Java byte-code obfuscation using Stratego

Huib van den Brink
Gerrit van den Geest

Center for Software Technology
Institute of Information and Computing Sciences
Utrecht University

25 April, 2005
1. Introduction

To prevent compiling a distribution of a program for every platform it has to run on, modern imperative language like Java and .NET compile in some intermediate form. For Java this intermediate form is called byte-code and .NET calls it Microsoft Intermediate Language (MSIL). Besides the advantages of compile-once-run-everywhere also secure execution of third-party programs in a sandboxenvironment is possible. But the drawback is that this intermediate format has to contain all or most information that’s also present in the original source code. Since such codes are relative easy to decompile, they increase the risk of malicious reverse engineering attacks.

There exists several ways to protect this code [3]. One way to protect reverse engineering of your code is to not distribute it completely, but only distribute an interface. This can be in the form of a web-service, a web-interface or some other kind of client-server implementation. Although the advantages of this approach are numerous, it’s not always what you want, because of limitations on network bandwidth, latency and availability.

Encryption of code is another possibility. However, unless the entire encryption and decryption process takes place in hardware, it is possible for the user to intercept and decrypt the code. Unfortunately, specialized hardware tends to limit the portability of programs.

Distribute programs in a form less vulnerable to decompilation is another possibility. When native object codes are supplied instead of Java byte-code, the task of decompilation is made more difficult, although not impossible. But when native object codes have to be supplied the above described advantages of an intermediate form such as byte-code are thrown away.

Code obfuscations transform a program to an equivalent one that it is more difficult to understand and reverse engineer. This approach preserves all the advantages of an intermediate compiled form such as byte-code or MSIL, but can never protect an application against all malicious reverse engineering attacks.
2. Some obfuscating transformations

After being obfuscated, the program still must produce the same results, although it may execute slower or have additional side effects, due to added code. There is a trade-off between the security provided by code obfuscation and the execution time-space penalty imposed on the transformed program.

Obfuscation transformations can be classified in the following three categories [3].

2.1. Layout obfuscations

Layout obfuscations are some simple transformations concerning the information provided by the naming of methods, variables, comments and debug info. Most current Java obfuscators target the lexical syntax of the application, such as source code formatting, comments removal, debug info removal and identifier scrambling. After this transformation the program still is executed in exact the same way. Data- and control obfuscation are more interesting transformations.

2.2. Data obfuscations

The obfuscation of data can be divided in three categories, data storage, data aggregation and data ordering. All three affect the manner of how data is accessed within the program.

2.2.1. Storage and encoding

Data encoding involves the use of variables. For instance one could rewrite the use of an variable $i$ in terms of $2i-1$.

```java
int i = 1;
while(i < 10)
{
    users[i] = i++;
}
```

Further, a local variable can be converted to a global and sometimes the other way around. A more advanced variant is the use of aliasing. Several global variables could be created pointing to the same object. Copying the references and using them under different names or setting some of them to null makes it difficult to understand.

Another use in this category is the use of cluttering up the use of standard API’s. The calls to the Java API are a good entry point to understand what happens in that part of the application. Obfuscating these standard calls are not possible without providing your own implementation of the API. However, it is possible to delegate all operations performed on these standard objects. For instance:

```java
Help()
{
    ok = new Button("ok");
    cancel = new Button("cancel");
    //Button createButton(String name)
    Object a_0 (String b_0)
}```
Placing the different operations all scattered throughout the program tends to be confusing.

### 2.2.2. Aggregation

An example of the aggregation of data is converting a two-dimensional-array to a one-dimensional-array.

### 2.2.3. Ordering

In stead of accessing a location of an array, the position also could be retrieved invoking an method returning the correct location.

```java
users[i] -> users[f(i)]
```

### 2.3. Control flow obfuscations

Control obfuscations [6] influences the order the statements are executed in. But this can also mean confusing the programmer by trying to let him think something has a meaning, while it actually has not. Inserting methods similar to each other, without ever invoking them or with invoking it and discarding the result, tend to be very difficult to understand. The cluttering prevents the programmer to be able to understand the whole picture of the application. Understanding an algorithm used is made virtually impossible using this kind of transformations.

#### 2.3.1. Aggregation

Aggregation influences the groupment of the statements. For example method inlining and interleaving methods. With method inlining a method call is replaced with the body of that method leading to lots of duplicate code.

```java
void a_1(Object b_1, Object b_2) { ((Button)b_1).addActionListener((ActionListener)b_2); }
```

Interleaving methods means fusing methods. Methods with the same amount and type of parameters and with the same return type are easy to combine. Let's consider the following example.
The three methods have the same structural type and could therefore easily be combined into one method.

As can be seen in the figure every call is being rewritten to an call with an extra parameter. This parameter is used in the if or switch to execute the right body.

The last technique discussed in this category is the clone method obfuscation. It confuses the programmer by simulating two different calls to two different methods, while in fact both the calls execute the same body.

In the example shown below an class A contains an add method. This method is invoked multiple times. This class and method however could be split up in two. In the result shown below the method and class are split up. Class B however still is of type A. B.add() contains the correct implementation and ensures that A.add() never will be invoked. The if always evaluates to false creating always an B object. A.add() just is an incorrect bogus implementation. Notice that B.add() and A.add2() have the same correct implementation, and hence the name method cloning. Finally the two different methods, with the same implementation, are invoked leading to the same result as the original code.

```java
private String getFirstName(long userId){
    return 7;
}

private String getLastName(long userId){
    return 8;
}

private String getNiceTime(long currentTimeMillis){
    return 9;
}

public String a_0(long b_0, int c_0)
{
    if(c_0 == 0)
        return 7;
    else if(c_0 == 1)
        return 8;
    else
        return 9;
}

As can be seen in the figure every call is being rewritten to an call with an extra parameter. This parameter is used in the if or switch to execute the right body.

The last technique discussed in this category is the clone method obfuscation. It confuses the programmer by simulating two different calls to two different methods, while in fact both the calls execute the same body.

In the example shown below an class A contains an add method. This method is invoked multiple times. This class and method however could be split up in two. In the result shown below the method and class are split up. Class B however still is of type A. B.add() contains the correct implementation and ensures that A.add() never will be invoked. The if always evaluates to false creating always an B object. A.add() just is an incorrect bogus implementation. Notice that B.add() and A.add2() have the same correct implementation, and hence the name method cloning. Finally the two different methods, with the same implementation, are invoked leading to the same result as the original code.

```
2.3.2. Ordering
Affects the order statements are executed in. Loops can be made to iterate backwards instead of forward.

2.3.3. Computation
Control computations are trying to hide the real control flow of an program. This can be achieved by for instance dead code insertion.

Another technique is affecting the flow with for instance loops. Java has no goto statement [4], however there is a Java bytecode goto instruction [5]. Now we can make use of the bytecode goto statement to create a non-reducible control flow graph. Control flow graphs produced from Java programs will always be reducible, but the Java bytecode can express non-reducible flow graphs. Expressing non-reducible flow graphs becomes very awkward in languages without gotos. A decompiler would have to turn a non-reducible flow graph into one with either duplicate code or with extraneous booleans. Turning a structured loop into a loop with multiple headers will achieve this.

2.4. Preventive transformation
This type of transformation doesn’t really obfuscate the source, but tries to attack the decompiler. One example is adding extra instructions after the RETURN byte code instruction. This transformation doesn’t affect the runtime behavior, for it is valid in byte code format, but makes the life of decompilers much harder.

Also the reconstruction of nice for loops can be obstructed by adding extra useless administration.
Another annoying transformation is the use of overloading. In Java methods can have the same name as long as the required parameters are of an different type (or different order of the types) regardless of the return value type [4]. In bytecode this requirement is not applicable, for the return type is also taken in consideration [5]. Therefore the following example is valid in byte code but not in Java code.

```java
public void String getName(String username) // valid
public void String getName(long userId)
public void int     getName(long userId)     // invalid
```

It is needless to say that it would be quite hard for the decompiler to detect this and decide which one should be ignored.
3. Implementation

This chapter describes the implementation of a simple Java byte-code obfuscator implemented in Stratego [7]. First the global architecture is discussed and later each of the different transformations applied is discussed.
We have implemented all layout obfuscations and a aggregation control obfuscation named method interleaving.

3.1. Architecture

The global architecture consists of a pipeline of different Stratego programs. The input of that pipeline is a list of files and the output is a list of files, this is achieved with the strategy xtc-multi-io-wrap. Below the complete source-code of this pipeline.

```plaintext
main =
    xtc-multi-io-wrap(map(xtc-transform("class2aterm", !["-c"]))
    ;map(xtc-transform("relative2label"))
    ;map(read-from)
    ;rename
    ;interleave-methods
    ;map(write-to)
    ;map(xtc-transform("label2relative"))
    ;map(xtc-transform("aterm2class"))
)
```

First the Stratego program class2aterm is mapped over each class-file, so each binary class-file is parsed into the aterm format. Each jump in the Java byte-code is absolute, class2aterm translates these jumps into relative jumps, but the transformations applied by the obfuscator needs labeled jumps. Hence we map the Stratego program relative2label over each file which replaces each relative jump with a labeled jump. Next we read each class-files and apply the obfuscating transformations.
After the transformations each class-file is written to the hard disk, and then the labeled jumps are replaced by relative ones with the label2relative. Finally the different aterms are transformed to binary class-files with aterm2class.
The Stratego programs class2aterm and aterm2class are part of The Dryad Front, available on www.strategoxt.org.

3.2. Layout obfuscations

Some layout information, such as line-numbers, comments and local-variable names are already thrown away by the java-compiler or are only meta-data in the class-file[5] for debugging purposes. Obfuscating this information is very trivial and can be achieved by just removing this meta-information. But global-variable- and method-names are part of the byte-code instructions and renaming those must be done very carefully.

3.2.1. Global-variable renaming

Global-variables in Java are called Fields in byte-code[5], so we use the name Field for a global-variable and FieldRef for the use of a global-variable in the rest of this chapter. Global variable renaming is easy, if you encounter a Field, just rename it and define a
new dynamic rule [1] to rename all FieldRefs of the current class and all its subclasses. But this is to straight-forward because if there is a Field which shadows another Field, two rewrite rules are generated for the FieldRefs, one defined in some parent-class and one in the current class. The simple solution is that if you encounter a Field, first check if there exists a rewrite rule for a Field with the same name, if true use the same new-name, if not generate a new rewrite rule. But if the obfuscator behaves like this shadowing information is still present in the obfuscated code, so this obfuscator only generates rename-rules for sub-classes which doesn’t shadow the current Field. The Stratego-code which achieves this is displayed below.

```
rename-field-ref-dec(|fldnm, tp, newfldnm) =
?class
; <rename-field-ref-dec(|fldnm, tp, newfldnm)>class
; <filter(not(has-field-name(|fldnm)))>
  <bagof-SubClasses>class => subclasses
; <all(shadow-rename-field-ref-dec(|fldnm ,tp ,newfldnm)>subclasses

has-field-name(|fldnm) =
?classname
;<NewFieldName>(classname, fldnm)
```

The name generated by the strategy new possibly shadows a Field from a third-party super-class. The chance is very small, but it is possible, the solution would be to generate a large random number as new-name, this would make the chance of a conflict smaller.

### 3.2.2. Method renaming

Method renaming is more complicated. Because methods that override must get the same new name. The simple solution is again that if you encounter a method, first check if there exists a rewrite rule for a method with the same name, if true use the same new name, if not generate a new rewrite rule. This solution is very powerful, but still preserves some information in the code, for example, it’s still easy to see which methods overriding each other.
This obfuscator applies a more advanced solution, it first checks if there are some unknown classes or interfaces in the inheritance hierarchy, because then it can’t rename any non-private or non-static methods. If not, the obfuscator checks if there are subclasses which import interfaces, the current method implements, then we have to use the obfuscated name of that interface method. Finally the obfuscator checks if the current method overrides another method, if so we have to use the obfuscated name of the overridden method. And otherwise we can generate a name for this method ourselves.

3.3. Control obfuscations

Because of time-constraints the only control obfuscations this obfuscator implements is method-interleaving.

3.3.1. Method-interleaving

We have also implemented a control obfuscation transformation named method interleaving. The idea of method interleaving is that methods of the same signature are interleaved in one method with a kind of switch parameter.

In our implementation only private methods with the same signature are grouped, these methods are then replaced by one method with an extra parameter added. This parameter is used to jump to the instructions of the formal method. The calls to the methods have to be rewritten to invoke the new interleaved function. Also an extra parameter has to be pushed on the stack to jump to the instructions of the formal method.

3.4. Labelling

The call to an interleaved method must be rewritten, because an extra argument must be pushed on the stack to switch to the right instructions in the interleaved method. Therefore an extra load-constant instruction has to be inserted. If the function-call is inside some control flow construction, the relative jumps of that control-flow are not valid anymore.

The bottom-up solution for this problem is to change the relative jumps to labelled jumps at the begin of the pipeline and the other way around at the end. The two Stratego-programs to achieve this are described below.

3.4.1. Relative to Label

The first Stratego-program takes as input the aterm of a class and does a generic traversal. If it encounters a method declaration it scopes the dynamic rule [1] LabelIndex to that method. And if it encounters instructions it first replaces all the relative jumps to labelled jumps and stores the location where a label has to be introduced in the scoped dynamic rule LabelIndex. Finally it introduces the labels into the instructions on the location stored in the LabelIndex rule.
3.4.2. Label to Relative

The second Stratego-program takes as input the aterm of a class and does a generic traversal. If it encounters a method declaration it scopes the dynamic rule `LabelLocation` to that method. And if it encounters instructions it first stores every label and the corresponding index in the scoped dynamic rule `LabelLocation` and immediately removes the label. Finally it iterates the instructions again and replaces every labeled jump with a relative jump, by looking up the label in the `LabelLocation` rule.

```stratego
relative-to-label =
  ?Method(_, _, _, _)
  ; { [LabelIndex : all(relative-to-label) ] }
<+ ?Instructions(_)
  ; ReplaceCounts
  ; Instructions(IntroduceLabels(0))
<+ all(relative-to-label)
```

So this program iterates the instructions twice, first to rewrite the jumps and second to introduce the labels on the right locations. Because jumps can either jump forward or backward this has to be done in two iterations. The complete code of this program can be found and page 14 of this document.

```stratego
label-to-relative =
  ?Method(_, _, _, _)
  ; { [LabelLocation : all(label-to-relative) ] }
<+ ?Instructions(_)
  ; Instructions(RemoveLabels(0))
  ; ReplaceLabels
<+ all(label-to-relative)
```

The complete code of this program can be found at page 15 of this document.
4. Discussion

In a relative small amount of time we achieved a great part of what non-commercial obfuscator basically do. Most of them rename methods and attributes. Sometimes they also do something with the control flow but that’s about it.

One of the things that made us a bit curious is the fact that the university of Auckland has published a lot of papers about obfuscation. And in some of those papers they describe Kava, a Konfusing Java program. However an implementation is nowhere to be found.

StrategoXT provided us means to separate functionality. All the different type of transformations easily could be bundled together, making it scalable and reusable. Even for testing purposes this sometimes was easy. Showing interleaving without doing variable and method renaming and without changing the Stratego code a lot sure saved us a lot of time.

Method inlining is a way to make a program less readable. To illustrate how transformations used in obfuscating could be reused, we show an optimisation used in the HotSpot VM. One of the important optimisations HotSpot performs is inlining frequently-called methods. That means that instead of being invoked with the overhead of a method call, the method's code is copied into the calling routine and executed as though it had been coded there. The smaller the method, the more sense inlining makes. A one-line method might spend more than twice as much time entering and exiting the routine as it does executing.

One of the things that are very important in the subject of obfuscating is the insight of a third party library. This functionality is provided by the Dryad library but is not yet used in this simple obfuscator, adding this functionality should not be a lot of work. This would complement the obfuscation process in an important way.

To complete the current small implementation there still is a lot of work to do. This includes the evaluation of external libraries, class and package renaming, making the current interleaving strategy solid enough to handle all possible situations. Implementing some of the other obfuscation techniques according to metrics given as parameter allowing an amount of performance penalty.

Our conclusion is that StrategoXT definitely is suitable for implementing an obfuscator. Especially the combination of Dryad makes it very useful and powerful. The idea in the beginning was to perform the obfuscation transformations on the bytecode itself. Our conclusion afterwards is that many of the transformations could just as well be performed on the source itself. After doing this project, it is in our believe that all obfuscation transformations as described by the university of Auckland [2] could be realised using StrategoXT.
I. References


II. relative2label.str

```plaintext
/**
 * Replaces relative jump instructions with labeled ones.
 * Not yet supported: LOOKUPSWITCH, TABLESWITCH, ExceptionTable?
 *
 * @author Huib van den Brink <ik@mijnadres.net>
 * @author Gerrit van den Geest <g.vandengeest@students.uu.nl>
 */
module relative2label
imports
  liblib
  dryad/bytecode/signature
strategies

main =
  io-wrap(relative-to-label)

relative-to-label =
  ?Method(_, _, _, _)
    ; (| LabelIndex : all(relative-to-label) |)
  <+ ?Instructions(_)
    ; ReplaceCounts
    ; Instructions(IntroduceLabels(|0))
  <+ all(relative-to-label)

ReplaceCounts :
  Instructions(intrs) -> Instructions(newintrs)
  where <map-with-index(CountToLabel <+ Snd)>intrs => newintrs

CountToLabel :
  (i, jump#([Count(relindex)])) -> jump#([LABEL(label)])
  where <newname>"label" => label
    ; <dec>i => index
    ; <add>(index, relindex) => absindex
    ; rules(LabelIndex : absindex -> LABEL(label))

IntroduceLabels(|index) :
  [instr | intrs] -> newintrs
  where if (<LabelIndex>index => label)
    then ![label, instr | <IntroduceLabels(|<inc>index)>intrs]
    else ![ instr | <IntroduceLabels(|<inc>index)>intrs]
  end => newintrs

IntroduceLabels(|index) :
  [] -> []
```

module label2relative

imports
  liblib
  dryad/bytecode/signature

strategies

main =
  io-wrap(label-to-relative)

label-to-relative =
  ?Method(_, _, _, _)
    ; { | LabelLocation : all(label-to-relative) |}
  <+ ?Instructions(_)          
    ; Instructions(RemoveLabels(|0))          
    ; ReplaceLabels
  <+ all(label-to-relative)

RemoveLabels(|index) :
  [instr| instrs] -> newinstrs
  where if (!instr => LABEL(label))
    then <RemoveLabels(|index)>instrs => newinstrs
           ; rules( LabelLocation : label -> index)
    else ![instr | <RemoveLabels(|<inc>index)>instrs] => newinstrs
  end

RemoveLabels(|index) :
  [] -> []

ReplaceLabels :
  Instructions(instrs) -> Instructions(newinstrs)
  where <map-with-index(LabelToCount <+ Snd)>instrs => newinstrs

LabelToCount :
  (i, jump#([LABEL(label)])) -> jump#([Count(relindex)])
  where <LabelLocation>label => absindex
    ; <dec>i => index
    ; <subt>(absindex, index) => relindex
/**
 * A simple obfuscator implemented in Stratego.
 *
 * @author Huib van den Brink <ik@mijnadres.net>
 * @author Gerrit van den Geest <g.vandengeest@students.uu.nl>
 */

module obf
imports
   liblib
   dryad/bytecode/signature
dryad/lib-ext/multi-options
dryad/lib-ext/xtc/multi-options
rename
interleave-methods

strategies

main =
   xtc-multi-io-wrap(map(xtc-transform(!"class2aterm", !"-c")))
   ;map(xtc-transform(!"relative2label"))
   ;map(read-from)
   ;rename
   ;interleave-methods
   ;map(write-to)
   ;map(xtc-transform(!"label2relative"))
   ;map(xtc-transform(!"aterm2class"))
   )
### V. rename.str

```plaintext
/**
 * Renaming obfuscating transformations
 *
 * Global variable renaming:
 * - In the analysis phase, a dynamic rule is defined for every field
 * - In the declaration phase all fields of a class are renamed
 *   including the fields used in subclasses, except for shadowed
 *   fields.
 * - The renaming of references takes place in the last traversal.
 *
 * Method renaming:
 * - The first traversal (analysis) defines a NewMethodName rule for
 *   every
 *   method to propose a new methodname.
 * - Just rename private methods.
 * - Don't rename methods with unknown implemented interfaces of
 *   subclasses, it could be an implementation.
 * - Don't rename methods with unknown superclasses, it could be an
 *   override.
 * - If it is an implementation or override of a known superclass or
 *   interface
 *   search for the new name of that method in the interface or
 *   superclass.
 * - The renaming of references takes place in the last traversal.
 * @author Huib van den Brink <ik@mijnadres.net>
 * @author Gerrit van den Geest <g.vandengeest@students.uu.nl>
 */

module rename
imports
  liblib
  dryad/bytecode/signature
methods
strategies

rename =
  analyse
  ;rename-dec
  ;do-rename

/**
 * Traversal for the analysis of the class-tree
 *
 */

analyse =
  ?ClassFile(_
    , _
    , _
    , ThisClass(classname)
    , SuperClass(Some(superclassname))
    , Interfaces(interfaces)
    , _
```
rename-dec =
  ThisClass(?classname)
  ; rules(CurrentClass: _ -> classname)
  ; rules(CurrentSubInterfaces: _ -> <get-sub-implemented-interfaces> <bagof-SubClasses>classname)
  ; rules(CurrentSuperClasses: _ -> <reverse><get-super-classes>classname)
  ; rules(CurrentSubClasses: _ -> <get-sub-classes>classname)
<< RenameField
<< RenameMethod
<< all(rename-dec)

RenameField :
  Field(AccessFlags(flags), Name(fldnm), tp, atrs) ->
  Field(AccessFlags(flags), Name(newfldnm), tp, atrs)
where <CurrentClass>() => currentclass
 ; <new>() => newfldnm
 ; if (is-private(|flags))
   then <rename-field-ref-dec(|fldnm, tp, newfldnm)>currentclass
   else <shadow-rename-field-ref-dec(|fldnm , tp , newfldnm)>currentclass
 end

shadow-rename-field-ref-dec(|fldnm, tp, newfldnm) =
  ?class
 ; <rename-field-ref-dec(|fldnm, tp, newfldnm)>class
; <filter(not(field-exists(|fldnm))))>
  <bagof-SubClasses>class => subclasses
; <all(shadow-rename-field-ref-dec(|fldnm
  , tp
  , newfldnm))>subclasses

rename-field-ref-dec(|fldnm, tp, newfldnm) =
  ?class
;rules(RenameFieldRef : FieldRef(Class(class), Name(fldnm), tp) ->
  FieldRef(Class(class), Name(newfldnm), tp))

field-exists(|fldnm) =
  ?classname
;<FieldExists>(classname, fldnm)

RenameMethod :
  Method(AccessFlags(flags), Name(name), desc, body) ->
  Method(AccessFlags(flags), Name(newname), desc, body)
where (not(special-java-method))
; <CurrentClass>() => currentclass
; <CurrentSubInterfaces>() => subinterfaces
; <CurrentSuperClasses>() => superclasses
; <CurrentSubClasses>() => subclasses
; if (is-private(|flags))
  then <NewMethodName>(currentclass, name, desc) => newname
    ; <rename-method-ref-dec(|name
      , desc
      , newname)>currentclass
  else if (<all(ParrentClass <+ ?"java.lang.Object")>subinterfaces)
    then if (<find-defined-methods(|name, desc)>
      subinterfaces => newname)
      then <map(rename-method-ref-dec(|name
        , desc
        , newname
      ))>[currentclass|subclasses]
    else if (<all(ParrentClass
      <+ ?"java.lang.Object")>superclasses)
      then if (<find-defined-methods(|name, desc)>
        superclasses => newname)
      then id
    else <NewMethodName>(currentclass, name, desc) => newname
      ;<map(rename-method-ref-dec(|name
        , desc
        , newname
      ))>[currentclass|subclasses]
  end
end
end
end
end

rename-method-ref-dec(|name, desc, newname) =
  ?class
;rules(RenameMethodRef :
  MethodRef(Class(class), Name(name), desc) ->
MethodRef(Class(class), Name(newname), desc)
RenameMethodRef :
  InterfaceMethodRef(Interface(class), Name(name), desc)
  ->
  InterfaceMethodRef(Interface(class), Name(newname), desc))

/**
 * Traversal for the actual renaming of the References
 * *
 */
do-rename =
  topdown(try(RenameFieldRef + RenameMethodRef))

/**
 * Helper fuctions
 * *
 */
is-private(|flags) =
  <fetch(?Private)>flags

get-super-classes =
  ?name
  ;if ( <ParrentClass>name => parentclass
    ; <ParrentInterfaces>name => parentinterfaces)
    then ![<flatten-list>[parentinterfaces
      ,parentclass
      ,<map(get-super-classes)>parentinterfaces
      ,<get-super-classes>parentclass]]
    else ![]
  end

get-sub-classes =
  ?classname
  ;if (<bagof-SubClasses>classname => subclasses)
    then ![<flatten-list>[subclasses
       ,<map(get-sub-classes)>subclasses]]
    else ![]
  end

find-defined-methods(|name, desc) =
  ?[x|xs]
  ;if (<NewMethodName>(x, name, desc) => newname)
    then !newname
    else <find-defined-methods(|name, desc)>xs
  end

get-sub-implemented-interfaces =
  if (!?sub|subs)
    then ![<flatten-list>[<get-implemented-interfaces>sub
        ,<get-sub-implemented-interfaces><bagof-SubClasses>sub
        ,<get-sub-implemented-interfaces>subs]]
    else ![]
  end

get-implemented-interfaces =
  ?name
;if (<ParrentInterfaces>name => interfaces)
    then <conc>(interfaces
        , <map(get-implemented-interfaces)>interfaces)
    else []
end
module interleave-methods

imports
liblib

strategies

interleave-methods=
    collect-methods
    ; group-by-signature
    ; all(try(declare-interleave-methods))
    ; refactor-calls
    
    /**
     * Strategy for collecting all the private methods.
     *
     */
    
collect-methods =
        ?ThisClass(classname)
        ; rules(CurrentClass: _ -> classname)
        ; all(collect-methods)
        <+ ?Method(AccessFlags(flags), nm, desc, body)
        ; where (is-private(|flags))
        ; <CurrentClass>() => classnm
        ; rules(MethodsOfClass:+ classnm -> Method(AccessFlags(flags), nm, desc, body))
        <+ all(collect-methods)

    /**
     * Group methods of the same class with the same signature
     *
     */
    
group-by-signature =
        ?ThisClass(classname)
        ; rules(MethodsOfClass:classname => methods)
        ; group>methods => groupedmethod
        ; filter(where(<gt>(<length><id>,1))))>groupedmethod => interleavableMethods
        ; map(interleavable-methods-dec(|classname)>interleavableMethods)
        <+ all(group-by-signature)

    interleavable-methods-dec(|classname)>interleavableMethods =
        ?methods
        ; rules(InterleavableMethods:+ classname -> methods)

    group =
        ?[];![]
        <+ ?[Method(flags, nm, desc, body)| xs]
        ; filter(?Method(flags, _, desc, _))> xs => matches
        ; filter(not(?Method(flags, _, desc, _)))> xs => non-matches
        ;![[Method(flags, nm, desc, body)|matches]| <group>non-matches]
/**
 * Generate and introduce new interleaved methods
 */

declare-interleave-methods :
ClassFile(minorv, majorv, flgs, ThisClass(classname), scls, intfs, flds, Methods(methods), atrs) ->
ClassFile(minorv, majorv, flgs, ThisClass(classname), scls, intfs, flds, Methods(<conc>(reducedmethods, imethods)), atrs)
where (rules(CurrentClass: _ -> classname)
; <bagof-InterleavableMethods>classname => im
; <all(try(InterleaveStart))>im => imethods
; <sym-diff>(methods, <concat>im) => reducedmethods)

InterleaveStart :
[Method(flags, methodname, MethodDescriptor(params, ret),
  Attributes([Code(_
    , _
    , Instructions(intrs)
    , ExceptionTable([])
    , Attributes([]))])
]| mthds] ->
Method(flags, Name(newname), MethodDescriptor(newparams, ret),
  Attributes([Code(MaxStack(None)
    , MaxLocals(None)
    , Instructions(newinstr)
    , ExceptionTable([])
    , Attributes([]))])
)

where <new>() => newname
; <CurrentClass>() => classname
; <concat>[params, [Int]] => newparams
; <length>newparams => switchindex
; <map(inclocals(|switchindex, 1))>intrs => newinstr
; !LABEL(<newname>"label") => nextlabel
Interleave(|newname, classname, switchindex, newdesc, nextnum, label) :
[Method(_, methodname, desc, Attributes([Code(_, _, Instructions(intrs), _, _)]))]
  => newinstr
where <map(inclocals(|switchindex, 1))>[label | intrs] => newinstr
  ; <refactor-call-dec(|classname
   , methodname
   , MethodDescriptor(params, ret)
   , newname
   , MethodDescriptor(newparams, ret)
   , 0
   )>()
)

Interleave(|newname, classname, switchindex, newdesc, nextnum, label) :
[Method(_, methodname, desc, Attributes([Code(_, _, Instructions(intrs), _, _)]))]
  => newinstr
where <map(inclocals(|switchindex, 1))>[label | intrs] => newinstr
  ; <refactor-call-dec(|classname
   , methodname
   , desc
   , newname
   , newdesc
   , nextnum
   )>()
)

refactor-call-dec(|classname, methodname, desc, newname, newdesc, switchnum) =
  rules (RefactorCall : INVOKESPECIAL(MethodRef(Class(classname),
    methodname, desc))
    =>
    [LDC(Int(switchnum)),
     INVOKESPECIAL(MethodRef(Class(classname), Name(newname), newdesc))]
)

inclocals(|from, inc) =
InclLocal("ILOAD", from, inc)
<+ InclLocal("LLOAD", from, inc)
<+ InclLocal("FLOAD", from, inc)
<+ InclLocal("DLOAD", from, inc)
<+ InclLocal("ALOAD", from, inc)
<+ InclLocal("ISTORE", from, inc)
<+ InclLocal("LSTORE", from, inc)
<+ InclLocal("FSTORE", from, inc)
IncLocal("DSTORE", from, inc)
IncLocal("ASTORE", from, inc)
IncIINC(from, inc)
inc
d

IncLocal(inc, from, inc) :
  load#([i]) -> load#([<add>(i, inc)]) where <geq>(i, from)

IncIINC(from, inc) :
  IINC(i, inc) -> IINC(<add>(i, inc), inc) where <geq>(i, from)

/**
 * Refactor the calls of the interleaved methods
 * Reset the MaxLocals of method containing the call
 */
refactor-calls =
  toptdown(try(RefactorMethods))

RefactorMethods :
  Method(flags, nm, desc, Attributes([Code(_, ml, instrs,
    ExceptionTable([]), Attributes([])])))
  ->
  Method(flags, nm, desc, Attributes([Code(MaxStack(None), ml,
    newinstrs, ExceptionTable([]), Attributes([])]))
    where <RefactorInstr>instrs => newinstrs

RefactorInstr :
  Instructions(instrs) -> Instructions(new_instrs)
    where <some(RefactorCall);flatten-list>instrs => new_instrs
VII. methods.str

```scala
/**
 * Describes special Java methods, such as methods of class
 * object, the entry point method and (static) constructors.
 *
 * @author Huib van den Brink <ik@mijnadres.net>
 * @author Gerrit van den Geest <g.vandengeest@students.uu.nl>
 */
module methods
imports
dryad/bytecode/signature

strategies

special-java-method =
  constructor + entrypoint-method + object-method

constructor =
  ?Method(
    _,
    Name("<init>")
    ,MethodDescriptor(_, Void)
    , _
  )

constructor =
  ?Method(
    _,
    Name("<clinit>")
    ,MethodDescriptor(_, Void)
    , _
  )

entrypoint-method =
  ?Method(
    AccessFlags([Public, Static])
    ,Name("main")
    ,MethodDescriptor([ArrayType(ObjectType("java.lang.String"))]
      , Void)
    , _
  )

entrypoint-method =
  ?Method(
    AccessFlags([Static, Public])
    ,Name("main")
    ,MethodDescriptor([ArrayType(ObjectType("java.lang.String"))] 
      , Void)
    , _
  )

object-method =
  ?Method(
    _
    , Name("clone")
  )
```

object-method = ?Method(
  , Name("equals")
  , MethodDescriptor([ObjectType("java.lang.Object")], Boolean)
  , -
)

object-method = ?Method(
  , Name("finalize")
  , MethodDescriptor([], Void)
  , -
)

object-method = ?Method(
  , Name("hashCode")
  , MethodDescriptor([], Int)
  , -
)

object-method = ?Method(
  , Name("toString")
  , MethodDescriptor([], ObjectType("java.lang.String"))
  , -
)