. The objective of the declassification technique is to reduce the number of classes which are instantiated and used in a program during its execution. The declassification technique is divided into two parts; analysis and transformation. Information from the usage counting analysis algorithm is used to determine how many classes are suitable for inlining. The transformation involves a source to source transformation and involves inlining the fields and methods of the inlinable class into its enclosing class. The declaration of the inlined class can then be removed from the program as it is no longer needed. By inlining the class in this way we are eliminating the need to create and maintain these 'intermediate classes' and instead we are extending the size of their enclosing class. This technique will change the hierarchical structure of the program by eliminating these inlinable classes.

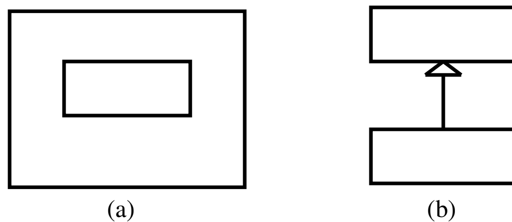
The optimised program code can then be compiled and run as normal. We investigate the potential of the declassification technique by performing the analysis and transformation on a number of reasonable-sized Java programs. The remainder of this paper is structured as follows. In Section 2, we describe the usage counting analysis which is used to identify classes for inlining. In Section 3, we describe the declassification transformation. Section 4 lists the restrictions placed on class inlining and Section 5 investigates the visibility constraints necessary. In Section 6 we evaluate the declassification technique. A number of extensions to the declassification technique are outlined in Section 7. In Section 8, we compare research in related areas, and Section 9 concludes.

2. Usage Counting Analysis

The central goal of the usage counting analysis technique is to identify classes which are *used exactly once* within the program. Examples of possible usages include a variable, field or parameter declared within the program which is the same type as the class. Two variables of this type are two usages of the class. A class which is used once within the program is considered a suitable class for transformation.

The usage counting analysis examines all top-level classes in the program to establish their suitability for inlining. A top-level class is a Java class which is not an inner class. There are four types of inner classes; static member classes, member classes, local classes and anonymous classes. Throughout this paper top-level classes will be referred to as classes and a distinction will only be made between top-level and inner classes when extensions to the declassification technique are discussed. It should be noted that although the enclosing class is the only class to use the inlinable class, it could instantiate this class one or more times within the program.

To aid the exposition of this technique, consider the example in Figure 1.



(a) (b)

Figure 1. The Picture and Circle classes

The Picture class has a field which stores an instance of the Circle class. This is illustrated in Figure 1 (a). The Circle class is a sub class of Shape. Figure 1 (b) illustrates the hierarchical relationship between Circle and Shape. The Circle class is a potential candidate for inlining, if it is established by the usage counting analysis technique that this class is used no more than once in the program. The Picture class is the enclosing class which uses it.

It is essential for the correct operation of the declassification technique that accurate information is available on the usages of the different classes within the program. This requires a very sophisticated type inference algorithm as Java programs, like other object-oriented programs, are difficult to reason about. The usage counting analysis technique uses the results from the type inference algorithm [9], which determines for each variable the set of classes to which it may be instantiated at run-time. It provides information on different types of variables; these include fields of objects, local variables, method parameters and return types. The usage counting analysis algorithm interrogates the type information available in the sets and supplements it with information gathered on the layout and structure of the classes in the Java programs. From these sources it is able to determine the usage counts of each class within the program. The pseudocode for the usage counting analysis algorithm is illustrated in Figure 2 and specifies the criteria by which usage counts are calculated for all top-level classes. A class with a count of one has been determined by the usage counting analysis as having only one use within the program.

It is essential that the declassification technique is safe. Its safety is assured by the fact that a class I is only inlined within another class E if the usage counting analysis algorithm determines that the usage count of the inlinable class I is one. There is as a result, an instance of the inlinable class I stored within one of the fields e.f of the enclosing class E. It is the only variable within the program that can store a reference to the inlinable class. This variable e.f could be instantiated to one or more consecutive instances of the inlinable class. There is, however, only one use of the inlinable class through the e.f reference. There are as a result, no alias relationships to the instance of the inlinable class, which could distort the sharing semantics of the program. Inlining is greatly complicated by alias relationships and it is more difficult to guarantee that such a program transformation will not result in invalid results. Consequently, the declassification technique does not have to deal with aliases and its analysis and transformation is significantly simplified, as a result.

```
Initialise all usage counts to zero
Apply Plevyak/Chiens algorithm to determine the set of
classes
        to which each variable may be instantiated

Start with the main program class
Count:
Begin
    Add 1 to the usage Count of current class
    If current class not marked
    Begin
        Mark current class.
```

```
            For each class in the set determined for
                  each field perform Count.
            For each class in the set determined for
                  each local variable perform Count.
            For each class in the set determined for each
                  method parameter perform Count.
            For each class in the set determined for each
                  method return type perform Count.
            For each class in the set determined for each
                  anonymous object perform Count.
            For each superclass perform Count.
      End
End
```

Figure 2. The pseudocode for the usage counting analysis algorithm

Figure 3 illustrates an example of the Picture and Circle classes in a simple Java program. There are four classes in the program; Picture, Circle, Square and Shape. The Circle and Square classes are subclasses of the Shape class. Each class has its own associated fields and methods. Again the Circle is used exactly once by the Picture class and is a suitable class for inlining. The Square class is not a suitable class because it is used twice by the Picture class.

```
class Picture {
     float count;
     Circle myCircle  = new Circle( );
     Square redSquare = new Square( );
     Square blueSquare = new Square( );

     public void initialPictureCircle( ) {
            myCircle.radius = 2;
            myCircle.colour = ‘Brown’; }

     float getCount( ) { return count; }
}

class Circle extends Shape {
     float radius = 1.0;
     int count = 2;

     int getCount( ) { return count; }

     public void calArea( )  { area = Math.PI * (radius *
                                   radius);}
}
class Square extends Shape {
     float width;

     public void calArea( )  { area = width * width;}
}

class Shape {
     float  area;
     String colour;
}
```

Figure 3. Program code

3. Transformation

The analysis will identify suitable classes for inlining. The transformation involves inlining the fields and methods of each ‘inlinable’ class within the enclosing class that has declared the instance. Any references to the instance of the inlined class are changed to reference its inlined fields and methods within the enclosing class object. Figure 4 illustrates the aggregate association between the Picture and Shape classes, following transformation. The circle class has been eliminated and the Picture class now inlines the Circle class.

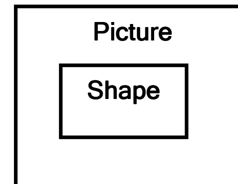


Figure 4. The Picture and Shape classes

The fields radius and count will be added to the Picture class. The methods getCount( ) and calArea( ) will also be added to the Picture class. In discussing the transformation, the following term is used; the inlined class variable is the field which is used to store the instance of the inlinable class within the enclosing class. The type of this variable is changed to the inlinable class’s superclass. The Picture class how creates an instance of the Circle class’s superclass in the variable myCircle. Figure 5 illustrates the example program given in Figure 3 following the transformation.

3.1. Fields

It may be necessary to change a field name in the inlinable class if there is a name clash. A name clash occurs when an inlinable field has the same name as one of the following:

1. One of the fields in the enclosing class;
2. One of the fields in any of the superclasses of the enclosing class;
3. One of the local variables in an enclosing class method;
4. One of the field constants of any of the interfaces the enclosing class or any of its superclasses implement.

A compiler error would occur if an inlined field clashes with a field in the enclosing class. The incorrect execution of the program could result if it clashes with a field in the enclosing class’s superclass. Another error could occur if the inlined field clashed with one of the local variables in an enclosing class method. This is because local variables

take precedence over fields. A reference to the inlinable class's field could after transformation reference a local variable if their names clashed. Finally, a clash with a field constant of one of the enclosing classes' interfaces, would result in the inlined field taking precedence over it. This would cause the incorrect execution of the program. The resolution of the name clash is achieved by replacing the variable name by a "gensym". A gensym is a unique name and it is implemented by concatenating the variable name with a random integer. Changes are necessary to both the inlinable class and the enclosing class to account for the new field names. If a field name has changed it is essential that changes are made to the field initializers and methods of the inlinable class. The fields of the inlinable class can be added directly if there is no clash between names.

Some changes are necessary to the enclosing class to reference the new fields which has been inlined. The inlined class variable is not removed, but changed to declare an instance of the superclass. Any references to fields belonging to this superclass can remain unchanged. Any references to fields not belonging to the superclass must be changed to reference the inlined fields directly.

The position in the enclosing class where the inlined fields are placed is important. They must be inlined immediately after the declaration of the inlined class variable. This is because these fields may be used in the inialization of one or more of the enclosing classes' fields. The position where the methods are placed is not important.

3.2. Methods

Similarly, it is necessary to change a method name in the inlinable class, if a name clash occurs. A name clash occurs when an inlinable method has the same name and number of parameters as one of the following:

1. One of the methods in the enclosing class;
2. One of the methods in any of the enclosing classes' superclasses.

To inline a method which has the same name and number of parameters could result in the incorrect execution of the program. The resolution of the name clash is achieved by renaming each method name to a unique name not used elsewhere in the program. The references in the inlinable class must be changed to reference the new method names. This is done in the same way as for fields. The amount of work needed to transform the program is reduced if the inlinable methods do not change name. The enclosing class must be changed again to reference the new inlined methods.

```
class Picture {
    float count;
    Shape myCircle;
    {circle35223( );}
    float radius;
    int count45143;
    Square redSquare   = new Square( );
    Square blueSquare = new Square( );

    public void initialPictureCircle( )              {
        radius = 2;
        myCircle.colour = 'Brown';                   }

    float getCount( ) { return count; }

    int getCount67454( ) { return count45143; }

    public void calArea ( )                          {
        myCircle.area = Math.PI *
        (radius * radius);                           }

    circle35223( )                                   {
        myCircle = new Shape( );
        radius = 1.0;
        count45143=2;                                }
}

class Square extends Shape{
    float width;
    public void calArea( )  { area = width * width;}
}

class Shape {
    float area;
    String colour;
}
```

Figure 5. Transformed program code

The pseudocode for the transformation technique is given in Figure 6. The technique assumes there is a clash of names between the inlinable and enclosing classes. The algorithm would be simpler if this was not the case.

For each class with a usage count equal to 1
Begin

Step A For each field of the inlinable class which clashes
Make the field name unique.

For each method of the inlinable class which clashes
Make the method name unique.

Step B Add inlinable class fields to enclosing class.
Add inlinable class methods to enclosing class.

Step C Change inlined class variable to be of the superclass type.
Add call to inlinable class's changed constructor method.

Step D For each field initializer and method in enclosing class
Perform ReferencesChangesInEnclosingClass.

Step E For each field initializer and method in inlinable class
Perform ReferencesChangesInInlinableClass.

Step F Perform ChangeConstructorMethods

Step G Combine enclosing class finalize method with inlinable class finalize method.

Step H Append inlinable interfaces to enclosing class interfaces.

End

```
ReferencesChangesInEnclosingClass:
Begin
    Change references to inlinable class fields and
        methods to reference the new inlined fields and
        methods.
    References to the inlinable class superclass fields and
        methods should remain unchanged.
    Change reference to inlinable superclass object to
        inlined class variable.
End

ReferencesChangesInInlinableClass:
Begin
    Change references to inlinable class fields and
        methods to reference the new names of the fields
        and methods.
    Change references to the inlinable class superclass
        fields and methods by referencing the inlined
        class variable.
    Change reference to inlinable superclass object to
        inlined class variable.
End

ChangeConstructorMethods:
Begin
    For each inlinable constructor method
    Begin
        Concatenate method name with a random interger
        number
        Instantiate the inlinable class's superclass
        For each inlinable instance field
            Add field body to constructor method body
        For each inlinable instance block initializer
            Add block initializer body to constructor
            method body
    End
End
```

Figure 6. The pseudocode for the transformation technique

A detailed explanation of the pseudocode is given by explaining how the sample program in Figure 5 is transformed step by step. Examples used to illustrate the steps involved, are taken from this sample program also.

Step A. There is a name clash between the field count in the Circle class and the field count in the enclosing class. The field name is made unique. For example: int count45143. All the other inlinable field names remain unchanged.

Step B. There is also a name clash between the method name getCount( ) in the Circle class and the method getCount( ) in the enclosing class. The method name is made unique. For example: getCount67454( ). All the other inlinable method names remain unchanged. The inlinable classes' fields and methods are added to the enclosing class. Some of the inlinable classes' methods are not added directly to the enclosing class but are combined with the enclosing classes' methods. Examples of these are the constructor, finalize and instance block initializer methods.

Step C. In Figure 3 the inlined class variable is of type Circle and an instance of this class is created. The transformation process changes the declaration type of the variable to its superclass Shape. For example: Shape myCircle. This is done to enable the enclosing class to access the fields and methods of the Circle classes' superclasses. The instantiation of the inlinable class is removed. A call is placed to the inlinable class's changed constructor method. This method instantiates the superclass and initializes the inlinable classes' fields as required. For example: circle35223( ).

Step D. The enclosing class Picture must be changed to reference its new inlined state. The changes are made to the enclosing classes' field initializers and methods. References to the inlinable class Circle are changed to reference the inlined field and method names directly. For example: radius = 2. No changes are necessary to any references to the fields and methods of the superclass Shape. For example: myCircle.colour = 'Brown'. Any reference to the superclass object must be changed to reference the inlined class variable myCircle, within the enclosing class's inlined state.

Step E. Changes must also be made to the inlinable class to account for the changes in the names of the fields and methods. Local variables in methods complicate the process of identifying field references. Any reference to the inlinable class's superclass object or its fields and methods must be done by referencing the superclass object stored in myCircle. For example: myCircle.area.

Step F. Each of the inlinable classes' constructor methods are changed to create an instance of the superclass and store it in the inlined class variable myCircle. For example: myCircle = new Shape( ). Each field body of the inlinable classes' instance fields are then added. If a call to an instance block initializer follows, its method body is added. These additions are added to the start of each

constructor method. Static field bodies or static block initializers are not added to the constructor method.

Step G. The finalize method of the inlinable class must be combined with that of the enclosing class.

Step H. Any interfaces which the inlinable class implements must be implemented by the enclosing class.

4. Restrictions

Five restrictions are placed on class inlining because it would be unsafe to allow it in these cases. They are as follows:

1. Class Instaniation

The variable which is used to store the instance of the inlinable class *cannot* be used to store an instance of any other class. For example an instance of the inlinable class's superclass is stored in the same variable as the instance of the inlinable class. To inline in this situation could result in the distortion of the program.

2. Method Overridding

The inlinable class could have one or more overridden methods. Dynamic binding occurs at run-time to establish which method is being called, whether it is the inlinable classes or the superclasses method. It is necessary, however, to establish which method is being called during the transformation process. This is because the sharing semantics have now changed. A restriction is therefore placed on class inlining to prevent a class being inlined if it has an overridden method *and* there is a reference to one of the overridden methods in either the enclosing or inlinable class.

3. Abstract Parent

To inline an inlinable class which has an immediate abstract superclass would be wrong as this abstract superclass cannot be instantiated.

4. Self Inlining

It is necessary that the declassification technique places a restriction on the inlining of a class within itself. This is because it would have a recursive effect which would cause the transformation to enter an infinite loop.

5. Reflection

Reflection is not handled by the declassification technique, but this can be easily checked. The Class.forName( ) method could be used in a program to invoke the inlined class. It is assumed that the code being transformed does not use reflection.

5. Access Modifiers

In Java, access modifiers are used to specify the visibility/access of a class within the program. The declassification technique should not break the privacy of objects and change the visibility of the inlined classes' fields and methods within the enclosing class. To weaken this visibility would be a change to how the inlined class is accessed and is considered unsafe. Access modifiers are *only* considered in the context of an inlinable class being inlined within an enclosing class within the same package. The transformation algorithm does not inline classes from different packages.

Access modifiers can be associated with classes, fields, methods and local variables. The only access modifier that can be associated with local variables is 'final'. Transforming an inlinable method with this access modifier will not change its visibility. Inlining a class with the following access modifiers; public, default and final does not change its visibility. An abstract modifier is not applicable as an instance of this class cannot be created. Inlining fields and methods with the following access modifiers; public, protected, default, private, static and final do not change their visibility. Any abstract method is automatically abstract itself and must be declared as such. Therefore inlining a class with any of the above access modifiers does not change the visibility of the fields and methods and is considered safe.

5.1. Other issues

The inlining of a class within its enclosing class eliminates the need to instantiate this inlinable class. This transformation does not only eliminate one instance of the inlinable class, the number of instantiations eliminated depends on the number of times the enclosing class is instantiated within the program. The more times the enclosing class is instantiated the greater the benefits will be to the declassification technique. One of the advantages of declassification is that it does not result in code bloating. This is supported by the empirical study, which shows that there is a negligible increase in code size after optimization. Some programs that had inlining capabilities show a small but important decrease in code size.

An inlinable class can be inlined if a static instance of it is created within the enclosing class. Each of the inlinable classes' fields and methods are added to the enclosing class with a static modifier. The inlinable classes' static fields and methods are added unchanged. The sharing semantics between the inlinable and enclosing class has not been changed after transformation.

6. Evaluation

The declassification technique was evaluated by analyzing and transforming a number of reasonably sized object-oriented programs in the SPEC98 [SPEC JVM, 1998] benchmark suite. SPEC98 is one of the most commonly used benchmark suites and a brief description of each is given below. One other medium sized Java program is also evaluated. The results of this evaluation are presented and from these results the benefits and costs of the technique are assessed. The benefits of the declassification technique are the run-time performance gains, reduced memory usage and the fact that there is no increase in code size. The cost could be the fact that the performance gain as a result of the optimization is not substantial enough when measured against the additional compile-time cost of carrying out the technique.

Test Programs
The industry standard SPEC98 benchmark suite has been used to conduct this study.

Check benchmark: This is a simple program to test various features of the JVM to ensure that it provides a suitable environment for Java programs.
Compress benchmark: This benchmark uses the modified Lempel-Ziv method (LZW), which finds common substrings and replaces them with a variable size code.
DB benchmark: This benchmark performs multiple database functions on a memory resident database.
Raytrace benchmark: This is a variant of _205_raytrace, a raytracer that works on a scene depicting a dinosaur, where two threads each render the scene in the input file time-test model, which is 340KB in size.

One other medium sized Java program is used as test data;
Declassification (declass) program
The declassification source program has been written entirely in Java.

6.1. Usage Counting Analysis

The test programs were analysed to calculate the number of suitable classes for inlining. Table 1 has three columns. The first illustrates the programs analysed. The second illustrates the number of top-level classes in each program, the third illustrates the number of classes which are suitable for declassification in each program. It is obvious from these results that there are few top-level classes which meet the criteria for inlining. It should be remembered that even though few examples of inlinable classes are found, these might still result in a lot less objects being created at run-time.

Table 1. Comparison of the number of inlinable classes in each program

Programs	Classes	Inlinable Classes
check	29	1
compress	25	1
db	19	1
raytrace	41	2
Declass	21	3

Check, compress and db have one inlinable class. The raytrace program has 2 inlinable classes out of 41 (5%). The declass program was the most suitable for optimization as it has 3 inlinable classes out of 21 (14%).

6.2. Transformation

In order to establish the effectiveness of the declassification technique, it is necessary to compare the run-time performance and memory consumption of the unoptimized and optimized versions of each test program. From these results, we can extrapolate the benefits of the declassification technique. To measure the effectiveness of the declassification technique we transformed the source code of the test programs. Each optimized program was then compiled using Java 2 SDK standard edition version 1.4. The unoptimized and optimized versions of each program were measured by the benchmark program available in SPEC98. Measurements were taken on a Pentium 200 with 32 megabytes of RAM and are the average of 10 runs. The percentage improvement (or disimprovement) is calculated by dividing the difference between the optimized and unoptimized measurements (eg. run-time memory usage) by the original unoptimized measurement. Tables 2, 3 and 4 show the percentage change. A negative number means the measure has decreased by that amount. A positive number means the measure has increased by that amount. A number of 0.00 means no change.

6.2.1. Memory Consumption. The percentage change in memory consumption of the optimized programs is illustrated in Table 2.

Table 2. Change in memory consumption of the test programs

Programs (in bytes)	Average Memory used Unoptimized	Average Memory used Optimized	% Memory Change
check	37084	37115	0.08
compress	6280938	5747851	-8.49
db	8684467	8684594	0.00
raytrace	4339080	4296789	-0.97
Declass	6717439	6243451	-7.06

It was hoped that the declassification technique would reduce the number of objects created by the program because of its ability to inline classes. It could be deduced that this has occurred in some of the test programs. The check and db programs in the empirical study show a negligible change in memory use between the optimized and unoptimized programs. The raytrace program had some inlining opportunities but only a small reduction in memory use after optimization. The declass program shows an important and significant 7% reduction in memory use following the inlining of three classes. The compress program shows a 8.5% decrease in memory use following the inlining of only one class. It could also be deduced that if more classes were inlined, the memory usage of the majority of programs would reduce further.

6.2.2. Run-time Performance. Table 3 shows that there is very little or no difference in the run-time performance of check, compress, db, and raytrace programs. It was hoped that the run-time average of the declass and raytrace programs would be decreased as a result of class inlining. This should reduce the number of memory dereferences as the fields of the inlinable class become local to the enclosing class. It should also reduce the number of dynamic dispatches necessary to execute the program.

Table 3. Changes in the run-time averages

Programs (in secs)	Average Run-Time Unoptimized	Average Run-Time Optimized	% Run-Time Change
check	54.9	54.9	0.00
compress	105.7	106.1	0.38
db	95.95	95.63	-0.33
raytrace	33.45	33.46	0.03
Declass	30.77	30.63	-0.45

The empirical study showed a negligible increase in the run-time average of the optimized compress program and only a 1% decrease occurred as a result of optimizing the declass program. It could be deduced from these results that the fields and methods of the inlinable classes within the compress and declass programs are not highly referenced by their enclosing classes. It should be noted that although there is not much of a change in the run-time performance for these programs, for longer runs, there may be more of a change because if there is less memory consumption as shown in Table 2 for the compress and declass programs, then less time should be consumed by garbage collection. It could also be deduced that the overall poor impact on the run-time performance is predominately due to the small number of classes found suitable for declassification.

6.2.3. Program code size. An important feature which can be attributed to the declassification technique is that there will be little to no increase in code size. This feature cannot be attributed to many other optimization techniques such as object inlining [Dolby, 1997]. This is supported by the empirical study, which shows that there is a negligible increase in code size after optimization. Some programs that had inlining capabilities show a small but important decrease in code size. The code size of the declass program reduced by 5% after declassification.

Table 4. Changes in the run-time averages

Programs	Unoptimized (bytes)	Optimized (bytes)	% Code Size Change
check	110,671	110,519	-0.14
compress	92,341	92,425	0.09
db	88,130	88,190	0.07
raytrace	131,833	131,100	-0.56
Declass	295,996	281,384	-4.94

7. Further Extensions

The declassification technique concentrates on top-level classes which are declared and instantiated as a field of an enclosing class. The usage counting analysis algorithm identified other top-level classes that have a usage count of one. The declassification technique could be extended to inline these top-level classes, it could also be extended to inline single usage inner classes.

7.1. Local Objects

The empirical study found that there are a considerable number of top-level classes that are instantiated as local variables in methods. A local object cannot be inlined successfully if a reference to the local object is returned from the method or passed as a parameter to another class instance. Figure 7 illustrates only the local objects which are used in this way. 11% of the db program classes and 8% of the compress program classes are being used as local objects. Each of these classes could be inlined within the method which created them by inlining its fields and expanding each of its method calls.

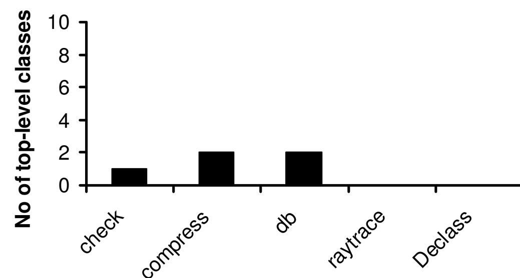


Figure 7. Number of top-level classes created in a method

An anonymous object which is instantiated in a method could be inlined within the method in a similar way to how you inline a local object. The information gathered by the declassification analysis and transformation algorithms could also be extended to facilitate the inlining of inner classes, i.e. member, local and anonymous classes. In Java each inner class is created as a normal top-level class by the JVM. This results in each inner class requiring space and time to be created on the heap. Eliminating inner classes should result in reduced memory consumption by the program and improved run-time performance.

8. Comparison with Related Work

There are a number of compile-time techniques that have been developed to optimise object-oriented programs. Method specialisation is a technique which is used to reduce the number of dynamic dispatches necessary in a program. A selective specialisation technique is presented in [5], whose central aim is to replace some of the dynamic calls to methods within a program with statically bound calls to specialised methods. The fundamental difference between selective specialisation and other method specialisation strategies is that it is selective about which methods it specialises. The selective specialisation technique in [5] increases the size of the Java program, the declassification technique does not. The technique in [5] uses both static and dynamic information gathered from program profiling. The declassification technique only uses static analysis. The complexity of the technique in [5] increases when it needs to recursively specialize methods, moving up through the call graph. The declassification technique does not involve recursion and is a simple and straightforward optimization technique.

Other areas of research have concentrated on object inlining. Object inlining aims to inline object(s) within another object or method, while the aim of the declassification technique is similar, it is intrinsically different because it involves inlining classes within other classes rather than object instances. For example, an object inlining technique is presented in [2] which can inline objects within methods. This technique, differs from the declassification technique because its central aim is to identify and inline objects which are created within a method. These are referred to as local objects. The declassification technique has the ability to inline top-level classes. A limitation of [2] is the fact that an object cannot be inlined if a reference to the inlined object is passed as a parameter to other objects or methods within the program. The solution to this problem is to reconstruct the object; the feasibility of this approach is discussed in [3].

A more powerful and less restrictive object inlining optimizing technique is presented in [6] which will automatically inline objects within container objects. The adaptive analysis technique also makes use of the precise information obtained from the algorithm in [9]. It is not restricted to inlining objects within methods, but has the considerable drawback that the inter-procedural data-flow analysis used by this technique is complex. This complexity is necessary because detailed information is required on the objects used within a program to enable program transformation. The declassification technique is a lot simpler and straightforward as its aim is to identify classes which are used exactly once in the program. These classes will then be inlined into their enclosing class. Complex calculations are not necessary because we do not need to check and deal with aliases in the analysis and transformation of each class. The technique in [6] in comparison needs to find and specialise uses of inlined objects and to ensure that inline allocation does not change aliasing relationships.

[11] states that an empirical study done on the object inlining technique in [6], revealed that the size of a program after inlining is almost identical to its original size. [12] estimates that there is on average a 20% increase in the size of a program. They cannot guarantee however, that this will always be the case and it is believed that there could be a substantially higher increase in the code size. This is because the inlined object's fields and methods are added to one or more container objects increasing their size and the class declaration of the inlined object still remains in the program. The declassification technique in comparison has a negligible increase in the program size, due to the fact that once the inlined classes' fields and methods are added to the enclosing class, its class declaration is deleted from the source code. [12] discusses a revised algorithm. [13] extends the research on automatic object inlining in Dolby's papers and investigates the possibility of several objects being inlined within a single field in succession. The declassification technique is different as it is inlining classes. It has however, the ability to inline a number of instances of the same class into the same field in succession.

9. Conclusion

This paper presents an optimisation technique which facilitates the automatic declassification of programs. Declassification involves the identification of suitable intermediate classes and the transformation of the source code to inline the fields and methods of each intermediate class within its enclosing class. The declaration of the inlined class can then be removed from the source program. The declassification technique is a lot simpler and less complicated than other object inlining techniques such as [6].

The declassification technique has the potential benefits associated with object inlining, but unfortunately few classes were found suitable for declassification. One of the reasons such a small number of inlinable classes were found could be the fact that four of the test programs

where taken from the SPEC98 benchmark suite. This benchmark suite is not particularly object-oriented as some of the programs are direct translations from the Fortran language. It would be interesting to see how the results of the declassification technique would alter if a more extensive empirical study was conducted that had a set of test programs which are intrinsically object-oriented. It should be noted however, that even if few classes are found suitable for inlining, this still *may* result in a lot less objects being created at run-time. This is discussed in section 5.1.

The empirical study showed that when suitable classes are found for inlining it can have a positive effect on the memory consumption of the program. Two of the programs in the empirical study showed a significant reduction of between 7% and 8.5% in memory consumption, as a result of declassification. The effect of declassification on the run-time performance of the test programs is negligible. It could be argued, however, that this is largely due to the small number of classes found suitable for declassification.

There are a number of significant features associated with this optimization technique. One of these is the fact that there is a negligible run-time cost associated with the technique. There is also little to no increase in the code size of a transformed program, if anything it will shrink in size. The analysis and transformation algorithms are less complicated than many other optimization techniques. Another feature is that it automatically inlines suitable classes and deletes the original class declaration, without any programmer intervention. The programmer does not have to explicitly declare that certain classes should be inlined.

Although the declassification technique was not overly successful in optimizing the test programs, further extensions to this technique could greatly improve its success. It was found that there are a number of local objects suitable for inlining within half of the test programs. The compress and db programs have two potential local objects and the check has one, this is an average of 7% of their overall classes. There are other further possibilities for extending the technique by inlining inner classes. These extensions combined with an intrinsically object-oriented set of test programs could greatly improve the effectiveness of the declassification technique.

References

[1] Gosling, J., Joy, B., Steele, G., The Java Language Specification, Addison-Wesley Longman, Inc, 1996.

[2] Budimlic, Z., Kennnedy, K., Optimising Java: Theory and Practive, Software: Practice and Experience 9, 6, June 1997, pp. 445-463.

[3] Budimlic, Z., Kennnedy, K., Static Interproedural Optimizations in Java, Center for Research on Parallel Computation, Rice University, Technical Report CRPC-TR98746, 1998.

[4] Calder, B., Grunwald, D., Zorn, B., Quantifying differences between C and C++ programs, Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994.

[5] Dean, J., Chambers, C., Grove, D., Selective Specialization for Object-Oriented Languages, In Proceeding of the ACM SIGPLAN ’95 Conference on Programming Language Design and Implementation, June 1995.

[6] Dolby, J., Automatic Inline Allocation of Objects, In Proceedings of the 1997 ACM SIGPLAN Conference Programming Language Design and Implementation, Las Vegas, Nevada, June 1997, pp. 7-17.

[7] Ellis, M., Stroustrup, B., The Annotated C++ Reference Manual, Addison-Wesley, 1990.

[8] Hamilton, G. W., Higher-order Deforestation, in Proceedings of the Eighth International Symposium on Programming, Logics, Implementation and Programs (PLILP ’96), Vol. 1140 of Lecture Notes in Computer Science, pp. 213-227.

[9] Plevyak, J., Chien, A., Precise concrete type inference of object-oriented programs, In proceedings of OOPSLA 1994, Object-Oriented Programming Systems, Languages and Architectures, pp. 324-340.

[10] Power B., Hamilton G.W., Declassification: Transforming Java Programs to Remove Intermediate Classes, In the Proceedings of the Workshop on Intermediate Representation Engineering for the Java Virtual Machine (SCI2001/ISAS2001), Vol. VII, 2001, pp. 111-116.

[11] Dolby, j., Chien, A., An evaluation of automatic object inline allocation techniques, In Proceedings of the Thirteenth Annual Conference on Object-Oriented Programming Languages, Systems and Applications (OOPSLA), Vancouver, British Columbia, October 1998. Available at http://www-csag.cs.uiuc.edu/papers/oopsla-98.ps.

[12] Dolby, j., Chien, A., An Automatic Object Inlining Optimization and its Evaluation, Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, Vancouver, British Columbia, Canada, May 2000, pp. 345 – 357.

[13] Laud, P., Analysis for Object Inlining in Java, JOSES: Java Optimization Strategies for Embedded Systems, Genoa, Italy, April 1 2001.

[14] Chilimbi, T., Hill, M., Larus, J., Cache-Conscious Structure Layout, In Proceedings of ACM SIGPLAN’99 Conference on Programming Language Design and Implementation, 1999

[15] Chilimbi, T., Hill, M., Larus, J., Cache-Conscious Structure Definition, In Proceedings of ACM SIGPLAN’99 Conference on Programming Language Design and Implementation, 1999