Program Transformation Project
The *Extract Method* refactoring in Java

Bogdan Dumitriu
José Pedro Rodrigues Magalhães

April 25, 2005

1 Description

Extract Method is the transformation that deals with selecting a piece of code and extracting it to a new method[1], accordingly removing the old code and invoking the new method. The behavior of the program must be preserved in the transformation, with the objectives being:

- increased readability, since the parent method becomes shorter, and the extracted code's purpose can be easily understood if given a meaningful name;
- reusability, since the extracted method might be useful somewhere else.

Our project deals with the implementation of *Extract Method* in Java, using Stratego [3] as our language of choice.

2 General architecture

2.1 Overview

The refactoring is achieved through a three-state process, namely:

1. Parsing an input Java source code file, which includes marking for what fragment to extract as well as the name for the new method;
2. Performing the actual transformation;
3. Pretty printing the result as Java source code.

The first part is performed with the aid of the SGLR parser fed with the parse table for Java and an extension for delimiting the region to extract.

The second one uses our own code for the transformation as well as components of **Dryad** [1], for ambiguous names reclassification and type checking.

The last part is done with the existing hand-crafted pretty printer for Java.

[1] or function, procedure or similar construct, depending on the language.

2.2 Information to collect

To perform the extraction, the following information must be considered:

- Is the extraction really possible?
- What are the various kinds of variables affected by or involved in the process?
- What exceptions are thrown in the fragment to extract and which of them, if any, are not caught by potential `catch` clauses that appear in the fragment?

2.2.1 Is extraction possible?

Some situations may not allow for extraction to be performed:

1. Does the code contain the markup for the fragment to extract? If not, then the input file did not contain the necessary information for extraction to be done.
2. Does the fragment contain any control-flow instructions? With our implementation, a fragment cannot contain any of the instructions `return`, `break`, `continue` or `label`. Although some configurations could be allowed (like a `return` statement at the end of the fragment or a loop that is fully enclosed in the fragment to extract and contains any number of `break` or `continue` statements), most situations will imply that the normal flow of the program is disrupted from the current block, thus not allowing the extraction (since the fragment can only direct the flow back to the parent method, by means of a `return`).
3. How many variables does the fragment modify? If the fragment modifies — by assignment — more than one external variable (that is, a variable declared before the fragment) which is also used after the fragment, then the extraction cannot be done, since a method can only return one variable. Some solutions would be possible, but they would imply too big a change to the original source code, thus contributing against the main purpose of the refactoring, namely to increase readability. One such solution could be to construct and use a holder class with setters and getters for all the changed variables. Another solution could be to extend the original class by adding some more fields to it, one for each variable and use those for variable passing between methods.

The extraction is possible only if all the conditions above are met.

2.2.2 Variable information

In order to generate the new method, three sets of variables need to be collected:

- Variables used in the fragment, but not assigned to in the fragment. These have to be sent as parameters to the new method;

- Variables whose value is changed by assignment in the fragment and are then used after the fragment as well. If one variable is changed inside the fragment, its value can be returned as the return value of the extracted method and assigned to the original variable in the calling fragment. If two or more variables of this type are changed the refactoring will be disallowed (as we have already discussed above);
- Variables declared in the fragment and used after it. Extracting the fragment to a different method would make the scope of the declaration restricted to that method. The effect is that the usage of such a variable after the fragment would result into a compilation error, due to the fact that the variable would no longer be declared in that scope. The simple solution to this problem is moving the declaration of the variable before the fragment, and passing the variable itself as a parameter to the extracted method.

2.2.3 Exceptions

For exception handling, we need to know all the exceptions that every instruction in the fragment can throw. Furthermore, we need to analyze which of those are already caught in any (possibly nested) `try - catch` clause, not forgetting to consider that catching an exception implies catching all the exceptions from the hierarchy of exceptions rooted in the explicitly caught one. After effectively knowing which exceptions are not caught, we have to add them to the `throws` clause of the new method.

3 Implementation details

3.1 One problem, two implementations

Our implementation actually consists of two separate solutions, although they share some commonalities. There is a naive implementation, the first one we attempted, and afterwards, with the gathered knowledge, we tried to come up with a more efficient and elegant solution, which would achieve the same effect with a single traversal on the program tree. However, since the second, improved version was mostly worked as a "proof of concept" based on the experience of the first one, we have not aimed to fully complete it. Notably, it misses exception handling. Therefore, we supply the two solutions: the first one[2], less efficient and perhaps less readable, but more powerful, and the second one[3], which is meant as a starting point of a reimplementation, but actually goes quite a bit further than that.

3.2 Parsing extended Java syntax

To know what fragment to extract and what name to give to the new method, we extended the syntax for Java with a new symbol (`@`) to mark both the beginning and the end of the piece of code to extract to a new method. The name to be be given to the extracted method is expected as a string after the first `@` symbol.

[2] file `extract-method.str`.
[3] file `better-extract-method.str`.

In terms of implementation, this is essentially a one-line work:

```
"@" Id BlockStm+ "@" -> BlockStm{cons("Extract")}
```

Of course this is followed by the generation of the new parse table (by combining the original Java syntax definition and our extension), and using the new parse table to parse files (instead of a plain invocation to parse-java).

3.3 Dryad issues

Our implementation introduces a new constructor in the Java representation: the Extract(name, code), which is used to name and mark the piece of code to extract. Unfortunately, **Dryad** is not expecting its input files to contain such markup, and thus fails when applied. We were then confronted with the following problem: Dryad cannot easily be extended to support extra constructors, but it had to be applied in order to reclassify ambiguous names and obtain type information. We solved this issue by converting the Extract markup to annotations: every statement inside the block to extract is annotated with a "extract-ann", name, where name is the name of the method to extract, and the first string is just a markup for later recognizing the annotation. After this we can apply dryad-reclassify-ambnames and dryad-type-checker, and then reconvert the annotations back into the Extract constructor.

Another problem with **Dryad** is that it currently does not support field variables distinction. So far, our project works correctly as long as there are no name clashes between the field names and the variable names whose declaration we might have to move up (the third category of variables described above). As soon as **Dryad** will classify field names in a distinguishable way from local variables, the project can be quickly updated with some minor modifications to support this.

3.4 extract-method.str overview

Our first implementation can be found in the file extract-method.str. Basically, it traverses the entire Java code to collect all the necessary information about variables and exceptions, binds all the information in the config hash table and retrieves it any time it needs.

Most strategies are commented in the source code itself, making it easy to understand how the whole information gathering and actual rewriting happens throughout the process.

Perhaps an interesting thing to discuss about this version, especially since it does not also appear in the other one, is the exception handling mechanism. The strategy for exception collection (define-exception-rules) is based on three dynamic rules:

- InTry - this rule will be defined as long as the current traversal point is within a (possibly nested) try block. It's simply a marker to help us know whether we are inside a try block or not;
- TempException - this rule is appended with each exception thrown by a method invocation. Also, every time a try-catch block is exited, but the current point of traversal is still in a higher-level try block, a TempException rule is defined for each uncaught exception;

- PermException - this rule will collect all the uncaught exceptions which will eventually have to appear in the throws clause of the extracted method. A PermException is defined when a try-catch block is exited and the current point of traversal is no longer inside a higher-level try block. All uncaught exceptions thrown in the block are appended to PermException.

You should note that TempException is scoped at the level of each try block, which means that all TempException's are automatically undefined when the try block is exited. The only TempException's which might still be defined after the traversal are the ones which are outside any try block. Therefore, a union of the bagof-TempException and the bagof-PermException is taken after the traversal.

3.5 better-extract-method.str overview

The second implementation, available in the better-extract-method.str file, provides an easier to understand solution, as well as more elegant and efficient. However, is not as complete as the first implementation, as explained before.

After similar "preparation" using Dryad, the bulk of this second approach is contained in a main traversal which deals with all the aspects of the problem. Collection of information is simulated by making heavy use of dynamic rules and rewriting is done either directly where possible, or by applying a dynamic rule defined elsewhere than the place of application, where the information needed for the rewriting is present. Careful scoping takes care that the result is indeed the desired one.

The main traversal has specialized behavior for:

1. Class declaration: if the class contains the fragment, then a new method will have to be added. This will be done by applying the RewriteClass dynamic rule, defined in the handler for the block which contains the fragment to extract (since it is there where we have the information we need for creating this rule).

2. Method declaration: this is only used to scope the VarType rule, since this is the scope for the types of the method parameters and the local variables defined inside the method.

3. Block level: many rules have to be scoped at the block level, and if this block contains the code to extract then this is the point where we have the necessary information to generate the rule to rewrite the class. The block itself also has to be rewritten so that the fragment of code to extract is replaced with a call to the new method. This is done by using the dynamic rule RewriteExtract, mentioned below, when discussing the specialized behavior for the fragment to extract.

4. Fragment to extract: here we define a rule to store the fragment in. This will allow us to retrieve it later, when we need to create the new method. A dynamic rule to rewrite the fragment to a call to the extracted method (possibly preceded by the list of declarations of those variables in the third category described in section 2.2.2) is then defined. This rule will be applied at the level of the block containing the fragment.

5. Variable declaration, use or assignment: here a variable is "added" to the appropriate dynamic rule, depending on its category, as defined in section 2.2.2.
6. Control flow: if control flow instructions are found in the fragment to extract, extraction will stop.

4 General remarks

Scope for variables

Why does moving the declarations of the third kind of variables (that is, variables declared in the fragment to extract and used after it) before the fragment work, you might wonder. The answer lies in the way scoping in defined in Java. We are going to consider the cases here:

- the variable is at the top level (i.e., not within an additional block) in the fragment to extract, which makes it visible in the rest of the fragment, as well as after it. In this case, moving the declaration before the fragment will ensure a slightly larger visibility scope. We have these cases:
  - the variable name is not used between the beginning of the fragment and the declaration. Then, moving its declaration up is harmless.
  - the variable name is used between the beginning of the fragment and the declaration (obviously, referring to another variable). In this case, simply moving the declaration up would render a different semantics, since the use would now refer to the moved up declaration. However, this can easily be solved. We know that the variable's use cannot refer to another local declaration or even parameter name, since Java doesn't allow redeclaration of variable names. Then, the variable must either refer to a field name, may it be static or non-static or to a static variable of another class (since Java 1.5, a class can be statically imported, thus making it possible to refer to a public/protected static variable of that class without prefixing it with the class name). In order to solve this case, we simply have to prefix the field name with `this.` (if it is a non-static field) or with the class name followed by a '.' (if it is a static field). We are currently not doing this, since Dryad doesn't yet offer support for distinguishing field names from local parameters or variables. But it can easily be done, once support is available.
- the variable declaration is nested in an extra block within the fragment. In this case, we can be sure that it won't be used after the fragment since the fragment can only be extracted as a new method if all blocks therein are fully contained inside it and since the scope of a variable declared in a block is limited to that block.

How clear is the code?

We have gone at great length to make our code as clear as possible, by commenting it quite thoroughly, by laying it out in a readable fashion, by separating things into strategies as much as possible, by using meaningful names, by

refactoring things which we realized could be done easier and by various other techniques. Last, but not least, the fact that we have created a brand new extra implementation should stand as proof for our drive to make the code clearer.

The only thing which might make it less clear is the heavy use of dynamic rules (in our second implementation), but given the complexity of the refactoring, there is hardly anything we could do about this.

And, agreed, the first solution might be made a bit more difficult to understand by the combination of a more imperative style of programming with the classic Stratego style, hence the reason for creating our second implementation.

Is Stratego appropriate for the transformation?

This issue can be regarded in two ways. On one hand, Stratego is the perfect tool for implementing such a refactoring, considering the amount of tools which it offers. Not only is parsing and pretty-printing fully covered by java-front, but it also provides a very useful toolset through the Dryad package, that came in extremely handy. We are sure that without such a supporting platform, the work involved by this type of refactoring is a very tedious task, to say the least.

On the other hand, the lack of support for more elaborate debugging combined with lengthy times for compilation of Stratego code to C makes working with Stratego on a project of this scale a living hell. We honestly believe we have spent towards half of the working hours adding `say` or `debug` statements and recompiling the code.

But, to end on a positive note, we are fully convinced that the advantages of Stratego clearly surpass its disadvantages, especially since the latter are mostly due to its current state of development, more than to structural faults.

References

[1] M. Bravenboer. **Dryad — The Tree Nymph** Available from http://catamaran.labs.cs.uu.nl/twiki/pt/bin/view/Stratego/TheDryad

[2] M. Fowler, et al. **Refactoring: Improving the Design of Existing Code (Hardcover).** Addison-Wesley Professional, 1999.

[3] E. Visser. **Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9.** In C. Lengauer et al., editors, Domain-Specific Program Generation, volume 3016 of Lecture Notes in Computer Science, pages 216–238. Spinger-Verlag, June 2004.

A File extract-method.str

```
module extract-method
imports

  dryad/structure/*
  dryad/jls/types/Subtyping
  dryad/type-check/*

  liblib

  JavaExtractMethod

signature
  constructors
    AfterExtract : List(A) -> B
```

```
    MarkedMethod : A -> B
    MarkedClass  : A -> B
    NoInit       : A
    Exceptions   : List(A) -> B

strategies

  /**
   * The main strategy. Calls sglri with the new parse table for Java
   * (including the notation for code to be extracted), then removes the
   * Extract markup to make it usable by dryad, then applies
   * dryad-reclassify-ambnames and then puts back the Extract markup.
   * After that, it extracts the fragment to a new method, if possible.
   * Rewriting will fail if extraction is not possible.
   */
  main =
    xtc-io-wrap(xtc-transform(!"sglri", ![" -p", "NewJava.tbl", "-s", "
         CompilationUnit"])
               ; read-from
               ; strip-extract
               ; write-to
               ; xtc-transform(!"dryad-reclassify-ambnames")
               ; xtc-transform(!"dryad-type-checker")
               ; read-from
               ; add-extract
               ; observables-wrap(extract-method)
               ; write-to
    )

strategies

  /**
   * Takes away the Extract markup and annotates the statements which are part
   * of the fragment to extract so that they can be restored later, after dryad
   * has been run on the code.
   */
  strip-extract =
    topdown(try(AnnotateExtract))

  /**
   * Puts back the Extract markup based on the annotations created by the
   * strip-extract strategy and adds an AfterExtract markup for the statements
   * that come after the fragment to extract.
   */
  add-extract =
    topdown(try(RestoreExtract))

  /**
   * Annotates all the statements in the body of an Extract, given the ATerm
   * marked by Extract.
   */
  annotate-extract =
    ?Extract(Id(theId), body)
    ; <map(\ x -> x {"extract-ann", theId} \)>body

  /**
   * Restores the Extract markup and adds the AfterExtract markup as well, given
   * the block in which the annotated statements are.
   */
  restore-extract(s) =
    where (filter(s) => extractStmts)
    ; where (split-fetch-keep(s) => (before, split {"extract-ann", theId}, after))
    ; <concat>[ before,
                [Extract(Id(theId), <map(\ x {_, _} -> x{} \)>extractStmts)],
                [AfterExtract(<drop(<length>extractStmts; dec)>after)]
              ]

rules

  AnnotateExtract :
    Block(body) -> Block(newBody)
    where <split-fetch-keep(annotate-extract)>body => (before, split, after)
        ; <concat>[before, split, after] => newBody

  AnnotateExtract :
    ConstrBody(x, body) -> ConstrBody(x, newBody)
    where <split-fetch-keep(annotate-extract)>body => (before, split, after)
        ; <concat>[before, split, after] => newBody

  RestoreExtract :
    Block(body) -> Block(newBody)
    where <restore-extract(?_ {"extract-ann", _})>body => newBody

  RestoreExtract :
    ConstrBody(x, body) -> ConstrBody(x, newBody)
    where <restore-extract(?_ {"extract-ann", _})>body => newBody

strategies

  /**
   * The main strategy for method extraction.
   */
  extract-method =
    mark-and-extract
    ; where(collect-various-info)
    ; where(change-var-lists)
    ; where(extraction-possible)
    ; perform-extraction
```

```
strategies

  /**
   * A custom traversal of a compilation unit which:
   *
   * - puts the Extract & AfterExtract ATerms in the config hash table.
   * - marks the method and the class which contain the fragment to be
   *   extracted with the MarkedClass & MarkedMethod constructors.
   */
  mark-and-extract =
       ?Extract(_, _)
       ; rules (ContainsExtract : x -> x)
       ; where(<set-config>("extract", <id>))

    <+ ?AfterExtract(_)
       ; where(<set-config>("afterExtract", <id>))

    <+ (?ConstrDec(_, _) + ?MethodDec(_, _))
       ; {| ContainsExtract :
              all(mark-and-extract)
              ; if ContainsExtract then
                  where(<set-config>("method", <id>))
                  ; !MarkedMethod(<id>)
                  ; rules (ContainsMarkedMethod : x -> x)
                else
                  id
                end
         |}

    <+ ?ClassDec(_, _)
       ; {| ContainsMarkedMethod :
              all(mark-and-extract)
              ; if ContainsMarkedMethod then
                  !MarkedClass(<id>)
                else
                  id
                end
         |}

    <+ all(mark-and-extract)

/**
 * Strategies related to collection of information about variables, exceptions
 * and so on.
 */
strategies

  collect-various-info =
    <get-config>"extract"
    ; <set-config>("assignedToVars", <assigned-to-variables>)
    ; <get-config>"extract"
    ; <set-config>("usedVars", <used-variables>)
    ; <get-config>"afterExtract"
    ; <set-config>("usedAfterVars", <used-variables>)
    ; <get-config>"method"
    ; <set-config>("types", <get-var-types>)
    ; <get-config>"extract"
    ; <set-config>("declaredVars", <declared-variables>)
    ; <get-config>"extract"
    ; <set-config>("exceptions", <get-exceptions>)

  change-var-lists =
    <get-config>"usedVars" => uv
    ; <get-config>"assignedToVars" => av
    ; <get-config>"usedAfterVars" => uav
    ; <get-config>"declaredVars" => dv

    ; <diff>(dv, uav) => dv'
    ; <diff>(uv, dv') => uv'
    ; <diff>(av, dv') => av'
    ; <isect>(av', uav) => av''
    ; <isect>(uav, dv) => uav'

    ; <set-config>("usedVars", uv')
    ; <set-config>("assignedToVars", av'')
    ; <set-config>("usedAfterVars", uav')

/**
 * Strategies related to exception handling.
 */
strategies

  /**
   * Returns a list of all the exceptions thrown in the fragment to which
   * it is applied.
   */
  get-exceptions =
    define-exception-rules
    ; <union>(<bagof-TempException>(), <bagof-PermException>())

  /**
   * A custom traversal which looks for exceptions and defines dynamic
   * rules for them.
   */
  define-exception-rules =
       ?Try(block, catches)
       ; {| TempException, InTry :
            rules(InTry : () -> ())
            ; Try(define-exception-rules, id)
```

```
            ; where( <map(?Catch(Param(_, <id>, _)), _))>catches
                     ; eliminate-caught-exceptions
                     ; ?exceptions
                   )
      |}
    ; Try(id, define-exception-rules)
    ; where(   <InTry>()
              < <map(\ x -> x where rules(TempException :+ () -> x ) \)>exceptions
              + <map(\ x -> x where rules(PermException :+ () -> x ) \)>exceptions
            )

  <+ ?Try(block, catches, finally)
    ; {| TempException, InTry :
          rules(InTry : () -> ())
          ; Try(define-exception-rules, id, id)
          ; where( <map(?Catch(Param(_, <id>, _)), _))>catches
                   ; eliminate-caught-exceptions
                   ; ?exceptions
                 )
      |}
    ; Try(id, define-exception-rules, define-exception-rules)
    ; where(   <InTry>()
              < <map(\ x -> x where rules(TempException :+ () -> x ) \)>exceptions
              + <map(\ x -> x where rules(PermException :+ () -> x ) \)>exceptions
            )

  <+ ?Invoke(methodid, args)
    ; Invoke(id, define-exception-rules)
    ; where(get-exceptions-of-invocation)

  <+ all(define-exception-rules)

/**
 * Defines TempException dynamic rules for all the exceptions thrown
 * by a method invocation.
 */
get-exceptions-of-invocation =
  ?e@Invoke(methodid, _)
  ; <dryad-tc-search-class-of-method>methodid => class
  ; <determine-method-signature(|class)>e
  ; get-exceptions-of-method
  ; map(\ x -> x where rules(TempException :+ () -> x) \)

/**
 * Returns a list with all the exceptions thrown by a method.
 */
get-exceptions-of-method =
  ?Method(_, _, _, Attributes(<fetch(?Exceptions(es))>))
  ; !es
  ; map(?Class(<id>); <bytecode-type-to-source-type>ObjectType(<id>))
  <+ ![]

eliminate-caught-exceptions =
  ?caughtExceptions
  ; <bagof-TempException>()
  ; filter(not(caught(|caughtExceptions)))

caught(|caughtExceptions) =
  ?exception
  ; <fetch({caughtException: ?caughtException; <is-subtype(|caughtException)>
       exception})>caughtExceptions

strategies

/**
 * A custom traversal of the compilation unit which performs the actual method
 * extraction.
 */
perform-extraction =
    ?MarkedClass(<id>)
    ; all(perform-extraction)
    ; ?ClassDec(cHead, ClassBody(cBody))
    ; <get-config>"extract" => Extract(Id(x), stmts)
    ; <get-config>"assignedToVars" => av
    ; (<eq>(<length>av, 1) < !av => [y]; <conc>(stmts, [ |[ return y; ]| ]) + !
        stmts) => emBody
    ; (<eq>(<length>av, 1) < !av => [y]; <lookup>(y, <get-config>"types") => (t,
        _) + !Void() => t)
    ; <get-config>"usedVars"
    ; map(\ x -> Param([], <Fst>(<lookup>(x, <get-config>"types")), Id(x)) \) =>
        param*
    ; <process-declarations>emBody => bstm*
    ; <get-config>"exceptions" => exceptions
    ; (<?[]>exceptions
      < !MethodDec(MethodDecHead([Private()], None(), t, Id(x), [param*], None),
          Block([ bstm* ])) => em
      + !MethodDec(MethodDecHead([Private()], None(), t, Id(x), [param*], Some(
          ThrowsDec(exceptions))), Block([ bstm* ])) => em
      )
    ; <conc>(cBody, [em]) => newCBody
    ; !ClassDec(cHead, ClassBody(newCBody))

  <+ ?ClassDec(_, _)
    // this is to avoid unnecessary traversal

  <+ ?MarkedMethod(<id>)
    ; all(perform-extraction)

  <+ ?MethodDec(_, _)
```

```
        // this is to avoid unnecessary traversal

    <+ ?Block(_)
        ; all(perform-extraction)
        ; ?Block(<id>)
        ; !Block(<flatten-list>)

    <+ ?ConstrBody(_, _)
        ; all(perform-extraction)
        ; ?ConstrBody(x, body)
        ; !ConstrBody(x, <flatten-list>body)

    <+ ?Extract(Id(x), _)
        ; <get-config>"assignedToVars" => av
        ; <get-config>"usedAfterVars" => uav
        ; <get-config>"usedVars"
        ; map(\ x -> ExprName(Id(x)) \) => e*
        ; <define-declarations>uav => decs
        ; (     <eq>(<length>av, 0)
                ; <conc>(decs, [ |[ x(e*); ]| ])
            <+ !av => [y]
                ; <conc>(decs, [ |[ y = x(e*); ]| ])
          )

    <+ ?AfterExtract(<id>)

    <+ all(perform-extraction)

  define-declarations =
    where(<get-config>"types" => typeList)
    ; map(\ x -> [ |[ ~<Fst>(<lookup>(x, typeList)) x; ]| ] \)

  process-declarations =
    map(?LocalVarDecStm(<id>) < generate-new-declarations + id)
    ; flatten-list

  generate-new-declarations =
    where(<get-config>"usedAfterVars" => uav)
    ; var-dec
    ; map({x, t, e: ?(x, (t, e))
          ; ( <elem>(x, uav)
              < (<?NoInit()>e < ![] + !|[ x = e; ]|)
              + (<?NoInit()>e < !LocalVarDecStm(lvdec|[ t x ]|) + !LocalVarDecStm(
                  lvdec|[ t x = e ]|))
            )
        })
    ; filter(not(?[]))

/**
 * Strategies related to collection of variables.
 */
strategies

  /**
   * Collects all the variables which are changed (by assigning to them) in
   * a fragment of code.
   */
  assigned-to-variables =
    generic-collect-vars(var-assign)

  /**
   * Collects all the variables which are used in a fragment of code.
   */
  used-variables =
    generic-collect-vars(var-use)

  /**
   * This is a generic strategy to collect free variables (i.e., those not
   * defined in the piece of code to which the strategy is applied) which match
   * the strategy var-match. This strategy should return a singleton list
   * containing the variable. The strategy can be applied to any piece of code.
   */
  generic-collect-vars(var-match) =
    where (new-iset => set)
    ; topdown(try(where(add-var(var-match | set))))
    ; <iset-elements>set => collectedVars
    ; <iset-destroy>set
    ; !collectedVars

  /**
   * Collects all the variables which are declared in a fragment of code.
   */
  declared-variables =
    where (new-iset => set)
    ; topdown(try(where(add-if-declared(|set))))
    ; <iset-elements>set => collectedDecs
    ; <iset-destroy>set
    ; !collectedDecs

  /**
   * If the var-dec strategy succeeds on the current term, then it adds each
   * variable in the list returned by var-dec to the set.
   */
  add-if-declared(|set) =
    where (var-dec => varDecs)
    ; <map(\
        (var, (type, init)) -> (var, (type, init))
        where <iset-add(|var)>set
      \)>varDecs
```

```
  /**
   * If the var-match strategy succeeds on the current term, then it adds the
       variable
   * returned by var-match to the set.
   */
  add-var(var-match | set) =
    var-match
    ; ?[x]
    ; <iset-add(|x)>set

  /**
   * Succeeds if applied to an assignment to a variable and returns the variable
   * name to which the assignment is made as a singleton list.
   */
  var-assign =
    ?Assign(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignMul(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignDiv(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignRemain(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignPlus(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignMinus(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignLeftShift(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignRightShift(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignURightShift(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignAnd(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignExcOr(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignOr(ExprName(Id(<make-singleton-list>)), _)
    <+ ?VarDec(Id(<make-singleton-list>), _)

  /**
   * Succeeds if applied to the usage of a variable and returns the used
   * variable as a singleton list.
   */
  var-use =
    ?ExprName(Id(<make-singleton-list>))

  make-singleton-list =
    ![<id>]

  /**
   * Returns a list containing (var, type) tuples with all the variables
   * declared in the ATerm the strategy is applied to.
   */
  get-var-types =
    collect-all(var-dec, conc, SkipNewScopes)
    ; flatten-list

  /**
   * If applied to a variable declaration, it returns a list of
   * (var, (type, initExpr)) tuples.
   */
  var-dec =
      ?LocalVarDec(_, type, varDecs)
      ; <get-var-type-tuples>(type, varDecs)
    <+ ?MethodDecHead(_, _, _, _, paramDecs, _)
      ; <get-param-type-tuples>paramDecs
    <+ ?ConstrDecHead(_, _, _, cparamDecs, _)
      ; <get-param-type-tuples>cparamDecs

  /**
   * Takes of tuple containing a type and a list of variable declarations and
   * returns a list of (var, type) tuples using the same type (received as a
   * parameter) for each of the variables in the list of variable declarations
   * (also received as a parameter).
   */
  get-var-type-tuples =
    ?(type, varDecs)
    ; <map(VarType(|type))>varDecs

  /**
   * Takes a list of parameter declarations and returs a simplified list which
   * only contains tuples with the names of the defined parameters and their
   * types.
   */
  get-param-type-tuples =
    map(ParamType)

rules

  VarType(|type) :
    VarDec(Id(x)) -> (x, (type, NoInit()))

  VarType(|type) :
    VarDec(Id(x), init) -> (x, (type, init))

  ParamType :
    Param(_, type, Id(x)) -> (x, (type, NoInit()))

  // for skipping anonymous classes
  SkipNewScopes :
    NewInstance(_, _, _, _) -> []

/**
 * Strategies for sanity checks on the fragment to extract.
 */
strategies

  /**
```

```
    * The strategy succeeds if the extract fragment can indeed be extracted. This
    * means that:
    *
    * - there are no control flow statements in the fragment (return, break,
    *   continue or label definitions).
    * - there is at most one single free variable to which an assignment is
    *   made.
    */
  extraction-possible =
    <get-config>"extract"
    ; (not(has-control-flow) <+ (say(!"\nExtraction cannot be done because of
        control flow.\n"); <exit>1))
    ; <get-config>"assignedToVars"
    ; (<leq>(<length>, 1) <+ (say(!"\nExtraction cannot be done because of more than
        one assignment to a variable in the fragment to extract.\n"); <exit>1))

  /**
   * The strategy succeeds if there are control flow statements (return, break,
   * continue or label definitions) in the term to which it is applied.
   */
  has-control-flow =
      ?Return(_)
    <+ ?Break(_)
    <+ ?Continue(_)
    <+ ?Labeled(_, _)
    <+ some(has-control-flow)
```

B File **better-extract-method.str**

```
module better-extract-method
imports

  dryad/structure/*
  dryad/jls/types/Subtyping
  dryad/type-check/*

  liblib

  JavaExtractMethod

signature
  constructors
    NoInit        : A

strategies

  /**
   * The main strategy. Calls sglri with the new parse table for Java
   * (including the notation for code to be extracted), then removes the
   * Extract markup to make it usable by dryad, then applies
   * dryad-reclassify-ambnames and then puts back the Extract markup.
   * After that, it extracts the fragment to a new method, if possible.
   * Rewriting will fail if extraction is not possible.
   */
  main =
    xtc-io-wrap(xtc-transform(!"sglri", ![-p", "NewJava.tbl", "-s", "
        CompilationUnit"])
              ; read-from
              ; strip-extract
              ; write-to
              ; xtc-transform(!"dryad-reclassify-ambnames")
              ; xtc-transform(!"dryad-type-checker")
              ; read-from
              ; add-extract
              ; where (prepare-rules)
              ; observables-wrap(extract-method)
              ; write-to
    )

strategies

  /**
   * Takes away the Extract markup and annotates the statements which are part
   * of the fragment to extract so that they can be restored later, after dryad
   * has been run on the code.
   */
  strip-extract =
    topdown(try(AnnotateExtract))

  /**
   * Puts back the Extract markup based on the annotations created by the
   * strip-extract strategy and adds an AfterExtract markup for the statements
   * that come after the fragment to extract.
   */
  add-extract =
    topdown(try(RestoreExtract))

  /**
   * Annotates all the statements in the body of an Extract, given the ATerm
   * marked by Extract.
   */
  annotate-extract =
```

```
    ?Extract(Id(theId), body)
    ; <map(\ x -> x {"extract-ann", theId} \)>body

  /**
   * Restores the Extract markup and adds the AfterExtract markup as well, given
   * the block in which the annotated statements are.
   */
  restore-extract(s) =
    where (filter(s) => extractStmts)
    ; where (split-fetch-keep(s) => (before, split {"extract-ann", theId}, after))
    ; <concat>[ before,
                [Extract(Id(theId), <map(\ x {_, _} -> x{} \)>extractStmts)],
                <drop(<length>extractStmts; dec)>after
              ]

rules

  AnnotateExtract :
    Block(body) -> Block(newBody)
    where <split-fetch-keep(annotate-extract)>body => (before, split, after)
        ; <concat>[before, split, after] => newBody

  AnnotateExtract :
    ConstrBody(x, body) -> ConstrBody(x, newBody)
    where <split-fetch-keep(annotate-extract)>body => (before, split, after)
        ; <concat>[before, split, after] => newBody

  RestoreExtract :
    Block(body) -> Block(newBody)
    where <restore-extract(?_ {"extract-ann", _})>body => newBody

  RestoreExtract :
    ConstrBody(x, body) -> ConstrBody(x, newBody)
    where <restore-extract(?_ {"extract-ann", _})>body => newBody

strategies

  /**
   * Prepares the dynamic rules, which are used to simulate lists, by
   * making them rewrite to the empty list.
   */
  prepare-rules =
    rules (UsedVars : () -> [])
    ; rules (AfterUsedVars : () -> [])
    ; rules (AssignedToVars : () -> [])
    ; rules (DeclaredVars : () -> [])

  /**
   * This is the main traversal, properly handling all cases where rules need
   * to be either scoped, defined or applied. See the documentation for a more
   * detailed explanation.
   */
  extract-method =
      ?ClassDec(_, _)
      ; handle-class

    <+ ?MethodDec(_, _)
      ; handle-method

    <+ ?Block(_)
      ; handle-block

    <+ ?Extract(_, _)
      ; handle-extract

    <+ var-dec
      ; handle-var-dec
      ; fail

    <+ var-use
      ; handle-var-use
      ; fail

    <+ var-assign
      ; handle-var-assign
      ; fail

    <+ is-control-flow
      ; handle-control-flow

    <+ all(extract-method)

  /**
   * The strategy that handles a class declaration.
   */
  handle-class =
    {| RewriteClass:
       all(extract-method)
     ; try(RewriteClass)
    |}

  /**
   * The strategy that handles a method declaration.
   */
  handle-method =
    {| VarType :
       all(extract-method)
    |}
```

```
/**
 * The strategy that handles a block.
 */
handle-block =
  {| InBlock, CurLabel, IsDeclared, ContainsExtract, InAfterExtract :
    where(new => label)
    ; rules(ContainsExtract :- ())
    ; rules(InBlock+label)
    ; rules(CurLabel : () -> label)
    ; all(extract-method)
    ; ?block
    ; try(<ContainsExtract>()
        ; change-var-lists
        ; define-rewrite-class-rule
        ; <rewrite-block>block
      )
  |}

/**
 * This strategy takes all the collected variable lists and transforms them
 * to what is actually needed to the rewrite process, through differences
 * and intersections of sets.
 */
change-var-lists =
  <UsedVars>() => uv
  ; <AssignedToVars>() => av
  ; <AfterUsedVars>() => auv
  ; <DeclaredVars>() => dv

  ; <diff>(dv, auv) => dv'
  ; <diff>(uv, dv') => uv'
  ; <diff>(av, dv') => av'
  ; <isect>(av', auv) => av''
  ; <isect>(auv, dv) => auv'

  ; rules(UsedVars : () -> uv')
  ; rules(AssignedToVars : () -> av'')
  ; rules(AfterUsedVars : () -> auv')

/**
 * This strategy defines the rule that rewrites the class (inserting the new
 * method).
 */
define-rewrite-class-rule =
  <ExtractedCode>() => (x, stmts)
  ; <AssignedToVars>() => av
  ; (<eq>(<length>av, 1) < !av => [y]; <conc>(stmts, [ |[ return y; ]| ]) + !stmts
      ) => emBody
  ; (<eq>(<length>av, 1) < !av => [y]; <VarType>y => (t, _) + !Void() => t)
  ; <AfterUsedVars>() => uav
  ; <UsedVars>()
  ; map(\ x -> Param([], <(VarType; Fst)>x, Id(x)) \) => param*
  ; <process-declarations>emBody => bstm*
  ; !|[ private t x(param*) { bstm* } ]| => em
  ; rules(
      RewriteClass :
        ClassDec(chead, ClassBody(cBody)) -> ClassDec(chead, ClassBody(newCBody))
        where <conc>(cBody, [em]) => newCBody
    )

/**
 * This strategy is used to decide if there is any variable declaration
 * inside the fragment that need to be moved up, because it is used after
 * the fragment.
 */
process-declarations =
  map(?LocalVarDecStm(<id>) < generate-new-declarations + id)
  ; flatten-list

/**
 * In conjunction with the previous strategy, this strategy will break
 * multiple variable declarations in a single line to multiple lines,
 * at the same time replacing a declaration + assignment with just the
 * assignment if that variable declaration has been moved up.
 */
generate-new-declarations =
  where(<AfterUsedVars>() => auv)
  ; var-dec
  ; map({x, t, e: ?(x, (t, e))
        ; ( <elem>(x, auv)
            < (<?NoInit()>e < ![] + !|[ x = e; ]|)
            + (<?NoInit()>e < !LocalVarDecStm(lvdec|[ t x ]|) + !LocalVarDecStm(
                lvdec|[ t x = e ]|))
          )
    })
  ; filter(not(?[]))

/**
 * The strategy applied to a block to rewrite it.
 */
rewrite-block =
  where(<AssignedToVars>() => av)
  ; where(<UsedVars>() => uv)
  ; topdown(try(RewriteExtract(|av, uv)))
  ; Block(flatten-list)

/**
 * The strategy that handles the fragment to extract.
 */
```

```
  handle-extract =
    {| InExtract :
      ?Extract(Id(x), stmts)
      ; rules(ExtractedCode : () -> (x, stmts))
      ; rules(InExtract : () -> ())
      ; rules(ContainsExtract : () -> ())
      ; all(extract-method)
      ; rules(InAfterExtract : () -> ())
      ; rules(RewriteExtract(|av, uv) : Extract(_, _) -> dec-call
              where <length>av => nrAv
                  ; <map(\ x -> ExprName(Id(x)) \)>uv => e*
                  ; (     (<eq>(nrAv, 0)
                           ; ![ x(e*); ]| => dec-call)
                    <+ (<eq>(nrAv, 1)
                        ; <AfterUsedVars>()
                        ; map(CreateDecs) => decs
                        ; !av => [y]
                        ; <conc>(decs, [ |[ y = x(e*); ]| ]) => dec-call)
                    <+ say(!"\nExtraction cannot be done because of more than one
                        assignment to a variable in the fragment to extract.\n")
                        ; <exit>1
                  )
        )
    |}

  /**
   * The strategy that handles a variable declaration.
   */
  handle-var-dec =
    map(declare-var-dec-rules)
    ; all(extract-method)

  /**
   * This strategy defines the VarTye for a given variable and also adds it to
   * the list of DeclaredVars, if we are inside the fragment to extract.
   */
  declare-var-dec-rules =
    ?(x, (t, init))
    ; rules(VarType : x -> (t, init))
    ; try(<InExtract>(); <DeclaredVars>() => dv; rules (DeclaredVars : () -> <union
        >(dv, [x])))

  /**
   * The strategy that handles a variable use.
   */
  handle-var-use =
    ?[x]
    ; try(<InExtract>(); <UsedVars>() => uv; rules (UsedVars : () -> <union>(uv, [x
        ])))
    ; try(<InAfterExtract>(); <AfterUsedVars>() => auv; rules (AfterUsedVars : () ->
         <union>(auv, [x])))

  /**
   * The strategy that handles a variable assignment.
   */
  handle-var-assign =
    ?[x]
    ; try(<InExtract>(); <AssignedToVars>() => av; rules (AssignedToVars : () -> <
        union>(av, [x])))

  /**
   * This strategy checks for the existance of control flow inside the
   * fragment, and exits with error code 1 if it happens to find it.
   */
  handle-control-flow =
    <InExtract>()
    ; say(!"\nExtraction cannot be done because of control flow.\n")
    ; <exit>1

rules

  CreateDecs : x -> |[ t x; ]|
    where VarType => (t,_)

strategies

  /**
   * Succeeds if applied to an assignment to a variable and returns the variable
   * name to which the assignment is made as a singleton list.
   */
  var-assign =
    ?Assign(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignMul(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignDiv(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignRemain(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignPlus(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignMinus(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignLeftShift(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignRightShift(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignURightShift(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignAnd(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignExcOr(ExprName(Id(<make-singleton-list>)), _)
    <+ ?AssignOr(ExprName(Id(<make-singleton-list>)), _)
    <+ ?VarDec(Id(<make-singleton-list>), _)

  /**
   * Succeeds if applied to the usage of a variable and returns the used
   * variable as a singleton list.
   */
```

```
  var-use =
    ?ExprName(Id(<make-singleton-list>))

  make-singleton-list =
    ![<id>]

  /**
   * Returns a list containing (var, type) tuples with all the variables
   * declared in the ATerm the strategy is applied to.
   */
  get-var-types =
    collect-all(var-dec, union, SkipNewScopes)
    ; flatten-list

  /**
   * If applied to a variable declaration, it returns a list of
   * (var, (type, initExpr)) tuples.
   */
  var-dec =
      ?LocalVarDec(_, type, varDecs)
      ; <get-var-type-tuples>(type, varDecs)
    <+ ?MethodDecHead(_, _, _, _, paramDecs, _)
      ; <get-param-type-tuples>paramDecs
    <+ ?ConstrDecHead(_, _, _, cparamDecs, _)
      ; <get-param-type-tuples>cparamDecs

  /**
   * Takes of tuple containing a type and a list of variable declarations and
   * returns a list of (var, type) tuples using the same type (received as a
   * parameter) for each of the variables in the list of variable declarations
   * (also received as a parameter).
   */
  get-var-type-tuples =
    ?(type, varDecs)
    ; <map(VarType(|type))>varDecs

  /**
   * Takes a list of parameter declarations and returs a simplified list which
   * only contains tuples with the names of the defined parameters and their
   * types.
   */
  get-param-type-tuples =
    map(ParamType)

rules

  VarType(|type) :
    VarDec(Id(x)) -> (x, (type, NoInit()))

  VarType(|type) :
    VarDec(Id(x), init) -> (x, (type, init))

  ParamType :
    Param(_, type, Id(x)) -> (x, (type, NoInit()))

  // for skipping anonymous classes
  SkipNewScopes :
    NewInstance(_, _, _, _) -> []

strategies

  /**
   * The strategy succeeds if the term to which it is applied is a control flow
   * statement (return, break, continue or label definition).
   */
  is-control-flow =
      ?Return(_)
    <+ ?Break(_)
    <+ ?Continue(_)
    <+ ?Labeled(_, _)
```