Title:	Roles of variables and program analysis
Candidate:	Craig Bishop
Degree:	MSc Computer Science
Year:	2005

Title:	Roles of variables and program analysis
Candidate:	Craig Bishop
Degree:	MSc Computer Science
Year:	2005
Numbers of words:	12,000

Abstract

Roles of Variables [1] is a new concept that can help novice programmers comprehension of the programming paradigm. This paper presents an extension to the BlueJ Java development environment that checks the roles played by primitive type variables in simple Java programs. The program uses data flow analysis of Java source code files to check the roles. The results, though based on a limited set of training data, show that data flow analysis can be used successfully to check the roles of variables where there is a sufficiently comprehensive bank of role checking rules. The software presented can be used as is with the training programs to help familiarise students with the roles of variables concept. The modular nature of the software means that it can be further trained as may be required to reliably recognize roles being played by variables in a wider range of Java programs.

Acknowledgements

Thank you to my supervisor Colin Johnson for his advice and encouragement during this project. I would also like to thank Ian Utting for his valuable explanation of how BlueJ extensions work. Special thanks to my wife Emma for her great support and patience throughout the past year, and to my children Elizabeth and Gabrielle for stopping me from spending too much time at the computer.

Contents

1 Introduction

Many students find software programming difficult to learn. Kuittinen & Sajaniemi suggest in [1] that this is in part due to the fact that programs deal with abstract entities that have little in common with everyday issues. Further they state that previous efforts to ease and enhance learning have concentrated mainly on improved teaching methods and presentation of materials. Roles of Variables is thus presented in [1] as a new concept that seeks to improve students' "programming knowledge" [2], by directly teaching how to construct programs from abstract concepts. The addition of visualisation and animation to the rules concept helps to reinforce students' knowledge [5], [9]. Furthermore, it introduces the possibility to use automatic detection of variable roles for the purposes of reverse engineering of more complex programs through automatic generation of program animations [11].

This project aims to develop a program that can be used to support the teaching of programming using Roles of Variables. The program does this by providing analysis and comment on the roles declared by programmers for given variables. The input to the program is compiled Java source code together with information from the programmer about the perceived role of each variable. The output from the program will be either confirmation that all variables are correctly playing their intended roles, or that there appears to be one or more roles declared incorrectly for one or more of the variables. In the event of an incorrect role being detected, the program will also provide justification for its assertion, the offending statement (within the source code) that leads to the assertion, and a suggestion for a more appropriate role.

In [11], Gerdt & Sajaniemi state that the cognitive as opposed to technical nature of Roles of Variables presents significant challenges in developing software, which can detect the roles being played by variables with a high degree of accuracy and consistency. The aim of the project is however, not to try to develop > 95% reliable role checking rules at the expense of working software. Instead a practical approach is adopted based on data flow analysis, with the aim to produce working software that will initially provide a role checking capability with around 70% reliability. The rules developed for this software are modular so that they can be built on as additional training information becomes available. The applicability of Roles of Variables to Objected Oriented programming has not yet been fully explored, though research will be presented in [12] – not yet published. For that reason, the software presented herein caters only for roles played by primitive type variables. In order to facilitate use of the software by teachers and novice programmers, the program is made available as an extension to the BlueJ programming environment.

The final version of the software is able to the check roles of variables in the training programs with 100% accuracy and consistency. However, it has not been possible in the time available to conduct extensive testing on other simple java programs to ascertain its overall reliability. Further, some limitations continue to exist with respect to the analysis and recognition of certain Java constructs (e.g. variable overriding, Switch/Case statements), and extra debugging will be required to accommodate the totality of conventions with respect to coding style. These limitations are discussed further in the body of this report.

This paper is organised as follows. Section 2 provides background information on the Roles of Variables concept, the current state of that work and its applicability to this project. Section 3 presents related information about Program Slicing [13], [14], [15], and how it can be used to extract information relevant for checking the roles played by variables. Section 4 outlines how the software interfaces with the BlueJ development environment. Section 5 details how slices are analysed, how variables' assignment and usage conditions are detected, and how rules are established to which variables having a given role should conform. Section 6 outlines the design of the analysis software, and section 7 concludes this report. Additional and supporting information can be found in Appendices 1 through 5.

2 Roles of variables

2.1 Background

In [3] 109 novice level programs (written by experienced programmers) were analysed. From this analysis 10 roles were identified that cover 99% of novice level program variable use. The roles identified represent typical usage of variables that repeatedly occur over many programs. The roles are described briefly in section 2.2, and are described in full in [2].

Traditionally, the teaching of programming involves the application of programming language to example programs [4]. Over time, this enables students to acquire “programming knowledge” i.e. how to construct programs from abstract concepts within a given programming paradigm [2]. Roles of Variables are programming knowledge that can be explicitly taught to students, and the results in [1], [6] suggest that that the use of roles provides students with a new conceptual framework that enables them to mentally process programs in a way similar to that of more experienced programmers. In particular, students taught using Roles of Variables demonstrate what Pennington describes in [7] as “deep knowledge” i.e. that relating to data flow and function, rather than “surface knowledge” relating to the operations and control structures that can be readily obtained by looking at the program source code. In addition, the results of [4] indicate that teachers of programming find the Roles of Variables concept relatively intuitive and also find it easy to assign roles consistently after a reasonably short familiarisation period. Even where role assignment is controversial, the concept provides good basis for discussion and clarification of program structure [4]. Further work looking at the impact on program construction has been presented in [10].

Animation and visualisation have been shown as useful to increase student’s understanding of programming language and program constructs [22]. The use of such aids in conjunction with Roles of Variables has been shown to further deepen programming knowledge of novice programmers in [5], [6] & [9]. In [11], Sajaniemi & Gerdt have investigated the automatic detection of variable roles for use in the generation of program animations for the purposes of reverse engineering. The idea being that if the role of a variable can be detected by a program, it would be possible using that program in conjunction with the software developed in [8], to automatically produce an animation illustrating the flow of data through the program being analysed, for the purposes of reverse engineering. Sajaniemi and Gerdt conclude in [11] that the combination of the non-exact cognitive concept that is Roles of Variables, and the exact nature of programming languages, makes the task of automatic detection of roles a non-trivial one. Further they suggest that automatic role analysis should be possible by performing data flow analysis combined with a machine learning strategy. Given the timescale available for this project, it is not considered feasible to design and implement a machine learning strategy that could lead to correct detection of roles in all cases with very high (e.g. over 95% reliability). Instead a more simplistic approach based on data flow analysis alone is adopted. The aim is to produce software that can detect incorrect role declarations and suggest correct ones for around 70% of variables 70% of the time. In addition, in the event that the variable does not appear to being playing any of the defined roles, the software can state that it is not clear what role is being played by the variable. The software could provide a useful role checking tool for novice programmers, and in the cases where the software incorrectly identifies variables as having erroneous role declarations, it could provide a useful basis for discussion about the structure of the program in question. It may also serve to highlight poor coding practice (see section 5).

In [12], Byckling, Gerdt, & Sajaniemi investigate extending the Roles of Variables concept to cover objects in Object Oriented programming. The results of this work have not been presented, or published at the time of writing, so it is not clear to what extent (if at all) the Roles of Variables concept is applicable to objects. In [1] Sajaniemi & Kuittinen say that the *role* of a variable characterizes its dynamic nature (i.e. the sequence of its successive values as related to other variables and external events), and thus that the way the value of a variable is used has no effect on the role. They state by way of example, that a variable whose value does not change is considered to be a *fixed value* whether it is used to limit the number of rounds in a loop or as a divisor in a single assignment. Given such an assertion, it could be argued that in some instances object references may take on one of the roles defined for primitive type variables. For example, if in the sequence of program events, an object reference has an object assigned to it once only, and thereafter only accessor methods are called on that object, one might consider the object to have the role *fixed value* (it is to all extents and purposes still the same object regardless of how it is used). Similarly an object reference holding a succession of objects dependent on its current properties compared to that of another object, could be said to be a *most wanted holder* (e.g. an object reference to the oldest instance of class Person in a succession of Person objects). Whether we might want to use the existing roles for such objects and how useful it would be to do so is a matter for discussion outside of this paper. On the other hand, if mutator methods are called on an object changing its internal state, it would appear difficult to see how any of the currently defined roles might be used. The object reference would still refer to the same instance of the object, but its internal state would be different (similar to an array whose internal values had altered). In any case, it is clear that further work is required on the applicability of Roles of Variables to Object Oriented programs, and for the purposes of this paper, only variables of (or in one case object references behaving as) primitive types are considered.

2.2 Role analysis

In programming, variables are not generally used in an ad-hoc fashion. Rather, there are a limited number of patterns that can be used to describe almost all usages of variables [4]. The Roles of Variables concept differs from the algorithmic patterns e.g. loop structures, often used in teaching in that it focuses on the flow of data through the variable in question, which is to a great extent independent of the algorithm as well as of the programming language [4]. So while a loop in java may be formed using different constructs such as those in 1a and 1b below, the role played by the variables `i` and `count` in each case will be the same. i.e. that of *stepper* (see role descriptions).

```
for (int i = 0; i < value; i++) {
    …
}
```

Figure 1a. "`i`" as stepper

```
while (count < value) {
    …
    count ++;
}
```

Figure 1b. "`count`" as stepper

The roles taken by variables are comprehensively described in [2]. The following outline descriptions of each role provide sufficient information for the purposes of this paper. They also introduce some of the ideas discussed in section 4 that relate to how variables playing a given role may be represented in java programs. Figure 2 (reproduced from [1]) shows the relationships between roles.

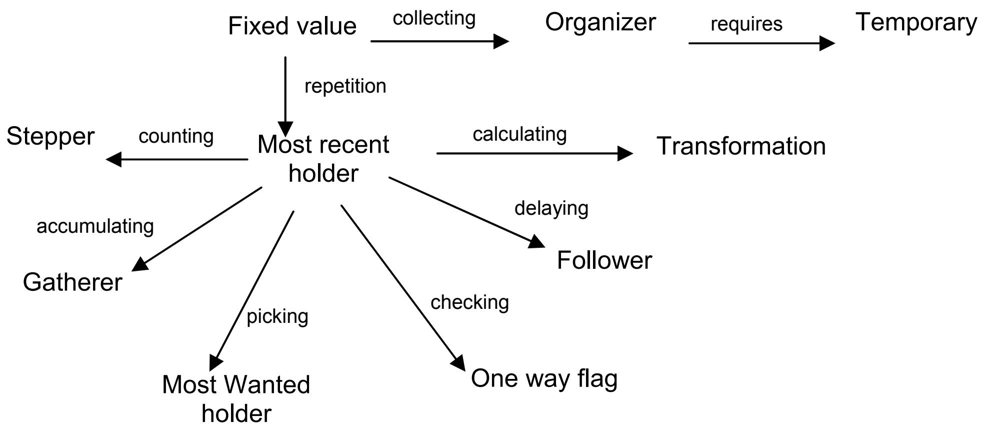


Figure 2. Roles relationships

1. **Fixed value.** The role of *fixed value* (aka constant), is played by any a variable whose value is fixed once it has been assigned, either by the user, the programmer, or by the final value of some other variable. The initialisation of such a variable may include some scaling or correction factor e.g. to make sure it is within required bounds if input by a user. The distinctive property of a *fixed value* variable is that it should not change after it has been initialised, though it may have two distinct values in its lifetime (its initial value, and final value following scaling/correction). This property may appear to be contradictory at first glance, but from the analysis in section 4, can be seen to relate to how and where the variable is used within a program. The observation that the use of the variable helps to determine its role, appears itself to contradict the statement in [1] that "the way the value of a variable is used has no effect on the role". However, in this context the use of the variable relates not to what end it is used, but where it is used relative to its being assigned. An example usage for a variable playing the role of *fixed value*, could be that of a user determined value limiting the number of values in a sequence output by a program generating Fibonacci numbers (see Fibonacci.java in Appendix 1.6).
2. **Organizer.** The role of *organizer* is one unique to an array that is used only for re-arranging its elements after initialisation. More generally, an array is said to play the same role as its constituent elements. For example, an array of *fixed values* would have the role *fixed value*, or an array of *gatherers* would have the role *gatherer*. An array with the role of *organizer* usually occurs with a variable having the role of *temporary* (to store elements on a temporary basis while they are being moved within the array). A typical example of an array playing the role of *organizer* would be that of the 'target' array in a Bubble Sort program (see BubbleSort.java in Appendix 1.1).
3. **Stepper**. A variable playing the role of *stepper* is one stepping through a succession of values that can be predicted as soon as the succession starts. In programming text books the term 'counter' is often used to describe such a pattern[1]. Typical usage for a *stepper* variable might be in a loop for counting through the contents of an array. The variables `i` and `count` in figures 1a and 1b above, are *steppers*. The future values of a *stepper* variable can be predicted if past values are known, and in most cases, the step between successive values is constant, unidirectional (boolean *steppers* are an exception to this – see section 4), and the value of the variables is tested against some limit.
4. **Most recent holder**. The role *most recent holder* is played by a variable holding the most recent value encountered in going through succession of values. In some respects it is similar to role of *stepper*, but the successive values cannot be predicted as there is no fixed relationship between them. The role of *most recent holder* can also be seen as a repetition of *fixed values* (see also figure 2), and as such implies assignment and usage within a loop. The analysis in section 4 shows how the relative positions of assignment and usage of such variables help to determine the roles they are playing. An example of a variable having the role *most recent holder*, would be one that holds the latest value input by a user, e.g. a succession of data input to a program providing statistical analysis of a range of values.
5. **Gatherer**. A *gatherer* is a variable that is used to accumulate the effect of individual values in going through a succession of values. Typically for a *gatherer*, each new value is obtained by combining new data with the previous value of the variable. This fact aids the identification of *gatherer* variables, as can be seen in section 4. An example of a variable having the role "*gatherer*" would be one storing the latest value to be output by the Fibonacci number generating program introduced in point 1 above, or the total sum of all values input the statistics program in point 4.
6. **Most wanted holder**. A *most wanted holder* is a variable holding the best value encountered so far in going through a succession of values. This implies assignment based on comparison of the variable's current value with a succession of values of another variable. As such the variable is likely to have its value assigned repetitively within a branch for which its current value forms part of the condition. An example of a

most wanted holder would be a variable in the statistics program holding the highest value input by the user.

7. **One way flag**. A *one way flag* is a two valued variable that cannot regain its initial value once its value has been changed. As such it is often used as a mechanism to register the occurrence of an event that should enable the transition from one part of a program to another. For example, a variable having the role *one way flag* may be set following the check on data input to a program via the terminal to indicate that the values input are within an acceptable range, and that the program can move on to perform calculations based on the data input.
8. **Transformation**. The role *transformation* is played by a variable whose value is always (repeatedly) set by the same calculation from value(s) of other variable(s)). As such it is generally tightly connected to another variable through some arithmetic expression. For example, the variable (`circ -` *transformation*) holding a succession of values for the circumference of a circle based on a succession of radii (`rad –` *most recent holder*) input by the user, and the *fixed value* (`pi`) set by the programmer:

   ```
   circ = 2 * pi * rad;
   ```
9. **Follower**. A *follower* is a variable that gets its succession of values by following the successive values of another variable. As such it is tightly connected with one other variable, usually taking its previous value. An example of a *follower* variable would be one in the Fibonacci program used to hold the previous value in the sequence for the purposes of calculating the subsequent value.
10. **Temporary**. The *temporary* is another typically described pattern in programming textbooks [1]. The role *temporary* is given to a variable holding some value for a very short time only. A *temporary* variable is often declared, assigned and used within a single loop and then discarded. It can be used to hold input data during error-checking or for storage of variables during re-arrangement within an *organizer*. An example of a *temporary* variable would be in the Fibonacci program to store the previous value in the sequence whilst the highest latest value was passed by a *gatherer* variable to a *follower* variable prior to calculation of the latest value in the sequence.

3 Program Slicing

3.1 Background

Section 2.1 discussed briefly the ways in which Roles of Variables can be used in reverse engineering. Chikosfsky and Cross in [19] describe reverse engineering of a system, as the process of using identified system components and their inter-relationships to create a different representation of a system, or representation at a higher level of abstraction. The automatic identification of roles being played by given variables can thus be seen itself as a form of reverse engineering, in that we are seeking to represent parts of the system (the variables being analysed) in a different form (i.e. in terms of their roles). Reverse engineering techniques are therefore useful to breakdown the source code for the purposes of role checking. In [14], Beck & Eichmann state that “reverse engineering drives the recreation of the design representation from the implementation”. In the case of the role checking software, the immediate feedback to the novice programmer of information confirming or questioning design assumptions should provide a different perspective of her and help to improve her “programming knowledge”.

In his work on reverse engineering, Weiser introduced the concept of “program slicing” to describe the grouping of non-sequential sets of statements by programmers involved in debugging [13]. He found that by examining the data flow of a program, he was able understand the mental abstractions of the programmers. Further he concluded that such non-sequential groupings applied to “units of data components”, i.e. the variables[14]. Program slicing has therefore been used in this project, both in the role checking procedure itself, but also for analysis of the training programs to help determine the individual characteristics of the different roles with respect to how and where annotated variables appear in the source code (see section 5).

3.2 Breaking down the source code

In [15], Moonen identifies two constructs that appear in programs. *Elementary* programming constructs are those such as read, write and assignment statements. *Control* constructs are those that decide which elementary construct is the next to be executed. Control constructs can be further sub-divided into *conditional* control constructs where the decision depends on the property of an object in memory, e.g. "`if x > 3 then`" and unconditional control constructs such as "`;`", which simply signify the end of the previous elementary construct. Usefully in Java, conditional control constructs always end in "`{`", unless only the immediately following statement is to be executed.

Expressing the source code as a sequence of elementary constructs ending in "`;`", and conditional control constructs that control the execution (or otherwise) of elementary constructs, as ending in "`{`", provides a useful mechanism for dividing correctly compiled source code into a hierarchy of statements. Each elementary construct ending in "`;`" can have an associated conditional control construct as a parent. Thus Java source code can be stored as a hierarchy of nested maps (linked maps can be used to preserve the order of storage). In these maps, elementary constructs ending in "`;`" act as keys to their own values, and conditional control constructs ending in "`{`" act as keys to other maps containing more elementary constructs and conditional control constructs as required. This is shown by the diagram in figure 3. The use of statements as keys in a map has the disadvantage that it is not possible to have multiple occurrences of the same statement at the same hierarchical level in the program (in the present software). However, it is thought that the use of loops and good programming practice should help to minimise the number of occasions in which statements are repeated at the same level in a program. Further, if this becomes a problem, it would be possible to generate a unique pseudo key for each statement and to store that in a separate look up table to ensure that all occurrences of a given statement are preserved.

The final part of breaking down the source code involves the removal of comment statements not having an impact on the flow of data through the program, and on the identification and storage of variables for which roles have been declared. Fortunately again in Java, comment statements are identified in a limited number of ways. The appearance of "`//`" anywhere in a line of source code means that the rest of the line is just comment. Similarly, the appearance of "`/*`" any where in the source code means that all text that follows is comment, up until the appearance of the characters "`*/`". It is therefore reasonably straight forward to identify and remove comment statements from the source code to be analysed. Once comment statements have been identified they can be analysed to look for the regular expression that identifies a variable role declaration (see section 3.2). The names of variables for which roles have been declared can then be stored together their declared roles. In the present software a map is used with the variable name acting as the key, and the variable role as the value. This has the obvious disadvantage that every variable to be checked in the program must have a different name, but one may perhaps question whether it is best practice to use variable overriding in novice level programs, and from the perspective of reverse engineering.

3.3 Slicing the program

As discussed in [3], [11] the role of a variable has more to do with way in which the data flows through the variable than value of the variable itself. Thus, for the purposes of the present project, there should not be any need to undertake the total decomposition of the source code described in [18] where every variable having an impact on the value of the analysed variable is isolated. Instead it should be sufficient to look for each variable, at where and under what circumstances it is assigned and used (though some provision may need to be made for indirect usage of the variable e.g. when two variables are used in combination to set a flag that acts as a condition for the loop).

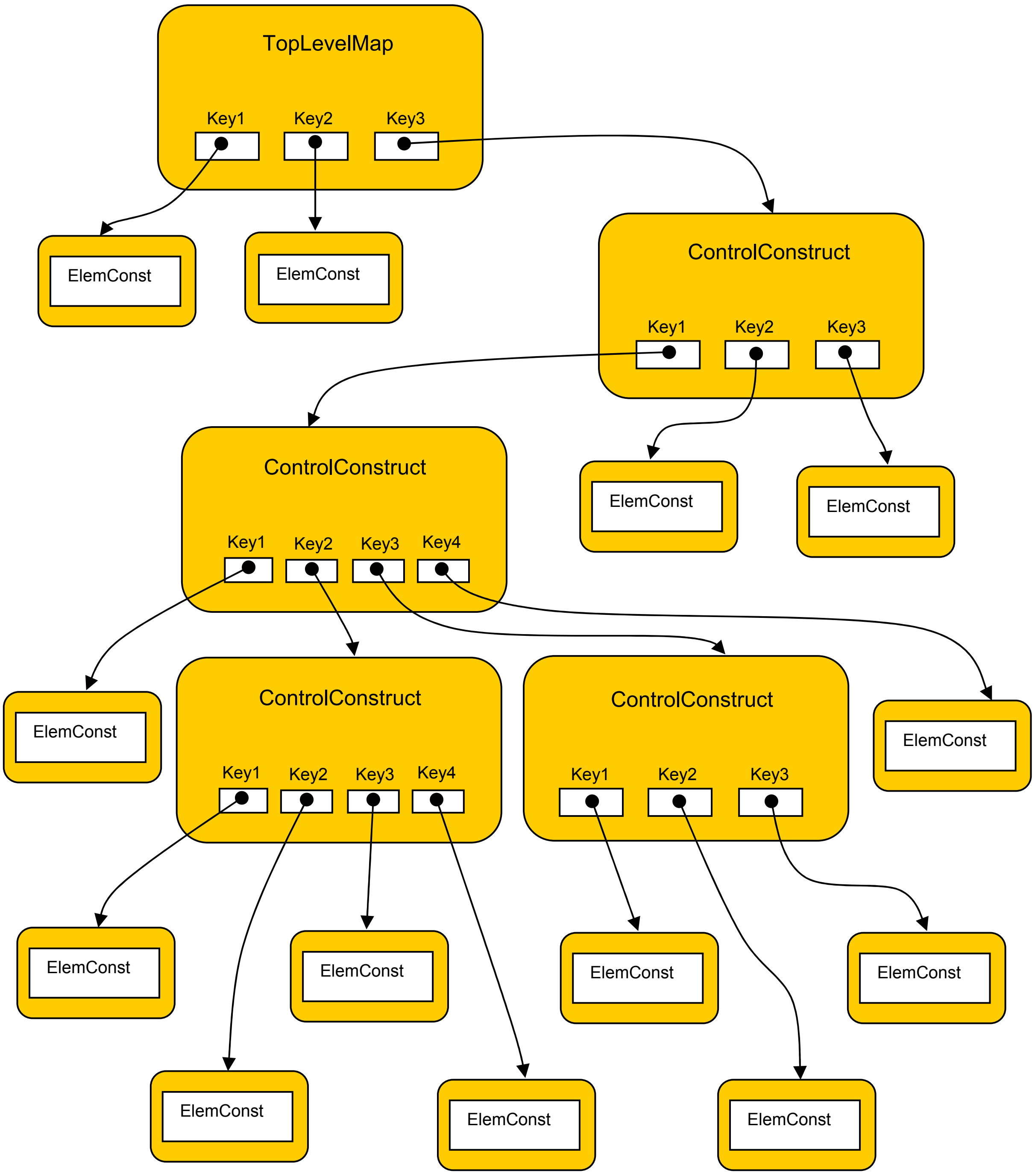
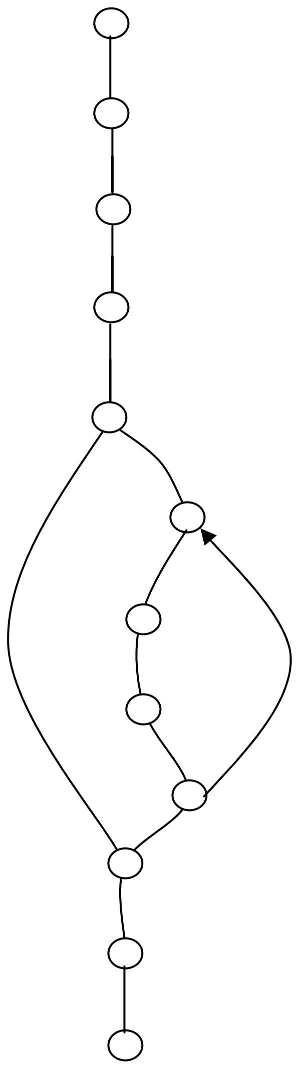


Figure 3. Source code map

Using the source code map (illustrated in figure 3), statements relating to each variable for which a role has been declared can be extracted. A sensible starting point for this in each program is the map containing the variables to be checked. Taking the key set of the variable map, the source code map can be traversed and each of the control constructs encountered stored in the order that they occur, until a construct (elementary, or control) containing the variable in question, is found. Once an occurrence of the given variable has been found, it is stored with its associated control constructs in a list. The map is traversed again looking for the next occurrence of the variable, with any control constructs and variable statements stored in a separate list from the first. This process can be repeated until every occurrence of the variable has been found. In this way it is possible to identify the position of each incidence of the variable being analysed in the program. This process can be repeated for all variables in the program for which a role has been declared. It is then possible to reconstruct the program from the perspective only of a given variable. This is the program slice. Figure 4, shows a program slice diagram for the variable last_fib from the Fibonacci program of appendix 1. Appendix 2 shows textual program slices for all variables in the training programs having the role *fixed value*.



```
public class Fibonacci {

    private int last_fib;

    public class Fibonacci() {

        last_fib = 1;

        if (number > 2) {

            for (i = 0; i < number; i++) {

                temp = last_fib;

                last_fib = fib;

            }

        }

    }

}
```

Figure 4. Program slice diagram for last_fib

3.4 Declaring roles

An initial consideration in designing the software to check consistency of roles being played by variables, is how in the source code to identify the variable and its role. Two possible mechanisms were considered for this:

1. Direct identification of the variable role by using a pre-fix to the variable name in a fashion similar to that of Hungarian Notation [16].
2. Annotation in the comments for each variable using a unique marker to facilitate identification by the analysing software.

Hungarian Notation was developed in the 1980's by Charles Simonyi of Microsoft. His idea was to create a naming convention for variables whereby the functional type of the variable was encoded in its name using a prefix of letters associated with it type. E.g. a variable `value` of type `float` would be identified by the name `fvalue`. Using this type of naming convention it is possible to encode the role of the variable directly with its name so that its role is obvious to the reader of the code. If applied to Role of Variables, `stCount`, `gtSum` and `mrhUserInput` could signify variables roles *stepper*, *gatherer* and *most recent holder*, respectively. Simonyi's notation became known as Hungarian Notation because the use of prefixes made variable names look a little as if they were not written in English, and because of Simonyi's Hungarian heritage. Thereby, lays one of its disadvantages, in that programmers not familiar with it can find it non-intuitive and difficult to understand [20]. This may not be such a problem in Java given the limited number of roles, as prefixes are not likely to get as long an unwieldy as they might when declaring variable types in a language such as C++. However, it would add some extra complexity to the naming of variables by novice programmers, and is not in line with existing Java naming conventions due to its limited relevance to OO languages that define objects and not types [20]. As such it may not be considered best practice for novice programmers learning Java for the first time, and could lead to confusion of code readers not familiar with Hungarian Notation or the role of variables concept. Another disadvantage of such a technique would be that in the event that the role for a variable had been incorrectly declared by the programmer and identified as such by the software, it would be necessary to rename every occurrence of the variable within the program to reflect its newly perceived role.

Hungarian Notation is essentially a commenting technique [21], and the second mechanism considered for highlighting variables and their roles is the use of unique markers within code comments. Identification of roles in code comments has advantages over a Hungarian Notation type mechanism, in that it is completely peripheral to the code and does not detract from its readability. Further it only requires the role for each variable to be declared once (this makes for easier correction in the event that a role has been incorrectly declared by the programmer). Given the advantages of the use of comments vs. Hungarian Notation, the second option is adopted. The expression "`%%`" was chosen as the unique marker to highlight each variable and its perceived role, because the author considered there to be no reason for such an expression to appear in Java comments (other than as some sort of identifier). Other multiple character strings such as "`$$`" would work just as well. By encapsulating the variable and its role in such a regular expression, it is possible to use the java.lang.String.split(String regex) method to extract that variable name and role, e.g. from the string "`%%variableName%%role%%`".

4 Incorporation with BlueJ

To maximise the usefulness and usability of the Roles of Variables checking software, it is integrated into the BlueJ development environment. Fortunately, BlueJ offers a useful extensions mechanism enabling developers to add new functionality to BlueJ as and when it is needed [17]. A full API is given that provides access to the BlueJ application via a proxy object. It also provides access to the classes for which role checking is required via a number of wrapper classes. In addition, [17] provides a useful guide of how to implement a simple extension, and this can be used as the basis for incorporation of the role checking software.

The roles checking software is seen as an optional, selectable feature for BlueJ that can be called following compilation of source code. Indeed the role checking software expects that the source code being checked has been successfully compiled (i.e. it has the correct syntax). When the Roles of Variables extension is installed, a tab is added to the extensions preferences sub-menu enabling the user to select:

- If Roles of Variables should be checked following compilation;
- If a reason should be given if an incorrect role declaration is identified; and
- If the software should suggest an appropriate role in the event that an erroneous declaration is identified.

The roles checking software can listen for compilation events from the BlueJ proxy object. If role checking has been selected a thread is started that analyses the recently compiled source code to check roles declared for variables. In the event that an erroneously declared role is identified, an editor object for the class in question is opened, and text is highlighted pointing to the variable in question and the offending statement.

5 Checking variable roles

5.1 Analysing statements

As part of their work on Roles of Variables, Sajaniemi & co provide a number of simple programs to communicate the concept, and in order to assess the ease with which it can be learned by Computer Science educators. A number of these programs are available at [2] as training programs for those unfamiliar with the concept, and as experimental data for studies leading to [4] and [12]. Seventeen different programs are identified (though generally only twelve have been used at any one time to train programmers in the Roles of Variables concept, or as experimental data). The programs are mainly written in Pascal though some are in Java. As discussed in section 2.1, teachers of programming appear able to obtain a sound grasp of the Role of Variables concept using twelve “training” programs within a short space of time, and to assign roles correctly and consistently to variables thereafter. The combination of seventeen training programs and the author’s own Java programming knowledge was therefore considered a reasonable basis from which to start training the software to recognise where roles have been declared incorrectly. Where applicable, Pascal programs provided at [2] have been translated into Java. The training programs can be found in Appendix 1.

In [15] Moonen describes a generic architecture for data flow analysis, and concludes that there is no need to be specific about control constructs within the program being analysed. Any loop can be modelled simply as a loop having a condition under which it should continue to run. Similarly any branch can be modelled as a branch and condition under which it should be followed in the sequence of executed statements. In [15], Moonen usefully differentiates between “variable definition” comprising input statements and those where the variable appears on the left hand side of equations, and “variable use” such as output statements and those where the variable appears on the right hand side of equations. According to [1], the role of a variable captures its behaviour by characterizing the dynamic nature of the variable, i.e. the sequence of its successive values as related to other variables and external events. The analysis of training programs undertaken in the present project does not entirely support the view in [1] that usage does has no effect on the role of a variable (see section 4), but the idea of separating statements involving variables into those in which the variable is defined and those in which the variable is used, provides a useful starting point for breaking down the sliced statements.

From the available training programs, four distinct categories of variable statement are identified as follows:

1. **Assignment statements** – i.e. those where the variable appears on the left hand side of an equation as described in [15]. Java appears to have some advantages over other programming languages such as Pascal in this respect, as direct assignment statements always contain an “=” operator on the left hand side of which appears the variable in question.[1]
2. **Usage statements** – i.e. those where the variable is either output to terminal (for use by the program user), directly used to assign some value to another variable, or as input to a method. This is roughly equivalent to the description of variable use in [15].

[1] Indirect assignment of a variable e.g. via input of an integer value to a method, is not covered in the present version of the software.

3. **Conditional statements** – i.e. those statements where the variable appears either as the condition for a loop, or as a condition for a branch[2] in the software.
4. **Other statements** – e.g. those where the variable is declared, or returned.

In order to expedite the process of analysing data flow in the training programs, a utility class was added to the analysis software. That class took each of the training files in turn and output slices for each variable to a file depending on the role declared for each variable. Ten different text files were thus created, one for each role. An example slice file for the role *fixed value* can be found in appendix 2. Each of those files was analysed and a table created for each role. Each table gives the variable name, its type, where and how it is assigned in the program, where and how it is used in the program, and other information as required for identifying conditions that indicate which roles are being played. Table 1 shows the table created for *fixed value.* Appendix 3 contains the 10 role tables for the training programs in their entirety.

Name	**Type**	**Assigned**	**Used**	**Other**
len	Int	Once only outside of loop	Not relevant	n/a
original	Int	Once only outside of loop	Not relevant	n/a
number	Int	Once only outside of loop	Not relevant	n/a
percent	Float	Once within loop	Outside of loop Branch condition within assignment loop	Conditional use sets assigning loop condition
factor	Float	Once only outside of loop	Not relevant	n/a
years	Int	Once within loop	Outside of loop Branch condition within assignment loop	Conditional use sets assigning loop condition
amount	float[ ]	Once only outside of loop	Not relevant	n/a
op	char[ ]	Once only outside of loop	Not relevant	n/a
digit	char[ ]	Once only outside of loop	Not relevant	n/a
letter	char[ ]	Once only outside of loop	Not relevant	n/a
size	Int	Once only	Not relevant	n/a

[2] Switch case statements are not currently dealt with by the software.

		outside of loop		
value	int[ ]	Initialised outside of loop Filled within loop	Within different loop having same signature	n/a
key	int	Once only outside of loop	Not relevant	n/a
hours	int	Once within loop	Outside of loop	n/a
mins	int	Once within loop	Outside of loop	n/a
secs	int	Once within loop	Outside of loop	n/a
value	int[ ]	Initialised outside of loop Filled within loop	In different loop Elsewhere outside of loop	n/a
c	char	Once only outside of loop	Not relevant	n/a
side	int	Initialised Within loop	As condition for assigning loop Outside of assigning loop	n/a

Table 1. Fixed value assignment and usage

Analysis of the tables in Appendix 3 highlights a number of commonalities in the way that variables playing certain types of role are assigned and used in the training programs. For example, analysis of table 1 above shows that in only a few instances is a variable playing the role of *fixed value* used within the same loop in which it is assigned. Closer inspection reveals that where *fixed value* variables are used within their assigning loop, it is either directly as the condition for the loop, or in combination with another variable via branch to set a variable having the role *one way flag* as the loop condition. In both cases, this use takes the form of error checking as described in section 2.2. Similar patterns of usage are observed for other roles as follows. This list is not exhaustive, but serves to illustrate the kind of patterns that emerge:

- *stepper*. The variable is either assigned within a “for” loop statement, incremented/decremented within a loop, or directly/indirectly toggled within a loop;
- *gatherer*. The variable appears either directly or indirectly on both sides of an assignment statement found within a loop.
- *most wanted holder*. The variable is used as a condition for the branch in which it is assigned a value.
- *transformation*. The variable is assigned within a loop with a combination of other variables, values and operators.

Given such examples, it becomes clear that in order to check the roles played by variables, it is necessary to identify the relative positions of assignment and use statements within the programs being analysed. The statements from each program slice are therefore analysed and grouped according to the four statement categories given above. Their whereabouts (line number) in the program slice is noted, as well as their relative position within the hierarchy of the program i.e. the method(s), loops, and/or branches in which they are found. It is noted that an assignment statement may in some cases also be a usage statement for a given variable, e.g. in the case of a statement where the variable appears on both sides of the "=" character. Similarly, an assignment statement may also be a conditional statement e.g. "`if (x > y) x = y;`". Given the relative importance of data flow through the variable compared to how the variable is used, priority in the analysis program is given to assignment statements. Thereafter, one of the conditions defining a usage statement for a given variable is that it has not already been identified as an assignment statement. Similarly, conditional statements should be neither assignment nor usage statements, and other statements should be all those that do not fit into the first three categories.

5.2 Checking conditions for assignment and use

Once the program slices have been analysed and all relevant statements sorted into one of four categories, a number of checks are made to establish the conditions under which variables are assigned and used. Flags are set depending on the result of each of these checks. The value of an individual flag is not sufficient on its own to accurately indicate the role being played by a variable, but when used in combination with the values of other flags, it is possible build up role checking rules for each of the roles that can be played.

In [1] Kuittinen and Sajaniemi emphasize the cognitive as opposed to technical nature of the roles concept. This aspect of the concept presents a number of challenges when it comes to checking the roles being played by variables. For example in the Fibonacci program (an extract of which can be found in figure 5a for ease of reference), the variable `fib` is playing the role *gatherer*, the variable `last_fib` is playing the role *follower*, and the variable `temp` is playing the role *temporary*.

```
1      public class Fibonacci {
2            …
3                  for (i = 3; i <= number; i++) {
4                        temp = last_fib;
5                        last_fib = fib;
6                        fib = fib + temp;
7                        System.out.println("Value " + i + " is: " + fib);
8                  }
9            …
```

Figure 5a. Extract from Fibonacci.java

In this code, the variable `fib` can be readily identified as a *gatherer* due to its appearing on both sides of assignment statement within a loop at line 6. There could however, be some confusion in the roles being played by `last_fib` and `temp` (not withstanding the fact that one of them has the variable name "`temp`"), as both are assigned in a loop only with the value of another variable. On first inspection both could appear to be playing either the role *follower* or *temporary*. However, the fact that a usage statement appears in the loop for `last_fib` before an assignment statement, implies that `last_fib` is initialised in the code prior to the loop (which is indeed the case). This suggests that the `last_fib` is more likely to have the role *follower*. What's more the variable `temp` does not appear anywhere in the program outside of the loop (aside from its declaration statement which in any case may be better placed in the loop for code readability). This is likely to support a role declaration of *temporary* for that variable.

By contrast, figure 5b shows code having exactly the same functionality, but with a change at line 6. In the revised code, the variable `fib` is assigned with the sum of `last_fib` and `temp`, after `last_fib` has been assigned with the value of `fib`. In this code, `fib` still has the role *gatherer*, but will appear to the current software as having the role *transformation*. In time the software could be enhanced to recognise `fib` as having the role *gatherer* in the code of 5b, even though it appears only indirectly on both sides of the assignment statement. At the time of writing however, the role checking software does not support such identification. This serves nicely to illustrate the crux of the problem at the heart of this project, i.e. how to use analysis of operation and control structures that can be readily obtained by looking that program source code, i.e. "surface knowledge" in [7], to identify what are essentially "deep" programming concepts such as the role played by variables. Even so, the fact that the software will question the correctness of the a role declaration of *gatherer* for the variable `fib`, in the code of 5b, and the fact that it will suggest the role `transformation`, could be seen as useful in highlighting poor coding practice in terms of code readability, with respect to a variable accumulating the effect of a succession of values.

```
1    public class Fibonacci {
2        …
3            for (i = 3; i <= number; i++) {
4                temp = last_fib;
5                last_fib = fib;
6                fib = last_fib + temp;
7                System.out.println("Value " + i + " is: " + fib);
8            }
9        …
```

Figure 5b. Extract from Fibonacci.java with re-arranged code

For the seventeen training programs in Appendix 1, twenty one different assignment/usage conditions are identified. The brief discussion following each condition is intended to illustrate the relevance of the condition for establishing role checking rules, but is not complete in that it does not cover every aspect of role checking for which the condition is relevant.

1. **Whether the variable is assigned in a loop.** If the variable is not assigned within a loop, this will in many cases indicate a role of *fixed value* (or perhaps *stepper* in the event that the variable is assigned within a "for" loop statement). On the other hand, if the variable is assigned a succession of values within a loop, this does not necessarily mean it is not playing the role *fixed value*, e.g. providing that the variable is not used with this succession of values elsewhere within assignment loop.
2. **Whether the variable is used in its assignment loop.** If the variable is used directly within the loop in which it is assigned, this should in most cases indicate, that the variable cannot have the role *fixed value*, though one could not infer which role the variable may have from this condition alone. Conversely, if the variable is not used directly, or as a condition within the loop within which it is assigned, this may indicate that the variable does not have a role involving the use of a succession of values such as *most recent holder.*
3. **Whether the variable is used conditionally in its assignment loop.** Similar to 2 above, but for conditional statements.
4. **Whether the variable is used either directly for its assigning loop condition.** If the variable is used as the condition for the loop in which it is assigned, but not within the loop itself, this may indicate that the variable has the role *one way flag*, when taken in combination with the values of other flags.
5. **Whether the variable is used indirectly for its assigning loop condition.** Indirect use of variable for its assignment loop condition may indicate that it has role *fixed*

value, rather than *most recent holder*, even though it is used in the loop in which it is assigned.

6. **Whether the variable is assigned in "for" loop statement.** If the variable is assigned its value within a “for” loop declaration statement, it is very likely to have the role *stepper*.
7. **Whether there is any direct usage of variable in the program.** If there is no direct usage of the variable in the program, i.e. it is only used conditionally, this may indicate when taken in combination with other conditions that the variable is a *one way flag*.
8. **Whether the variable is assigned in a branch for which it is part of the condition.** If the variable is part of the condition for the branch in which it is assigned, this can be taken to imply the setting of its new value is dependent on comparison of its current value with some other variable. In many cases this will indicate that the variable has the role *most wanted holder*.
9. **Whether the variable appears on both sides of assignment statement.** If the variable being checked appears on both sides of the assignment statement, this is likely to indicate that it has either the role of *gatherer*, or *stepper*, depending on the other operators appearing on the right hand sides of the equation.
10. **Whether the variable appears indirectly on both sides of assignment statement.** A variable may implicitly appear on both sides of an assignment statement in the case where the operators `+=`, `-=`, etc are used. It is therefore also necessary to look for this condition to aid checking for *gatherer* and *stepper* roles.
11. **Whether the variable is toggled within a loop.** If the variable is toggled within a loop this will in most cases indicate that the role being played is that of *stepper*.
12. **Whether the variable is assigned twice in loop, one assignment being within a nested loop.** In the case where the variable is of type boolean and is assigned with a different value in each loop, this may indicate an indirect toggling of the variable within the loop and hence that the variable has the role of *stepper*.
13. **Whether the variable is incremented/decremented within a loop.** Similarly to the toggle statement. The increment, or decrement of a variable within a loop is likely to indicate that it has the role *stepper*.
14. **Whether the variable is used outside of loop in which it is assigned.** If a variable is used outside of the loop in which it is assigned, this should signal that the role being played is not that of *temporary*.
15. **Whether the variable is assigned in loop before it is used that loop.** This condition may be used in combination with other conditions to help differentiate between the roles *follower* and *temporary*, as in both cases the variables are assigned directly with the values of other variables (as discussed above).
16. **Whether the variable is used conditionally for a loop outside of its assignment loop.** If a variable is not used conditionally outside the loop in which it is assigned, this may in combination with other conditions indicate that the variable has the role *one way flag*.
17. **Whether the variable appears in array organizing type statement.** This condition is used to help differentiate between variables of type `array` playing the role organizer and those playing other role, such as a *fixed value*. In that case an organizer statement is identified as one occurring within a loop, and having the variable on both sides of the assignment statement with different indexes. E.g.

    ```
    target[j] = target[j+1];
    ```
18. **Whether the variable is of type array.** If a variable is not of type array, it is immediately clear that it cannot have the role *organizer*.

19. **Whether the variable is assigned within a loop with a combination of other variables, value and operators.** If this condition is true it may indicate that the role being played by the variable is that of *transformation*.
20. **Whether the variable is assigned with the output from a method call.** If a variable is assigned with the output from a method call, then it is likely that it will not have the role *follower, or transformation*.
21. **Whether variable is assigned with a value resulting from instantiation of a new object, or directly with boolean value.** If this condition is true, then the variable being checked is unlikely to have the role *follower*, or *transformation*.

The above conditions are those identified as necessary to check the roles declared in the seventeen training programs. They are not complete for all java programs, and analysis of additional programs will highlight other assignment/use conditions that should be checked if roles are to be unequivocally identified for the variables therein.

It has not been possible during the course of this project to tailor the analysis software to cater for all the different coding styles adopted by individual programmers. For example, figure 6 shows an extract from a program Calories.java (adapted from [23] and not one of the training programs). In this extract the variables `gramsOfFat` and `totalCalories` and declared and assigned within a single statement. An earlier version of the role checking software correctly recognised the statement at lines 6 and 7 as an assignment statement for both the variables. However, due to the positioning of `totalCalories` after the "=" character assigning the value "`0`" to variable `gramsOfFat`, the software incorrectly identified the role *gatherer* for the variable `totalCalories`. This was because the variable appeared to be on both sides of an assignment statement. Whilst this bug has been remedied, other such bugs are likely to be identified as other programs are analysed. The software will need to be further tuned in order to perform with adequate reliability for a large range of input programs.

```
1      public class Calories {
2             public static final double CALORIES_PER_GRAM = 9.0;
3
4             public static void main (String[] args){
5                    …
6                    int   gramsOfFat = 0,
7                    totalCalories = 0;
8                    …
9             }
```

Figure 6. Extract from Calories.java

5.3 Establishing rules

Rules for checking each role are established based on the combination of the conditions outlined in 5.2 that are applicable to each of the variables in the training programs. In each case, it is assumed that the role declaration for a given variable is correct unless a specific condition is breached that indicates a possibly incorrect declaration. Thus, rules for checking the validity of each role declaration are built up as a series of boolean equations. For example, assume for a given variable, that if any of the following conditions hold, the variable cannot have the role *fixed value*.

A. The variable is incremented/decremented within a loop
B. The variable is toggled within a loop
C. The variable is appears on both sides of an assignment statement within a loop
D. The variable is used within a loop in which it is also assigned

If R signifies an incorrect role declaration, then:

$R = A \vee B \vee C \vee D$

The actual rules defined for each of the roles following analysis of the training programs are in most cases significantly more complex than the example given above. A complete listing can be found in Appendix 4. In the event that an incorrect role declaration is detected, the variable is checked against each of the other rules in turn to see if there is a match. If there is not, the user will be informed that the software is unsure of the role being played by the variable. The mutual exclusivity of the rule sets means that where the role is not recognized, or unspecified (by leaving white space between 2nd and 3rd markers e.g. "`%% %%`"), the program is able to suggest a probable role for the variable in question, though at present it does not identify a supporting statement, or provide a reason for that suggestion.

In developing the rule sets, steps are taken to ensure that as far as possible the rule set for each role is mutually exclusive of the other rule sets. This is to stop incidences where a variable appears to conform to the rule set of more than one role. One exception to this is the case of the rule set for *most recent holder*. In that rule set, one of the rules indicating that the variable is not a most recent holder is as follows:

$R = A \wedge B$,

where R signifies and incorrect role declaration, A signifies that the variable is of type array, and B signifies that the variable is not playing the role *fixed value*. This break with convention is to enable the program to recognize in the case of Histogram.java (see Appendix 1), that the float Array (`amount`) has the role *fixed value* and not the role *most recent holder*. The problem in that case is that the array appears to be used within the loop in which it is assigned, even though only one of the array values is being used and that value is actually assigned only once (i.e. it is a *fixed value*). At present, the variable is reckoned by the analysis software to have the role *fixed value*, only because even though it appears to be used in the loop within which it is assigned, it is also of type Array. What's more, the analysis program recognizes the variable as not having the role *most recent holder*, only because one of the rules is that the variable shall not be an Array and also conform to the rule set for *fixed value*. This combination of rules is circular and self satisfying. What more, it effectively means that it is not possible for a variable to be of type Array and have the role *most recent holder*. Clearly this is not an adequate rule for either *fixed value*, or *most recent holder*, and it serves to highlight the additional difficulties faced when trying to define rules that hold for variables of type array and other primitive types. It also perhaps serves to highlight the sort of problem that may be encountered if trying to deal with other more complex variables such as those that refer to an object. Further analysis is therefore required for variables of type Array, perhaps to the extent that additional categories of variable statement should be defined for Array type variables, such as "Array value assignment", and "Array value usage". This may facilitate differentiation between statements in which the variable assigned, e.g. "`amount = new float[12];`" and those in which an individual value is added to the Array, e.g. "`amount[month-1] = getInput();`".

It is desirable to add information to the output of the program that can be used to illustrate why a declared role is thought to be incorrect, so prioritisation is used for the rules within the rule set. Prioritisation is also required, because some rules are more specific than others. E.g. if a variable having a declared role of *fixed value* is actually a *gatherer*, it may fail the *fixed value* check because it is used within the loop within which it is assigned, as well as because the variable appears on both sides of an assignment statement. In such cases it is preferable to identify the reason for the incorrect role declaration as the detection of a specific *gatherer* (on both side of assignment) type statement, rather than the more generic reason that a *fixed value* variable shall not be used in the same loop in which it is assigned. This prioritisation also supports the suggestion of roles to the programmer, as the offending statement (e.g. "variable appears on both sides of assignment statement") can be presented with the incorrect role detection announcement, reason for the detection, and suggested role for the variable.

The reader should also bear in mind that the defined rules have been established based on a limited amount of test data, and as such have been clearly biased to fit that data. Considerably more training data is required to fine tune the rules in order to provide consistently reliable role checking functionality. Nevertheless, the rules developed thus far should be adequate to correctly identify roles being played by variables in many simple programs, providing that reasonable programming practice has been adhered to. Just as poorly written code is difficult for third party programmers to understand, so it is for the role checking program to analyse correctly. Regardless of the potential short comings of the developed software with respect to third party generated java programs, the software could be used with the seventeen training programs alone to help students become more familiar with the Roles of Variables concept, in order to apply it to their own program design.

6 Program design, implementation, and test

The role checking program is written in Java to facilitate its use as an extension to the BlueJ development environment. The program comprises six packages as illustrated in figure 7. Each package groups together classes having related functionality. A complete listing of the source code can be found in Appendix 5. The main package contains the classes RolesOfVariables and RoleAnalyser. The class RolesOfVariables is the interface to the BlueJ development environment. RolesOfVariables extends the Extension class from the blueJ.extensions API, and as such contains methods facilitating interface with BlueJ. It instantiates the PreferenceSetter class in the main.blueJExt package, which creates a new options tab under tools/preferences/extensions to select which roles of variables functionality should be run (detection, reason for error, suggested role). RolesOfVariables includes an inner class CompileEventListener that implements the CompileListener from the BlueJ API. When source code to be analysed is compiled successfully and the role checking software is enabled, the CompileEventListener starts a new thread that creates an instance of RoleAnalyser which returns a map of role checking results for each source code file to be checked. In the event that an error in role declaration is detected, an editor for the class is opened and the offending statement (as passed by the RoleAnalsyer) is highlighted in the source code together with a message dependent on the options selected for the role checking software.

The RoleAnalyser class is the main class of the role checking software. It takes as its input an Array of successfully compiled files, and returns a HashMap containing an ArrayList for each file being checked. Each ArrayList contains a HashMap of the role checking results for the variables, and a LinkedHashMap of the roles declared for each of the variables. When created, it instantiates a sequence of classes in the main.progAnal package to break down code and analyse program slices for each variable. It then instantiates the RuleApplyer class to check conditions of variable assignment and usage and to apply the rule set for each role as appropriate.

The SourceSorter is instantiated by the RoleAnalyser. It takes as its input the source code of the program to be analysed. It returns a LinkedHashMap containing the source code broken into hierarchy of nested LinkedHashMaps as described in section 3.2. It also returns the RoleHolder containing variables and their declared roles.

The StatementGetter is instantiated by the RoleAnalyser. It extends ProgramAnalyser, and takes as its inputs the source map and the variables for which roles have been declared. It returns a LinkedHashMap, containing an ArrayList of ArrayLists for each variable in the analysed program. These give the precise location of each occurrence of a statement containing the variable in question, in terms of its hierarchy within the program, as described in section 3.3.

The ProgramSlicer is instantiated by the RoleAnalyser. It takes as its inputs the set of variables, and the statement map returned by the StatementGetter. It returns a HashMap containing an ArrayList of statements comprising the program slice for each variable, as described in section 3.3.

The MethodGetter is instantiated by the RoleAnalyser. It extends ProgramAnalyser, and takes as its input the program slice generated by ProgramSlicer. It returns an ArrayList of all method names in the analysed source code.

The StatementAnalyser is instantiated by the RoleAnalyser. It extends ProgramAnalsyer, and takes as its inputs the set of variables and the program slice map returned by the ProgramSlicer. It returns a HashMap containing a LinkedHashMap for each variable. Each LinkedHashMap contains statements analysed and grouped according to the four categories defined in section 5.1. The line number of each statement is given within the program slice is given together with the location of each statement in terms of the program hierarchy, as described in 5.1.

The ProgramAnalyser holds methods that are common the to StatementGetter, MethodGetter, StatementAnalyser and ConditionChecker classes.

The RuleApplyer class is located in the main.rule package and is instantiated by the RoleAnalyser. It takes as its input the map of analysed statements returned by the StatementAnalsyer, the map of variables and their declared roles, and a list of the methods in the analysed file. The RuleApplyer instantiates one of ten role checkers for each variable depending on the declared role. If a non-recognized role is declared for a variable, the role checker OtherRole is created. Each role checker returns a HashMap for each analysed file, containing an ArrayList for each variable returned by the role checker.

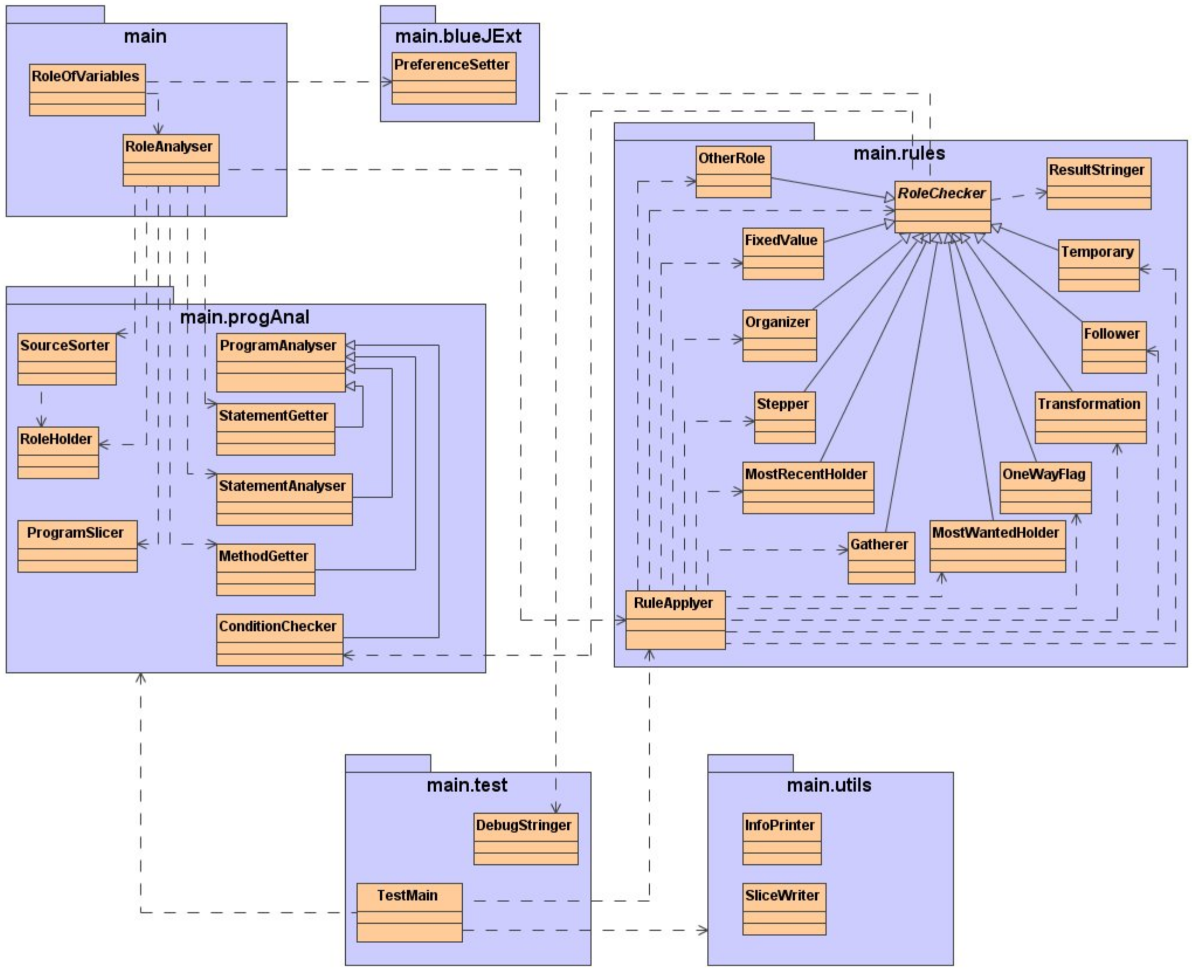


Figure 7. Class diagram for Roles of Variables software

Each role checker extends the abstract class RoleChecker, and takes as its inputs the analysed map, the list of variables, and the list of methods. It sets flags depending on the conditions listed in section 5.2, by instantiating the ConditionChecker in package main.progAnal, and calling its methods as required. Each role checker checks whether the role being played corresponds to its type (e.g. FixedValue checks that role is *fixed value*). If not, it tests to see what role may be being played, by executing the rule sets for each of the other roles. This process could be made more efficient if the reason for incorrect role detection is used to control order that subsequent rule sets are executed, but this has not been implemented in the current software. The ResultStringer is used by role checkers generate a message suggesting the role being played in the event that an incorrect role is detected, and to generate an offending statement in the event that a detected error is due to the lack of a given statement rather than its presence (see rules for Organizer & Stepper). Each role checker returns an ArrayList for the variable stating whether the declaration is correct, and in the event that it is not, also supplying the reason for incorrect declaration, the offending statement, and the a suggested role.

The ConditionChecker extends the ProgramAnalyser and is found in the main.progAnal package as although it is used for role checking, it further analyses the program source code. It takes as its inputs a variable to be checked, the list of methods in the source code, and the analysed map returned by the StatementAnalyser. It analyses how and where the variable is assigned and used within the program, and returns evidence of this in terms of statements and their locations within the source code, when given conditions are fulfilled.

The packages main.test and main.utils are not included in the roles_of_variables.jar file, but were used in the development of the software. In the main.test package, the TestMain class contains tests for overall role checking functionality. In addition the class contains test for source break down, relevant statement identification and program slicing. These latter tests check outputs from the program against external files. The DebugStringer class generates detailed messages relating to variable assignment and use for debugging purposes. In the main.Utils package, the InfoPrinter class contains methods to output the results from various stages of the role checking process, to aid with debugging. The SliceWriter class generates program slices as described in section 5.1.

Exhaustive testing was undertaken for each of the training programs, by creating ten directories (i.e. one for each role type) in each of which all variables were declared with the same role. E.g. a directory was created where all variables annotated with the role *fixed value*, another was created in which all variables were said to have the role *gatherer*, and so on. Checks were made to ensure that all incorrectly declared variables are identified as such and that software suggests the correct role for those variables.

7 Conclusions

Roles of Variables is a new concept that has been shown in [1] & [6] to improve students' "programming knowledge" [2], by directly teaching how to construct programs from abstract concepts. This paper has presented an extension to the BlueJ programming environment that checks the roles played by primitive type variables in simple Java programs, and prompts the programmer in the event that the role of the variable appears to be different from that perceived by the programmer. In the event that role is not recognized, or not specified, the program will identify an unrecognized role declaration and suggest a possible role for the variable in question. This provides some scope for investigation into its potential use with animation software for auto generation of comprehension aiding program animations.

Whilst the program analyses the roles played by variables in the seventeen training programs with 100% reliability, initial testing has confirmed that the program is clearly biased towards this training data. Further, it has highlighted bugs, and anomalies in individual programming styles, that have required correction. In addition, some limitations continue to exist with respect to the Java programming paradigm (not withstanding the non-support of object references). This paper concurs with [11] that automatic role checking of variables is a non-trivial task, but it also shows how data flow analysis can be used successfully to check the roles being played, providing that a sufficient bank of role checking rules exists for the files to be analysed.

Further training of the software is required to improve the reliability with which roles can be checked, both in terms of catering for additional variable assignment and usage conditions, and in terms of catering for the totality of Java coding conventions. This would require a vastly increased amount of training data to be collected, and would probably benefit from the addition of a machine learning strategy as suggested in [11], because the amount of data required to be analysed for >95% reliability may prove unwieldy for manual analysis by one or more individuals. Finally, the results of studies yet to be published in [12], may provide the basis for the addition of role checking rules for variables referring to objects.

References

[1] Kuittinen M., Sajaniemi J. (2004) Teaching Roles of Variables in Elementary Programming Courses. ITiCSE 2004, *Proceedings of the 9th Annual Conference on Innovation and Technology in Computer Science Education, Leeds, UK, June 2004. Association for Computing Machinery, 57-61.*

[2] J. Sajaniemi. (2003) *Roles of variables home page.* Available: www.cs.joensuu.fi/˜saja/var roles/ (Accessed Nov. 12th, 2003).

[3] J. Sajaniemi. An empirical analysis of Roles of Variables in novice-level procedural programs. *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, pages 37–39. IEEE Computer Society, 2002.

[4] Ben-Ari M., Sajaniemi J. (2004) Roles of Variables from the perspective of CS Educators. *ITiCSE 2004, Proceedings of the 9th Annual Conference on Innovation and Technology in Computer Science Education, Leeds, UK, June 2004. Association for Computing Machinery, 52-56.*

[5] Stützle T., Sajaniemi J. (2005) An Empirical Evaluation of Visual Metaphors in the Animation of Roles of Variables. *Informing Science Journal, 8, 87-100.*

[6] Sajaniemi J., Kuittinen M. (2005) An Experiment on Using Roles of Variables in Teaching Introductory Programming. *Computer Science Education 15(1), 59-82.*

[7] N.Pennington. (1987) Comprehension strategies in programming. *G. M. Olson, S. Sheppard, and E. Soloway, editors, Empirical Studies of Programmers: Second Workshop, pages 100–113. Ablex Publishing Company.*

[8] J. Sajaniemi and M. Kuittinen. Program Animation Based on the Roles of Variables. *Proceedings of the ACM 2003 Symposium on Software Visualization (SoftVis 2003), pages 7-16. Association for Computing Machinery, 2003.*

[9] Nevalainen S., Sajaniemi J. (2005) Short-Term Effects of Graphical versus Textual Visualisation of Variables on Program Perception. *Accepted to the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005), Brighton, UK, June 2005.*

[10] Byckling P., Sajaniemi J. (2005) Using Roles of Variables in Teaching: Effects on Program Construction. *Accepted to the 17th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2005), Brighton, UK, June 2005.*

[11] Gerdt P., Sajaniemi J. (2004) An Approach to Automatic Detection of Variable Roles in Program Animation. *Proceedings of the Third Program Visualization Workshop (ed. A. Korhonen), Research Report CS-RR-407, Department of Computer Science, University of Warwick, UK, 86-93.*

[12] Byckling P., Gerdt P., Sajaniemi J. (2005) Roles of Variables in Object-Oriented Programming. *Accepted to the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005) Educators Symposium, San Diego, California, USA, October 2005.*

[13] WEISER, M. (1984) Program slicing. *IEEE Trans. Softw. Eng. SE-lo, 4 (July 1984), 352-357.*

[14] Jon Beck, David Eichmann. (1993) Program and interface slicing for reverse engineering, *Proceedings of the 15th international conference on Sofrtware Engineering, p.5-9, 518, May 17-21, 1993, Baltimore, Maryland, United States.*

[15] Moonen L. (1996) A Generic Architecture for Data Flow Analysis to Support Reverse Engineering. *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam, 25-26 September 1997.*

[16] Simonyi C. (1999) *Explication of the Hungarian notation identifier naming convention.* Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs600/html/HungaNotat.asp (Accessed July 26, 2005)

[17] Blue J (2005) *BlueJ - How to write extensions.* Available: www.bluej.org/extensions/extensions.html. (Accessed July 25, 2005).

[18] K. Gallagher, J. Lyle. (1991). "Using program slicing in software maintenance," *IEEE Trans. Softw. Eng., 17(8), pp 751- 761, Aug. 1991.*

[19] E. Chikofsky and J. Cross. (1990) "Reverse engineering and design recovery: a taxonomy;" *IEEE Softw., 7(l), pp 13-17, Jan. 1990.*

[20] Kuro5hin.org. (2002) *Hungarian Notation.* Available: www.kuro5hin.org/story/2002/4/12/153445/601 . (Accessed August 22, 2005)

[21] Onslow R, Davis B, Martin R, C. (1998) *Hungarian Notation – The Good, The Bad, and The Ugly*. Available http://ootips.org/hungarian-notation.html (Accessed August 22, 2005)

[22] P. Mulholland. (1998) A Principled Approach to the Evaluation of SV: A Case Study in Prolog. *J. Stasko, J. Dominique, M. H. Brown, and B. A. Price, editors, Software Visualization - Programming as a Multimedia Experience, pages 439-451. The MIT Press.*

[23] Mcann, L. (2001) *How to Develop a Simple Java Program*. Available: http://www.cs.arizona.edu/people/mccann/develop_java.html (Accessed August 30, 2005)

Bibliography

Barnes and Kolling, (2005). *Objects First With Java – A practical introduction using BlueJ*, Pearson Education.

Covers the basics of Java programming and use of BlueJ.

Pressman and Ince, (2000). *Software Engineering – A practitioner's approach, European Adaptation*, McGraw Hill.

Covers good Software Engineering practice.

Nielson, H. R. Nielson, and C. Hankin, (1998). *Principles of Program Analysis.* Springer-Verlag, Heidelberg, 1998.

Covers the principles of Program Analysis.

Avison, D.E, (1989) *The Project Report – A guide for students*, Available: www.cs.kent.ac.uk.chain.kent.ac.uk/teaching/prospectus/project-report/contents.html, (Accessed September 1, 2005).

Provides guidance on good report writing practice.

Appendix 1 RoVs Training Programs

A1.1. BubbleSort.Java

```
public class BubbleSort {
   public int[] sort(int[] source) {
      /* variables */
      int i, j, temp; //%%temp%%temporary
      boolean swapped; //%%swapped%%stepper%%
      int len = source.length; //%%len%%fixed value%%
      int[] target = new int[len]; //%%target%%Organizer%%
      for (i = 0; i < len; i++) {
         target[i] = source[i];
      }
      for (i = len-1;  i > 0; i--) {
         swapped = false;
         for (j = 0; j < i; j++) {
            if (target[j] > target[j+1]) {
               temp = target[j];
               target[j] = target[j+1];
               target[j+1] = temp;
               swapped = true;
            }
         }
         if (!swapped) {
            break;
         }
      }
      return target;
   }
}
```

A1.2. Closest.Java

```
import java.io.*;
public class Closest {
   int original, // %%original%%fixed value%%
   closest;      // %%closest%%most wanted holder%%

   public void findClosest() {
      int data;         // %%data%%most recent holder%%
      System.out.print("Enter any number: ");
      original = getInput();
      System.out.print("Enter a positive number (negative to end): ");
      data = getInput();
      closest = data;
      while (data >= 0) {
         if (Math.abs(data-original) < Math.abs(closest-original)
            closest = data;
         System.out.print("Enter a positive number (negative to end): +
            ");
         data = getInput();
      }
      if (closest < 0) System.out.println("No positive value entered.");
      else System.out.println("The closest to " + original + " was " +
         closest)
   }

   public static void main(String argv[]) {
      Closest c = new Closest();
      c.findClosest();
   }

   public int getInput() {
      BufferedReader in = new
         BufferedReader(newInputStreamReader(System.in));
      int returnInt = -1;
      try {
         returnInt = in.read() - 48;
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
      return returnInt;
   }
}
```

A1.3. DiceGame.Java

```
import java.util.Random;

public class DiceGame {
    int die1;// Values of dice thrown %%die1%%most recent holder%%
    int die2;    //%%die2%%most recent holder%%
    int total1;// Sums of values of each player %%total1%%gatherer%%
    int total2; //%%total2%%gatherer%%
    boolean firstPlayer; // First player's
turn ? %%firstPlayer%%stepper%%
    Random r = null;

    private int randomize(int i) {
        if (r == null)
        r = new Random();
        return r.nextInt(i);
    }

    public void play() {
        total1 = 0;
        total2 = 0;
        firstPlayer = true;
        while(total1 < 100 && total2 < 100) {
            die1 = randomize(6) + 1;
            die2 = randomize(6) + 1;
            if (firstPlayer)
            System.out.print("First");
            else
            System.out.print("Second");
            System.out.println(" player throws: " + die1 + ", " + die2);
            if (firstPlayer) total1 = total1 + die1 + die2;
            else total2 = total2 + die1 + die2;
            firstPlayer = !firstPlayer;
        }
        System.out.println("First player: " + total1);
        System.out.println("Second player: " + total2);
        if (total1 > total2) System.out.print("First");
        else System.out.print("Second");
        System.out.println(" player won the game!");
    }


    public static void main(String argv[]) {
        DiceGame dc = new DiceGame();
        dc.play();
    }
}
```

A1.4 DivMod7.java

```
import java.io.*;
public class DivMod7 {
   private final int divisor = 7; // Divisor to be used
   public void calculateDivMod7() {
      float value,           // Input value read
      q, r,            // Current quotient and remainder
      lq, lr;          //Largest quotient and remainder
      //%%value%%most recent holder%%
      //%%q%%transformation%%
      //%%r%%transformation%%
      //%%lq%%most wanted holder%%
      //%%lr%%most wanted holder%%
      lq = 0;
      lr = 0;
      System.out.print("Enter value (negative number to terminate): ");
      value = getInput();
      while (value >= 0) {
         q = value / divisor;
         System.out.print("Quotient = " + q);
         r = value % divisor;
         System.out.println(", Remainder = " + r);
         if (q > lq) lq = q;
         if (r > lr) lr = r;
         System.out.print("Enter value (negative number to terminate)"
            ":");
         value = getInput();
      }
      System.out.println("Largest quotient is " + lq +
         " and largest remainder is " + lr);
   }
   public static void main(String argv[]) {
      DivMod7 dv7 = new DivMod7();
      dv7.calculateDivMod7();
   }

   public float getInput(){
      float returnFloat = 0;
      BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
      try {
         String inputString = in.readLine();
         Float tempFloat = new Float(inputString);
         returnFloat = tempFloat.floatValue();
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
      return returnFloat;
   }
}
```

A1.5 Doubles.java

```
import java.io.*;
public class Doubles {
   private int counter; //%%counter%%stepper%%
   private int number; //%%number%%most recent holder%%

   public Doubles() {
      do {
         System.out.print("Give amount of loops: ");
         counter = getInput();
      }
      while (counter < 0);
      while (counter > 0) {
         System.out.print("Give some number: ");
         number = getInput();
         System.out.println("Two times " + number + " is " + 2*number);
         counter = counter - 1;
      }
   }

   public int getInput() {
      int returnInt = 0;
      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      try {
         returnInt = new Integer(in.readLine()).intValue();
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
      return returnInt;
   }
}
```

A1.6 Fibonacci.java

```
import java.io.*;
public class Fibonacci {
   private int last_fib;   /*%%last_fib%%follower%%*/
   private int fib; //%%fib%%gatherer%%
   private int temp;       //%%temp%%temporary%%
   private int number;     //%%number%%fixed value%%
   private int i;          //%%i%%stepper%%
   /**
     * Constructor for objects of class Fibonachi
    */
   public Fibonacci()
      last_fib = 1;
      fib = 1;
      number = getNumber();
      if (number <= 2) {
         System.out.println("The first and second numbers are 1.");
      } else {
         System.out.println("Value 1 is: 1");
         System.out.println("Value 2 is: 1");
         for (i = 3; i <= number; i++) {
            temp = last_fib;
            last_fib = fib;
            fib = fib + temp;
            System.out.println("Value " + i + " is: " + fib);
         }
      }
   }

   /**
    * Read number of integers in sequence from terminal
    * @return  The number of values to be in the sequence
    */
   private int getNumber() {
      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      int n = 0;
      while (n < 1) {
         String tempN = "";
         System.out.print("Enter number in sequence: ");
         try {
            tempN = in.readLine(); /* this is a comment,
            so is this,
            this too*/} catch (IOException e) {
            System.out.println(e.getMessage());
         }
         Integer tempInt = new Integer(tempN);
         n = tempInt.intValue();
      }
      return n;
   }
}
```

A1.7 Growth.java

```
import java.io.*;

public class Growth {
   //Growth of capital on a bank account
   private float capital; //Capital on the bank
account %%capital%%gatherer%%
   private float percent; //Interest rate %%percent%%fixed value%%
   private float factor; //Factor for yearly growth
   //%%factor%%fixed value%%
   private float interest; //Interest in current year
   //%%interest%%transformation%%
   private int years; //Time to consider %%years%%fixed value%%
   private int i; //Year counter %%i%%stepper%%
   private boolean inputOk; //Is input valid ? %%inputOk%%One Way Flag%%

   public Growth() {
      System.out.println("Enter capital (positive or negative): ");
      capital = getFloat();
      inputOk = false;
         while (!inputOk) {
            System.out.print("Enter interest rate (%): ");
            percent = getFloat();
            System.out.print("Enter time (years) : ");
            years = getInt();
            if (percent > 0 && years > 0) inputOk = true;
            if (!inputOk) {
               System.out.println();
               System.out.println("Invalid data. Re-enter.");
            }
         }
      System.out.println();
      factor = percent / 100;
      for (i = 1; i <= years; i++) {
         interest = capital * factor;
         capital = capital + interest;
         System.out.println("After " + i + " years: interest is " +
            interest + " and total capital is " + capital);
      }
   }

   private float getFloat() {
      float returnFloat = 0;
      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      try {
         String inputString = in.readLine();
         Float tempFloat = new Float(inputString);
         returnFloat = tempFloat.floatValue();
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
      return returnFloat;
   }

   private int getInt() {
      int returnInt = 0;
      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      try {
         returnInt = new Integer(in.readLine()).intValue();
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
      return returnInt;
   }
}
```

A1.8 Histogram.java

```
import java.io.*;

public class Histogram {
   //Draw a histogram

   private final int longest = 40;  //Longest bar
   private float[] amount = new float[12]; //Data for drawing
      //%%amount%%fixed value%%
   private float max; //Maximum data element %%max%%most wanted holder%%
   private int month; //Current month %%month%%stepper%%
   private int i; //%%i%%stepper%%

   public Histogram() {
      max = 0;
      for (month = 1; month <= 12; month ++) {
         System.out.println("Enter amount for month " + month + ": ");
         amount[month - 1] = getInput();
         if (max < amount[month - 1]) {
            max = amount[month - 1];
         }
      }
      System.out.println();
      for (month = 1; month <= 12; month++) {
         System.out.print(month + ": ");
         if (month < 10) System.out.print(" ");
         for (i = 0; i <= amount[month-1] / max * longest; i++)
                  System.out.print('*');
         System.out.println();
      }
   }

   private float getInput(){
      float returnFloat = 0;
      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      try {
         String inputString = in.readLine();
         Float tempFloat = new Float(inputString);
         returnFloat = tempFloat.floatValue();
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
      return returnFloat;
   }
}
```

A1.9 Lexical.java

```
import java.util.Set;
import java.io.*;

public class Lexical {

//Lexical type recognition; spaces ignored

   private String lt; //Lexical type of current character
      //%%lt%%most recent holder%%
   private char c; //Current character from input
   //%%c%%most recent holder%%
   //%%op%%fixed value%%
   //%%digit%%fixed value%%
   //%%letter%%fixed value%%

   public Lexical() {
      char[] op = {43,45,42,47};
      char[] digit = {48,49,50,51,52,53,54,55,56,57};
      char[] letter = getLetters();
      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      System.out.print("Enter expression, terminate with '.' : ");
      String stringChar = "";
      try {
         stringChar = in.readLine();
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
      int i = 0;
      do {
         c = stringChar.charAt(i);
         if (c == ' ') lt = "Space";
         else if (c == 46) lt = "Period";
         else if (is(c, op)) lt = "Op";
         else if (is(c, digit)) lt = "Digit";
         else if (is(c, letter)) lt = "Letter";
         else lt = "Error";
         if (!lt.equals("Space")) System.out.println(lt);
         i ++;
      } while (!lt.equals("Error") && !lt.equals("Period"));
   }

   private char[] getLetters() {
      char[] returnArray = new char[52];
      int i = 0;
      for (char ch = 65; ch <= 122; ch++) {
         returnArray[i] = ch;
         i ++;
         if (i == 26) ch = 96;
      }
      return returnArray;
   }

   private boolean is(char inputChar, char[] inputArray) {
      boolean is = false;
      for (int i = 0; i < inputArray.length; i++) {
         char tempChar = inputArray[i];
         if (inputChar == tempChar) {
            is = true;
            break;
         }
      }
      return is;
   }
}
```

A1.10 Multiplication.java

```
import java.io.*;
   public class Multiplication {
   //Print a multiplication table

   private int size;// Size of table %%size%%fixed value%%
   private int row; // Row index     %%row%%stepper%%
   private int col; // Column index  %%col%%stepper%%

   public Multiplication() {
      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      System.out.print("Enter size: ");
      try {
         size = new Integer(in.readLine()).intValue();
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
      System.out.println("              Multiplication table ");
      System.out.print("\t");
      for (col = 1; col <= 10; col++) System.out.print(col + "\t");
      System.out.println();
      for (row = 1; row <= size; row++) {
         System.out.print(row + "\t");
         for (col = 1; col <= 10; col++) System.out.print(row*col +
            "\t");
         System.out.println();
      }
   }
}
```

A1.11 Number.java

```
import java.io.*;

public class Number {
   //Read a number with embedded commas, e.g., 123,456
   private final char sep = 44;
   private char c; //The character read %%c%%most recent holder%%
   private int val; //Accumulated value of number %%val%%gatherer%%

   public Number() {
      val = 0;
      System.out.print("Enter number: ");
      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      String inputLine = "";
      try {
         inputLine = in.readLine();
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
      int i = 0;
      do {
         c = inputLine.charAt(i);
         if (c != sep) val = val * 10 + c - 48;
         i++;
      }while ((c >= 48 && c <= 57 || c == sep) &&
         i < inputLine.length());
      System.out.println("Value is " + val);
   }
}
```

A1.12 Occur.java

```
import java.io.*;
public class Occur {
   //Count occurences of a value in an array

   private final int last = 7;
   private int[] value; //Array of values %%value%%fixed value%%
   private int i; //Array index %%i%%stepper%%
   private int key; //Value to search for %%key%%fixed value%%
   private int count; //Count of occurences %%count%%stepper%%

   public Occur() {
      value = new int[last];
      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      System.out.println("Enter " + last + " values: ");
      for (i = 0; i < last; i++) {
         try {
            value[i] = new Integer(in.readLine()).intValue();
         } catch(IOException e) {
            System.out.println(e.getMessage());
         }
      }
      System.out.println();
      System.out.print("Enter value to search for: ");
      try {
         key = new Integer(in.readLine()).intValue();
      } catch(IOException e) {
         System.out.println(e.getMessage());
      }
      count = 0;
      for (i = 0; i < last; i++) if (value[i] == key) count ++;
      System.out.println("There are " + count + " occurences of " +
            key + " in the array");
   }
}
```

A1.13 ProgramTime.java

```
import java.io.*;
public class ProgramTime{
   //Transform time to seconds
   private int hours; //%%hours%%fixed value%%
   private int mins; //%%mins%%fixed value%%
   private int secs; //%%secs%%fixed value%%

   public ProgramTime() {
      BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
      System.out.print("Enter time separated by ':'(hour min sec , " +
         "e.g., 13:30:5): ");
      String time = null;
      try {
         time = in.readLine();
      } catch(IOException e) {
         System.out.println(e.getMessage());
      }
      boolean isHour = true;
      String tempString = "";
      for (int i = 0; i < time.length(); i++) {
         tempString += time.substring(i, i+1);
         if (tempString.endsWith(":")) {
            tempString = tempString.substring(0,
               tempString.length()-1);
            if (isHour) {
               hours = new Integer(tempString).intValue();
               isHour = false;
               tempString = "";
            }
            else if (!isHour) {
               mins = new Integer(tempString).intValue();
               tempString = "";
            }
         } else if(i == time.length()-1) {
            secs = new Integer(tempString).intValue();
         }
      }
      int seconds = secs + 60 * (mins + 60 * hours);
      if (hours < 10) {
         if (mins < 10) {
               System.out.println("At 0" + hours + ":0" + mins + ":" +
                  secs + "\t" + seconds + "seconds has elapsed.");
         } else {
            System.out.println("At 0" + hours + ":" + mins + ":" + secs
               + "\t" + seconds + "seconds has elapsed.");
         }
      } else {
         if (mins < 10) {
            System.out.println("At " + hours + ":0" + mins + ":" +
               secs + "\t" + seconds + "seconds has elapsed.");
         } else {
            System.out.println("At " + hours + ":" + mins + ":" +
               secs + "\t " + seconds + " seconds has elapsed.");
         }
      }
   }
}
```

A1.14 Saw.java

```
import java.io.*;
public class Saw {
   //Read a sequence of values and check if they form a 'saw'
   // i.e. adjacent values go up and then down.
   private final int last = 7;
   private int[] value = new int[last]; //Values to be checked
   //%%value%%fixed value%%
   private int i; //Index of array %%i%%stepper%%
   private boolean up; //Current direction is up ? %%up%%stepper%%
   private boolean ok; //Does saw property still hold ?
   //%%ok%%one way flag%%

   public Saw() {
      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      System.out.println("Enter " + last + " values:");
      for (i=0; i < last; i++) {
         try {
            value[i] = new Integer(in.readLine()).intValue();
         } catch (IOException e) {
            System.out.println(e.getMessage());
         }
      }
      if (value[0] < value[1]) up = true;
      if (value[0] != value[1]) ok = true;
      i = 1;
      while (ok && i < last - 1) {
         ok = (up && (value[i] > value[i+1])) ||
            (!up && (value[i] < value[i+1]));
         up = !up;
         i ++;
      }
      System.out.print("Values ");
      if (!ok) System.out.print("do not ");
      System.out.print("form a saw.");
   }
}
```

A1.15 SmoothedAverage.java

```
import java.io.*;
public class SmoothedAverage {
   // Largest average of three consecutive months
   public void calculateAverage() {
      int month;              // Current month %%month%%stepper%%
      float current, previous, preceding, average, largest;
      //Data for three months
      //%%current%%most recent holder%%mrh
      //%%previous%%follower%%
      //%%preceding%%follower%%follower
      //Current average%%average%%transformation%%transformation
      //Largest one found so far%%largest%%most wanted holder%%
      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      try {
         System.out.print("Enter 1. value: ");
         preceding = new Float(in.readLine()).floatValue();
         System.out.print("Enter 2. value: ");
         previous = new Float(in.readLine()).floatValue();
         System.out.print("Enter 3. value: ");
         current = new Float(in.readLine()).floatValue();
         largest = (current + previous + preceding) / 3;
         for (month = 4; month <= 12; month++) {
            preceding = previous;
            previous = current;
            System.out.print("Enter " + month + ". value: ");
            current = new Float(in.readLine()).floatValue();
            average = (current + previous + preceding) / 3;
            if (average > largest) largest = average;
         }
         System.out.println("Largest three month average was " +
            largest);
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
   }

   public static void main(String argv[]) {
      SmoothedAverage sa = new SmoothedAverage();
      sa.calculateAverage();
   }
}
```

A1.16 Square.java

```
import java.io.*;
public class Square {
   public static void main(String argv[]) {
      final int maxSide = 78;// Max length for sides
      char c;                // Char to be used for drawing
                             //%%c%%fixed value%%
      int side;              // Length of sides %%side%%fixed value%%
      int i, j;              // Counters for side lengths %%i%%stepper%%
                             //%%j%%stepper

      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      System.out.print("Enter character for drawing: ");
      try {
         c = in.readLine().charAt(0);
         System.out.print("Enter side length: ");
         side = new Integer(in.readLine()).intValue();
         while (side < 0 || side > maxSide) {
            System.out.print("Length incorrect. Re-enter: ");
            side = new Integer(in.readLine()).intValue();
         }
         System.out.println();
         for (i = 1; i <= side; i++) {
            for (j = 1; j <= side; j++)
               System.out.print(c + " ");
            System.out.println();
         }
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
   }
}
```

A1.17 TwoLargest.java

```
import java.io.*;

public class TwoLargest {
   //Largest and second largest input value
   private int value;      //Input value %%value%%most recent holder%%
   private int largest;    //Largest input value so far
                           //%%largest%%most wanted holder%%
   private int second;     //Second largest so far
                           //%%second%%most wanted holder%%

   public TwoLargest() {
      BufferedReader in = new BufferedReader(new
         InputStreamReader(System.in));
      largest = -1;
      second  = -1;
      try {
         value = new Integer(in.readLine()).intValue();
         while (value >= 0) {
            if (value > largest) {
               second = largest;
               largest = value;
            }
            else if (value > second) second = value;
            System.out.print("Enter value (negative to end): ");
            value = new Integer(in.readLine()).intValue();
         }
         if (largest > 0) System.out.println("Largest was " + largest);
         if (second  > 0) System.out.println("Second largest was " +
            second);
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
   }
}
```

Appendix 2 Example program slices for *fixed value* variables

A2.1 Fixed Value

```
Variable: len
public class BubbleSort {
   public int[] sort(int[] source) {
      int len = source.length;
      int[] target = new int[len];
      for (i = 0; i < len; i++) {
      }
      for (i = len-1;  i > 0; i--) {
      for (j = 0; j < i; j++) {
         if (target[j] > target[j+1]) {
         }
      }
         if (!swapped) {
         }
      }
   }
}
```

```
Variable: original
public class Closest {
   int original, closest;
   public void findClosest() {
      original = getInput();
      while (data >= 0) {
         if (Math.abs(data-original) < Math.abs(closest-original))
         closest = data;
      }
      else System.out.println("The closest to " + original + " was " +
         closest);
   }
   public static void main(String argv[]) {
   }
   public int getInput() {
      try {
      }
      catch (IOException e) {
      }
   }
}
```

```
Variable: number
public class Fibonachi {
   private int number;
   public Fibonachi() {
      number = getNumber();
      if (number <= 2) {
      }
      else {
         for (i = 3; i <= number; i++) {
         }
      }
   }
   private int getNumber() {
      while (n < 1) {
         try {
         }
         catch (IOException e) {
         }
      }
   }
}
```

Variable: percent
```
public class Growth {
   private float percent;
   public Growth() {
      while (!inputOk) {
         percent = getFloat();
         if (percent > 0 && years > 0) inputOk = true;
         if (!inputOk) {
         }
      }
      factor = percent / 100;
      for (i = 1; i <= years; i++) {
      }
   }
   private float getFloat() {
      try {
      }
      catch (IOException e) {
      }
   }
   private int getInt() {
      try {
      }
      catch (IOException e) {
      }
   }
}
```

Variable: factor
```
public class Growth {
   private float factor;
   public Growth() {
      while (!inputOk) {
         if (!inputOk) {
         }
      }
      factor = percent / 100;
      for (i = 1; i <= years; i++) {
         interest = capital * factor;
      }
   }
   private float getFloat() {
      try {
      }
      catch (IOException e) {
      }
   }
   private int getInt() {
      try {
      }
      catch (IOException e) {
      }
   }
}
```

Variable: years
```
public class Growth {
   private int years;
   public Growth() {
      while (!inputOk) {
         years = getInt();
         if (percent > 0 && years > 0) inputOk = true;
         if (!inputOk) {
         }
      }
        for (i = 1; i <= years; i++) {
        }
    }
    private float getFloat() {
        try {
        }
        catch (IOException e) {
        }
    }
    private int getInt() {
        try {
        }
        catch (IOException e) {
        }
    }
}
```

Variable: amount
```
public class Histogram {
   private float[] amount = new float[12];
   public Histogram() {
      for (month = 1; month <= 12; month ++) {
         amount[month - 1] = getInput();
         if (max < amount[month - 1]) {
            max = amount[month - 1];
         }
      }
      for (month = 1; month <= 12; month++) {
         for (i = 0; i <= amount[month-1] / max * longest; i++)
                  System.out.print('*');
      }
   }
   private float getInput(){
      try {
      }
      catch (IOException e) {
      }
   }
}
```

Variable: op
```
public class Lexical {
   public Lexical() {
      char[] op = {43,45,42,47};
      try {
      }
      catch (IOException e) {
      }
      do {
         else if (is(c, op)) lt = "Op";
      }
      while (!lt.equals("Error") && !lt.equals("Period"));
   }
   private char[] getLetters() {
      for (char ch = 65; ch <= 122; ch++) {
      }
   }
   private boolean is(char inputChar, char[] inputArray) {
      for (int i = 0; i < inputArray.length; i++) {
         if (inputChar == tempChar) {
         }
       }
    }
}
```

Variable: digit
```
public class Lexical {
   public Lexical() {
      char[] digit = {48,49,50,51,52,53,54,55,56,57};
      try {
      }
      catch (IOException e) {
      }
      do {
         else if (is(c, digit)) lt = "Digit";
      }
      while (!lt.equals("Error") && !lt.equals("Period"));
   }
   private char[] getLetters() {
      for (char ch = 65; ch <= 122; ch++) {
      }
   }
   private boolean is(char inputChar, char[] inputArray) {
      for (int i = 0; i < inputArray.length; i++) {
         if (inputChar == tempChar) {
         }
      }
   }
}
```

Variable: letter
```
public class Lexical {
   public Lexical() {
      char[] letter = getLetters();
      try {
      }
      catch (IOException e) {
      }
      do {
         else if (is(c, letter)) lt = "Letter";
      }
      while (!lt.equals("Error") && !lt.equals("Period"));
   }
   private char[] getLetters() {
      for (char ch = 65; ch <= 122; ch++) {
      }
   }
   private boolean is(char inputChar, char[] inputArray) {
      for (int i = 0; i < inputArray.length; i++) {
         if (inputChar == tempChar) {
         }
      }
   }
}
```

Variable: size
```
public class Multiplication {
   private int size;
   public Multiplication() {
      try {
         size = new Integer(in.readLine()).intValue();
      }
      catch (IOException e) {
      }
      for (row = 1; row <= size; row++) {
      }
   }
}
```

Variable: value
```
public class Occur {
   private int[] value;
   public Occur() {
      value = new int[last];
      for (i = 0; i < last; i++) {
         try {
            value[i] = new Integer(in.readLine()).intValue();
         }
         catch(IOException e) {
         }
      }
      try {
      }
      catch(IOException e) {
      }
      for (i = 0; i < last; i++) if (value[i] == key) count ++;
   }
}
```

Variable: key
```
public class Occur {
   private int key;
   public Occur() {
      for (i = 0; i < last; i++) {
         try {
         }
         catch(IOException e) {
         }
      }
      try {
         key = new Integer(in.readLine()).intValue();
      }
      catch(IOException e) {
      }
      for (i = 0; i < last; i++) if (value[i] == key) count ++;
      System.out.println("There are " + count + " occurences of " +
         key + " in the array");
   }
}
```

Variable: hours
```
public class ProgramTime{
   private int hours;
   public ProgramTime() {
      try {
      }
      catch(IOException e) {
      }
      for (int i = 0; i < time.length(); i++) {
         if (tempString.endsWith(":")) {
            if (isHour) {
               hours = new Integer(tempString).intValue();
            }
            else if (!isHour) {
            }
         }
      }
      int seconds = secs + 60 * (mins + 60 * hours);
      if (hours < 10) {
         if (mins < 10) {
            System.out.println("At 0" + hours + ":0" + mins + ":" +
               secs + "\t" + seconds + "seconds has elapsed.");
         }
         else {
            System.out.println("At 0" + hours + ":" + mins + ":" +
               secs + "\t" + seconds + "seconds has elapsed.");
         }
      }
      else {
         if (mins < 10) {
            System.out.println("At " + hours + ":0" + mins + ":" +
               secs + "\t" +    seconds + "seconds has elapsed.");
         }
         else {
            System.out.println("At " + hours + ":" + mins + ":" +
               secs + "\t " + seconds + " seconds has elapsed.");
         }
      }
   }
}
```

Variable: mins

```
public class ProgramTime{
   private int mins;
   public ProgramTime() {
      try {
      }
      catch(IOException e) {
      }
      for (int i = 0; i < time.length(); i++) {
         if (tempString.endsWith(":")) {
            if (isHour) {
            }
            else if (!isHour) {
               mins = new Integer(tempString).intValue();
            }
         }
      }
      int seconds = secs + 60 * (mins + 60 * hours);
      if (hours < 10) {
         if (mins < 10) {
            System.out.println("At 0" + hours + ":0" + mins + ":" +
               secs + "\t" + seconds + "seconds has elapsed.");
         }
         else {
            System.out.println("At 0" + hours + ":" + mins + ":" +
               secs +"\t" + seconds + "seconds has elapsed.");
         }
      }
      else {
         if (mins < 10) {
            System.out.println("At " + hours + ":0" + mins + ":" +
               secs +"\t" + seconds + "seconds has elapsed.");
         }
         else {
            System.out.println("At " + hours + ":" + mins + ":" +
               secs +"\t " + seconds + " seconds has elapsed.");
         }
      }
   }
}
```

Variable: secs
```
public class ProgramTime{
   private int secs;
   public ProgramTime() {
      try {
      }
      catch(IOException e) {
      }
      for (int i = 0; i < time.length(); i++) {
         if (tempString.endsWith(":")) {
            if (isHour) {
            }
            else if (!isHour) {
            }
         }
         else if(i == time.length()-1)
            secs = new   Integer(tempString).intValue();
         int seconds = secs + 60 * (mins + 60 * hours);
         if (hours < 10) {
            if (mins < 10) {
               System.out.println("At 0" + hours + ":0" + mins + ":" +
                  secs + "\t" + seconds + "seconds has elapsed.");
            }
            else {
               System.out.println("At 0" + hours + ":" + mins + ":" +
                  secs + "\t" + seconds + "seconds has elapsed.");
            }
         }
         else {
            if (mins < 10) {
               System.out.println("At " + hours + ":0" + mins + ":" +
                  secs + "\t" +seconds + "seconds has elapsed.");
            }
            else {
               System.out.println("At " + hours + ":" + mins + ":" +
                  secs + "\t " + seconds + " seconds has elapsed.");
            }
         }
      }
   }
```

Variable: value
```
public class Saw {
   private int[] value = new int[last];
   public Saw() {
      for (i=0; i < last; i++) {
         try {
            value[i] = new Integer(in.readLine()).intValue();
         }
         catch (IOException e) {
         }
      }
      if (value[0] < value[1]) up = true;
      if (value[0] != value[1]) ok = true;
      while (ok && i < last - 1) {
         ok = (up && (value[i] > value[i+1])) ||
            (!up && (value[i] < value[i+1]));
      }
   }
}
```

Variable: c
```
public class Square {
   public static void main(String argv[]) {
      char c;
      try {
         c = in.readLine().charAt(0);
         while (side < 0 || side > maxSide) {
         }
         for (i = 1; i <= side; i++) {
            for (j = 1; j <= side; j++)System.out.print(c + " ");
         }
      }
      catch (IOException e) {
      }
   }
}
```

Variable: side
```
public class Square {
   public static void main(String argv[]) {
      int side;
      try {
         side = new Integer(in.readLine()).intValue();
         while (side < 0 || side > maxSide) {
            side = new Integer(in.readLine()).intValue();
         }
         for (i = 1; i <= side; i++) {
            for (j = 1; j <= side; j++)System.out.print(c + " ");
         }
      }
      catch (IOException e) {
      }
   }
}
```

Appendix 3 Roles analysis tables

A3.1 Fixed Value

Name	Type	Assigned	Used	Other
len	int	Once only outside of loop	Not relevant	n/a
original	int	Once only outside of loop	Not relevant	n/a
number	int	Once only outside of loop	Not relevant	n/a
percent	float	Once within loop	Outside of loop Branch condition within assignment loop	Conditional use sets assigning loop condition
factor	float	Once only outside of loop	Not relevant	n/a
years	int	Once within loop	Outside of loop Branch condition within assignment loop	Conditional use sets assigning loop condition
amount	float[ ]	Once only outside of loop	Not relevant	n/a
op	char[ ]	Once only outside of loop	Not relevant	n/a
digit	char[ ]	Once only outside of loop	Not relevant	n/a
letter	char[ ]	Once only outside of loop	Not relevant	n/a
size	int	Once only outside of loop	Not relevant	n/a
value	int[ ]	Initialised outside of loop	Within different loop having same signature	n/a

		Filled within loop		
key	int	Once only outside of loop	Not relevant	n/a
hours	int	Once within loop	Outside of loop	n/a
mins	int	Once within loop	Outside of loop	n/a
secs	int	Once within loop	Outside of loop	n/a
value	int[ ]	Initialised outside of loop Filled within loop	In different loop Elsewhere outside of loop	n/a
c	char	Once only outside of loop	Not relevant	n/a
side	int	Initialised Within loop	As condition for assigning loop Outside of assigning loop	n/a

A3.2 Organizer

Name	**Type**	**Assigned**	**Used**	**Other**
Target	int[]	Initialised Filled in loop In second loop – only with its own values	In other loop, as part of assignment As branch condition for assignment with itself in second loop	n/a

A3.3 Stepper

Name	**Type**	**Assigned**	**Used**	**Other**
swapped	boolean	Within loop - false Within nested loop – true	Branch conditionally within assignment loop	Conditional use leads to “break” statement to end loop
firstPlayer	boolean	Initialised Within loop – toggled	Within loop	n/a
counter	int	Within loop Within other loop – decremented	As condition for initial assigning loop As condition for second assigning loop	n/a
I	int	As part of loop statement	As part of loop statement	n/a
I	int	As part of loop statement	As part of loop statement	n/a
month	int	As part of loop statement As part of loop statement	Within loop statement Within loop statement	n/a
I	int	As part of loop statement	Within loop statement	n/a
Row	int	As part of loop statement	Within loop statement Within assigning loop	n/a
Col	int	As part of loop statement	Within loop statement Within assigning loop	n/a
i	int	As part of	Within loop	n/a

		loop statement As part of loop statement	statement Within assigning loop Within loop statement Within assigning loop	
Count	int	Intialised Within loop – incremented	Within loop	n/a
I	int	As part of loop statement Initialised Within second loop - incremented	Within loop Within second loop	n/a
Up	Boolean	Initialised Within loop – toggled	Within assigning loop	n/a
month	int	As part of loop statement	As part of loop statement	n/a
I	int	As part of loop statement	As part of loop statement	n/a
J	int	As part of loop statement	As part of loop statement	n/a

A3.4 Most Recent Holder

Name	Type	Assigned	Used	Other
Data	int	Initialised Within loop	After initialisation Within assigning loop As condition for assigning loop	n/a
die1	int	Within loop	Within assigning loop	Declared outside of loop
die2	int	Within loop	Within assigning loop	Declared outside of loop
value	float	Initialised Within loop	After initialisation Within assigning loop As condition for assigning loop	n/a
number	int	Within loop	Within assigning loop	n/a
Lt	String	Within loop	Within assigning loop As condition for assigning loop	n/a
C	char	Within loop	Within assigning loop	Declared outside of loop
C	char	Within loop	Within assigning loop As condition for assigning loop	n/a
current	float	Initialised Within loop	After initialisation Within assigning loop	n/a
Value	int	Initialised Within loop	Within assigning loop As condition for assigning loop	n/a

A3.5 Gatherer

Name	**Type**	**Assigned**	**Used**	**Other**
Total1	int	Initialised outside loop Inside loop	Inside assigning loop to change value of itself Outside of loop Branch condition outside loop	Variable could either appear on both sides of assignment statement, or there could be +=, -= etc and other no arithmetic characters after the equals
Total2	int	Initialised outside loop Inside loop	Inside assigning loop to change value of itself Branch condition outside loop	See above
Fib	int	Initialised outside loop Inside loop	Inside assigning loop to change value of itself Inside assigning loop	See above
capital	float	Initialised outside of loop Inside loop	Inside assigning loop to change value of itself Inside assigning loop	See above
Val	int	Initialise outside of loop Inside loop	Inside assigning loop to change value of itself Outside of assigning loop	See above

A3.6 Most wanted holder

Name	Type	Assigned	Used	Other
closest	int	Initialised outside of loop In branch within loop	Conditionally for branch in which assigned Conditionally outside of assignment loop	n/a
Lq	float	Initialised outside of loop In branch within loop	Conditionally for branch in which assigned Outside of assignment loop	n/a
Lr	float	Initialised outside of loop In branch within loop	Conditionally for branch in which assigned Outside of assignment loop	n/a
Max	float	Initialised outside of loop In branch within loop	Conditionally for branch in which assigned Conditionally outside of assignment loop	n/a
largest	int	Initialised outside of loop In branch within loop	Conditionally for branch in which assigned Outside of loop	n/a
largest	int	Initialised outside of loop In branch within loop	Conditionally for branch in which assigned Outside of assignment loop Conditionally outside of assignment loop	n/a
second	int	Initialised outside of loop In branch within loop	Conditionally for one branch in which assigned Outside of assignment loop	n/a

		In branch within loop	Conditionally outside of assignment loop	

A3.7 One way flag

Name	Type	Assigned	Used	Other
inputOk	boolean	Initialised outside of loop Within loop	As condition for assignment loop Branch condition within assignment loop	n/a
Ok	boolean	Initialised outside of loop Within loop	As condition for assignment loop Branch condition once outside of assignment loop	n/a

A3.8 Transformation

Name	Type	Assigned	Used	Other
Q	float	Within loop	Within loop Branch condition within assignment loop	Assigned with combination of other variables, values and operators
R	float	Within loop	Within loop Branch condition within assignment loop	Assigned with combination of other variables, values and operators
interest	float	Within loop	Within assignment loop Within assignment loop	Assigned with combination of other variables, values and operators
average	float	Within loop	Within assignment loop Branch condition within assignment loop	Assigned with combination of other variables, values and operators

A3.9 Follower

Name	Type	Assigned	Used	Other
last_fib	int	Initialised outside of loop Within loop	Within assignment loop	Assignment in loop with value of single variable Used before assigned in loop
previous	int	Initialised outside of loop Within loop	Within assignment loop Within assignment loop Outside of assignment loop	Assignment in loop with value of single variable Used before assigned in loop
preceding	int	Initialised outside of loop Within loop	Within assignment loop Outside of assignment loop	Assignment in loop with value of single variable Used after assigned in loop

A3.10 Temporary

Name	Type	Assigned	Used	Other
Temp	int	Within loop	Within assignment loop	Assignment in loop with value of single variable Used after assigned in loop
temp	int	Within loop	Within assignment loop	Used after assigned in loop

Appendix 4 Rules for roles

A4.1 Logical propositions

The conditions listed **A** – **U** represent logical propositions used to establish whether a variable is playing a given role. **X** signifies that the variable has the role *fixed value*. Text explaining the relevance of these propositions can be found in the main body of the report at section 5.1.

A - variable is assigned in a loop.

B - variable is used in its assignment loop.

C - variable is used conditionally in its assignment loop.

D - variable is used directly for its assigning loop condition.

E - variable is used indirectly for its assigning loop condition.

F - variable is assigned in "for" loop statement.

G - variable is used directly in the program.

H - variable is assigned in a branch for which it is part of the condition.

I - variable appears directly on both sides of assignment statement.

J - variable appears indirectly on both sides of assignment statement.

K - variable is directly toggled within a loop.

L - variable is indirectly toggled within loop.

M - variable is incremented/decremented within a loop.

N - variable is used outside of loop in which it is assigned.

O - variable is assigned in loop before it is used in that loop.

P - variable is used conditionally for a loop outside of its assignment loop.

Q - variable appears in array organizing type statement.

R - variable is of type array.

S - variable is assigned within a loop with a combination of other variables, values and operators.

T - variable is assigned with the output from a method call.

U - variable is assigned with a value resulting from instantiation of a new object or directly with boolean value.

X – variable is playing role *fixed value*.

A4.2 Rules for each role

In the following rules, **Y** signifies an incorrect role declaration. The order in which individual logical propositions appear signifies the order in which they are checked by the program and hence the priority they are afforded (see section 5.2. of the main report for further details). If any of the conditions is true, the remaining conditions are not checked. In the following, "^" is signifies "AND" and "v" signifies "OR". Brackets are included to show where propositions are checked together in the program (and a single offending statement and error message provided as a result), as well as to ensure logical integrity. The tables indicate for each condition under which the variable is not deemed to be playing its perceived role, the reason that is provided by the analysing program via BlueJ. The reader should bear in mind that the role checking rules are based mainly on the conditions under which variable appear in the training programs, and that there will be exceptions to the rules in other Java programs. These will require further modification of the rule sets. Conversely, some of the specific rules listed are probably not be required for 100% reliable role checking of the training programs, e.g. "`I`" for variables having the role *organizer*, but this has not yet been tested.

A4.2.1 Fixed Value

```
Y =   (Q^R) v M v K v I v J v (S^B) v
      F v L v H v (¬T^¬U^B) v (B^¬R) v
      (C^¬(DvE)^¬R) v (¬G^D^¬P)
```

Condition under which variable is adjudged not to be fixed value	**Message generated**
(Q^R)	"appears to be organizer statement"
M	"incremented/decremented within loop"
K	"toggled within loop"
I	"appears on both sides of assignment"
J	"indirectly appears on both sides of assignment"
(S^B)	"assigned in loop with combination of other variables, operators and constants"
F	"assigned in for loop statement"
L	"appears to be indirectly toggled within loop"
H	"used as condition for branch in which assigned"
(¬T^¬U^B)	"always assigned in loop with value of other variable"
(B^¬R)	"used in loop in which assigned"
(C^¬(DvE)^¬R)	"used as condition in loop in which assigned"
(¬G^D^¬P)	"condition for loop in which assigned and limited use outside of loop"

A4.2.2 Organizer

Y = ¬R v ¬Q v I

Condition under which variable is adjudged not to be fixed value	**Message generated**
¬R	`"does not appear to be array or is not used directly as array"`
¬Q	`"no organizer type statements found"`
¬I	`"no statements found where variable on both sides of assignment"`

A4.2.3 Stepper

Y = ¬M ^ ¬F ^ ¬K ^ ¬L

Condition under which variable is adjudged not to be fixed value	**Message generated**
¬M ^ ¬F ^ ¬K ^ ¬L	`"no stepper type assign statements found"`

A4.2.4 Most Recent Holder

$$\begin{aligned} Y = \; & F \vee \neg A \vee (Q \wedge R) \vee \\ & (Q \wedge X) \vee M \vee K \vee I \vee \\ & J \vee S \vee H \vee L \vee (\neg G \wedge D \wedge \neg P) \vee \\ & (\neg T \wedge \neg U) \vee (\neg B \wedge (\neg C \vee D \vee E)) \end{aligned}$$

Condition under which variable is adjudged not to be fixed value	Message generated
F	`"assigned in for loop statement"`
¬A	`"not assigned in loop"`
(Q^R)	`"appears to be organizer statement"`
(Q^X)	`"is array filled in loop"`
M	`"incremented/decremented within loop"`
K	`"toggled within loop"`
I	`"appears on both sides of assignment"`
J	`"indirectly appears on both sides of assignment"`
S	`"assigned in loop with combination of other variables, operators and constants"`
H	`"used as condition for branch in which assigned"`
L	`"appears to be indirectly toggled within loop"`
(¬G^D^¬P)	`"condition for loop in which assigned and limited use outside of loop"`
(¬T^¬U)	`"always assigned in loop with value of other variable"`
(¬B^(¬C v D v E))	`"not used in loop in which assigned"`

A4.2.5 Gatherer

$Y = F \vee \neg A \vee (Q \wedge R) \vee M \vee K \vee L \vee$
$H \vee (S \wedge \neg I \wedge \neg J) \vee (\neg T \wedge \neg U \wedge \neg I \wedge \neg J) \vee$
$(\neg G \wedge D \wedge \neg P \wedge \neg I \wedge \neg J) \vee (\neg I \wedge \neg J)$

Condition under which variable is adjudged not to be fixed value	**Message generated**
F	"assigned in for loop statement"
¬A	"not assigned in loop"
(Q^R)	"appears to be organizer statement"
M	"incremented/decremented within loop"
K	"toggled within loop"
L	"appears to be indirectly toggled within loop"
H	"used as condition for branch in which assigned"
(S^¬I^¬J)	"assigned in loop with combination of other variables, operators and constants"
(¬T^¬U^¬I^¬J)	"always assigned in loop with value of other variable"
(¬G^D^¬P^¬I^¬J)	"condition for loop in which assigned and limited use outside of loop"
(¬I^¬J)	"not found directly, or indirectly on both sides of assignment statement"

A4.2.6 Most wanted holder

$Y = F \vee (Q \wedge R) \vee M \vee K \vee$
$I \vee J \vee L \vee \neg H$

Condition under which variable is adjudged not to be fixed value	**Message generated**
F	"assigned in for loop statement"
(Q^R)	"appears to be organizer statement"
M	"incremented/decremented within loop"
K	"toggled within loop"
I	"appears on both sides of assignment"
J	"indirectly appears on both sides of assignment"
L	"appears to be indirectly toggled within loop"
¬H	"not assigned in branch, or is not condition for branch in which it is assigned"

A4.2.7 One way flag

$$\begin{aligned} Y = \; & F \vee \neg A \vee (Q \wedge R) \vee \\ & (Q \wedge \neg R) \vee M \vee K \vee I \vee \\ & J \vee L \vee H \vee (S \wedge \neg D) \vee \\ & (\neg T \wedge \neg U) \vee G \vee \neg D \vee P \end{aligned}$$

Condition under which variable is adjudged not to be fixed value	**Message generated**
F	"assigned in for loop statement"
¬A	"not assigned in loop"
(Q^R)	"appears to be organizer statement"
(Q^¬R)	"does not appear to be used as one way flag"
M	"incremented/decremented within loop"
K	"toggled within loop"
I	"appears on both sides of assignment"
J	"indirectly appears on both sides of assignment"
L	"appears to be indirectly toggled within loop"
H	"used as condition for branch in which assigned"
(S^¬D)	"assigned in loop with combination of other variables, operators and constants"
(¬T^¬U)	"always assigned in loop with value of other variable"
G	"direct use of variable"
¬D	"not used directly for assign loop condition"
P	"used for loop condition outside of loop in which it is assigned"

A4.2.8 Transformation

$$
\begin{aligned}
\mathbf{Y} = \;& \mathbf{F} \vee \neg\mathbf{A} \vee (\mathbf{Q}\wedge\mathbf{R}) \vee \\
& \mathbf{M} \vee \mathbf{K} \vee \mathbf{I} \vee \mathbf{J} \vee \\
& \mathbf{L} \vee \mathbf{H} \vee (\neg\mathbf{T}\wedge\neg\mathbf{U}\wedge\neg\mathbf{S}) \vee \\
& \neg\mathbf{T} \vee \neg\mathbf{S} \vee (\neg\mathbf{G}\wedge\mathbf{D}\wedge\neg\mathbf{P})
\end{aligned}
$$

Condition under which variable is adjudged not to be fixed value	**Message generated**
F	"assigned in for loop statement"
¬A	"not assigned in loop"
(Q^R)	"appears to be organizer statement"
M	"incremented/decremented within loop"
K	"toggled within loop"
I	"appears on both sides of assignment"
J	"indirectly appears on both sides of assignment"
L	"appears to be indirectly toggled within loop"
H	"used as condition for branch in which assigned"
(¬T^¬U^¬S)	"always assigned in loop with value of other variable"
¬T	"assigned with output from call to method"
¬S	"assignment statement contains no operator characters"
(¬G^D^¬P)	"condition for loop in which assigned and limited use outside of loop"

A4.2.9 Follower

$$
\begin{aligned}
Y = \; & F \vee \neg A \vee (Q \wedge R) \vee \\
& M \vee K \vee I \vee J \vee \\
& L \vee H \vee S \vee T \vee (\neg G \wedge D \wedge \neg P) \vee \\
& U \vee (O \wedge \neg N) \vee \neg B
\end{aligned}
$$

Condition under which variable is adjudged not to be fixed value	**Message generated**
F	"assigned in for loop statement"
¬A	"not assigned in loop"
(Q^R)	"appears to be organizer statement"
M	"incremented/decremented within loop"
K	"toggled within loop"
I	"appears on both sides of assignment"
J	"indirectly appears on both sides of assignment"
L	"appears to be indirectly toggled within loop"
H	"used as condition for branch in which assigned"
S	"assigned in loop with combination of other variables, operators and constants"
T	"assigned with output from call to method"
(¬G^D^¬P)	"condition for loop in which assigned and limited use outside of loop"
U	"assigned directly with value, instantiation of object, or call to method of other object"
(O^¬N)	"assigned before use in loop and not used outside of assign loop"
¬B	"not used in loop in which assigned"

A4.2.10 Temporary

$$
\begin{aligned}
Y = \; & F \vee \neg A \vee (Q \wedge R) \vee \\
& M \vee K \vee I \vee J \vee \\
& L \vee H \vee S \vee T \vee (\neg G \wedge D \wedge \neg P) \vee \\
& U \vee \neg O \vee N \vee \neg B
\end{aligned}
$$

Condition under which variable is adjudged not to be fixed value	**Message generated**
F	"assigned in for loop statement"
¬A	"not assigned in loop"
(Q^R)	"appears to be organizer statement"
M	"incremented/decremented within loop"
K	"toggled within loop"
I	"appears on both sides of assignment"
J	"indirectly appears on both sides of assignment"
L	"appears to be indirectly toggled within loop"
H	"used as condition for branch in which assigned"
S	"assigned in loop with combination of other variables, operators and constants"
T	"assigned with output from call to method"
(¬G^D^¬P)	"condition for loop in which assigned and limited use outside of loop"
U	"assigned directly with value, instantiation of object, or call to method of other object"
¬O	"not assigned in loop before use"
N	"used outside of loop in which it is assigned"
¬B	"not used in loop in which assigned"

Appendix 5 Program Listings

The extension roles_of_variables.jar comprises class from the packages main, main.progAnal, main.rules, and main.BlueJExt. The main.test and main.utils packages were used for development and test. Please note that due to the margins constraints for this document, the layout of the code is not exactly as it appeared in the tools in which it was developed (Eclipse and BlueJ). Every attempt has been made to ensure that the code is still readable and still has correct syntax, but there may be some anomalies for which the author apologises.

A5.1 main

A5.1.1 RolesOfVariables.java

```
package main;

/*
 * Created on 12-Jul-2005
 */

/**
 * @author cbishop
 */
import bluej.extensions.*;
import bluej.extensions.editor.Editor;
import bluej.extensions.event.*;
import bluej.extensions.editor.TextLocation;

import java.io.File;
import java.net.URL;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Set;
import java.util.LinkedHashMap;

import main.blueJExt.PreferenceSetter;
import main.progAnal.ProgramAnalyser;

public class RolesOfVariables extends Extension {

   private PreferenceSetter rolePreferences;
   private BlueJ bluej;

   /**
    * Starts extension
    */
   public void startup(BlueJ blueJ) {
      bluej = blueJ;
      bluej.addCompileListener(new CompileEventListener());
      rolePreferences = new PreferenceSetter(bluej);
      bluej.setPreferenceGenerator(rolePreferences);
   }

   /**
    * @return version number of extension
    */
   public String getVersion() {
      return ("Version 1.0. 2005.09");
   }

   /**
    * @return name of extension
    */
   public String getName() {
      return ("Roles of Variables");
```

```
}



/**
 * Outputs terminate message to debug.txt
 */
public void terminate() {
   System.out.println("Roles of variables terminates");
}

/**
 * @return description of extension
 */
public String getDescription() {
   return ("Extension to analyse roles of variables" +
      "\nand notify if roles annotated correctly or +
      "otherwise.");
}

/**
 * @return true - assume for now that extension is compatible with
 *  current
 *        BlueJ version.
 */
public boolean isCompatible() {
   return true;
}

/**
 * @return URL of Roles of Variables website – in the absence of any
 * URL associated with this extension
 */
public URL getURL() {
   try {
      return new URL("http://www.cs.joensuu.fi/~saja/var_roles/");
   } catch (Exception e) {
      // The link is either dead or otherwise unreachable
      System.out.println("Roles of Variables: getURL: Exception="
            + e.getMessage());
      return null;
   }
}

/*
 * Display results of role checking
 */
private void displayResults(HashMap roleResults) {
   BPackage currentPackage = bluej.getCurrentPackage();
   Set fileNames = roleResults.keySet();
   Iterator it = fileNames.iterator();
   HashMap roleMap = (HashMap) roleResults.get("role map");
   try {
      while (it.hasNext()) {
         String fileName = (String) it.next();
         if (!fileName.equals("role map")) {
            String className = fileName.substring(0,
                  fileName.length() - 5);
            BClass currentClass =
               currentPackage.getBClass(className);
            Editor editor = currentClass.getEditor();
            ArrayList fileResults = (ArrayList) roleResults
                  .get(fileName);
            HashMap checkedResults = (HashMap) fileResults.get(0);
            LinkedHashMap roles = (LinkedHashMap)
               fileResults.get(1);
            Set variables = checkedResults.keySet();
            Iterator iter = variables.iterator();
            while (iter.hasNext()) {
               String variable = (String) iter.next();
               Integer roleInt = (Integer) roles.get(variable);
               String role = (String) roleMap.get(roleInt);
               ArrayList checkedRole = (ArrayList) checkedResults
                      .get(variable);
               Boolean isOK = (Boolean) checkedRole.get(0);
               if (!isOK.booleanValue()) {
                  highlightText((String) checkedRole.get(1),
                     editor);
                  String errorString = "";
                  if (role != null) {
                     errorString = "Possible incorrect role" +
                        " annotation of '" + role +
                        "' for variable '" + variable + "'";
                  } else {
                     errorString = "Possible incorrect role " +
                        "annotation " + "for variable '" +
                        variable + "'";
                  }
                  if (rolePreferences.giveReason()) {
                     errorString += " - " + checkedRole.get(2)
                           + "\n";
                  } else
                     errorString += "\n";
                  if (rolePreferences.suggestRole()) {
                     errorString += checkedRole.get(3);
                  }
                  editor.showMessage(errorString);
                  editor.setVisible(true);
                  System.out.println("message shown: " +
                     errorString);
                  return; //if role annotation problem return
               }
            }
            editor.showMessage("Class compiled - no syntax " +
               "errors.\n" + "Role annotations OK.");
         }
      }
   } catch (PackageNotFoundException e) {
      System.out.println(e.getMessage());
   } catch (ProjectNotOpenException e) {
      System.out.println(e.getMessage());
   }
}

/*
 * Highlight offending statement returned from role checking
 */
private void highlightText(String offendingStatement, Editor editor)
{
   int lines = editor.getLineCount();
   int textLength = editor.getTextLength();
   ProgramAnalyser progAnal = new ProgramAnalyser();
   for (int i = 0; i < lines - 1; i++) {
      String lineToCheck = editor.getText(new TextLocation(i, 0),
            new TextLocation(i + 1, 0));
      if (progAnal.contains(lineToCheck, offendingStatement)) {
         editor.setSelection(new TextLocation(i, 0),
            new TextLocation(i + 1, 0));
         System.out.println("text highlighted");
      }
   }
}

/*
 * Inner class for compile listener @author cbishop
 *
 */
class CompileEventListener implements CompileListener {
   public void compileStarted(CompileEvent e) {
   }

   public void compileError(CompileEvent e) {
   }

   public void compileWarning(CompileEvent e) {
   }

   /*
    * Start new thread for role checking when source code has been
    * compiled successfully
    */
   public void compileSucceeded(CompileEvent e) {
       System.out.println("compile succeeded");
       if (rolePreferences.checkRoles()) {
           File[] files = e.getFiles();
           RoleThread roleThread = new RoleThread(files);
           roleThread.start();
       }
   }

   public void compileFailed(CompileEvent e) {
   }
}

/*
* Inner class for role checking thread @author cbishop
*/
class RoleThread extends Thread {

   File[] files;

   RoleThread(File[] files) {
       this.files = files;
   }

   /**
    * Start new thread for role checking
    */
   public void run() {
       System.out.println("thread started");
       RoleAnalyser roleAnalyser = new RoleAnalyser(files);
       HashMap roleResults = roleAnalyser.checkRoles();
       displayResults(roleResults);
   }
}
}
```

A5.1.2 RoleAnalyser.java

```
/*
 * Created on 10-Jun-2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package main;

import java.io.File;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Set;

import main.progAnal.MethodGetter;
import main.progAnal.StatementAnalyser;
import main.progAnal.RoleHolder;
import main.progAnal.SourceSorter;
import main.progAnal.StatementGetter;
import main.progAnal.ProgramSlicer;
import main.rules.RuleApplyer;

/**
 * @author cbishop
 *
 * Main class to run role analyser program
 */
public class RoleAnalyser {

   private SourceSorter sourceSorter;
   private LinkedHashMap brokenSource;
   private StatementGetter statementGetter;
   private static final String[] PROGRAMS = { "BubbleSort.java",
      "Closest.java", "DiceGame.java", "DivMod7.java", "Doubles.java",
      "Fibonachi.java", "Growth.java", "Histogram.java", "Lexical.java",
      "Multiplication.java", "Number.java", "Occur.java",
      "ProgramTime.java", "Saw.java", "SmoothedAverage.java",
      "Square.java", "TwoLargest.java" };
   private static final String[] ROLES = { "Fixed Value", "Organizer",
      "Stepper", "Most Recent Holder", "Gatherer",
      "Most Wanted Holder", "One Way Flag", "Transformation",
      "Follower", "Temporary" };
   private HashMap roleMap;
   private File[] files;

   public RoleAnalyser(File[] inputFiles)
   {
      files = inputFiles;
   }

   /**
    * Check roles played by variables
    *
    * @return HashMap containing results of role checking
    */
   public HashMap checkRoles() {
      HashMap returnMap = new HashMap();
      for (int i = 0; i < files.length; i++) {
         ArrayList fileResults = new ArrayList();
         String fileName = files[i].toString();
         roleMap = initialiseMap();
         brokenSource = getSortedSource(fileName);
         //printer.printSource(brokenSource);//debug
         Set variables = getVariables();
        LinkedHashMap variableStatements = getStatementMap(brokenSource,
               variables);
         RoleHolder roleHolder = sourceSorter.getRoleHolder();
         LinkedHashMap roles = roleHolder.getRoles();
```

```
            HashMap sortedStatements = sortStatements(variables,
                  variableStatements);
            HashMap analysedStatements =
               getStatementAnalysis(sortedStatements, variables);
            //printer.printAnalysedStatements(analysedStatements); //debug
            ArrayList methods = getMethods(sortedStatements);
            HashMap checkedRoles = getResults(analysedStatements, roles,
               methods);
            //add results of analysis for given file
            fileResults.add(checkedRoles);
            //add annotated roles for each variable in given file
            fileResults.add(roles);
            //add fileResults to return map for given file
            returnMap.put(files[i].getName(), fileResults);
         }
         returnMap.put("role map", roleMap);//add role map to return map
         return returnMap;
      }

      /*
       * Return hierarchy of source code statements
       */
      private LinkedHashMap getSortedSource(String fileName) {
         sourceSorter = new SourceSorter();
         LinkedHashMap returnMap = sourceSorter.sortSource(fileName);
         return returnMap;
      }

      /*
       * Return Set of variables
       */
      private Set getVariables() {
         RoleHolder roles = sourceSorter.getRoleHolder();
         Set returnSet = roles.getVariables();
         return returnSet;
      }

      /*
       * Return map of variable statements
       */
      private LinkedHashMap getStatementMap(LinkedHashMap sourceMap,
            Set variables) {
         StatementGetter statementGetter = new StatementGetter();
         LinkedHashMap returnMap =
            statementGetter.getStatements(brokenSource, variables);
         return returnMap;
      }

      /*
       * Return map of program slices
       */
      private HashMap sortStatements(Set variables,
            LinkedHashMap relevantStatements) {
         ProgramSlicer programSlicer = new ProgramSlicer();
         HashMap returnMap = programSlicer.sortStatements(variables,
               relevantStatements);
         return returnMap;
      }

      /*
       * Return map of analysed statements
       */
      private HashMap getStatementAnalysis(HashMap statements,
            Set variables) {
         HashMap analysedStatements = new HashMap();
         StatementAnalyser statementAnalyser =
            new StatementAnalyser(statements, variables);
         analysedStatements = statementAnalyser.getStatementAnalysis();
         return analysedStatements;
      }
```

```
    /*
     * Return map of results from role checking
     */
    private HashMap getResults(HashMap analysedStatements,
          LinkedHashMap roles, ArrayList methods) {
       HashMap checkedRoles = new HashMap();
       RuleApplyer ruleApplyer = new RuleApplyer();
       checkedRoles = ruleApplyer.applyRules(analysedStatements, roles,
             methods);
       return checkedRoles;
    }

    /*
     * Initialised map with roles and associated int values
     */
    private HashMap initialiseMap() {
       HashMap returnMap = new HashMap();
       for (int i = 0; i < 10; i++) {
          returnMap.put(new Integer(i + 1), ROLES[i]);
       }
       return returnMap;
    }

    /*
     * Return list of methods in analysed program
     */
    private ArrayList getMethods(HashMap sortedStatements) {
       ArrayList returnArray = new ArrayList();
       MethodGetter methodGetter = new MethodGetter();
       returnArray = methodGetter.getMethods(sortedStatements);
       return returnArray;
    }
}
```

A5.2 main.progAnal

A5.2.1 SourceSorter.java

```
package main.progAnal;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.LinkedHashMap;
import java.util.Stack;

/**
 * Class to take java source code file as input and output ArrayList
 * containing hierarchical breakdown of code into methods, branches,
 * sub-branches, etc
 *
 * @author Craig Bishop
 * @version 090607
 */
public class SourceSorter {
   private LinkedHashMap currentMap;
   private Stack hashMaps;
   private String inputLine;
   private int bracketCount;
   private boolean openQuote;
   private RoleHolder roles;

   /**
    * Constructor for objects of class SourceBreakDown
    */
   public SourceSorter() {
   }

   /**
    * Method to break down input file into ArrayList of Strings with
    * separate entry in list for each level in the code hierarchy.
    *
    * @param fielname
    *           String giving name of file to be checked
    * @return LinkedHashMap containing a breakdown of the input source
    * code
    */
   public LinkedHashMap sortSource(String fileName) {
      roles = new RoleHolder();
      currentMap = new LinkedHashMap();
      hashMaps = new Stack();
      inputLine = "";
      String tempLine;
      boolean openQuote = false;
      try {
         FileInputStream fileIn = new FileInputStream(fileName);
         BufferedReader input =
            new BufferedReader(new InputStreamReader(
               fileIn));
         do {
            tempLine = input.readLine();
            if (tempLine != null) {
               String trimLine = tempLine.trim();
               processLine(trimLine);
            }
         } while (tempLine != null);
      } catch (FileNotFoundException e) {
         System.out.println(e.getMessage());
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
```

```java
      return currentMap;
   }

   /**
    * Method to process each line for addition to LinkedHashMap
    *
    * @param lineToProcess
    *            the line to be processed.
    */
   private void processLine(String lineToProcess) {
      for (int i = 0; i < lineToProcess.length(); i++) {
         inputLine += lineToProcess.substring(i, i + 1); //add first
                                                // character of
                                                // string to
                                                // inputLine
         if (lineToProcess.charAt(i) == 34) {
            openQuote = !openQuote;
         }
         //check that string to be processed is not within inverted
         //commas.
         if (!openQuote) {
         //check if previous string is comment
            if (lineToProcess.substring(i).startsWith("*/")) {
               inputLine = ""; //remove comment string prior to "*/"
               // increment i so that "/" is not added to input line
               // next time round
               i++;
               //check if subsequent string is comment
               } else if (comment(lineToProcess.substring(i))) {
               String commentLine = lineToProcess.substring(i);
               if (roleDeclaration(commentLine)) {
                  roles.noteRole(commentLine);
               }
               if (commentLine.startsWith("//")) {
                  inputLine = inputLine.substring(0,
                  // remove first character from inputLine
                  inputLine.length() - 1);
                  i = lineToProcess.length();
               }
            //check if character within brackets
            } else if (lineToProcess.substring(i, i + 1).equals("(")) {
               bracketCount++;
            } else if (lineToProcess.substring(i, i + 1).equals(")")) {
               bracketCount--;
            //only treat ";" as end line if not within brackets
            } else if (lineToProcess.substring(i, i + 1).equals(";")
                  && bracketCount == 0) {
               inputLine = inputLine.trim();
               currentMap.put(inputLine, inputLine);
               inputLine = "";
            } else if (lineToProcess.substring(i, i + 1).equals("{")
                  && !lineToProcess.endsWith("};")) {
               LinkedHashMap tempMap = new LinkedHashMap();
               inputLine = inputLine.trim();
               currentMap.put(inputLine, tempMap);
               hashMaps.push(currentMap);
               currentMap = (LinkedHashMap) currentMap.get(inputLine);
               inputLine = "";
            } else if (lineToProcess.substring(i, i + 1).equals("}")
                  && !lineToProcess.endsWith("};")) {
               inputLine = inputLine.trim();
               currentMap.put(inputLine, inputLine);
               currentMap = (LinkedHashMap) hashMaps.pop();
               inputLine = "";
            }
         }
      }
   }
```

```
    /*
     * Return whether line is comment
     */
    private boolean comment(String lineString) {
       boolean isComment = false;
       if (lineString.startsWith("/*") || lineString.startsWith("//")) {
          isComment = true;
       }
       return isComment;
    }

    /*
     * Return whether role is declared for variable in comment line
     */
    private boolean roleDeclaration(String commentLine) {
       boolean roleDeclared = false;
       for (int i = 0; i < commentLine.length() - 1; i++) {
          if (commentLine.substring(i, i + 2).equals("%%")) {
             roleDeclared = true;
          }
       }
       return roleDeclared;
    }

    public RoleHolder getRoleHolder() {
       return roles;
    }
}
```

A5.2.2 RoleHolder.java

```
/*
 * Created on 10-Jun-2005
 *
 */
package main.progAnal;

import java.util.LinkedHashMap;
import java.util.Iterator;
import java.util.Set;

/**
 * @author cbishop Class to note/hold variables and their roles
 */
public class RoleHolder {

   private LinkedHashMap varRoles;

   /**
    * Constructor for RoleHolder
    */
   public RoleHolder() {
      varRoles = new LinkedHashMap();
   }

   /**
    * Note role declared for given variable in comment string
    *
    * @param roleString
    *            String containing role declaration
    */
   public void noteRole(String roleString) {
      String[] stringArray = new String[3];
      stringArray = roleString.split("%%");
      if (stringArray.length >= 3) {
         Integer roleValue = new Integer(getValue(stringArray[2]));
         //if (roleValue.intValue() > 0) {
         varRoles.put(stringArray[1], roleValue);
         //}
      }
   }

   /*
    * Return int value for declared role
    */
   private int getValue(String role) {
      if (role.equalsIgnoreCase("Fixed value")) {
         return 1;
      } else if (role.equalsIgnoreCase("Organizer")) {
         return 2;
      } else if (role.equalsIgnoreCase("Stepper")) {
         return 3;
      } else if (role.equalsIgnoreCase("Most recent holder")) {
         return 4;
      } else if (role.equalsIgnoreCase("Gatherer")) {
         return 5;
      } else if (role.equalsIgnoreCase("Most wanted holder")) {
         return 6;
      } else if (role.equalsIgnoreCase("One way flag")) {
         return 7;
      } else if (role.equalsIgnoreCase("Transformation")) {
         return 8;
      } else if (role.equalsIgnoreCase("Follower")) {
         return 9;
      } else if (role.equalsIgnoreCase("Temporary")) {
         return 10;
      } else {
         return -1;
      }
    }

    /**
     * Print variables and their roles
     *
     * @return String giving variables and their roles
     */
    public String toString() {
       Set roleSet = (Set) varRoles.keySet();
       Iterator roleIt = roleSet.iterator();
       String roleString = "";
       while (roleIt.hasNext()) {
          String var = (String) roleIt.next();
          roleString += "Variable: " + var + ", Role: " +
             varRoles.get(var) + "\n";
       }
       return roleString;
    }

    /**
     * Return set of variables
     *
     * @return Set Variables
     */
    public Set getVariables() {
       Set variables = varRoles.keySet();
       return variables;
    }

    /**
     * Return map of variables and their roles
     *
     * @return LinkedHashMap Variables as keys and their roles as values
     */
    public LinkedHashMap getRoles() {
       return varRoles;
    }
}
```

A5.2.3 ProgramAnalyser.java

```
/*
 * Created on 01-Jul-2005
 */
package main.progAnal;

import java.util.ArrayList;

/**
 * Class containing common methods used by other program analyser classes
 *
 * @author cbishop
 */
public class ProgramAnalyser {
   private static final char[] FOLLOWING_CHARS = { 32, 37, 38, 41, 42,
      43, 44, 45, 46, 47, 59, 60, 61, 62, 91, 93, 94, 124 };
   private static final char[] PREVIOUS_CHARS = { 32, 33, 37, 38, 40,
      41, 42, 43, 45, 47, 60, 61, 62, 91, 93, 94, 124 };
   private static final char[] ARITHMETIC_CHARS = { 32, 37, 40, 41, 42,
      43, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 59 };
   private static final char[] OPERATORS = { 33, 37, 38, 42, 43, 45, 47,
      94, 124 };
   private static final char[] OTHER_CHARS = { 34, 46, 40, 41 };

   /**
    * Return whether input variable name found in input statement refers
    * to the variable in question, or is simply a substring of a longer
    * identifier
    *
    * @param inputStatement
    *            String containing statement
    * @param potentialSubstring
    *            String variable
    * @return boolean true is variable is substring
    */
   public boolean subString(String inputStatement,
         String potentialSubstring) {
      boolean isSubstring = true;
      for (int i = 0; i < inputStatement.length(); i++) {
         String tempString = inputStatement.substring(i);
         if (tempString.startsWith(potentialSubstring)) {
            char followingChar = tempString.charAt(potentialSubstring
                  .length());
            if (i == 0 && isPermitted(followingChar, true)) {
               isSubstring = false;
               break;
            } else if (i > 0) {
               char previousChar = inputStatement.charAt(i - 1);
               if (isPermitted(previousChar, false)
                     && isPermitted(followingChar, true)) {
                  isSubstring = false;
                  break;
               }
            }
         }
      }
      return isSubstring;
   }

   /*
    * Return whether characters prior to or following the variable name
    * in the statement are permitted, or not. If not permitted, then
    * variable name is part of substring.
    */
   private boolean isPermitted(char charToCheck, boolean following) {
      boolean isPermitted = false;
      if (following) {
         for (int i = 0; i < FOLLOWING_CHARS.length; i++) {
            char checker = FOLLOWING_CHARS[i];
            if (charToCheck == FOLLOWING_CHARS[i]) {
               isPermitted = true;
               break;
            }
         }
      } else {
         for (int i = 0; i < PREVIOUS_CHARS.length; i++) {
            char checker = PREVIOUS_CHARS[i];
            if (charToCheck == PREVIOUS_CHARS[i]) {
               isPermitted = true;
               break;
            }
         }
      }
      return isPermitted;
   }

   /**
    * Remove white space from input String
    *
    * @param inputString
    * @return String with white space removed
    */
   public String removeSpaces(String inputString) {
      String outputString = "";
      for (int i = 0; i < inputString.length(); i++) {
         char tempChar = inputString.charAt(i);
         if (tempChar != 32) { //look for white space
            outputString += inputString.substring(i, i + 1);
         }
      }
      return outputString;
   }

   /**
    * Return whether input statement contain the variable
    *
    * @param statement
    *            String being statement to analyse
    * @param var
    *            String being name of variable
    * @return boolean true if statement contains the variable
    */
   public boolean contains(String statement, String var) {
      boolean contains = false;
      for (int i = 0; i < statement.length(); i++) {
         String statementSubstring = statement.substring(i);
         if (statementSubstring.startsWith(var)) {
            contains = true;
            break;
         }
      }
      return contains;
   }

   /**
    * Return whether variable in statement is enclosed in brackets
    *
    * @param statement
    *            String being statement to analyse
    * @param var
    *            String being name of variable
    * @return boolean true if variable is enclosed in brackets
    */
   public boolean inBrackets(String statement, String var) {
      boolean bracketed = false;
      int bracketCount = 0;
      for (int i = 0; i < statement.length(); i++) {
         String statementSubstring = statement.substring(i);
         if (statement.charAt(i) == 40)
```

```
                bracketCount++;
            else if (statement.charAt(i) == 41)
                bracketCount--;
            if (statementSubstring.startsWith(var) && bracketCount > 0) {
                bracketed = true;
                break;
            } else if (statementSubstring.startsWith(var) &&
                bracketCount == 0) {
                bracketed = false;
            }
        }
        return bracketed;
    }

    /**
     * Return whether statement is a loop statement or not
     *
     * @param statement
     *            String being statement to analyse
     * @return boolean true if statement is loop statement
     */
    public boolean loop(String statement) {
        boolean isLoop = false;
        if (statement.startsWith("while") || statement.startsWith("for")
                || statement.startsWith("do")) {
            isLoop = true;
        }
        return isLoop;
    }

    /**
     * Return whether statement is "else" statement
     *
     * @param stringToCheck
     *            String being statement to analyse
     * @return boolean true if statement is "else"
     */
    public boolean isElse(String stringToCheck) {
        boolean isElse = false;
        if (stringToCheck.startsWith("else ")
                || stringToCheck.startsWith("else{")
                || stringToCheck.startsWith("else(")
                && !stringToCheck.startsWith("else if")) {
            isElse = true;
        }
        return isElse;
    }

    /**
     * Return whether statement is "if" statement
     *
     * @param stringToCheck
     *            String being statement to analyse
     * @return boolean true if statement is "if"
     */
    public boolean isIf(String stringToCheck) {
        boolean isIf = false;
        if (stringToCheck.startsWith("if ")
                || stringToCheck.startsWith("if(")
                || stringToCheck.startsWith("else if ")
                || stringToCheck.startsWith("else if(")) {
            isIf = true;
        }
        return isIf;
    }

    /**
     * Return whether statement is branch statement
     *
     * @param statementToCheck
     *            String being statement to analyse
     * @return boolean true if statement is branch statement
     */
    public boolean branch(String statementToCheck) {
       boolean isBranch = false;
       if (isIf(statementToCheck) || isElse(statementToCheck)) {
          isBranch = true;
       }
       return isBranch;
    }

    /**
     * Return whether statement is control construct
     *
     * @param stringToCheck
     *            String being statement to analyse
     * @return boolean true if statement is "if"
     */
    public boolean control(String stringToCheck) {
       boolean isControl = false;
       if (loop(stringToCheck) || branch(stringToCheck)) {
          isControl = true;
       }
       return isControl;
    }

    /**
     * Return whether statement is usage statement where value of
     * variable is output to terminal for use by program user
     *
     * @param inputStatement
     *            String being statement to analyse
     * @param variable
     *            String being variable in question
     * @return boolean true if statement is print usage statement
     */
    public boolean printUse(String inputStatement, String variable) {
       boolean isPrintUse = false;
       if (inputStatement.startsWith("System.out.print")
             && inBrackets(inputStatement, variable)
             && contains(inputStatement, variable)
             && !subString(inputStatement, variable)) {
          isPrintUse = true;
       }
       return isPrintUse;
    }

    /**
     * Return whether statement is throw statement
     *
     * @param statementArray
     *            String being statement to analyse
     * @return boolean true is statement is throw statement
     */
    public boolean throwStatement(ArrayList statementArray) {
       boolean isThrow = false;
       for (int i = 0; i < statementArray.size(); i++) {
          String tempStatement = (String) statementArray.get(i);
          if (tempStatement.startsWith("throw")) {
             isThrow = true;
          }
       }
       return isThrow;
    }

    /**
     * Return whether statement is try or catch statement
     *
     * @param statement
     *            String being statement to analyse
```

```
 * @return boolean true is statement is try/catch
 */
public boolean tryCatchStatement(String statement) {
   boolean isTryCatch = false;
   if (statement.startsWith("try") || statement.startsWith("catch"))
   {
      isTryCatch = true;
   }
   return isTryCatch;
}

/**
 * Return whether statement is method signature
 *
 * @param statement
 *          String being statement to analyse
 * @return boolean true is statement is method signature
 */
public boolean methodStatement(String statement) {
   boolean methodStatement = false;
   if ((statement.startsWith("private")
      || statement.startsWith("public"))
      && contains(statement, "(") && !contains(statement, " class ")
      && statement.endsWith("{")) {
      methodStatement = true;
   }
   return methodStatement;

}

/**
 * Return whether statement is class declaration
 *
 * @param statement
 *          String being statement to analyse
 * @return boolean true is statement is class declaration
 */
public boolean isClass(String statement) {
   boolean classDeclaration = false;
   for (int i = 0; i < statement.length(); i++) {
      if (statement.substring(i).startsWith("class")
            && !subString(statement, "class")) {
         classDeclaration = true;
      }
   }
   return classDeclaration;
}

/**
 * Return true if input statement is already in ArrayList of
 * statements
 *
 * @param statement
 *          String being statment to analyse
 * @param inputStatements
 *          ArrayList of statements
 * @return boolean true if statement is in ArrayList
 */
public boolean isInArray(String statement, ArrayList inputStatements)
{
   boolean isInArray = false;
   for (int i = 0; i < inputStatements.size(); i++) {
      String arrayStatement = (String) inputStatements.get(i);
      if (statement.equals(arrayStatement)) {
         isInArray = true;
         break;
      }
   }
   return isInArray;
}
```

```java
/**
 * Return true if input statement is an arithmetic expression
 * @param statementToCheck
 *            String being statement to analyse
 * @return boolean true if statement is arithmetic expression
 */
public boolean arithExp(String statementToCheck) {
   boolean arithExp = true;
   for (int i = 0; i < statementToCheck.length(); i++) {
      boolean charCheck = false;
      char charToCheck = statementToCheck.charAt(i);
      for (int j = 0; j < ARITHMETIC_CHARS.length; j++) {
         if (charToCheck == ARITHMETIC_CHARS[j]) {
            charCheck = true; //check if particular character is
                              // arithmetic char
            j = ARITHMETIC_CHARS.length;
         }
      }
      //if any character is not arithmetic char, set arithExp false
      if (!charCheck) {
         arithExp = false;
         break;
      }
   }
   return arithExp;
}

/**
 * Return true if input statement containing variable is an
 * arithmetic expression
 *
 * @param statementToCheck
 *            String being statement to analyse
 * @param var
 *            String being variable
 * @return boolean true if statement is arithmetic expression
 */
public boolean arithExp(String statementToCheck, String var) {
   boolean arithExp = true;
   for (int i = 0; i < statementToCheck.length(); i++) {
      String statementSubstring = statementToCheck.substring(i);
      if (statementSubstring.startsWith(var + "[")) {
         statementToCheck =
            statementSubstring.substring(var.length());
         for (int j = 0; j < statementToCheck.length(); j++) {
            if (statementToCheck.substring(j).equals("]")) {
               statementToCheck = statementToCheck.substring(j);
            }
         }
         i = 0;
      } else if (statementSubstring.startsWith(var)) {
         statementToCheck =
            statementSubstring.substring(var.length());
         i = 0;
      }
      boolean charCheck = false;
      char charToCheck = statementToCheck.charAt(i);
      for (int j = 0; j < ARITHMETIC_CHARS.length; j++) {
         if (charToCheck == ARITHMETIC_CHARS[j]) {
            charCheck = true; //check if particular character is
                              // arithmetic char
            j = ARITHMETIC_CHARS.length;
         }
      }
      //if any character is not arithmetic char, set arithExp false
      if (!charCheck) {
         arithExp = false;
         break;
      }
```

```
        }
        return arithExp;
    }

    /**
     * Return part of input statement that occurs after the "=" sign
     *
     * @param inputStatement
     *            String being statement to analyse
     * @return String representing part of statement after "="
     */
    public String afterEquals(String inputStatement) {
        String returnString = "";
        int bracketCount = 0;
        for (int k = 0; k < inputStatement.length(); k++) {
            String statementSubstring = inputStatement.substring(k);
            if (statementSubstring.startsWith("=") && bracketCount == 0) {
                returnString = statementSubstring;
                break;
            }
            if (statementSubstring.startsWith("(")) {
                bracketCount++;
            }
            if (statementSubstring.startsWith(")")) {
                bracketCount--;
            }
        }
        return returnString;
    }

    /**
     * Return whether input statement is operator or other statement
     *
     * @param statement
     *            String being statement to analyse
     * @param what
     *            String stating what sort of characters to look for
     * @return boolean true if statement contains any of the characters
     */
    public boolean is(String statement, String what) {
        boolean is = false;
        for (int i = 0; i < statement.length(); i++) {
            char charToCheck = statement.charAt(i);
            boolean contains = false;
            if (what.equals("operator")) {
                for (int j = 0; j < OPERATORS.length; j++) {
                //see if character from statement is an operator character
                    if (charToCheck == OPERATORS[j]) {
                        contains = true;
                        j = OPERATORS.length;
                    }
                }
            } else if (what.equals("other")) {
                for (int j = 0; j < OTHER_CHARS.length; j++) {
                    //see if character from statement is an other character
                    if (charToCheck == OTHER_CHARS[j]) {
                        contains = true;
                        j = OTHER_CHARS.length;
                    }
                }
            }
            //if one character in statement is operator/other character
            if (contains) {
                is = true;
                break;
            }
        }
        return is;
    }
}
```

A5.2.4 StatementGetter.java

```
/*
 * Created on 13-Jun-2005
 *
 */
package main.progAnal;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Set;
import java.util.Stack;

/**
 * @author cbishop
 *
 */
public class StatementGetter extends ProgramAnalyser {

   private String inputKey;
   private LinkedHashMap inputMap;
   private Set keySet;
   private Iterator iter;
   private Stack stack;
   private Set variables;

   /**
    * Constructor for StatementGetter
    *
    */
   public StatementGetter() {
      super();
   }

   /**
    * Return map of control and variable containing statements
    *
    * @param inputCode
    *            LinkedHashMap being hierarchical store of source code
    * @param inputVars
    *            Set of variable names
    * @return LinkedHashMap of variable statements
    */
   public LinkedHashMap getStatements(LinkedHashMap inputCode,
         Set inputVars) {
      inputMap = inputCode;
      variables = inputVars;
      keySet = inputCode.keySet();
      stack = new Stack();
      LinkedHashMap variableStatements = getStatementMap(inputMap);
      return variableStatements;
   }

   /*
    * Iterate through set of declared variables and return map
    * containing relevant statements for each one
    */
   private LinkedHashMap getStatementMap(LinkedHashMap mapToAnalyse) {
      LinkedHashMap returnMap = new LinkedHashMap();
      Iterator it = variables.iterator();
      //for each variable, search code for variable statements
      while (it.hasNext()) {
         String variable = (String) it.next();
         ArrayList tempArray = fillArray(variable, mapToAnalyse);
         returnMap.put(variable, tempArray);
      }
      return returnMap;

   }
```

```
/*
 * Return ArrayList of relevant statements for a given variable
 */
private ArrayList fillArray(String variableName,
      LinkedHashMap mapToAnalyse) {
   ArrayList returnArray = new ArrayList();
   keySet = mapToAnalyse.keySet();
   iter = keySet.iterator();
   stack.clear();
   ArrayList tempArray = new ArrayList();
   int lineCounter = 0;
   int latestPosition = 0;
   boolean openQuote = false; //flag for if string is in ""s.
   while (iter.hasNext()) {
      boolean variablePresent = false;
      String tempStatement = (String) iter.next();
      lineCounter++;
      for (int i = 0; i < tempStatement.length(); i++) {
         String tempString = tempStatement.substring(i);
         if (tempStatement.charAt(i) == 34) {
            openQuote = !openQuote;
         }
         if (tempString.startsWith(variableName)
               && !subString(tempStatement, variableName)
               && lineCounter > latestPosition && !openQuote) {
            variablePresent = true;
            mapToAnalyse = inputMap; //start search from top of map
                                 // again
            latestPosition = lineCounter;
            lineCounter = 0;
            keySet = mapToAnalyse.keySet();
            iter = keySet.iterator();
            stack.clear();
            break;
         }
      }
      if (variablePresent) {
         tempArray.add(tempStatement);
         returnArray.add(tempArray);
         tempArray = new ArrayList();
      } else if (tempStatement.endsWith("{")) {
         tempArray.add(tempStatement);
         mapToAnalyse = goDownLevel(mapToAnalyse, tempStatement);
      } else if (tempStatement.endsWith("}")) {
         tempArray.add(tempStatement);
         mapToAnalyse = goUpLevel();
      //add all while if statements without following '{' to cater
      // for "do" loops
      } else if ((tempStatement.startsWith("while ")
            || tempStatement.startsWith("while("))
            && lineCounter > latestPosition) {
         //in case condition depends on variale set by relevant
         // variable during loop
         tempArray.add(tempStatement);
      }
   }
   //add final array, even if not variable statement present, in case
   //relevant while statement follows last use of variable
   returnArray.add(tempArray);
   return returnArray;
}

/*
 * Go down one level in source code hierarchy
 */
private LinkedHashMap goDownLevel(LinkedHashMap mapToTraverse,
      String mapKey) {
   stack.push(mapToTraverse); //put current input map onto stack
   stack.push(iter); //put current iterator onto stack
```

```
      mapToTraverse = (LinkedHashMap) mapToTraverse.get(mapKey);
      keySet = mapToTraverse.keySet();
      iter = keySet.iterator();
      return mapToTraverse;
   }

   /*
    * Go up one level in source code hiearchy
    */
   private LinkedHashMap goUpLevel() {
      LinkedHashMap returnMap;
      //pop higher level iterator of stack
      iter = (Iterator) stack.pop();
      //pop higher level input map off stack
      returnMap = (LinkedHashMap) stack.pop();
      return returnMap;
   }
}
```

A5.2.5 ProgramSlicer.java

```
/*
 * Created on 29-Jun-2005
 */
package main.progAnal;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Set;

/**
 * Class to produce program slice for each variable
 *
 * @author cbishop
 */
public class ProgramSlicer {

   HashMap statementMap;

   Iterator iter;

   /**
    * Return map containing program slices for each variable
    *
    * @param variables
    *            Set of variable name
    * @param relevantStatements
    *            LinkedHashMap containing statement for program slice
    * @return HashMap of program slices
    */
   public HashMap sortStatements(Set variables,
         LinkedHashMap relevantStatements) {
      statementMap = new HashMap();
      iter = variables.iterator();
      while (iter.hasNext()) {
         String keyString = (String) iter.next();
         ArrayList parentArray = (ArrayList) relevantStatements
               .get(keyString);
         ArrayList variableArray = getVariableStatements(parentArray);
         statementMap.put(keyString, variableArray);
      }
      return statementMap;
   }

   /*
    * Return ArrayList of program slice statements for each variable
    */
   private ArrayList getVariableStatements(ArrayList inputArray) {
      ArrayList sortedArray = new ArrayList();
      ArrayList previousControlStatements = new ArrayList();
      int bracketCount = 0;
      for (int i = 0; i < inputArray.size(); i++) {
         ArrayList statementArray = (ArrayList) inputArray.get(i);
         int startIndex = 0;
         for (int count = 0; count < statementArray.size(); count++) {
            String statementString =
               (String) (statementArray.get(count));
            if (!previous(statementString, previousControlStatements,
                  startIndex)) {
               if (statementString.endsWith("}")) {
                  bracketCount--;
                  previousControlStatements.add(statementString);
                  startIndex++;
               }
               sortedArray.add(statementString);
               if (statementString.endsWith("{")) {
                  bracketCount++;
```

```
                    for (int j = startIndex;
                        j < previousControlStatements.size(); j++) {
                        previousControlStatements.remove(startIndex);
                        sortedArray.add("}");
                        bracketCount--;
                    }
                    previousControlStatements.add(statementString);
                    startIndex++;
                }
            } else {
                startIndex++; //so don't remove relevant statement from
                              // previous control statements
            }
        }
        if (i == inputArray.size() - 1) {
            for (int k = 0; k < bracketCount; k++) {
                sortedArray.add("}");
            }
        }
    }
    return sortedArray;
}

/*
 * Return whether control statement has already been added to program
 * slice
 */
private boolean previous(String statement,
        ArrayList previousStatements, int start) {
    boolean isPrevious = false;
    if (previousStatements.size() > 0) {
        for (int prevIndex = 0; prevIndex < previousStatements.size();
            prevIndex++) {
            String tempString =
                (String) previousStatements.get(prevIndex);
            if (statement.equals(tempString) && start <= prevIndex) {
                isPrevious = true;
                break;
            }
        }
    }
    return isPrevious;
}
}
```

A5.2.6 StatementAnalyser.java

```
/*
 * Created on 20-Jun-2005
 */
package main.progAnal;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Set;
import java.util.Stack;

/**
 * Class to analyse statements and return which category of statement
they are
 *
 * @author cbishop
 */
public class StatementAnalyser extends ProgramAnalyser {

   private HashMap statements;
   private Set variables;
   private ArrayList assignmentStatements;
   private ArrayList usageStatements;
   private ArrayList conditionalUsageStatements;
   private ArrayList otherStatements;

   /**
    * Constructor for StatementAnalyser
    *
    * @param inputStatements
    *            Statements for all varaiables to be analysed
    * @param variables
    *            Set of variables
    */
   public StatementAnalyser(HashMap inputStatements, Set variables) {
      statements = inputStatements;
      this.variables = variables;
   }

   /**
    * Get analysis of variables whereabouts in source code
    *
    * @return Map containing analysis for all variables
    */
   public HashMap getStatementAnalysis() {
      HashMap analysedStatements = new HashMap();
      Iterator it = variables.iterator();
      ArrayList variableStatements = new ArrayList();
      while (it.hasNext()) {
         String variable = (String) it.next();
         variableStatements = (ArrayList) statements.get(variable);
         LinkedHashMap analysedMap = analyseStatements(variable,
               variableStatements);
         analysedStatements.put(variable, analysedMap);
      }
      return analysedStatements;
   }

   /*
    * Get map of analysed statements for given input variable
    */
   private LinkedHashMap analyseStatements(String var,
         ArrayList varStatements) {
      LinkedHashMap analysedMap = new LinkedHashMap();
      assignmentStatements = find(var, varStatements, "Assignment");
      usageStatements = find(var, varStatements, "Usage");
      conditionalUsageStatements = find(var, varStatements,
            "Conditional usage");
      otherStatements = find(var, varStatements, "Other");
      HashMap assignmentWhereaboutsMap =
         whereabouts(assignmentStatements, varStatements);
      //System.out.println(var + " map size = " +
      // assignmentWhereaboutsMap.size()); //debug
      HashMap usageWhereaboutsMap = whereabouts(usageStatements,
            varStatements);
      HashMap conditionalWhereaboutsMap = whereabouts(
            conditionalUsageStatements, varStatements);
      HashMap otherWhereaboutsMap = whereabouts(otherStatements,
            varStatements);
      analysedMap.put("assignment", assignmentWhereaboutsMap);
      analysedMap.put("usage", usageWhereaboutsMap);
      analysedMap.put("conditional", conditionalWhereaboutsMap);
      analysedMap.put("other", otherWhereaboutsMap);
      return analysedMap;
   }

   /*
    * Find whereabouts of statements for a given statement type
    */
   private ArrayList find(String var, ArrayList varStatements,
         String what) {
      ArrayList returnArray = new ArrayList();
      //System.out.println(what + " statements :" + var); //debug
      for (int i = 0; i < varStatements.size(); i++) {
         String statement = (String) varStatements.get(i);
         if (what.equals("Assignment")
               || (what.equals("Usage")
               && !isInArray(statement, assignmentStatements))
               || (what.equals("Conditional usage")
               && !isInArray(statement, assignmentStatements)
               && !isInArray(statement, usageStatements))
               || (what.equals("Other")
               && !isInArray(statement, assignmentStatements)
               && !isInArray(statement, usageStatements)
               && !isInArray(statement, conditionalUsageStatements))) {
            String tempString =
                  retrieveStatement(var, statement, what);
            if (tempString != null
                  && !isInArray(tempString, returnArray)) {
               returnArray.add(tempString);
            }
         }
      }
      return returnArray;
   }

   /*
    * Analyse input statement and return whether it is a given type.
    */
   private String retrieveStatement(String var, String statement,
         String what) {
      String returnString = null;
      int bracketCount = 0;
      int window = var.length() + 3;
      String preEquals = "";
      for (int j = 0; j < statement.length(); j++) {
         String statementSubstring = statement.substring(j);
         preEquals += statement.substring(j, j + 1);
         String noSpace = removeSpaces(statementSubstring);
         String windowedPreEquals = "";
         //create sliding window for var to prevent erroneous detection
         // of var prior to "=".
         if (preEquals.length() > window) {
            windowedPreEquals = preEquals.substring(preEquals.length()
                  - window, preEquals.length());
         }
         if (statementSubstring.startsWith("(")) {
```

```java
                bracketCount++;
            } else if (statementSubstring.startsWith(")")) {
                bracketCount--;
            }
            if (what.equals("Assignment")
                    && bracketCount == 0
                    && !arrayIndex(var, statement)
                    && (assignmentCharacter(statementSubstring)
                            && statement.startsWith(var)
                            && !subString(preEquals, var)
                            || assignmentCharacter(statementSubstring)
                            && !subString(windowedPreEquals, var) || noSpace
                            .startsWith("]=")
                            && !subString(preEquals, var)
                            && contains(preEquals, var + "["))) {
                returnString = statement;
                break;
            } else if (what.equals("Usage")
                    && (bracketCount == 0
                    && statementSubstring.startsWith("=")
                    ^ printUse(statementSubstring, var))
                    && (contains(statement, var)
                    && !subString(statementSubstring, var)
                    || contains(preEquals, "[" + var)
                    || contains(preEquals, var + "]")
                    && !subString(preEquals, var))) {
                returnString = statement;
                break;
            } else if (what.equals("Conditional usage")
                    && contains(statement, var)
                    && !subString(statement, var)
                    && control(statement)) {
                returnString = statement;
                break;
            } else if (what.equals("Other") && contains(statement, var)
                    && !subString(statement, var)) {
                returnString = statement;
                break;
            }
        }
        return returnString;
    }

    /*
     * Return true if input string starts with character that qualifies
     * is as variable assignment statement
     */
    private boolean assignmentCharacter(String substring) {
        boolean appropriateCharacter = false;
        if (substring.startsWith("=") || substring.startsWith("++")
                || substring.startsWith("--")) {
            appropriateCharacter = true;
        }
        return appropriateCharacter;
    }

    /*
     * Return true if input string starts with character that qualifies
     * is as array assignment statement
     */
    private boolean arrayIndex(String var, String statement) {
        boolean isIndex = false;
        for (int i = 0; i < statement.length(); i++) {
            String statementSubstring = statement.substring(i);
            if (statementSubstring.startsWith("[" + var)
                    && !subString(statement, var)) {
                isIndex = true;
                break;
            }
        }
        return isIndex;
    }

    /*
     * Get whereabouts map for a given variable @param inputStatements
     * Assignment, usage, conditional, or other statements for a given
     * variable
     * @param statementBank Program sliced statements for variable
     */
    private HashMap whereabouts(ArrayList inputStatements,
            ArrayList statementBank) {
        HashMap returnMap = new HashMap();
        Iterator it = inputStatements.iterator();
        while (it.hasNext()) {
            String statement = (String) it.next();
            ArrayList analysedArray = buildArray(statement, statementBank);
            returnMap.put(statement, analysedArray);
        }
        return returnMap;
    }

    /*
     * Return array containing whereabouts of given input statement in
     * program sliced statements
     */
    private ArrayList buildArray(String statement,
            ArrayList statementBank) {
        int lineCount = 1;
        int latestPosition = 0;
        int foundCount = 0;
        ArrayList indexCard = new ArrayList();
        Stack doStack = new Stack();
        ArrayList returnArray = new ArrayList();
        int curlyCount = 0;
        while (lineCount <= statementBank.size()) {
            String bankedStatement =
                    (String) statementBank.get(lineCount - 1);
            if (bankedStatement.endsWith("{")) {
                returnArray.add(new Integer(lineCount));
                //add statement line index to return array
                returnArray.add(bankedStatement);
                //add statement to return array
                if (bankedStatement.startsWith("do")) {
                    doStack.push(new Integer(curlyCount));
                    //note position of do loop in hierarchy
                }
                curlyCount++;
            } else if (bankedStatement.endsWith("}")) {
                for (int p = 0; p < 2; p++) {
                    returnArray.remove(returnArray.size() - 1);
                    //remove line index and signature from end of array
                }
                curlyCount--;
                if (!doStack.empty()) {
                    Integer tempInt = (Integer) doStack.peek();
                    if (curlyCount == tempInt.intValue()) {
                        doStack.pop();
                        //remove do loop indicator from top of stack

                    }
                }
            }
            if (statement.equals(bankedStatement)
                    && lineCount > latestPosition) {
                if (!doStack.empty()) {
                //add while parts of outstanding do loops
                    int doWhileCount = lineCount;
                    //starting from next line
                    for (int j = 0; j < doStack.size(); j++) {
                        Integer doCount = (Integer) doStack.pop();
```

```
                        while (doWhileCount < statementBank.size()) {
                            String testString = (String) statementBank
                                    .get(doWhileCount);
                            if (curlyCount == doCount.intValue()) {
                                String tempString = "do {} " + testString;
                                int index = (2 * doCount.intValue()) + 1;
                                returnArray.add(index, tempString);
                                //replace existing do statement with do while
                                //statement
                                returnArray.remove(index + 1);
                                doWhileCount = statementBank.size();
                            }
                            if (testString.endsWith("{"))
                                curlyCount++;
                            else if (testString.endsWith("}"))
                                curlyCount--;
                            doWhileCount++;
                        }
                    }
                }
                latestPosition = lineCount;
                //add line number after previous found line number
                returnArray.add(foundCount, new Integer(lineCount));
                lineCount = 0;
                //note where new found statement number will be located in
                //returnArray
                indexCard.add(new Integer(foundCount + 1));
                //to ensure that future statement line numbers are added
                //before latest loop line numbers
                foundCount = returnArray.size();
            }
            lineCount++;
        }
        indexCard.add(new Integer(foundCount + 1));
        returnArray.add(0, indexCard); //add index beginning of array
        return returnArray;
    }
}
```

A5.2.7 MethodGetter.java

```
/*
 * Created on 15-Jul-2005
 */
package main.progAnal;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Set;

/**
 * @author cbishop
 */
public class MethodGetter extends ProgramAnalyser {

   /**
    * Return list of method names in source code
    *
    * @param sortedStatements
    *            HashMap being program slice for variable
    * @return ArrayList of method names in program
    */
   public ArrayList getMethods(HashMap sortedStatements) {
      ArrayList methods = new ArrayList();
      Set keySet = sortedStatements.keySet();
      Iterator it = keySet.iterator();
      //get first available variable
      String variable = (String) it.next();
      ArrayList sortedArray =
         (ArrayList) sortedStatements.get(variable);
      it = sortedArray.iterator();
      while (it.hasNext()) {
         String statement = (String) it.next();
         if (methodStatement(statement)) {
            //add name of method to methods
            int j = 0;
            String reverseName = "";
            for (int i = 0; i < statement.length(); i++) {
               String statementSubstring = statement.substring(i);
               //where method name ends
               if (statementSubstring.startsWith("(")) {
                  j = i - 1;
                  i = statement.length();
               }
            }
            //so that possible gap between method name
            // and '(' is not interpreted as method name
            int position = 0;
            //work backwards adding method name to
            // reverseName until beginning of method name
            while (j > 0) {
               String charString = statement.substring(j, j + 1);
               if (charString.equals(" ") && position > 0) {
                  //look for space before method name
                  j = 0;
               }
               reverseName += charString;
               j--;
               position++;
            }
            String methodName = "";
            for (j = reverseName.length() - 1; j >= 0; j--) {
               methodName += reverseName.substring(j, j + 1);
            }
            methods.add(methodName);
            //System.out.println(methodName); debug
         }
      }
```

```
        return methods;
    }
}
```

A5.2.8 ConditionChecker.java

```
/*
 * Created on 06-Jul-2005
 */
package main.progAnal;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Set;

/**
 * Class to check conditions under which variables are assigned and used
 *
 * @author cbishop
 */
public class ConditionChecker extends ProgramAnalyser {

   private String variable;

   private LinkedHashMap analysedMap;

   private ArrayList methods;

   /**
    * Constructor for ConditionChecker
    *
    * @param var
    *            Name of input variable
    * @param inputMap
    *            Analysed map of variable statements
    * @param methodNames
    *            List of method names in class
    */
   public ConditionChecker(LinkedHashMap inputMap, String var,
         ArrayList methodNames) {
      variable = var;
      analysedMap = inputMap;
      methods = methodNames;
   }

   /**
    * Return whether there is no assignment statement for variable
    *
    * @return boolean true is there is no assignment statement
    */
   public boolean noAssignmentStatement() {
      boolean noAssignment = false;
      HashMap assignmentWhereaboutsMap = (HashMap) analysedMap
            .get("assignment");
      if (assignmentWhereaboutsMap.size() < 1) {
         noAssignment = true;
      }
      return noAssignment;
   }

   /**
    * Return whether there is only one assignment statement for the
    * variable in the program
    *
    * @return boolean true if there is only one assignment statement
    */
   public boolean onlyOneAssignmentStatement() {
      boolean onlyOneAssignment = false;
      HashMap assignmentWhereaboutsMap = (HashMap) analysedMap
            .get("assignment");
      if (assignmentWhereaboutsMap.size() <= 1) {
         onlyOneAssignment = true;
      }
      return onlyOneAssignment;
   }

   /**
    * Return map of statements found in loops or branches
    *
    * @param what
    *            String being the category of statement
    * @param where
    *            String signifying whether to look in loops or branches
    * @return HashMap containing statement and loop/branch statement in
    * which it is found
    */
   public HashMap foundIn(String what, String where) {
      HashMap foundIn = new HashMap();
      HashMap whereaboutsMap = (HashMap) analysedMap.get(what);
      Set statementSet = whereaboutsMap.keySet();
      Iterator it = statementSet.iterator();
      while (it.hasNext()) {
         ArrayList statementIn = new ArrayList();
         String statement = (String) it.next();//get statement
         ArrayList specificOccuranceIn = new ArrayList();
         //get array for given statement
         ArrayList occuranceArray = (ArrayList) whereaboutsMap
                 .get(statement);
         boolean found = false;
         if (loop(statement) && where.equals("loop")) {
             //add line number of start of loop to array
             specificOccuranceIn.add(occuranceArray.get(1));
             //add loop statement to array
             specificOccuranceIn.add(statement);
             found = true;
         }
         //get index for statement array
         ArrayList indexArray = (ArrayList) occuranceArray.get(0);
         for (int j = 0; j < indexArray.size() - 1; j++) {
             //get position in array of statement line number
             Integer index = (Integer) indexArray.get(j);
             //get posiition in array of next statement line number
             Integer nextIndex = (Integer) indexArray.get(j + 1);
             for (int i = index.intValue() + 1;
                     i < nextIndex.intValue(); i += 2) {
                 //get branch, loop, method, etc within which assignment
                 // found
                 String statementString =
                     (String) occuranceArray.get(i + 1);
                 if (where.equals("loop")) {
                     if (loop(statementString)) { //isolate loops
                         //add line number of start of loop to array
                         specificOccuranceIn.add(occuranceArray.get(i));
                         //add loop statement to array
                         specificOccuranceIn.add(statementString);
                         found = true;
                     }
                 } else if (where.equals("branch")) {
                     if (branch(statementString)) {
                         //add line number of start of loop to array
                         specificOccuranceIn.add(occuranceArray.get(i));
                         //add loop statement to array
                         specificOccuranceIn.add(statementString);
                         found = true;
                     }
                 }
             }
             if (found) {
                 //if assignment in loop add line number of assignment
                 // statement
                 specificOccuranceIn.add(0, occuranceArray.get(index
                     .intValue()));
                 statementIn.add(specificOccuranceIn);
             }
         }
         if (found) {
             foundIn.put(statement, statementIn);
         }
     }
     return foundIn;
 }

 /**
  * Return variable statements occuring in assignment loops together
  * with loop signatures and line numbers
  *
  * @param assignmentsInLoop
  *            HashMap of assignment statements found in loops
  * @param inputLoopStatements
  *            HashMap of loop statements
  * @return HashMap containing assignment statements and which loops
  * they are in
  */
 public HashMap foundInAssignmentLoop(HashMap assignmentsInLoop,
         HashMap inputLoopStatements) {
     HashMap foundInAssignmentLoop = new HashMap();
     Set statementSet = inputLoopStatements.keySet();
     Iterator it = statementSet.iterator();
     while (it.hasNext()) {
         String statement = (String) it.next();
         ArrayList statementInLoop = (ArrayList) inputLoopStatements
                 .get(statement);
         ArrayList specificStatement = compareLoopStatements(
                 assignmentsInLoop, statementInLoop);
         if (specificStatement.size() > 0) {
             foundInAssignmentLoop.put(statement, specificStatement);
         }
     }
     return foundInAssignmentLoop;
 }

 /*
  * Return array list containing occurance arrays for a given
  * statement
  */
 private ArrayList compareLoopStatements(HashMap assignmentsInLoop,
         ArrayList statementInLoop) {
     ArrayList specificStatement = new ArrayList();
     Set statementSet = assignmentsInLoop.keySet();
     Iterator it = statementSet.iterator();
     while (it.hasNext()) {
         String statement = (String) it.next();
         ArrayList assignStatInLoop =
             (ArrayList) assignmentsInLoop.get(statement);
         for (int i = 0; i < statementInLoop.size(); i++) {
             for (int j = 0; j < assignStatInLoop.size(); j++) {
                 ArrayList compares = (ArrayList) statementInLoop.get(i);
                 ArrayList assigns = (ArrayList) assignStatInLoop.get(j);
                 ArrayList inAssignLoop =
                     lookForCommon(compares, assigns);
                 if (inAssignLoop.size() > 0) {
                     inAssignLoop.add(2, statement); //add assignment
                     // statement to array
                     specificStatement.add(inAssignLoop);
                 }
             }
         }
     }
     return specificStatement;
 }
```

```
/*
 * Return array list containing details of occurance for given
 * statement
 */
private ArrayList lookForCommon(ArrayList compares,
      ArrayList assigns) {
   ArrayList returnArray = new ArrayList();
   boolean found = false;
   for (int i = 1; i < compares.size(); i += 2) {
      Integer comp = (Integer) compares.get(i);
      for (int j = 1; j < assigns.size(); j += 2) {
         Integer ass = (Integer) assigns.get(j);
         if (comp.intValue() == ass.intValue()) {
            found = true;
            returnArray.add(comp);
            returnArray.add(compares.get(i + 1));
         }
      }
   }
   if (found) {
      //add line number of compare statement
      returnArray.add(0, compares.get(0));
      //add line number of assign statement
      returnArray.add(1, assigns.get(0));

   }
   return returnArray;
}

/**
 * Return ArrayList of statements where use in assignment loop is
 * directly or indirectly related to loop condition
 *
 * @param foundInAssignmentLoop
 *           HashMap of statement found in assignment loop
 * @return ArrayList of statements that relate to loop condition
 */
public ArrayList useForLoopCondition(HashMap foundInAssignmentLoop) {
   ArrayList useForLoopCondition = new ArrayList();
   Set statementSet = foundInAssignmentLoop.keySet();
   Iterator it = statementSet.iterator();
   while (it.hasNext()) {
      String statement = (String) it.next();
      ArrayList statementInLoop = (ArrayList) foundInAssignmentLoop
            .get(statement);
      for (int i = 0; i < statementInLoop.size(); i++) {
         ArrayList specificOccurance = (ArrayList) statementInLoop
               .get(i);
         for (int j = 4; j < specificOccurance.size(); j += 2) {
            String loopStatement =
               (String) specificOccurance.get(j);
            if (statement.equals(loopStatement)) {
               useForLoopCondition.add(statement);
            }
         }
      }
   }
   return useForLoopCondition;
}

/**
 * Return true if assignment loop is dependent on assignment
 * statement following conditional use for a given variable
 *
 * @param foundInAssignmentLoop
 *           HashMap of statements found in assignment loop
 * @return boolean
 */
public boolean indirectUse(HashMap foundInAssignmentLoop) {
   boolean indirectUse = false;
```

```
        Set statementSet = foundInAssignmentLoop.keySet();
        Iterator it = statementSet.iterator();
        while (it.hasNext()) {
            String statement = (String) it.next();
            ArrayList statementInLoop = (ArrayList) foundInAssignmentLoop
                    .get(statement);
            for (int i = 0; i < statementInLoop.size(); i++) {
                ArrayList specificOccurance = (ArrayList) statementInLoop
                        .get(i);
                for (int j = 4; j < specificOccurance.size(); j += 2) {
                    String loopStatement =
                        (String) specificOccurance.get(j);
                    int bracketCount = 0;
                    boolean afterBrackets = false;
                    String indirectAssign = "";
                    for (int k = 0; k < statement.length(); k++) {
                        String statementSubstring = statement.substring(k);
                        if (bracketCount == 0 && afterBrackets) {
                            indirectAssign += statement.substring(k, k + 1);
                            if (indirectAssign.endsWith("=")) {
                                //remove "="
                                indirectAssign = indirectAssign.substring(0,
                                        indirectAssign.length() - 1);
                                indirectAssign = indirectAssign.trim();
                                if (contains(statement, indirectAssign)
                                        && !subString(statement, indirectAssign)
                                        && statementSet.size() < 2) {
                                    indirectUse = true;
                                    break;
                                }
                            }
                        }
                        if (statementSubstring.startsWith("(")) {
                            bracketCount++;
                            afterBrackets = true;
                        }
                        if (statementSubstring.startsWith(")")) {
                            bracketCount--;
                        }
                    }
                }
            }
        }
        return indirectUse;
    }

    /**
     * Return list of statements where variable is assigned in a "for"
     * loop declaration
     *
     * @param what
     *            String signfying the category of statement
     * @return ArrayList of "for" loop assignments
     */
    public ArrayList assignmentInFor(String what) {
        ArrayList assignInFor = new ArrayList();
        HashMap whereaboutsMap = (HashMap) analysedMap.get(what);
        Set statementSet = whereaboutsMap.keySet();
        Iterator it = statementSet.iterator();
        while (it.hasNext()) {
            String statement = (String) it.next();
            String noSpace = removeSpaces(statement);
            if (statement.startsWith("for")
                    && inBrackets(noSpace, variable)
                    && contains(noSpace, variable + "=")
                    && !subString(statement, variable)) {
                assignInFor.add(statement);
            }
        }
        return assignInFor;
```

```
}

/**
 * Return if variable is not used directly in the program
 *
 * @return boolean
 */
public boolean noDirectUsage() {
   boolean noDirectUsage = false;
   HashMap usageWhereaboutsMap = (HashMap) analysedMap.get("usage");
   if (usageWhereaboutsMap.size() < 1) {
      noDirectUsage = true;
   }
   return noDirectUsage;
}

/**
 * Return map containing assignment statements and branch statements
 * where variable is condition for branch
 *
 * @return HashMap
 */
public HashMap conditionAssignStatement() {
   HashMap conditionForAssignmentBranch = new HashMap();
   HashMap assignments = (HashMap) analysedMap.get("assignment");
   ArrayList conditionAssignment = new ArrayList();
   Set statementSet = assignments.keySet();
   Iterator it = statementSet.iterator();
   //check all assignment statements to see if they are branch
   //statements
   while (it.hasNext()) {
      String assignmentStatement = (String) it.next();
      boolean conditionalAssignment = false;
      //check if variable is condition for its assignment
      if (branch(assignmentStatement)
            && inBrackets(assignmentStatement, variable)) {
         ArrayList statementIn = new ArrayList();
         //get array for given statement
         ArrayList occuranceArray = (ArrayList) assignments
               .get(assignmentStatement);
         //add line number of assignment statement
         conditionAssignment.add(occuranceArray.get(1));
         //add line number of conditional statement
         conditionAssignment.add(occuranceArray.get(1));
         //add assignment statement to array list
         conditionAssignment.add(0, assignmentStatement);
         conditionalAssignment = true;
      }
      if (conditionalAssignment)
         conditionForAssignmentBranch.put(assignmentStatement,
               conditionAssignment);
   }
   return conditionForAssignmentBranch;
}

/**
 * Return map containing incidences where assignment in branch is
 * also in loop
 *
 * @param assignmentsInBranch
 *            HashMap containing assignmentsInBranch
 * @param inputLoopStatements
 *            HashMap containing loop statements
 * @return HashMap
 */
public HashMap conditionForAssignmentBranch(
      HashMap assignmentsInBranch, HashMap inputLoopStatements) {
   HashMap conditionForAssignmentBranch = new HashMap();
   Set inputSet = inputLoopStatements.keySet();
   Iterator it = inputSet.iterator();
```

```
      while (it.hasNext()) {
      //get statement to look for
         String statement = (String) it.next();
         Set branchAssignments = assignmentsInBranch.keySet();
         Iterator iter = branchAssignments.iterator();
         ArrayList conditionAssignment = new ArrayList();
         while (iter.hasNext()) {
            String assignStatement = (String) iter.next();
            ArrayList statementInBranch = (ArrayList)
               assignmentsInBranch.get(assignStatement);
            for (int i = 0; i < statementInBranch.size(); i++) {
               ArrayList specificOccurance =
                  (ArrayList) statementInBranch.get(i);
               //add assignment statement to array list
               conditionAssignment.add(assignStatement);
               //add line number for assignment statement
               conditionAssignment.add(specificOccurance.get(0));
               boolean isCommon = false;
               for (int j = 2; j < specificOccurance.size(); j += 2) {
                  String conditionalStatement =
                     (String) specificOccurance.get(j);
                  if (statement.equals(conditionalStatement)) {
                     //add line number for branch
                     conditionAssignment.add(specificOccurance
                           .get(j - 1));
                     isCommon = true;
                     j = specificOccurance.size();
                  }
               }
               //if no common statement found, remove assignment
               //statement and line number
               if (!isCommon) {
                  conditionAssignment
                        .remove(conditionAssignment.size() - 1);
                  conditionAssignment
                        .remove(conditionAssignment.size() - 1);
               }
            }
         }
         //check if assignment in branches for which variable is
         //condition
         if (conditionAssignment.size() > 0) {
            conditionForAssignmentBranch.put(statement,
               conditionAssignment);
         }
      }
      return conditionForAssignmentBranch;
   }

   /**
    * Return list of statement of a given type
    *
    * @param assignmentsInLoop
    *            HashMap of assignment statements found in loops
    * @param statementType
    *            String being the type of statement to look for
    * @return ArrayList of statements
    */
   public ArrayList statementTypeCheck(HashMap assignmentsInLoop,
         String statementType) {
      ArrayList returnArray = new ArrayList();
      Set statementSet = assignmentsInLoop.keySet();
      Iterator it = statementSet.iterator();
      //check all assignment statements to see if variable appears on
      //both sides of statement
      while (it.hasNext()) {
         String statement = (String) it.next();
         if (statementType.equals("bothSides")
               && onBothSides(statement)) {
            returnArray.add(statement);
```

```
        } else if (statementType.equals("indirect")
                && indirectBothSides(statement)) {
            returnArray.add(statement);
        } else if (statementType.equals("toggle")
                && toggleStatement(statement)) {
            returnArray.add(statement);
        } else if (statementType.equals("incDec")
                && incDecEtc(statement)) {
            returnArray.add(statement);
        }
    }
    return returnArray;
}

/*
 * Check if variable appears on both sides of assignment statement
 */
private boolean onBothSides(String statement) {
    boolean onBothSides = false;
    String afterEquals = afterEquals(statement);
    if (contains(afterEquals, variable) && !contains(afterEquals, ",")
            && !subString(afterEquals, variable)) {
        onBothSides = true;
    }
    return onBothSides;
}

/*
 * Check if variable appears in directly on both sides of assignment
 * statement
 */
private boolean indirectBothSides(String statement) {
    boolean indirectBothSides = false;
    int bracketCount = 0;
    for (int k = 0; k < statement.length(); k++) {
        String statementSubstring = statement.substring(k);
        if (statementSubstring.startsWith(variable)
                && bracketCount == 0) {
            String afterVar =
                statementSubstring.substring(variable.length());
            afterVar = removeSpaces(afterVar);
            if (afterVar.startsWith("+=") || afterVar.startsWith("-=")
                    || afterVar.startsWith("*=")
                    || afterVar.startsWith("/=")
                    || afterVar.startsWith("%=")
                    && !arithExp(afterVar.substring(2))) {
                indirectBothSides = true;
                k = statement.length(); //end loop if incDec detected
            }
        }
        if (statementSubstring.startsWith("(")) {
            bracketCount++;
        }
        if (statementSubstring.startsWith(")")) {
            bracketCount--;
        }
    }
    return indirectBothSides;
}

/*
 * Return true if variable is toggled within statement
 */
private boolean toggleStatement(String statement) {
    boolean toggleStatement = false;
    String afterEquals = afterEquals(statement);
    if (contains(afterEquals, "!" + variable)
            && !subString(afterEquals, "!" + variable)) {
        toggleStatement = true;
    }
```

```
      return toggleStatement;
   }

   /*
    * Return true if variable is incremented/decremented in statement
    */
   private boolean incDecEtc(String statement) {
      boolean incDecStatement = false;
      int bracketCount = 0;
      for (int k = 0; k < statement.length(); k++) {
         String statementSubstring = statement.substring(k);
         if (statementSubstring.startsWith(variable)
               && bracketCount == 0) {
            String afterVar =
               statementSubstring.substring(variable.length());
            afterVar = removeSpaces(afterVar);
            if (afterVar.startsWith("++")
                  || afterVar.startsWith("--")
                  || afterVar.startsWith("**")
                  || afterVar.startsWith("//")
                  || (afterVar.startsWith("+=")
                  || afterVar.startsWith("-=")
                  || afterVar.startsWith("*=")
                  || afterVar.startsWith("/=")
                  || afterVar.startsWith("%=")
                  && arithExp(afterVar.substring(2)))
                  || (afterVar.startsWith("=")
                  && onBothSides(statement)
                  && arithExp(afterVar.substring(1), variable))) {
               incDecStatement = true;
               k = statement.length(); //end loop if incDec detected
            }
         }
         if (statementSubstring.startsWith("(")) {
            bracketCount++;
         }
         if (statementSubstring.startsWith(")")) {
            bracketCount--;
         }
      }
      return incDecStatement;
   }

   /**
    * Return map of statements where variable is used outside of the
    * loop in which it is assigned
    *
    * @param inAssignLoop
    *            HashMap containing usage/conditional statement found in
    *            assignment loop
    * @param what
    *            String being category of statement
    * @return HashMap
    */
   public HashMap outsideAssignLoop(HashMap inAssignLoop, String what) {
      HashMap outsideAssignLoop = new HashMap();
      HashMap whereaboutsMap = (HashMap) analysedMap.get(what);
      Set statementSet = whereaboutsMap.keySet();
      Iterator it = statementSet.iterator();
      while (it.hasNext()) {
         ArrayList statementIn = new ArrayList();
         String statement = (String) it.next();//get statement
         boolean foundOutsideAssign = false;
         ArrayList specificOccuranceOutside = new ArrayList();
         if ((loop(statement) && what.equals("conditional"))
               || what.equals("usage")) {
            //get array for given statement
            ArrayList occuranceArray = (ArrayList) whereaboutsMap
                  .get(statement);
            //get index for statement array
```

```
                ArrayList indexArray = (ArrayList) occuranceArray.get(0);
                for (int j = 0; j < indexArray.size() - 1; j++) {
                    //get position in array of statement line number
                    Integer index = (Integer) indexArray.get(j);
                    //get branch, loop, method, etc within which assignment
                    // found
                    Integer statementLineNo = (Integer) occuranceArray
                            .get(index.intValue());
                    specificOccuranceOutside = foundOutside(statementLineNo,
                            inAssignLoop);
                }
            }
            //if statement found outside assign loop
            if (specificOccuranceOutside.size() > 0) {
                //add array of line numbers for that statement to return
                //map
                outsideAssignLoop.put(statement, specificOccuranceOutside);
            }
        }
        return outsideAssignLoop;
    }

    /*
     * Return list of statements found outside assignment loop
     */
    private ArrayList foundOutside(Integer statementLineNo,
            HashMap foundInAssignLoop) {
        ArrayList returnArray = new ArrayList();
        Set foundSet = foundInAssignLoop.keySet();
        Iterator it = foundSet.iterator();
        boolean matchFound = false;
        while (it.hasNext()) {
            //get statement found in assign loop
            String foundStatement = (String) it.next();
            //get array for given statement
            ArrayList statementInLoop = (ArrayList) foundInAssignLoop
                    .get(foundStatement);
            for (int i = 0; i < statementInLoop.size(); i++) {
                //get array list for occurance of statement in loop
                ArrayList specificOccurance = (ArrayList) statementInLoop
                        .get(i);
                //get line number of statement
                Integer statementInLoopLineNo = (Integer) specificOccurance
                        .get(0);
                //see if input statement found in assign loop
                if (statementLineNo.intValue() == statementInLoopLineNo
                        .intValue()) {
                    matchFound = true;
                }
            }
        }
        //if input statement line number not found in assign
        //loop add input statement line number to return array
        if (!matchFound) {
            returnArray.add(statementLineNo);
        }
        return returnArray;
    }

    /**
     * Return list of statements where variable appears to be being used
     * like an array
     *
     * @return ArrayList
     */
    public ArrayList arrayCheck() {
        ArrayList isArray = new ArrayList();
        String[] whatArray = { "assignment", "usage", "conditional",
            "other" };
        for (int i = 0; i < whatArray.length; i++) {
            HashMap whereaboutsMap =
               (HashMap) analysedMap.get(whatArray[i]);
            Set statementSet = whereaboutsMap.keySet();
            Iterator it = statementSet.iterator();
            while (it.hasNext()) {
               String statement = (String) it.next();//get statement
               String noSpace = removeSpaces(statement);
               if (contains(noSpace, variable + "[")
                     && !subString(noSpace, variable)) {
                  isArray.add(statement);
                  break;
               } else if (contains(noSpace, "]" + variable)) {
                  isArray.add(statement);
                  break;
               }
            }
         }
         return isArray;
      }

      /**
       * Return list of statements where variable appears to be being used
       * as organizer
       *
       * @return ArrayList
       */
      public ArrayList reorganize() {
         ArrayList returnArray = new ArrayList();
         HashMap whereaboutsMap = (HashMap) analysedMap.get("assignment");
         Set statementSet = whereaboutsMap.keySet();
         Iterator it = statementSet.iterator();
         boolean notReorganize = false;
         while (it.hasNext()) {
            String statement = (String) it.next();//get statement
            String afterEquals = afterEquals(statement);
            String noSpace = removeSpaces(afterEquals);
            if (contains(noSpace, variable + "[")
                  && !subString(afterEquals, variable)
                  && !arithExp(noSpace, variable)) {
               returnArray.add(statement);
            } else if (contains(noSpace, variable)
                  || arithExp(noSpace, variable)
                  && !subString(afterEquals, variable)) {
               notReorganize = true;
               returnArray.clear();
               break;
            }
         }
         return returnArray;
      }

      /**
       * Return list of statements where variable appears to be being used
       * as transformation
       *
       * @param assignmentsInLoop HashMap containing assignments in loops
       * @return ArrayList
       */
      public ArrayList transform(HashMap assignmentsInLoop) {
         ArrayList returnArray = new ArrayList();
         Set statementSet = assignmentsInLoop.keySet();
         Iterator it = statementSet.iterator();
         boolean isTransform = true;
         while (it.hasNext()) {
            String statement = (String) it.next();
            returnArray.add(statement);
            String afterEquals = afterEquals(statement);
            for (int i = 0; i < methods.size(); i++) {
               String method = (String) methods.get(i);
               //check if method is called on right hand side of
               //assignment and if so clear array
               if (contains(afterEquals, method + "(")) {
                  returnArray.clear();
                  break;
               }
            }
            //if any statement is not transformation clear array and end
            //method
            if (!is(afterEquals, "operator")) {
               returnArray.clear();
               break;
            }
         }
         return returnArray;
      }

      /**
       * Return list of statements in which variable is indirectly toggled
       * by being set twice within a loop, once in a nested loop, with
       * opposing values
       *
       * @param assignmentsInLoop
       *            HashMap of assignment statements in loop
       * @return ArrayList
       */
      public ArrayList nestedBooleanAssign(HashMap assignmentsInLoop) {
         ArrayList returnArray = new ArrayList();
         Set statementSet = assignmentsInLoop.keySet();
         Iterator it = statementSet.iterator();
         boolean isBoolean = false;
         boolean isTrue = false;
         ArrayList compareArray = new ArrayList();
         while (it.hasNext()) {
            String statement = (String) it.next();
            String afterEquals = afterEquals(statement);
            if (contains(afterEquals, "true")
                  && !subString(afterEquals, "true")) {
               isBoolean = true;
               isTrue = true;
            } else if (contains(afterEquals, "false")
                  && !subString(afterEquals, "false")) {
               isBoolean = true;
            }
            if (isBoolean) {
               ArrayList statementInLoop = (ArrayList) assignmentsInLoop
                     .get(statement);
               boolean isNested = false;
               for (int i = 0; i < statementInLoop.size(); i++) {
                  ArrayList specificOccurance =
                     (ArrayList) statementInLoop.get(i);
                  if (specificOccurance.size() > 3) {
                     isNested = true;
                  }
                  compareArray.add(statement); //add statement in question
                  //add outer most line number of loop
                  compareArray.add(specificOccurance.get(1));
                  //add whether assignment is true or false
                  compareArray.add(new Boolean(isTrue));
                  //add whether assign statement is in nest loop
                  compareArray.add(new Boolean(isNested));
               }
            }
         }
         if (compareArray.size() > 4) {
            returnArray = compareStatements(compareArray);
         }
         return returnArray;
      }
```

```
    /*
     * Return list of statement stating whether the each assignment
statement is
     * in a nested loop or not
     */
    private ArrayList compareStatements(ArrayList compareArray) {
        ArrayList returnArray = new ArrayList();
        for (int i = 0; i < compareArray.size(); i += 4) {
            Boolean isNested = (Boolean) compareArray.get(i + 3);
            if (isNested.booleanValue()) {
                Integer comparedLineNum =
                    (Integer) compareArray.get(i + 1);
                Boolean comparedValue = (Boolean) compareArray.get(i + 2);
                //compare line number with loop lines for rested of array
                for (int j = i + 4; j < compareArray.size(); j += 4) {
                    Boolean compareNested =
                        (Boolean) compareArray.get(j + 3);
                    Integer compareInteger =
                        (Integer) compareArray.get(j + 1);
                    Boolean compareValue =
                        (Boolean) compareArray.get(j + 2);
                    if (!compareNested.booleanValue()
                            && comparedLineNum.intValue() ==
                                compareInteger.intValue()
                            && comparedValue != compareValue) {
                        returnArray.add(compareArray.get(i));
                        returnArray.add("nested");
                        returnArray.add(compareArray.get(j));
                        break;
                    }
                }
            } else {
                Integer comparedLineNum =
                    (Integer) compareArray.get(i + 1);
                Boolean comparedValue = (Boolean) compareArray.get(i + 2);
                //compare line number with loop lines for rested of array
                for (int j = i + 4; j < compareArray.size(); j++) {
                    Boolean compareNested =
                        (Boolean) compareArray.get(j + 3);
                    Integer compareInteger =
                        (Integer) compareArray.get(j + 1);
                    Boolean compareValue =
                        (Boolean) compareArray.get(j + 2);
                    if (compareNested.booleanValue()
                            && comparedLineNum.intValue() == compareInteger
                                    .intValue()
                            && comparedValue != compareValue) {
                        returnArray.add(compareArray.get(i));
                        returnArray.add("not nested");
                        returnArray.add(compareArray.get(j));
                        break;
                    }
                }
            }
        }
        return returnArray;
    }

    /**
     * Return list of statement in which variable is assigned with output
     * from method call
     *
     * @param assignmentsInLoop
     *            HashMap of assignment statements found in loops
     * @return ArrayList
     */
    public ArrayList assignedWithMethod(HashMap assignmentsInLoop) {
        ArrayList returnArray = new ArrayList();
        Set statementSet = assignmentsInLoop.keySet();
        Iterator it = statementSet.iterator();
        boolean isTransform = true;
        while (it.hasNext()) {
            String statement = (String) it.next();
            String afterEquals = afterEquals(statement);
            for (int i = 0; i < methods.size(); i++) {
                String method = (String) methods.get(i);
                //check if method is called on right hand side of
                //assignment
                if (contains(afterEquals, method + "(")) {
                    returnArray.add(statement);
                }
            }
        }
        return returnArray;
    }

    /**
     * Return list of statements where variable is assigned with value in
     * loop
     * before it is used in that loop
     *
     * @param usageInAssignments
     * @return ArrayList
     */
    public ArrayList assignBeforeUse(HashMap usageInAssignments) {
        ArrayList returnArray = new ArrayList();
        Set usageSet = usageInAssignments.keySet();
        Iterator it = usageSet.iterator();
        while (it.hasNext()) {
            String usageStatement = (String) it.next();
            ArrayList usageInLoop = (ArrayList) usageInAssignments
                    .get(usageStatement);
            for (int i = 0; i < usageInLoop.size(); i++) {
                ArrayList usageOccurance = (ArrayList) usageInLoop.get(i);
                Integer assLineNum = (Integer) usageOccurance.get(1);
                int assL = assLineNum.intValue();
                //get outer most loop line number
                Integer useLineNum = (Integer) usageOccurance.get(0);
                int useL = useLineNum.intValue();
                if (assLineNum.intValue() < useLineNum.intValue()) {
                    returnArray.add(usageStatement);
                    returnArray.add(usageOccurance.get(2));
                    for (int j = 4; j < usageOccurance.size(); j += 2) {
                        returnArray.add(usageOccurance.get(j));
                    }
                } else {
                    returnArray.clear();
                }
            }
        }
        return returnArray;
    }

    /**
     * Return list of statements where variable is assigned with value
     * from instantiation of new object, or assigned directly with
     * boolean value
     *
     * @param assignmentsInLoop
     *            HashMap of assignment statement found in loops
     * @return ArrayList
     */
    public ArrayList otherAssignment(HashMap assignmentsInLoop) {
        ArrayList returnArray = new ArrayList();
        Set usageSet = assignmentsInLoop.keySet();
        Iterator it = usageSet.iterator();
        while (it.hasNext()) {
            String assignStatement = (String) it.next();
            String afterEquals = afterEquals(assignStatement);
            String noSpace = removeSpaces(afterEquals);
            if (is(afterEquals, "other")) {
               returnArray.add(assignStatement);
            }
            //check if value assigned by instantiating new object
            else if (contains(afterEquals, "new")
                  && !subString(afterEquals, "new")) {
               returnArray.add(assignStatement);
            } else if (noSpace.equals("=true;")) {
               returnArray.add(assignStatement);
            } else if (noSpace.equals("=false;")) {

            }
         }
         return returnArray;
      }
   }
```

A5.3 main.rules.java

A5.3.1 RuleApplyer.java

```
/*
 * Created on 05-Jul-2005
 *
 */
package main.rules;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Set;

/**
 * @author cbishop
 */
public class RuleApplyer {

    /**
     * Return map of results from role checking
     *
     * @param analysedStatements
     *            Map containing whereabouts of variable statements
     * @param roles
     *            Map containing variables and their declared roles
     * @return HashMap containing checked roles for each variable
     */
    public HashMap applyRules(HashMap analysedStatements,
            LinkedHashMap roles, ArrayList methods) {
        HashMap checkedRoles = new HashMap();
        ArrayList checkedRole = new ArrayList();
        Set variableSet = roles.keySet();
        Iterator it = variableSet.iterator();
        while (it.hasNext()) {
            String variable = (String) it.next();
            Integer roleValue = (Integer) roles.get(variable);
            LinkedHashMap analysedMap =
                (LinkedHashMap) analysedStatements.get(variable);
            RoleChecker roleChecker;
            switch (roleValue.intValue()) {
            case 1:
                roleChecker = new FixedValue(analysedMap,
                        variable, methods);
                break;
            case 2:
                roleChecker = new Organizer(analysedMap,
                        variable, methods);
                break;
            case 3:
                roleChecker = new Stepper(analysedMap, variable, methods);
                break;
            case 4:
                roleChecker = new MostRecentHolder(analysedMap, variable,
                        methods);
                break;
            case 5:
                roleChecker = new Gatherer(analysedMap, variable, methods);
                break;
            case 6:
                roleChecker = new MostWantedHolder(analysedMap, variable,
                        methods);
                break;
            case 7:
                roleChecker = new OneWayFlag(analysedMap, variable,
                        methods);
                break;
            case 8:
                roleChecker = new Transformation(analysedMap, variable,
                        methods);
                break;
            case 9:
                roleChecker = new Follower(analysedMap, variable, methods);
                break;
            case 10:
                roleChecker = new Temporary(analysedMap, variable,
                        methods);
                break;
            default:
                roleChecker = new OtherRole(analysedMap, variable,
                        methods);
            }
            checkedRole = roleChecker.checkRole();
            checkedRoles.put(variable, checkedRole);
        }
        return checkedRoles;
    }
}
```

A5.3.2 RoleChecker.java

```
/*
 * Created on 06-Jul-2005
 *
 */
package main.rules;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Set;
import main.progAnal.ConditionChecker;

/**
 * @author cbishop
 */
public abstract class RoleChecker {

   protected String variable;
   protected LinkedHashMap analysedMap;
   protected ArrayList methods;
   protected ConditionChecker conditionChecker;
   protected ResultStringer resultStringer;
   //condition storage
   protected HashMap assignmentsInLoop;
   protected HashMap usagesInLoop;
   protected HashMap conditionalsInLoop;
   protected HashMap usageInAssignmentLoop;
   protected HashMap conditionalInAssignmentLoop;
   protected HashMap assignmentsInBranch;
   protected HashMap conditionalAssignmentBranch;
   protected ArrayList onBothSides;
   protected ArrayList toggleStatements;
   protected ArrayList incDecStatements;
   protected HashMap useOutsideAssign;
   protected ArrayList assignBeforeUse;
   protected HashMap condOutsideAssign;
   protected ArrayList indirectBothSides;
   protected ArrayList reorganize;
   protected ArrayList transformations;
   protected ArrayList nestedBooleanAssignment;
   protected ArrayList methodAssignments;
   protected ArrayList otherAssignments;
   protected ArrayList assignmentInFor;
   protected ArrayList useForAssignLoopCondition;
   protected ArrayList isArray;
   //condition flags
   protected boolean onlyOneAssignment;
   protected boolean isAssignedInFor;
   protected boolean assignedInLoop;
   protected boolean usedInAssignLoop;
   protected boolean conditionalUseInAssign;
   protected boolean indirectUse;
   protected boolean noDirectUsage;
   protected boolean conditionForAssignBranch;
   protected boolean onBothSidesOfAssign;
   protected boolean toggleStatement;
   protected boolean incDecStatement;
   protected boolean usedOutsideAssign;
   protected boolean assignedBeforeUse;
   protected boolean loopUseNoAssign;
   protected boolean assignCondLoop;
   protected boolean assignmentLoopConditionUse;
   protected boolean indirectBS;
   protected boolean isReorganize;
   protected boolean isArrayVariable;
   protected boolean isTransform;
   protected boolean nestedBooleanAssign;
   protected boolean methodAssignment;
   protected boolean otherAssign;
   //result Flags
   protected boolean isFixedValue;
   protected boolean isOrganizer;
   protected boolean isStepper;
   protected boolean isMostRecentHolder;
   protected boolean isGatherer;
   protected boolean isMostWantedHolder;
   protected boolean isOneWayFlag;
   protected boolean isTransformation;
   protected boolean isFollower;
   protected boolean isTemporary;
   //Explanatory strings
   private String offendingStatement;
   private String reason;

   /**
    * Constructor for RoleChecker
    *
    * @param inputMap
    *           LinkedHashMap containing analysed statements
    * @param var
    *           String being variable for which role is to be checked
    * @param meths
    *           ArrayList of method names in source code
    */
   public RoleChecker(LinkedHashMap inputMap, String var,
         ArrayList meths) {
      variable = var;
      analysedMap = inputMap;
      methods = meths;
      conditionChecker = new ConditionChecker(analysedMap, variable,
         methods);
      resultStringer = new ResultStringer(variable);
      initialiseConditionFlags();
      getConditions();
      setConditionFlags();
   }

   /**
    * Abstract method to be completed in each Class extending
    *  RoleChecker
    *
    * @return ArrayList of results from role checking
    */
   public abstract ArrayList checkRole();

   /*
    * Initialise condition flags
    */
   protected void initialiseConditionFlags() {
      //variable assigned in loop
      assignedInLoop = false;
      //variable used in assignment loop
      usedInAssignLoop = false;
      //variable used conditionally in assignment loop
      conditionalUseInAssign = false;
      //variable used indirectly for assignment loop condition
      indirectUse = false;
      //variable only assigned in "for" loop statement
      isAssignedInFor = false;
      //no direct usage of variable
      noDirectUsage = false;
      //assignment in branch for which variable is condition
      conditionForAssignBranch = false;
      //variable appears on both sides of assignment statement
      onBothSidesOfAssign = false;
      //variable toggled within loop
```

```
        toggleStatement = false;
        //variable incremented etc within loop
        incDecStatement = false;
        //variable is not used outside of loop in which it is assigned
        usedOutsideAssign = false;
        //variable is not assigned before it is used in assignment loop
        assignedBeforeUse = false;
        //variable is used as condition for assignment loop
        assignCondLoop = false;
        //variable used conditionally for loop outside of assignment loop
        loopUseNoAssign = false;
        //variable not used indirectly on both sides of assignment
        //statement
        indirectBS = false;
        //variable is reorganize array
        isReorganize = false;
        //variable is Array
        isArrayVariable = false;
        //variable is not used in transformation statement
        isTransform = false;
        //variable is not assigned twice in loop, one assignment being in
        // nested loop
        nestedBooleanAssign = false;
        //variable is not assigned with output from method
        methodAssignment = false;
        otherAssign = false;
    }

    /*
     * Get conditions under which variables are assigned and used in
     * program
     */
    protected void getConditions() {
        onlyOneAssignment = conditionChecker.onlyOneAssignmentStatement();
        //find assignments in loop
        assignmentsInLoop = conditionChecker.foundIn("assignment",
                "loop");
        //find direct usage in loop
        usagesInLoop = conditionChecker.foundIn("usage", "loop");
        //find conditional usage in loop
        conditionalsInLoop = conditionChecker.foundIn("conditional",
                "loop");
        //find direct usage in assignment loop
        usageInAssignmentLoop = conditionChecker.foundInAssignmentLoop(
                assignmentsInLoop, usagesInLoop);
        //find conditional usage in assignment loop
        conditionalInAssignmentLoop =
                conditionChecker.foundInAssignmentLoop(assignmentsInLoop,
                conditionalsInLoop);
        //find assignment in branch
        assignmentsInBranch = conditionChecker.foundIn("assignment",
                "branch");
        //find incidents where assignment is dependent on condition of
        //variable
        conditionalAssignmentBranch =
                conditionChecker.conditionForAssignmentBranch(
                assignmentsInBranch, conditionalsInLoop);
        //if not incidents where assignment dependent on condition of
        //variable
        // check assignment statement for leading branch condition
        if (conditionalAssignmentBranch.size() < 1)
            conditionalAssignmentBranch =
                    conditionChecker.conditionAssignStatement();
        //find assignment statements where variable appears on both sides
        //of "="
        onBothSides = conditionChecker.statementTypeCheck((HashMap)
                analysedMap.get("assignment"), "bothSides");
        //find assignment statements in loop where variable is toggled
        toggleStatements = conditionChecker.statementTypeCheck(
                assignmentsInLoop, "toggle");
        //find assignment statements in loop where variable is
        //incremented, decremented, etc
        incDecStatements = conditionChecker.statementTypeCheck(
              assignmentsInLoop, "incDec");
        //find use outside of assignment loop
        useOutsideAssign = conditionChecker.outsideAssignLoop(
              usageInAssignmentLoop, "usage");
        //find whether variable used before assigned in loop
        assignBeforeUse = conditionChecker
              .assignBeforeUse(usageInAssignmentLoop);
        //find conditional use for loop statements loops outside of
        //assignment loop
        condOutsideAssign = conditionChecker.outsideAssignLoop(
              conditionalInAssignmentLoop, "conditional");
        //check if variable only assigned value as part of "for" loop
        //statement
        assignmentInFor = conditionChecker.assignmentInFor("usage");
        //check if variable only assigned value as part of "for" loop
        //statement
        if (assignmentInFor.size() < 1)
           assignmentInFor =
                 conditionChecker.assignmentInFor("conditional");
        //find assignment statements in loop where variable appears
        //indirectly on both sides of "=")
        indirectBothSides = conditionChecker.statementTypeCheck(
              assignmentsInLoop, "indirect");
        reorganize = conditionChecker.reorganize();
        //check if variable is array
        isArray = conditionChecker.arrayCheck();
        //check if variable is only assigned by transformation statement
        transformations = conditionChecker.transform(assignmentsInLoop);
        nestedBooleanAssignment =
              conditionChecker.nestedBooleanAssign( assignmentsInLoop);
        //check is variable is assigned with output from method
        methodAssignments =
              conditionChecker.assignedWithMethod(assignmentsInLoop);
        //check if variable is assigned with value other than of other
        //variable
        otherAssignments = conditionChecker.otherAssignment(
              assignmentsInLoop);
     }

     /*
      * Set condition flags depending on conditions
      */
     protected void setConditionFlags() {
        if (assignmentsInLoop.size() > 0)
           assignedInLoop = true;
        if (usageInAssignmentLoop.size() > 0)
           usedInAssignLoop = true;
        if (conditionalInAssignmentLoop.size() > 0)
           conditionalUseInAssign = true;
        if (conditionalAssignmentBranch.size() > 0)
           conditionForAssignBranch = true;
        if (onBothSides.size() > 0)
           onBothSidesOfAssign = true;
        if (toggleStatements.size() > 0)
           toggleStatement = true;
        if (incDecStatements.size() > 0)
           incDecStatement = true;
        if (conditionalUseInAssign) {
           useForAssignLoopCondition = conditionChecker
                 .useForLoopCondition(conditionalInAssignmentLoop);
           assignCondLoop = false;
           if (useForAssignLoopCondition.size() > 0)
              assignCondLoop = true;
           if (!assignCondLoop) {
              indirectUse = conditionChecker.indirectUse(
                    conditionalInAssignmentLoop);
           }
        }
        if (assignmentInFor.size() > 0)
           isAssignedInFor = true;
        if (conditionChecker.noDirectUsage() && !onBothSidesOfAssign)
           noDirectUsage = true;
        if (useOutsideAssign.size() > 0)
           usedOutsideAssign = true;
        if (assignBeforeUse.size() > 0)
           assignedBeforeUse = true;
        if (condOutsideAssign.size() > 0)
           loopUseNoAssign = true;
           assignmentLoopConditionUse = (indirectUse || assignCondLoop);
        if (indirectBothSides.size() > 0)
           indirectBS = true;
        if (reorganize.size() > 0)
           isReorganize = true;
        if (transformations.size() > 0)
           isTransform = true;
        if (nestedBooleanAssignment.size() > 0)
           nestedBooleanAssign = true;
        if (methodAssignments.size() > 0)
           methodAssignment = true;
        if (otherAssignments.size() > 0)
           otherAssign = true;
        if (isArray.size() > 0)
           isArrayVariable = true;
     }

     /*
      * Return if variable is fixed value
      */
     protected boolean isFixedValue() {
        fixedValue();
        return isFixedValue;
     }

     /*
      * Return if variable is organizer
      */
     protected boolean isOrganizer() {
        organizer();
        return isOrganizer;
     }

     /*
      * Return if variable is stepper
      */
     protected boolean isStepper() {
        stepper();
        return isStepper;
     }

     /*
      * Return if variable is most recent holder
      */
     protected boolean isMostRecentHolder() {
        mostRecentHolder();
        return isMostRecentHolder;
     }

     /*
      * Return if variable is gatherer
      */
     protected boolean isGatherer() {
        gatherer();
        return isGatherer;
     }

     /*
      * Return if variable is most wanted holder
```

```
     */
    protected boolean isMostWantedHolder() {
       mostWantedHolder();
       return isMostWantedHolder;
    }

    /*
     * Return if variable is one way flag
     */
    protected boolean isOneWayFlag() {
       oneWayFlag();
       return isOneWayFlag;
    }

    /*
     * Return if variable is transformation
     */
    protected boolean isTransformation() {
       transformation();
       return isTransformation;
    }

    /*
     * Return if variable is follower
     */
    protected boolean isFollower() {
       follower();
       return isFollower;
    }

    /*
     * Return if variable is temporary
     */
    protected boolean isTemporary() {
       temporary();
       return isTemporary;
    }

    /*
     * Test whether variable is fixed value
     */
    protected ArrayList fixedValue() {
       ArrayList returnArray = new ArrayList();
       offendingStatement = "";
       reason = "";
       isFixedValue = true;
       if (isArrayVariable && isReorganize) {
          isFixedValue = false;
          setMessage(1);
       } else if (incDecStatement) {
          isFixedValue = false;
          setMessage(2);
       } else if (toggleStatement) {
          isFixedValue = false;
          setMessage(3);
       } else if (onBothSidesOfAssign) {
          isFixedValue = false;
          setMessage(4);
       } else if (indirectBS) {
          isFixedValue = false;
          setMessage(5);
       } else if (isTransform && usedInAssignLoop) {
          isFixedValue = false;
          setMessage(6);
       } else if (isAssignedInFor) {
          isFixedValue = false;
          setMessage(7);
       } else if (nestedBooleanAssign) {
          isFixedValue = false;
          setMessage(8);
       } else if (conditionForAssignBranch) {
          isFixedValue = false;
          setMessage(9);
       } else if (!methodAssignment && !otherAssign && usedInAssignLoop)
       {
          isFixedValue = false;
          setMessage(10);
       } else if (usedInAssignLoop && !isArrayVariable) {
          isFixedValue = false;
          Set keySet = usageInAssignmentLoop.keySet();
          Iterator it = keySet.iterator();
          offendingStatement = (String) it.next();
          reason = "used in loop in which assigned";
       } else if (conditionalUseInAssign && !assignmentLoopConditionUse
               && !isArrayVariable) {
          isFixedValue = false;
          Set keySet = conditionalInAssignmentLoop.keySet();
          Iterator it = keySet.iterator();
          offendingStatement = (String) it.next();
          reason = "used as condition in loop in which assigned";
       } else if (noDirectUsage && assignCondLoop && !loopUseNoAssign) {
          isFixedValue = false;
          setMessage(11);
       }
       returnArray.add(offendingStatement);
       returnArray.add(reason);
       returnArray.add(0, new Boolean(isFixedValue));
       return returnArray;
    }

    /*
     * Test whether variable is organizer
     */
    protected ArrayList organizer() {
       ArrayList returnArray = new ArrayList();
       offendingStatement = "";
       reason = "";
       isOrganizer = true;
       if (!isArrayVariable) {
          isOrganizer = false;
          offendingStatement = r
                  esultStringer.offendingStatement("organizer");
          reason = "does not appear to be array or is not used directly"+
                  " as array";
       } else if (!isReorganize) {
          isOrganizer = false;
          offendingStatement =
                  resultStringer.offendingStatement("organizer");
          reason = "no organizer type statements found";
       } else if (!onBothSidesOfAssign) {
          isOrganizer = false;
          offendingStatement =
                  resultStringer.offendingStatement("organizer");
          reason = "no statements found where variable on both sides of"+
                  " assignment";
       }
       returnArray.add(offendingStatement);
       returnArray.add(reason);
       returnArray.add(0, new Boolean(isOrganizer));
       return returnArray;
    }

    /*
     * Test whether variable is stepper
     */
    protected ArrayList stepper() {
       ArrayList returnArray = new ArrayList();
       offendingStatement = "";
       reason = "";
       isStepper = true;
```

```
        if (!incDecStatement && !isAssignedInFor && !toggleStatement
                && !nestedBooleanAssign) {
            isStepper = false;
            offendingStatement =
                    resultStringer.offendingStatement("stepper");
            reason = "no stepper type assign statements found";
        }
        returnArray.add(offendingStatement);
        returnArray.add(reason);
        returnArray.add(0, new Boolean(isStepper));
        return returnArray;
    }

    /*
     * Test whether variable is most recent holder
     */
    protected ArrayList mostRecentHolder() {
        ArrayList returnArray = new ArrayList();
        offendingStatement = "";
        reason = "";
        isMostRecentHolder = true;
        if (isAssignedInFor) {
            isMostRecentHolder = false;
            setMessage(7);
        } else if (!assignedInLoop) {
            isMostRecentHolder = false;
            setMessage(12);
        } else if (isArrayVariable && isReorganize) {
            isMostRecentHolder = false;
            setMessage(1);
        } else if (isArrayVariable && isFixedValue()) {
            isMostRecentHolder = false;
            Set keySet = assignmentsInLoop.keySet();
            Iterator it = keySet.iterator();
            offendingStatement = (String) it.next();
            reason = "is array filled in loop";
        } else if (incDecStatement) {
            isMostRecentHolder = false;
            setMessage(2);
        } else if (toggleStatement) {
            isMostRecentHolder = false;
            setMessage(3);
        } else if (onBothSidesOfAssign) {
            isMostRecentHolder = false;
            setMessage(4);
        } else if (indirectBS) {
            isMostRecentHolder = false;
            setMessage(5);
        } else if (isTransform) {
            isMostRecentHolder = false;
            setMessage(6);
        } else if (conditionForAssignBranch) {
            isMostRecentHolder = false;
            setMessage(9);
        } else if (nestedBooleanAssign) {
            isMostRecentHolder = false;
            setMessage(8);
        } else if (noDirectUsage && assignCondLoop && !loopUseNoAssign) {
            isMostRecentHolder = false;
            setMessage(11);
        } else if (!methodAssignment && !otherAssign) {
            isMostRecentHolder = false;
            setMessage(10);
        } else if (!usedInAssignLoop
                && (!conditionalUseInAssign || assignmentLoopConditionUse))
        {
            isMostRecentHolder = false;
            Set keySet = assignmentsInLoop.keySet();
            Iterator it = keySet.iterator();
            setMessage(13);
        }
        returnArray.add(offendingStatement);
        returnArray.add(reason);
        returnArray.add(0, new Boolean(isMostRecentHolder));
        return returnArray;
    }

    /*
     * Test whether variable is gatherer
     */
    protected ArrayList gatherer() {
        ArrayList returnArray = new ArrayList();
        offendingStatement = "";
        reason = "";
        isGatherer = true;
        if (isAssignedInFor) {
            isGatherer = false;
            setMessage(7);
        } else if (!assignedInLoop) {
            isGatherer = false;
            setMessage(12);
        } else if (isArrayVariable && isReorganize) {
            isGatherer = false;
            setMessage(1);
        } else if (incDecStatement) {
            isGatherer = false;
            setMessage(2);
        } else if (toggleStatement) {
            isGatherer = false;
            setMessage(3);
        } else if (nestedBooleanAssign) {
            isGatherer = false;
            setMessage(8);
        } else if (conditionForAssignBranch) {
            isGatherer = false;
            setMessage(9);
        } else if (isTransform && !onBothSidesOfAssign && !indirectBS) {
            isGatherer = false;
            setMessage(6);
        } else if (noDirectUsage && assignCondLoop && !loopUseNoAssign
                && !onBothSidesOfAssign && !indirectBS) {
            isGatherer = false;
            setMessage(11);
        } else if (!methodAssignment && !otherAssign
                && !onBothSidesOfAssign
                && !indirectBS) {
            isGatherer = false;
            setMessage(10);
        } else if (!onBothSidesOfAssign && !indirectBS) {
            isGatherer = false;
            Set keySet = assignmentsInLoop.keySet();
            Iterator it = keySet.iterator();
            offendingStatement = (String) it.next();
            reason = "not found directly, or indirectly on both sides of" +
                    " assignment statement";
        }
        returnArray.add(offendingStatement);
        returnArray.add(reason);
        returnArray.add(0, new Boolean(isGatherer));
        return returnArray;
    }

    /*
     * Test whether variable is most wanted holder
     */
    protected ArrayList mostWantedHolder() {
        ArrayList returnArray = new ArrayList();
        offendingStatement = "";
        reason = "";
        isMostWantedHolder = true;
```

```
        if (isAssignedInFor) {
            isMostWantedHolder = false;
            setMessage(7);
        } else if (isArrayVariable && isReorganize) {
            isMostWantedHolder = false;
            setMessage(1);
        } else if (incDecStatement) {
            isMostWantedHolder = false;
            setMessage(2);
        } else if (toggleStatement) {
            isMostWantedHolder = false;
            setMessage(3);
        } else if (onBothSidesOfAssign) {
            isMostWantedHolder = false;
            setMessage(4);
        } else if (indirectBS) {
            isMostWantedHolder = false;
            setMessage(5);
        } else if (nestedBooleanAssign) {
            isMostWantedHolder = false;
            setMessage(8);
        } else if (!conditionForAssignBranch) {
            isMostWantedHolder = false;
            HashMap assignments = (HashMap) analysedMap.get("assignment");
            Set keySet = assignments.keySet();
            Iterator it = keySet.iterator();
            offendingStatement = (String) it.next();
            reason = "not assigned in branch, or is not condition for" +
                    " branch in which it is assigned";
        }
        returnArray.add(offendingStatement);
        returnArray.add(reason);
        returnArray.add(0, new Boolean(isMostWantedHolder));
        return returnArray;
    }

    /*
     * Test whether variable is one way flag
     */
    protected ArrayList oneWayFlag() {
        ArrayList returnArray = new ArrayList();
        offendingStatement = "";
        reason = "";
        isOneWayFlag = true;
        if (isAssignedInFor) {
            isOneWayFlag = false;
            setMessage(7);
        } else if (!assignedInLoop) {
            isOneWayFlag = false;
            setMessage(12);
        } else if (isArrayVariable && isReorganize) {
            isOneWayFlag = false;
            setMessage(1);
        } else if (isArrayVariable && !isReorganize) {
            isOneWayFlag = false;
            Set keySet = assignmentsInLoop.keySet();
            Iterator it = keySet.iterator();
            offendingStatement = (String) it.next();
            reason = "does not appear to be used as one way flag";
        } else if (incDecStatement) {
            isOneWayFlag = false;
            setMessage(2);
        } else if (toggleStatement) {
            isOneWayFlag = false;
            setMessage(3);
        } else if (onBothSidesOfAssign) {
            isOneWayFlag = false;
            setMessage(4);
        } else if (indirectBS) {
            isOneWayFlag = false;
            setMessage(5);
        } else if (nestedBooleanAssign) {
            isOneWayFlag = false;
            setMessage(8);
        } else if (conditionForAssignBranch) {
            isOneWayFlag = false;
            setMessage(9);
        } else if (isTransform && !assignCondLoop) {
            isOneWayFlag = false;
            setMessage(6);
        } else if (!methodAssignment && !otherAssign) {
            isOneWayFlag = false;
            setMessage(10);
        } else if (!noDirectUsage) {
            isOneWayFlag = false;
            HashMap usages = (HashMap) analysedMap.get("usage");
            Set keySet = usages.keySet();
            Iterator it = keySet.iterator();
            offendingStatement = (String) it.next();
            reason = "direct use of variable";
        } else if (!assignCondLoop) {
            isOneWayFlag = false;
            offendingStatement =
                    resultStringer.offendingStatement("one way flag");
            reason = "not used directly for assign loop condition";
        } else if (loopUseNoAssign) {
            isOneWayFlag = false;
            offendingStatement =
                    resultStringer.offendingStatement("one way flag");
            reason = "used for loop condition outside of loop in which it"+
                    " is assigned";
        }
        returnArray.add(offendingStatement);
        returnArray.add(reason);
        returnArray.add(0, new Boolean(isOneWayFlag));
        return returnArray;
    }

    /*
     * Test whether variable is transformation
     */
    protected ArrayList transformation() {
        ArrayList returnArray = new ArrayList();
        offendingStatement = "";
        reason = "";
        isTransformation = true;
        if (isAssignedInFor) {
            isTransformation = false;
            setMessage(7);
        } else if (!assignedInLoop) {
            isTransformation = false;
            setMessage(12);
        } else if (isArrayVariable && isReorganize) {
            isTransformation = false;
            setMessage(1);
        } else if (incDecStatement) {
            isTransformation = false;
            setMessage(2);
        } else if (toggleStatement) {
            isTransformation = false;
            setMessage(3);
        } else if (onBothSidesOfAssign) {
            isTransformation = false;
            setMessage(4);
        } else if (indirectBS) {
            isTransformation = false;
            setMessage(5);
        } else if (nestedBooleanAssign) {
            isTransformation = false;
            setMessage(8);
        } else if (conditionForAssignBranch) {
            isTransformation = false;
            setMessage(9);
        } else if (!methodAssignment && !otherAssign && !isTransform) {
            isTransformation = false;
            setMessage(10);
        } else if (methodAssignment) {
            isTransformation = false;
            setMessage(14);
        } else if (!isTransform) {
            isTransformation = false;
            Set keySet = assignmentsInLoop.keySet();
            Iterator it = keySet.iterator();
            offendingStatement = (String) it.next();
            reason = "assignment statement contains no operator" +
                    "characters";
        } else if (noDirectUsage && assignCondLoop && !loopUseNoAssign) {
            isTransformation = false;
            setMessage(11);
        }
        returnArray.add(offendingStatement);
        returnArray.add(reason);
        returnArray.add(0, new Boolean(isTransformation));
        return returnArray;
    }

    /*
     * Test whether variable is follower
     */
    protected ArrayList follower() {
        ArrayList returnArray = new ArrayList();
        offendingStatement = "";
        reason = "";
        isFollower = true;
        if (isAssignedInFor) {
            isFollower = false;
            setMessage(7);
        } else if (!assignedInLoop) {
            isFollower = false;
            setMessage(12);
        } else if (isArrayVariable && isReorganize) {
            isFollower = false;
            setMessage(1);
        } else if (incDecStatement) {
            isFollower = false;
            setMessage(2);
        } else if (toggleStatement) {
            isFollower = false;
            setMessage(3);
        } else if (onBothSidesOfAssign) {
            isFollower = false;
            setMessage(4);
        } else if (indirectBS) {
            isFollower = false;
            setMessage(5);
        } else if (nestedBooleanAssign) {
            isFollower = false;
            setMessage(8);
        } else if (conditionForAssignBranch) {
            isFollower = false;
            setMessage(9);
        } else if (isTransform) {
            isFollower = false;
            setMessage(6);
        } else if (methodAssignment) {
            isFollower = false;
            setMessage(14);
        } else if (noDirectUsage && assignCondLoop && !loopUseNoAssign) {
            isFollower = false;
            setMessage(11);
        } else if (otherAssign) {
            isFollower = false;
            setMessage(15);
        } else if (assignedBeforeUse && !usedOutsideAssign) {
            isFollower = false;
            offendingStatement = (String) assignBeforeUse.get(0);
            reason = "assigned before use in loop and not used outside of"+
                    " assign loop";
        } else if (!usedInAssignLoop) {
            isFollower = false;
            setMessage(13);
        }
        returnArray.add(offendingStatement);
        returnArray.add(reason);
        returnArray.add(0, new Boolean(isFollower));
        return returnArray;
    }

    /*
     * Test whether variable is temporary
     */
    protected ArrayList temporary() {
        ArrayList returnArray = new ArrayList();
        offendingStatement = "";
        reason = "";
        isTemporary = true;
        if (isAssignedInFor) {
            isTemporary = false;
            setMessage(7);
        } else if (!assignedInLoop) {
            isTemporary = false;
            setMessage(12);
        } else if (isArrayVariable && isReorganize) {
            isTemporary = false;
            setMessage(1);
        } else if (incDecStatement) {
            isTemporary = false;
            setMessage(2);
        } else if (toggleStatement) {
            isTemporary = false;
            setMessage(3);
        } else if (onBothSidesOfAssign) {
            isTemporary = false;
            setMessage(4);
        } else if (indirectBS) {
            isTemporary = false;
            setMessage(5);
        } else if (nestedBooleanAssign) {
            isTemporary = false;
            setMessage(8);
        } else if (conditionForAssignBranch) {
            isTemporary = false;
            setMessage(9);
        } else if (isTransform) {
            isTemporary = false;
            setMessage(6);
        } else if (methodAssignment) {
            isTemporary = false;
            setMessage(14);
        } else if (noDirectUsage && assignCondLoop && !loopUseNoAssign) {
            isTemporary = false;
            setMessage(11);
        } else if (otherAssign) {
            isTemporary = false;
            setMessage(15);
        } else if (!assignedBeforeUse) {
            isTemporary = false;
            Set keySet = usageInAssignmentLoop.keySet();
            Iterator it = keySet.iterator();
            offendingStatement = (String) it.next();
            reason = "not assigned in loop before use";
        } else if (usedOutsideAssign) {
            isTemporary = false;
            Set keySet = useOutsideAssign.keySet();
            Iterator it = keySet.iterator();
            offendingStatement = (String) it.next();
            reason = "used outside of loop in which it is assigned";
        } else if (!usedInAssignLoop) {
            isTemporary = false;
            setMessage(13);
        }
        returnArray.add(offendingStatement);
        returnArray.add(reason);
        returnArray.add(0, new Boolean(isTemporary));
        return returnArray;
    }

    /*
     * Set message applicable condition detected
     */
    private void setMessage(int reasonInt) {
        Iterator it;
        Set keySet;
        switch (reasonInt) {
        case 1:
            offendingStatement = (String) reorganize.get(0);
            reason = "appears to be organizer statement";
            break;
        case 2:
            offendingStatement = (String) incDecStatements.get(0);
            reason = "incremented/decremented within loop";
            break;
        case 3:
            offendingStatement = (String) toggleStatements.get(0);
            reason = "toggled within loop";
            break;
        case 4:
            offendingStatement = (String) onBothSides.get(0);
            reason = "appears on both sides of assignment";
            break;
        case 5:
            offendingStatement = (String) indirectBothSides.get(0);
            reason = "indirectly appears on both sides of assignment";
            break;
        case 6:
            offendingStatement = (String) transformations.get(0);
            reason = "assigned in loop with combination of other " +
                "variables, operators and constants";
            break;
        case 7:
            offendingStatement = (String) assignmentInFor.get(0);
            reason = "assigned in for loop statement";
            break;
        case 8:
            String statementType = (String) nestedBooleanAssignment.get(1);
            if (statementType.equals("nested")) {
                offendingStatement =
                        (String) nestedBooleanAssignment.get(0);
            } else {
                offendingStatement =
                        (String) nestedBooleanAssignment.get(2);
            }
            reason = "appears to be indirectly toggled within loop";
            break;
        case 9:
            keySet = conditionalAssignmentBranch.keySet();
            it = keySet.iterator();
            offendingStatement = (String) it.next();
            reason = "used as condition for branch in which assigned";
            break;
        case 10:
            keySet = assignmentsInLoop.keySet();
            it = keySet.iterator();
            offendingStatement = (String) it.next();
            reason = "always assigned in loop with value of other " +
                    variable";
            break;
        case 11:
            offendingStatement = (String) useForAssignLoopCondition.get(0);
            reason = "condition for loop in which assigned and limited " +
                    "use outside of loop";
            break;
        case 12:
            HashMap assignments = (HashMap) analysedMap.get("assignment");
            keySet = assignments.keySet();
            it = keySet.iterator();
            offendingStatement = (String) it.next();
            reason = "not assigned in loop";
            break;
        case 13:
            HashMap usages = (HashMap) analysedMap.get("usage");
            keySet = usages.keySet();
            if (!keySet.isEmpty()) {
                it = keySet.iterator();
                offendingStatement = (String) it.next();
            } else {
                HashMap conditionals =
                        (HashMap) analysedMap.get("conditional");
                keySet = conditionals.keySet();
                it = keySet.iterator();
                offendingStatement = (String) it.next();
            }
            reason = "not used in loop in which assigned";
            break;
        case 14:
            offendingStatement = (String) methodAssignments.get(0);
            reason = "assigned with output from call to method";
            break;
        case 15:
            offendingStatement = (String) otherAssignments.get(0);
            reason = "assigned directly with value, instantiation of " +
                    "object, or call to method of other object";
            break;
        }
    }
}
```

A5.3.3 ResultStringer.java

```
/*
 * Created on 12-Jul-2005
 */
package main.rules;

/**
 * @author cbishop
 */
public class ResultStringer {

   private String variable;

   /**
    * Constructor for ResultStringer
    *
    * @param var
    *            Variable name
    */
   public ResultStringer(String var) {
      variable = var;
   }

   /**
    * Return statement to highlight where no offending statement has
    * been explicitly detected
    *
    * @param role
    *            String being role declared for variable in source code
    * @return String
    */
   public String offendingStatement(String role) {
      return variable + "%%" + role;
   }

   /**
    * Return string stating that variable appears to be fixed value
    *
    * @return String
    */
   public String fixedValue() {
      return "variable " + variable + " appears to be 'fixed value'";
   }

   /**
    * Return string stating that variable appears to be organizer
    *
    * @return String
    */
   public String organizer() {
      return "variable " + variable + " appears to be 'organizer'";
   }

   /**
    * Return string stating that variable appears to be stepper
    *
    * @return String
    */
   public String stepper() {
      return "variable " + variable + " appears to be 'stepper'";
   }

   /**
    * Return string stating that variable appears to be most recent
    * holder
    *
    * @return String
```

```
     */
    public String mostRecentHolder() {
       return "variable " + variable + " appears to be " +
             "'most recent holder'";
    }

    /**
     * Return string stating that variable appears to be gatherer
     *
     * @return String
     */
    public String gatherer() {
       return "variable " + variable + " appears to be 'gatherer'";
    }

    /**
     * Return string stating that variable appears to be most wanted
     * holder
     *
     * @return String
     */
    public String mostWantedHolder() {
       return "variable " + variable + " appears to be" +
             " 'most wanted holder'";
    }

    /**
     * Return string stating that variable appears to be one way flag
     *
     * @return String
     */
    public String oneWayFlag() {
       return "variable " + variable + " appears to be 'one way flag'";
    }

    /**
     * Return string stating that variable appears to be transformation
     *
     * @return String
     */
    public String transformation() {
       return "variable " + variable + " appears to be 'transformation'";
    }

    /**
     * Return string stating that variable appears to be follower
     *
     * @return String
     */
    public String follower() {
       return "variable " + variable + " appears to be 'follower'";
    }

    /**
     * Return string stating that variable appears to be temporary
     *
     * @return String
     */
    public String temporary() {
       return "variable " + variable + " appears to be 'temporary'";
    }
```

```
/**
     * Return string stating that variable appears to have other role
     *
     * @return String
     */
    public String other() {
       return "not clear which role variable " + variable + " is +
             " playing";
    }
}
```

A5.3.4 Fixed Value.java

```
/*
 * Created on 20-Jun-2005
 */
package main.rules;

import java.util.ArrayList;
import java.util.LinkedHashMap;

/**
 * @author cbishop
 */
public class FixedValue extends RoleChecker {

   /**
    * Constructor for FixedValue
    *
    * @param analysedMap
    *            LinkedHashMap of analysed statements
    * @param variable
    *            String being variable name
    * @param methods
    *            ArrayList of method names
    */
   public FixedValue(LinkedHashMap analysedMap, String variable,
         ArrayList methods) {
      super(analysedMap, variable, methods);
   }

   /**
    * Return list of result from role check
    *
    * @return ArrayList
    */
   public ArrayList checkRole() {
      ArrayList results = fixedValue();
      if (!isFixedValue()) {
         if (isOrganizer()) {
            results.add(resultStringer.organizer());
         } else if (isStepper()) {
            results.add(resultStringer.stepper());
         } else if (isMostRecentHolder()) {
            results.add(resultStringer.mostRecentHolder());
         } else if (isGatherer()) {
            results.add(resultStringer.gatherer());
         } else if (isMostWantedHolder()) {
            results.add(resultStringer.mostWantedHolder());
         } else if (isOneWayFlag()) {
            results.add(resultStringer.oneWayFlag());
         } else if (isTransformation()) {
            results.add(resultStringer.transformation());
         } else if (isFollower()) {
            results.add(resultStringer.follower());
         } else if (isTemporary()) {
            results.add(resultStringer.temporary());
         } else {
            results.add(resultStringer.other());
         }
      }
      return results;
   }
}

/*
 * debug rules for fixed value
 * if (onBothSidesOfAssign || indirectBS) { result =
 * false; results.add(debugStringer.bothSides(onBothSides, true));
 * } else if (incDecStatement) { result = false;
 * results.add(debugStringer.incDec(incDecStatements));
 * } else if(toggleStatement) { result = false;
 * results.add(debugStringer.toggle(toggleStatements));
 * } else if(isAssignedInFor) { result = false;
 * results.add(debugStringer.assignedInFor(true));
 * } else if(conditionForAssignBranch) { result = false;
 * results.add(debugStringer.conditionForAssignBranch(
 * conditionalAssignmentBranch));
 * } else if(usedInAssignLoop && !isArray) { result = false;
 * results.add(debugStringer.showFoundInAssignmentLoop(
 * usageInAssignmentLoop, "usage"));
 * } else if(conditionalUseInAssign && !assignmentLoopConditionUse
 * && !isArray) { result = false;
 * results.add(debugStringer.showFoundInAssignmentLoop(
 * conditionalInAssignmentLoop, "conditional"));
 * } else if(noDirectUsage && useForAssignLoopCondition &&
 * !loopUseNoAssign) { result = false; results.add(
 * debugStringer.noDirectUse()); }
 */
```

A5.3.5 Organizer.java

```
/*
 * Created on 20-Jun-2005
 */
package main.rules;

import java.util.ArrayList;
import java.util.LinkedHashMap;

/**
 * @author cbishop
 */
public class Organizer extends RoleChecker {

   /**
    * Constructor for Organizer
    *
    * @param analysedMap
    *            LinkedHashMap of analysed statements
    * @param variable
    *            String being variable name
    * @param methods
    *            ArrayList of method names
    */
   public Organizer(LinkedHashMap analysedMap, String variable,
         ArrayList methods) {
      super(analysedMap, variable, methods);
   }

   /**
    * Return list of result from role check
    *
    * @return ArrayList
    */
   public ArrayList checkRole() {
      ArrayList results = organizer();
      if (!isOrganizer()) {
         if (isFixedValue()) {
            results.add(resultStringer.fixedValue());
         } else if (isStepper()) {
            results.add(resultStringer.stepper());
         } else if (isMostRecentHolder()) {
            results.add(resultStringer.mostRecentHolder());
         } else if (isGatherer()) {
            results.add(resultStringer.gatherer());
         } else if (isMostWantedHolder()) {
            results.add(resultStringer.mostWantedHolder());
         } else if (isOneWayFlag()) {
            results.add(resultStringer.oneWayFlag());
         } else if (isTransformation()) {
            results.add(resultStringer.transformation());
         } else if (isFollower()) {
            results.add(resultStringer.follower());
         } else if (isTemporary()) {
            results.add(resultStringer.temporary());
         } else {
            results.add(resultStringer.other());
         }
      }
      return results;
   }
}

/*
 * debug rules for organizer
 *
 * if (!isArray) { result = false;
 * results.add(debugStringer.notArray()); }
 * if (!isReorganize) { result = false;
 * results.add(debugStringer.notOrganizer()); }
 * if (!onBothSidesOfAssign) { result = false;
 * results.add(debugStringer.bothSides(onBothSides, false)); }
 * results.add(0, new Boolean(result)); return results;
 */
```

A5.3.6 Stepper.java

```
/*
 * Created on 20-Jun-2005
 */
package main.rules;

import java.util.ArrayList;
import java.util.LinkedHashMap;

/**
 * @author cbishop
 */
public class Stepper extends RoleChecker {
   /**
    * Constructor for Stepper
    *
    * @param analysedMap
    *            LinkedHashMap of analysed statements
    * @param variable
    *            String being variable name
    * @param methods
    *            ArrayList of method names
    */
   public Stepper(LinkedHashMap analysedMap, String variable,
         ArrayList methods) {
      super(analysedMap, variable, methods);
   }

   /**
    * Return list of result from role check
    *
    * @return ArrayList
    */
   public ArrayList checkRole() {
      ArrayList results = stepper();
      if (!isStepper()) {
         if (isFixedValue()) {
            results.add(resultStringer.fixedValue());
         } else if (isOrganizer()) {
            results.add(resultStringer.organizer());
         } else if (isMostRecentHolder()) {
            results.add(resultStringer.mostRecentHolder());
         } else if (isGatherer()) {
            results.add(resultStringer.gatherer());
         } else if (isMostWantedHolder()) {
            results.add(resultStringer.mostWantedHolder());
         } else if (isOneWayFlag()) {
            results.add(resultStringer.oneWayFlag());
         } else if (isTransformation()) {
            results.add(resultStringer.transformation());
         } else if (isFollower()) {
            results.add(resultStringer.follower());
         } else if (isTemporary()) {
            results.add(resultStringer.temporary());
         } else {
            results.add(resultStringer.other());
         }
      }
      return results;
   }
}

/*
 * debug rules for stepper
 *
 * if (!incDecStatement) { results.add(debugStringer.incDec()+
 * " and "); }
 * if !isAssignedInFor) {
```

```
* results.add(debugStringer.assignedInFor(false) + " and "); }
* if (!toggleStatement) { results.add(debugStringer.toggle() +
* " and "); }
* if (!nestedBooleanAssign) {
* results.add(debugStringer.nestedBooleanAssign()); }
* if (!incDecStatement && !isAssignedInFor && !toggleStatement &&
* !nestedBooleanAssign) { result = false; } results.add(0, new
* Boolean(result)); return results;
*/
```

A5.3.7 MostRecentHolder.java

```
/*
 * Created on 20-Jun-2005
 */
package main.rules;

import java.util.ArrayList;
import java.util.LinkedHashMap;

/**
 * @author cbishop
 */
public class MostRecentHolder extends RoleChecker {
   /**
    * Constructor for MostRecentHolder
    *
    * @param analysedMap
    *            LinkedHashMap of analysed statements
    * @param variable
    *            String being variable name
    * @param methods
    *            ArrayList of method names
    */
   public MostRecentHolder(LinkedHashMap analysedMap, String variable,
         ArrayList methods) {
      super(analysedMap, variable, methods);
   }

   /**
    * Return list of result from role check
    *
    * @return ArrayList
    */
   public ArrayList checkRole() {
      ArrayList results = mostRecentHolder();
      if (!isMostRecentHolder()) {
         if (isFixedValue()) {
            results.add(resultStringer.fixedValue());
         } else if (isOrganizer()) {
            results.add(resultStringer.organizer());
         } else if (isStepper()) {
            results.add(resultStringer.stepper());
         } else if (isGatherer()) {
            results.add(resultStringer.gatherer());
         } else if (isMostWantedHolder()) {
            results.add(resultStringer.mostWantedHolder());
         } else if (isOneWayFlag()) {
            results.add(resultStringer.oneWayFlag());
         } else if (isTransformation()) {
            results.add(resultStringer.transformation());
         } else if (isFollower()) {
            results.add(resultStringer.follower());
         } else if (isTemporary()) {
            results.add(resultStringer.temporary());
         } else {
            results.add(resultStringer.other());
         }
      }
      return results;
   }
}

/*
 * debug rules for most recent holder
 *
 * if (!assignedInLoop) { result = false;
 * results.add(debugStringer.notAssignedInLoop());
 * } else if (isArray) {
```

```
 * result = false; results.add(debugStringer.arrayAssignInLoop());
 * } else if (onBothSidesOfAssign || indirectBS) { result = false;
 * results.add(debugStringer.bothSides(onBothSides, true));
 * } else if (incDecStatement) { result = false;
 * results.add(debugStringer.incDec(incDecStatements));
 * } else if (toggleStatement) { result = false;
 * results.add(debugStringer.toggle(toggleStatements));
 * } else if (isTransform) { result = false;
 * results.add(debugStringer.isTransformation(transformations));
 * } else if(conditionForAssignBranch) { result = false;
 * results.add(debugStringer.conditionForAssignBranch(
 * conditionalAssignmentBranch));
 * } else if (nestedBooleanAssign) { result = false;
 * results.add(debugStringer.nestedBooleanAssign(
 * nestedBooleanAssignment));
 * } else if (!methodAssignment && !otherAssign) { result = false;
 * results.add(debugStringer.notMethodAssign());
 * } else if (!usedInAssignLoop &&
 * (!conditionalUseInAssign || assignmentLoopConditionUse)) {
 * result = false;
 * results.add(debugStringer.notUsedInAssign()); }
 * results.add(0, new Boolean(result)); return results; }
 */
```

A5.3.8 Gatherer.java

```
/*
 * Created on 20-Jun-2005
 */
package main.rules;

import java.util.ArrayList;
import java.util.LinkedHashMap;

/**
 * @author cbishop
 */
public class Gatherer extends RoleChecker {

   /**
    * Constructor for Gatherer
    *
    * @param analysedMap
    *            LinkedHashMap of analysed statements
    * @param variable
    *            String being variable name
    * @param methods
    *            ArrayList of method names
    */
   public Gatherer(LinkedHashMap analysedMap, String variable,
         ArrayList methods) {
      super(analysedMap, variable, methods);
   }

   /**
    * Return list of result from role check
    *
    * @return ArrayList
    */
   public ArrayList checkRole() {
      ArrayList results = gatherer();
      if (!isGatherer()) {
         if (isFixedValue()) {
            results.add(resultStringer.fixedValue());
         } else if (isOrganizer()) {
            results.add(resultStringer.organizer());
         } else if (isStepper()) {
            results.add(resultStringer.stepper());
         } else if (isMostRecentHolder()) {
            results.add(resultStringer.mostRecentHolder());
         } else if (isMostWantedHolder()) {
            results.add(resultStringer.mostWantedHolder());
         } else if (isOneWayFlag()) {
            results.add(resultStringer.oneWayFlag());
         } else if (isTransformation()) {
            results.add(resultStringer.transformation());
         } else if (isFollower()) {
            results.add(resultStringer.follower());
         } else if (isTemporary()) {
            results.add(resultStringer.temporary());
         } else {
            results.add(resultStringer.other());
         }
      }
      return results;
   }
}

/*
 * debug rules for gatherer
 *
 * if (!onBothSidesOfAssign && !indirectBS) { result = false;
 * results.add(debugStringer.bothSides(onBothSides, false)); }
 * if (isArray && isReorganize) { result = false;
 * results.add(debugStringer.isOrganizer(reorganize)); }
 * if (toggleStatement) { result = false;
 * results.add(debugStringer.toggle(toggleStatements)); }
 * if (incDecStatement) { result = false;
 * results.add(debugStringer.incDec(incDecStatements)); }
 * results.add(0, new Boolean(result)); return results;
 */
```

A5.3.9 MostWantedHolder.java

```
/*
 * Created on 20-Jun-2005
 */
package main.rules;

import java.util.ArrayList;
import java.util.LinkedHashMap;

/**
 * @author cbishop
 */
public class MostWantedHolder extends RoleChecker {
   /**
    * Constructor for MostWantedHolder
    *
    * @param analysedMap
    *            LinkedHashMap of analysed statements
    * @param variable
    *            String being variable name
    * @param methods
    *            ArrayList of method names
    */
   public MostWantedHolder(LinkedHashMap analysedMap, String variable,
         ArrayList methods) {
      super(analysedMap, variable, methods);
   }

   /**
    * Return list of result from role check
    *
    * @return ArrayList
    */
   public ArrayList checkRole() {
      ArrayList results = mostWantedHolder();
      if (!isMostWantedHolder()) {
         if (isFixedValue()) {
            results.add(resultStringer.fixedValue());
         } else if (isOrganizer()) {
            results.add(resultStringer.organizer());
         } else if (isStepper()) {
            results.add(resultStringer.stepper());
         } else if (isMostRecentHolder()) {
            results.add(resultStringer.mostRecentHolder());
         } else if (isGatherer()) {
            results.add(resultStringer.gatherer());
         } else if (isOneWayFlag()) {
            results.add(resultStringer.oneWayFlag());
         } else if (isTransformation()) {
            results.add(resultStringer.transformation());
         } else if (isFollower()) {
            results.add(resultStringer.follower());
         } else if (isTemporary()) {
            results.add(resultStringer.temporary());
         } else {
            results.add(resultStringer.other());
         }
      }
      return results;
   }
}

/*
 * debug rules for most wanted holder
 *
 * if (!conditionForAssignBranch) { result = false;
 * results.add(debugStringer.conditionForAssignBranch()); } else if
 * (onBothSidesOfAssign || indirectBS) { result = false;
 * results.add(debugStringer.bothSides(onBothSides, true)); } else if
 * (incDecStatement) { result = false;
 * results.add(debugStringer.incDec(incDecStatements)); } else if
 * (toggleStatement) { result = false;
 * results.add(debugStringer.toggle(toggleStatements)); }
 * results.add(0, new Boolean(result)); return results; }
 */
```

A5.3.10 OneWayFlag.java

```java
/*
 * Created on 20-Jun-2005
 */
package main.rules;

import java.util.ArrayList;
import java.util.LinkedHashMap;

/**
 * @author cbishop
 */
public class OneWayFlag extends RoleChecker {

    /**
     * Constructor for OneWayFlag
     *
     * @param analysedMap
     *            LinkedHashMap of analysed statements
     * @param variable
     *            String being variable name
     * @param methods
     *            ArrayList of method names
     */
    public OneWayFlag(LinkedHashMap analysedMap, String variable,
            ArrayList methods) {
        super(analysedMap, variable, methods);
    }

    /**
     * Return list of result from role check
     *
     * @return ArrayList
     */
    public ArrayList checkRole() {
        ArrayList results = oneWayFlag();
        if (!isOneWayFlag()) {
            if (isFixedValue()) {
                results.add(resultStringer.fixedValue());
            } else if (isOrganizer()) {
                results.add(resultStringer.organizer());
            } else if (isStepper()) {
                results.add(resultStringer.stepper());
            } else if (isMostRecentHolder()) {
                results.add(resultStringer.mostRecentHolder());
            } else if (isGatherer()) {
                results.add(resultStringer.gatherer());
            } else if (isMostWantedHolder()) {
                results.add(resultStringer.mostWantedHolder());
            } else if (isTransformation()) {
                results.add(resultStringer.transformation());
            } else if (isFollower()) {
                results.add(resultStringer.follower());
            } else if (isTemporary()) {
                results.add(resultStringer.temporary());
            } else {
                results.add(resultStringer.other());
            }
        }
        return results;
    }
}

/*
 * debug rules for one way flag
 *
 * if (!assignedInLoop) { result = false;
 * results.add(debugStringer.notAssignedInLoop());
 * } else if(!noDirectUsage) {
```

```
 * result = false; results.add(debugStringer.directUse());
 * } else if (onBothSidesOfAssign || indirectBS) { result = false;
 * results.add(debugStringer.bothSides(onBothSides, true));
 * } else if (incDecStatement) { result = false;
 * results.add(debugStringer.incDec(incDecStatements));
 * } else if (toggleStatement) { result = false;
 * results.add(debugStringer.toggle(toggleStatements));
 * } else if (transformation) { result = false;
 * results.add(debugStringer.isTransformation(transformations));
 * } else if(conditionForAssignBranch) { result = false;
 * results.add(debugStringer.conditionForAssignBranch(
 * conditionalAssignmentBranch));
 * } else if (nestedBooleanAssign) { result = false;
 * results.add(debugStringer.nestedBooleanAssign(
 * nestedBooleanAssignment));
 * } else if (!useForAssignLoopCondition) { result = false;
 * results.add(debugStringer.notUsedForAssignLoopCondition());
 * } else if (loopUseNoAssign) { result = false;
 * results.add(debugStringer.loopUseNoAssign()); }
 * results.add(0, new Boolean(result)); return results;
 *
 */
```

A5.3.11 Transformation.java

```
/*
 * Created on 20-Jun-2005
 */
package main.rules;

import java.util.ArrayList;
import java.util.LinkedHashMap;

/**
 * @author cbishop
 */
public class Transformation extends RoleChecker {
    /**
     * Constructor for Transformation
     *
     * @param analysedMap
     *            LinkedHashMap of analysed statements
     * @param variable
     *            String being variable name
     * @param methods
     *            ArrayList of method names
     */
    public Transformation(LinkedHashMap analysedMap, String variable,
          ArrayList methods) {
       super(analysedMap, variable, methods);
    }

    /**
     * Return list of result from role check
     *
     * @return ArrayList
     */
    public ArrayList checkRole() {
       ArrayList results = transformation();
       if (!isTransformation()) {
          if (isFixedValue()) {
             results.add(resultStringer.fixedValue());
          } else if (isOrganizer()) {
             results.add(resultStringer.organizer());
          } else if (isStepper()) {
             results.add(resultStringer.stepper());
          } else if (isMostRecentHolder()) {
             results.add(resultStringer.mostRecentHolder());
          } else if (isGatherer()) {
             results.add(resultStringer.gatherer());
          } else if (isMostWantedHolder()) {
             results.add(resultStringer.mostWantedHolder());
          } else if (isOneWayFlag()) {
             results.add(resultStringer.oneWayFlag());
          } else if (isFollower()) {
             results.add(resultStringer.follower());
          } else if (isTemporary()) {
             results.add(resultStringer.temporary());
          } else {
             results.add(resultStringer.other());
          }
       }
       return results;
    }
}

/*
 * debug rules for transformation
 *
 *
 * if (!assignedInLoop) { result = false;
 * results.add(debugStringer.notAssignedInLoop()); }
 * if (onBothSidesOfAssign || indirectBS) { result = false;
 * results.add(debugStringer.bothSides(onBothSides, true)); }
 * if (incDecStatement) { result = false;
 * results.add(debugStringer.incDec(incDecStatements)); }
 ( if (toggleStatement) { result = false;
 * results.add(debugStringer.toggle(toggleStatements)); }
 * if (!transformation) { result = false;
 * results.add(debugStringer.notTransform()); }
 * if(conditionForAssignBranch) { result = false;
 * results.add(debugStringer.conditionForAssignBranch(
 * conditionalAssignmentBranch)); }
 * results.add(0, new Boolean(result));
 * return results;
 */
```

A5.3.12 Follower.java

```
/*
 * Created on 20-Jun-2005
 */
package main.rules;

import java.util.ArrayList;
import java.util.LinkedHashMap;

/**
 * @author cbishop
 */
public class Follower extends RoleChecker {
   /**
    * Constructor for Follower
    *
    * @param analysedMap
    *            LinkedHashMap of analysed statements
    * @param variable
    *            String being variable name
    * @param methods
    *            ArrayList of method names
    */
   public Follower(LinkedHashMap analysedMap, String variable,
         ArrayList methods) {
      super(analysedMap, variable, methods);
   }

   /**
    * Return list of result from role check
    *
    * @return ArrayList
    */
   public ArrayList checkRole() {
      ArrayList results = follower();
      if (!isFollower()) {
         if (isFixedValue()) {
            results.add(resultStringer.fixedValue());
         } else if (isOrganizer()) {
            results.add(resultStringer.organizer());
         } else if (isStepper()) {
            results.add(resultStringer.stepper());
         } else if (isMostRecentHolder()) {
            results.add(resultStringer.mostRecentHolder());
         } else if (isGatherer()) {
            results.add(resultStringer.gatherer());
         } else if (isMostWantedHolder()) {
            results.add(resultStringer.mostWantedHolder());
         } else if (isOneWayFlag()) {
            results.add(resultStringer.oneWayFlag());
         } else if (isTransformation()) {
            results.add(resultStringer.transformation());
         } else if (isTemporary()) {
            results.add(resultStringer.temporary());
         } else {
            results.add(resultStringer.other());
         }
      }
      return results;
   }
}

/*
 * debug rules for follower
 *
 * if (!assignedInLoop) { result = false;
 * results.add(debugStringer.notAssignedInLoop()); }
 * if (onBothSidesOfAssign || indirectBS) { result = false;
 * results.add(debugStringer.bothSides(onBothSides, true)); }
 * if (incDecStatement) { result = false;
 * results.add(debugStringer.incDec(incDecStatements)); }
 * if (toggleStatement) {
 * result = false; results.add(debugStringer.toggle(toggleStatements)); }
 * if (isTransform) { result = false;
 * results.add(debugStringer.isTransformation(transformations)); }
 * if(conditionForAssignBranch) { result = false;
 * results.add(debugStringer.conditionForAssignBranch(
 * conditionalAssignmentBranch)); }
 * if (nestedBooleanAssign) { result = false;
 * results.add(debugStringer.nestedBooleanAssign(
 * nestedBooleanAssignment)); }
 * if (methodAssignment) { result = false;
 * results.add(debugStringer.methodAssigns(methodAssignments)); }
 * if (!usedInAssignLoop) { result = false;
 * results.add(debugStringer.notUsedInAssign()); }
 * if (assignedBeforeUse && !usedOutsideAssign) { result = false;
 * results.add(debugStringer.temporary(assignBeforeUse)); }
 * results.add(0, new Boolean(result));
 */
```

A5.3.13 Temporary.java

```
/*
 * Created on 20-Jun-2005
 */
package main.rules;

import java.util.ArrayList;
import java.util.LinkedHashMap;

/**
 * @author cbishop
 */
public class Temporary extends RoleChecker {
   /**
    * Constructor for Temporary
    *
    * @param analysedMap
    *            LinkedHashMap of analysed statements
    * @param variable
    *            String being variable name
    * @param methods
    *            ArrayList of method names
    */
   public Temporary(LinkedHashMap analysedMap, String variable,
         ArrayList methods) {
      super(analysedMap, variable, methods);
   }

   /**
    * Return list of result from role check
    *
    * @return ArrayList
    */
   public ArrayList checkRole() {
      ArrayList results = temporary();
      if (!isTemporary()) {
         if (isFixedValue()) {
            results.add(resultStringer.fixedValue());
         } else if (isOrganizer()) {
            results.add(resultStringer.organizer());
         } else if (isStepper()) {
            results.add(resultStringer.stepper());
         } else if (isMostRecentHolder()) {
            results.add(resultStringer.mostRecentHolder());
         } else if (isGatherer()) {
            results.add(resultStringer.gatherer());
         } else if (isMostWantedHolder()) {
            results.add(resultStringer.mostWantedHolder());
         } else if (isOneWayFlag()) {
            results.add(resultStringer.oneWayFlag());
         } else if (isTransformation()) {
            results.add(resultStringer.transformation());
         } else if (isFollower()) {
            results.add(resultStringer.follower());
         } else {
            results.add(resultStringer.other());
         }
      }
      return results;
   }
}

/*
 * debug rules for temporary
 *
 * if (!assignedInLoop) { result = false;
 * results.add(debugStringer.notAssignedInLoop()); }
 * if (onBothSidesOfAssign || indirectBS) { result = false;
 * results.add(debugStringer.bothSides(onBothSides, true)); }
 * if (incDecStatement) { result = false;
 * results.add(debugStringer.incDec(incDecStatements)); }
 * if (toggleStatement) {
 * result = false; results.add(debugStringer.toggle(toggleStatements)); }
 * if (transformation) { result = false;
 * results.add(debugStringer.isTransformation(transformations)); }
 * if(conditionForAssignBranch) { result = false;
 * results.add(debugStringer.conditionForAssignBranch(
 * conditionalAssignmentBranch)); }
 * if (nestedBooleanAssign) { result = false;
 * results.add(debugStringer.nestedBooleanAssign(
 * nestedBooleanAssignment)); }
 * if (methodAssignment) { result = false;
 * results.add(debugStringer.methodAssigns(methodAssignments)); } if
 * (!usedInAssignLoop) { result = false;
 * results.add(debugStringer.notUsedInAssign()); }
 * if (usedOutsideAssign) {
 * result = false; results.add(debugStringer.usedOutsideAssign()); } if
 * (!assignedBeforeUse) { result = false;
 * results.add(debugStringer.assignBeforeUse()); }
 * results.add(0, new Boolean(result)); return results;
 */
```

A5.3.14 OtherRole.java

```
/*
 * Created on 30-Aug-2005
 */
package main.rules;

import java.util.ArrayList;
import java.util.LinkedHashMap;

/**
 * @author cbishop
 */
public class OtherRole extends RoleChecker {

   /**
    * Constructor for OtherRole
    *
    * @param analysedMap
    *            LinkedHashMap of analysed statements
    * @param variable
    *            String being variable name
    * @param methods
    *            ArrayList of method names
    */
   public OtherRole(LinkedHashMap analysedMap, String variable,
         ArrayList methods) {
      super(analysedMap, variable, methods);
   }

   /**
    * Return list of result from role check
    *
    * @return ArrayList
    */
   public ArrayList checkRole() {
      String offendingStatement = resultStringer.offendingStatement("");
      String reason = "role declared for variable is not recognized";
      Boolean result = new Boolean(false);
      ArrayList results = new ArrayList();
      results.add(result);
      results.add(offendingStatement);
      results.add(reason);
      if (isFixedValue()) {
         results.add(resultStringer.fixedValue());
      } else if (isOrganizer()) {
         results.add(resultStringer.organizer());
      } else if (isStepper()) {
         results.add(resultStringer.stepper());
      } else if (isMostRecentHolder()) {
         results.add(resultStringer.mostRecentHolder());
      } else if (isGatherer()) {
         results.add(resultStringer.gatherer());
      } else if (isMostWantedHolder()) {
         results.add(resultStringer.mostWantedHolder());
      } else if (isOneWayFlag()) {
         results.add(resultStringer.oneWayFlag());
      } else if (isTransformation()) {
         results.add(resultStringer.transformation());
      } else if (isFollower()) {
         results.add(resultStringer.follower());
      } else if (isTemporary()) {
         results.add(resultStringer.temporary());
      } else {
         results.add(resultStringer.other());
      }
      return results;
   }
}
```

A5.4 main.BlueJExt

A5.4.1 PreferenceSetter.java

```
package main.blueJExt;

import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JCheckBox;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;
import java.awt.GridLayout;

import bluej.extensions.BlueJ;
import bluej.extensions.PreferenceGenerator;

/**
 * @author cbishop
 *
 */
public class PreferenceSetter implements PreferenceGenerator {
   private JPanel myPanel;
   private JCheckBox checkRoles;
   private JCheckBox giveReason;
   private JCheckBox suggestRole;
   private BlueJ bluej;

   public static final String CHECK_ROLES =
         "Check roles with compilation";
   public static final String GIVE_REASON =
         "Give reason if incorrect role detected";
   public static final String SUGGEST_ROLE =
         "Suggest role for incorrectly annotated role";

   /**
    * Construct the panel, and initialise it from any stored values
    *
    * @param bluej BlueJ object
    */
   public PreferenceSetter(BlueJ bluej) {
      this.bluej = bluej;
      myPanel = new JPanel(new GridLayout(0, 2));
      checkRoles = new JCheckBox();
      giveReason = new JCheckBox();
      suggestRole = new JCheckBox();
      checkRoles.addItemListener(new ItemEventListener());
      myPanel.add(new JLabel("    Check roles with compilation"));
      myPanel.add(checkRoles);
      myPanel.add(new JLabel("    Give reason if incorrect role " +
            "detected"));
      myPanel.add(giveReason);
      myPanel.add(new JLabel("    Suggest role for incorrectly " +
            "annotated role"));
      myPanel.add(suggestRole);
      // Load the default value
      loadValues();
      if (checkRoles()) {
         giveReason.setEnabled(true);
         suggestRole.setEnabled(true);
      } else {
         giveReason.setEnabled(false);
         suggestRole.setEnabled(false);
      }
   }

   /**
    * @return JPanel preference panel
    */
```

```
public JPanel getPanel() {
   return myPanel;
}

/**
 * Save preferences for extension
 */
public void saveValues() {
   // Save the preference value in the BlueJ properties file
   if (checkRoles.isSelected()) {
      bluej.setExtensionPropertyString(CHECK_ROLES, "true");
   } else {
      bluej.setExtensionPropertyString(CHECK_ROLES, "false");
   }
   if (giveReason.isSelected()) {
      bluej.setExtensionPropertyString(GIVE_REASON, "true");
   } else {
      bluej.setExtensionPropertyString(GIVE_REASON, "false");
   }
   if (suggestRole.isSelected()) {
      bluej.setExtensionPropertyString(SUGGEST_ROLE, "true");
   } else {
      bluej.setExtensionPropertyString(SUGGEST_ROLE, "false");
   }
}

/**
 * Load preferences for extension
 */
public void loadValues() {
   // Load the property value from the BlueJ proerties file,
   // default to an empty string
   String cr = bluej.getExtensionPropertyString(CHECK_ROLES, "");
   if (cr.equals("true")) {
      checkRoles.setSelected(true);
   } else
      checkRoles.setSelected(false);
   String gr = bluej.getExtensionPropertyString(GIVE_REASON, "");
   if (gr.equals("true")) {
      giveReason.setSelected(true);
   } else
      giveReason.setSelected(false);
   String sr = bluej.getExtensionPropertyString(SUGGEST_ROLE, "");
   if (sr.equals("true")) {
      suggestRole.setSelected(true);
   } else
      suggestRole.setSelected(false);
}

/**
 * @return whether role checking is required
 */
public boolean checkRoles() {
   return checkRoles.isSelected();
}

/**
 * @return whether to give reason for incorrect role adjudication
 */
public boolean giveReason() {
   return giveReason.isSelected();
}

/**
 * @return whether to suggest correct role for variable
 */
public boolean suggestRole() {
   return suggestRole.isSelected();
}
```

```
    /**
     * inner class to disabled reason and role suggestion if role
     * checking not selected
     *
     * @author cbishop
     */
    class ItemEventListener implements ItemListener {

       public void itemStateChanged(ItemEvent e) {
          if (e.getStateChange() == 1) {
             giveReason.setEnabled(true);
             suggestRole.setEnabled(true);
          } else {
             giveReason.setEnabled(false);
             suggestRole.setEnabled(false);
          }
       }
    }
}
```

A5.6 main.test

A5.6.1 TestMain.java

```
/*
 * Created on 29-Jun-2005
 */
package main.test;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Set;
import java.util.Stack;
import main.RoleAnalyser;
import main.progAnal.MethodGetter;
import main.progAnal.RoleHolder;
import main.progAnal.SourceSorter;
import main.progAnal.StatementAnalyser;
import main.progAnal.StatementGetter;
import main.progAnal.ProgramSlicer;
import main.rules.RuleApplyer;
import main.utils.InfoPrinter;

/**
 * @author cbishop
 *
 */
public class TestMain {
   private static final String[] FILES = { "BubbleSort.java",
         "Closest.java", "DiceGame.java", "DivMod7.java",
         "Doubles.java", "Fibonacci.java", "Growth.java",
         "Histogram.java", "Lexical.java", "Multiplication.java",
         "Number.java", "Occur.java", "ProgramTime.java", "Saw.java",
         "SmoothedAverage.java", "Square.java", "TwoLargest.java" };
   private static final String[] ROLES = { "Fixed Value", "Organizer",
         "Stepper", "Most Recent Holder", "Gatherer",
         "Most Wanted Holder", "One Way Flag", "Transformation",
         "Follower", "Temporary" };
   private SourceSorter sourceSorter;
   private StatementGetter statementGetter;
   private LinkedHashMap brokenSource;
   private Set variables;
   private LinkedHashMap relevantStatements;
   private LinkedHashMap variableStatements;
   private HashMap sortedStatements;
   private LinkedHashMap roles;
   private String file;
   private HashMap roleMap;

   public static void main(String[] args) {
      TestMain testMain = new TestMain(FILES);
   }

   /**
    * Constructor for test class
    *
    * @param files
    *            String[] giving files names of all training programs
    */
   public TestMain(String[] files) {
      //runAll();
```

```
        testRA(files);
        //testRoV(files);
    }

    /*
     * Test RoV software in same way that RolesOfVariables class will use
     * RoleAnalyser
     */
    private void testRoV(String[] fileStrings) {
        File[] files = new File[fileStrings.length];
        for (int i = 0; i < fileStrings.length; i++) {
            files[i] = new File("../../java programs/RoVs/" +
                    fileStrings[i]);
        }
        RoleAnalyser roleAnalyser = new RoleAnalyser(files);
        HashMap roleResults = roleAnalyser.checkRoles();
        Set fileNames = roleResults.keySet();
        Iterator it = fileNames.iterator();
        HashMap roleMap = (HashMap) roleResults.get("role map");
        while (it.hasNext()) {
            String fileName = (String) it.next();
            if (!fileName.equals("role map")) {
                ArrayList fileResults =
                        (ArrayList) roleResults.get(fileName);
                HashMap checkedResults = (HashMap) fileResults.get(0);
                LinkedHashMap roles = (LinkedHashMap) fileResults.get(1);
                Set variables = checkedResults.keySet();
                Iterator iter = variables.iterator();
                while (iter.hasNext()) {
                    String variable = (String) iter.next();
                    Integer roleInt = (Integer) roles.get(variable);
                    String role = (String) roleMap.get(roleInt);
                    ArrayList checkedRole = (ArrayList) checkedResults
                            .get(variable);
                    Boolean isOK = (Boolean) checkedRole.get(0);
                    if (!isOK.booleanValue()) {
                        String errorString = "Possible incorrect role " +
                                "anotation of '" + role + "' for variable '" +
                                variable + "'.\n";
                        errorString += checkedRole.get(1) + "\n";
                        errorString += " - " + checkedRole.get(2) + "\n";
                        //add reason
                        errorString += checkedRole.get(3);
                        //add role suggestion
                        System.out.println(errorString + "\n");
                    } else {
                        System.out.println("Variable '" + variable
                                + "', role annotation OK.\n");
                    }
                }
            }
        }
    }

    /*
     * Test RoV in similar way to way RoleAnalyser instantiates program
     * analysis and rules
     */
    private void testRA(String[] files) {
        InfoPrinter printer = new InfoPrinter();
        for (int i = 0; i < files.length; i++) {
            file = files[i];
            roleMap = initialiseMap();
            SourceSorter sourceSorter = new SourceSorter();
            brokenSource =
                    sourceSorter.sortSource("../../java programs/RoVs/"
                    + file);
            RoleHolder roleHolder = sourceSorter.getRoleHolder();
            variables = roleHolder.getVariables();
            roles = roleHolder.getRoles();
            StatementGetter statementGetter = new StatementGetter();
            variableStatements =
                    statementGetter.getStatements(brokenSource, variables);
            ProgramSlicer programSlicer = new ProgramSlicer();
            sortedStatements = programSlicer.sortStatements(variables,
                    variableStatements);
            //printer.printStatements(variables, sortedStatements);
            StatementAnalyser statementAnalyser = new StatementAnalyser(
                    sortedStatements, variables);
            HashMap analysedStatements = statementAnalyser
                    .getStatementAnalysis();
            //printer.printAnalysedStatements(analysedStatements);
            MethodGetter methodGetter = new MethodGetter();
            ArrayList methods = methodGetter.getMethods(sortedStatements);
            RuleApplyer ruleApplyer = new RuleApplyer();
            HashMap checkedRoles = new HashMap();
            checkedRoles = ruleApplyer.applyRules(analysedStatements,
                    roles, methods);
            Iterator it = variables.iterator();
            while (it.hasNext()) {
                String variable = (String) it.next();
                ArrayList checkResults =
                        (ArrayList) checkedRoles.get(variable);
                Integer roleKey = (Integer) roles.get(variable);
                String role = (String) roleMap.get(roleKey);
                printer.printTestResults(checkResults, variable, role);
            }
        }
    }

    /*
     * Initialise map associating role within specific int keys.
     */
    private HashMap initialiseMap() {
        HashMap returnMap = new HashMap();
        for (int i = 0; i < 10; i++) {
        returnMap.put(new Integer(i + 1), ROLES[i]);
        }
        return returnMap;
    }

    /*
     * Run all program slicing tests
     */
    private void runAll() {
        testSourceBreakdown();
        testArrayStatements();
        testSortedStatements();
    }

    /*
     * Compare broken source with expected source break down and report
     * errors
     */
    private void testSourceBreakdown() {
        System.out.println("Source Test: " + file);
        ArrayList returnArray = new ArrayList();
        Set keySet = (Set) brokenSource.keySet();
        Stack iterators = new Stack();
        Stack maps = new Stack();
        Iterator it = keySet.iterator();
        int lineCounter = 1;
        boolean testOk = true;
        try {
            FileInputStream fileIn = new FileInputStream(
                    "../../test files/source files/" + file);
            BufferedReader input =
                    new BufferedReader(new InputStreamReader(fileIn));
            while (it.hasNext()) {
                String key = (String) it.next();
            String testLine = input.readLine();
            String testKey = removeSpaces(key);
            testLine = removeSpaces(testLine);
            if (!testKey.equals(testLine)) {
               System.out.println("Error at line :" + lineCounter +
                     ": " + testKey + "\t" + testLine);
               testOk = false;
            }
            if (key.endsWith("{")) {
               maps.push(brokenSource);
               iterators.push(it);
               brokenSource = (LinkedHashMap) brokenSource.get(key);
               keySet = brokenSource.keySet();
               it = keySet.iterator();
            } else if (key.endsWith("}")) {
               brokenSource = (LinkedHashMap) maps.pop();
               it = (Iterator) iterators.pop();
            }
            lineCounter++;
         }
         if (testOk) {
            System.out.println("Passed!");
         }
      } catch (FileNotFoundException e) {
         System.out.println(e.getMessage());
      } catch (IOException e) {
         System.out.println(e.getMessage());
      }
      System.out.println();
   }

   /*
    * Compare statement ArrayLists with expected ArrayList for each
    * variable and report errors
    */
   private void testArrayStatements() {
      Iterator iter = variables.iterator();
      System.out.println("Array Test: " + file);
      boolean testOk = true;
      try {
         FileInputStream fileIn = new FileInputStream(
               "../../test files/array files/" + file);
         BufferedReader input =
               new BufferedReader(new InputStreamReader(fileIn));
         int lineCounter = 1;
         while (iter.hasNext()) {
            String keyString = (String) iter.next();
            ArrayList tempArray =
                  (ArrayList) variableStatements.get(keyString);
            System.out.println(keyString);
            for (int i = 0; i < tempArray.size(); i++) {
               ArrayList variableArray = (ArrayList) tempArray.get(i);
               for (int count = 0; count < variableArray.size();
                     count++) {
                  String variableString =
                        (String) (variableArray.get(count));
                  String testString = input.readLine();
                  if (!variableString.equals(testString)) {
                     System.out.println("Error at line " +
                           lineCounter + "\t" + variableString +
                           "\t" + testString);
                     testOk = false;
                  }
               }
            }
            lineCounter++;
         }
         if (testOk) {
            System.out.println("Passed!\n");
         }
```

```
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    /*
     * Compare program slices for each variable with expected program
     * slices
     */
    private void testSortedStatements() {
        System.out.println("Statement Test: " + file);
        Iterator iter = variables.iterator();
        boolean testOk = true;
        int lineCounter = 1;
        try {
            FileInputStream fileIn = new FileInputStream(
                    "../../test files/statement files/" + file);
            BufferedReader input =
                    new BufferedReader(new InputStreamReader(fileIn));
            while (iter.hasNext()) {
                String keyString = (String) iter.next();
                System.out.println(keyString);
                ArrayList statementArray =
                        (ArrayList) sortedStatements.get(keyString);
                for (int i = 0; i < statementArray.size(); i++) {
                    String statementString =
                            (String) (statementArray.get(i));
                    String testString = input.readLine();
                    if (!statementString.equals(testString)) {
                        System.out.println("Error at line " + lineCounter +
                                "\t" + statementString + "\t" + testString);
                        testOk = false;
                    }
                }
                lineCounter++;
            }
        } catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
        if (testOk) {
            System.out.println("Passed!\n");
        }
    }

    private String removeSpaces(String inputString) {
        String outputString = "";
        for (int i = 0; i < inputString.length(); i++) {
            char tempChar = inputString.charAt(i);
            if (tempChar != 32) { //look for white space
                outputString += inputString.substring(i, i + 1);
            }
        }
        return outputString;
    }
}
```

A5.6.2 DebugStringer.java

```
/*
 * Created on 12-Jul-2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package main.test;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Set;

/**
 * @author cbishop
 *
 */
public class DebugStringer {

   private String variable;

   private LinkedHashMap analysedMap;

   /**
    * Constructor for DebugStringer
    *
    * @param inputMap
    *            LinkedHashMap containing analysed statements
    * @param var
    *            String variable for which string to be generated
    */
   public DebugStringer(LinkedHashMap inputMap, String var) {
      variable = var;
      analysedMap = inputMap;
   }

   /**
    * Show statement for variable found in loop
    *
    * @param foundInLoop
    *            HashMap containing all statements for a given variable
    * found in a loop
    * @param what
    *            String specifying what sort of statement to look for
    * @return String giving statement and its location in the program
    * slice
    */
   public String showFoundInLoop(HashMap foundInLoop, String what) {
      String returnString = "";
      boolean inLoop = false;
      Set foundSet = foundInLoop.keySet();
      Iterator it = foundSet.iterator();
      while (it.hasNext()) {
         String statementInQuestion = (String) it.next();
         ArrayList statementInLoop =
               (ArrayList) foundInLoop.get(statementInQuestion);
         returnString += variable + " " + what +
               " statement in loop is: " + statementInQuestion + "\n";
         inLoop = true;
         for (int i = 0; i < statementInLoop.size(); i++) {
            ArrayList specificOccurance =
                  (ArrayList) statementInLoop.get(i);
            returnString += "at line " + specificOccurance.get(0) +
                  " in loop: ";
            for (int j = 1; j < specificOccurance.size() - 1; j += 2) {
               returnString += specificOccurance.get(j + 1) +
                     " at line: ";
```

```
                returnString += specificOccurance.get(j) + "\n";
            }
        }
    }
    return returnString;
}

/**
 * Show statements found in loop in which variable is assigned
 *
 * @param foundInLoop
 *            HashMap containing all statements found in assignment
 * loop
 * @param what
 *            String The sort of statemen to look for
 * @return String giving statement and its location in the program
 * slice
 */
public String showFoundInAssignmentLoop(HashMap foundInLoop,
        String what) {
    String returnString = "";
    //boolean inLoop = false;
    Set foundSet = foundInLoop.keySet();
    Iterator it = foundSet.iterator();
    while (it.hasNext()) {
        String statementInQuestion = (String) it.next();
        ArrayList statementInLoop =
                (ArrayList) foundInLoop.get(statementInQuestion);
        //inLoop = true;
        for (int i = 0; i < statementInLoop.size(); i++) {
            ArrayList specificOccurance =
                    (ArrayList) statementInLoop.get(i);
            returnString += what + " statement '" +
                    statementInQuestion + "' found at line: " +
                    specificOccurance.get(0);
            for (int j = 3; j < specificOccurance.size() - 1; j += 2) {
                returnString += "\nin loop '" +
                        specificOccurance.get(j + 1) + "' (at line: ";
                returnString += specificOccurance.get(j) + ")";
            }
            returnString += "\nassignment statement '"
                    + specificOccurance.get(2) + "' found at line "
                    + specificOccurance.get(1) + "\n";
        }
    }
    return returnString;
}

/**
 * Show if variable assigned in for loop statement
 *
 * @param is
 *            boolean Whether statement is assigned in for loop
 * statement
 * @return String stating whether statement is assigned in for loop
 *         statement
 */
public String assignedInFor(boolean is) {
    String returnString;
    if (is)
        returnString = "variable " + variable +
                " is assigned in 'for' loop statement";
    else
        returnString = "variable " + variable +
                " is not assigned in 'for' loop statement";
    return returnString;
}

/**
 * Show whether there is any direct use of the variable in the
```

```
     * program
     *
     * @return String indicating that there is no direct use of variable
     * in the program
     */
    public String noDirectUse() {
       String returnString = "limited or no conditional use of " +
             "variable in assignment loop and direct condition for " +
             "assignment loop";
       return returnString;
    }

    /**
     * Show when toggle statement is found in loop
     *
     * @param toggleStatements
     *            ArrayList of toggleStatements for variable
     * @return String givin toggle statement found in a loop
     */
    public String toggle(ArrayList toggleStatements) {
       String returnString = "";
       for (int i = 0; i < toggleStatements.size(); i++) {
          returnString += "toggle statement " + toggleStatements.get(i) +
                " found in loop\n";
       }
       return returnString;
    }

    /**
     * Show that variable is not toggled within loop
     *
     * @return String
     */
    public String toggle() {
       return "variable is not toggled in loop";
    }

    /**
     * Show that variable is condition for its assignment branch
     *
     * @param conditionalAssignmentBranch
     *            HashMap containing assignmentBranch condition
     * statements
     * @return String giving offending statement and whereabouts in
     * program slice
     */
    public String conditionForAssignBranch(
          HashMap conditionalAssignmentBranch) {
       String returnString = "";
       Set statementSet = conditionalAssignmentBranch.keySet();
       Iterator it = statementSet.iterator();
       while (it.hasNext()) {
          String statement = (String) it.next();
          ArrayList branchStatements =
             (ArrayList) conditionalAssignmentBranch.get(statement);
          for (int i = 0; i < branchStatements.size() - 1; i += 3) {
             returnString += "assignment statement '"+
                   branchStatements.get(i) + "'\n";
             returnString += "found at line " +
                   branchStatements.get(i + 1) +
                   " in branch '" + statement + "' at line " +
                   branchStatements.get(i + 2) + "\n";
          }
          returnString += "variable appears to be condition for branch" +
                " in which it is assigned\n";
       }
       return returnString;
    }

    /**
```

```
     * State that variable is not assigned in branch, or condition for
     * assignment branch
     *
     * @return Message to that effect
     */
    public String conditionForAssignBranch() {
       return "variable is not assigned in branch, or is not " +
             "condition for branch in which it is assigned\n";
    }

    /**
     * Show that variable appears on both side of assignment statement
     *
     * @param onBothSides
     *            ArrayList containing statements in question
     * @param is
     *            whether variable does appear on both sides or not
     * @return String stating whether variable appears on both sides and
     * if so giving offending statement
     */
    public String bothSides(ArrayList onBothSides, boolean is) {
       String returnString = "";
       if (is) {
          Iterator it = onBothSides.iterator();
          while (it.hasNext()) {
             String statement = (String) it.next();
             returnString += "variable " + variable +
                   " appears on both sides of assignment statement " +
                   statement + "\n";
          }
       } else
          returnString += returnString += "variable " +
                + variable " does not appear on both sides of " +
                "assignment statement\n";
       return returnString;
    }

    /**
     * Show that variable is incremented/decremented in loop
     *
     * @param incDecStatements
     *            ArrayList of incDec statements
     * @return String giving offending statements
     */
    public String incDec(ArrayList incDecStatements) {
       String returnString = "";
       Iterator it = incDecStatements.iterator();
       while (it.hasNext()) {
          String statement = (String) it.next();
          returnString += "variable " + variable +
                " is incremented/decremented etc within" +
                " loop in statement " + statement + "\n";
       }
       return returnString;
    }

    /**
     * State that variable is not incremented or decremented within loop
     *
     * @return String to that effect
     */
    public String incDec() {
       String returnString = "variable " + variable +
             " is not incremented/decremented in loop statement";
       return returnString;
    }

    /**
     * State that variable does not appear to be array
     *
     * @return String to that effect
     */
    public String notArray() {
       String returnString = "variable " + variable +
             " is not array, or is not used directly as an array\n";
       return returnString;
    }

    /**
     * State that variable is not assigned within a loop
     *
     * @return String to that effect
     */
    public String notAssignedInLoop() {
       String returnString = "variable " + variable
             + " is not assigned within loop\n";
       return returnString;
    }

    /**
     * State that there appear to be no organizer statements for variable
     *
     * @return String to that effect
     */
    public String notOrganizer() {
       String returnString = "there are no organizer type assignment" +
             "statements for variable " + variable + "\n";
       return returnString;
    }

    /**
     * Show that variable appears in organizer type statements
     *
     * @param organizerStatements
     *            ArrayList of organizer statements
     * @return String confirming this and giving organizer statements in
     *         question
     */
    public String isOrganizer(ArrayList organizerStatements) {
       String returnString = "variable " + variable
             + " is used as organizer in statement ";
       for (int i = 0; i < organizerStatements.size(); i++) {
          String statement = (String) organizerStatements.get(i) + "\n";
          returnString += statement;
       }
       return returnString;
    }

    /**
     * State that variable does not appear to be transformation
     *
     * @return String to that effect
     */
    public String notTransform() {
       String returnString = "one or more assignment statement either" +
             "contains no operator characters, or call to method";
       return returnString;
    }

    /**
     * Show that variable is transformation
     *
     * @param transformations
     *            ArrayList containing transformation statements
     * @return String showing transformation type statement for variable
     */
    public String isTransformation(ArrayList transformations) {
       String returnString = "";
       for (int i = 0; i < transformations.size(); i++) {
          returnString += "statement " + transformations.get(i) +
                " appears to be transformation statement\n";
        }
        return returnString;
    }

    /**
     * State that variable does appear to be indirectly toggled within
     * loop
     *
     * @return String to that effect
     */
    public String nestedBooleanAssign() {
        return "variable " + variable + " is not boolean assigned " +
              "with alternate values in nested and outer loops";
    }

    /**
     * Show that variable appears to be indirectly toggled within loop
     *
     * @param nestedAssigns
     *            ArrayList of nested boolean assignments
     * @return String showing statements in question
     */
    public String nestedBooleanAssign(ArrayList nestedAssigns) {
        String returnString = "";
        for (int i = 0; i < nestedAssigns.size(); i += 3) {
            String statementType = (String) nestedAssigns.get(i + 1);
            if (statementType.equals("nested")) {
                returnString += "assignment statement " +
                        nestedAssigns.get(i) + " found in nested loop in " +
                        loop containing statement " +
                        nestedAssigns.get(i + 2);
            } else {
                returnString += "assignment statement " +
                        nestedAssigns.get(i + 2) +
                        " found in nested loop in loop containing statement"
                        + " " + nestedAssigns.get(i);
            }
        }
        return returnString;
    }

    /**
     * Show that variable appears to be assigned with return from method
     * call
     *
     * @param methodAssigns
     *            ArrayList of assignment statements within method calls
     * on right hand side
     * @return String showing variable and statements in question
     */
    public String methodAssigns(ArrayList methodAssigns) {
        String returnString = "";
        for (int i = 0; i < methodAssigns.size(); i++) {
            String statement = (String) methodAssigns.get(i);
            returnString += "variable " + variable +
                    " appears to be assigned with output " +
                    "from method in statement " + statement + "\n";
        }
        return returnString;
    }

    /**
     * State that variable is not used in assignment loop
     *
     * @return String to that effect
     */
    public String notUsedInAssign() {
        return "variable " + variable + " does not appear to be used " +
            "in same loop in which it is assigned\n";
    }

    /**
     * Show that variable is assigned before use in loop
     *
     * @param assignBeforeUse
     *            ArrayList of statements
     * @return String showing specific statements and fact that variable
     * not used outside of loop
     */
    public String temporary(ArrayList assignBeforeUse) {
       String returnString = "";
       returnString += "variable " + variable + " used in statement '" +
             assignBeforeUse.get(0) + "'\n" + "before assignment '" +
             assignBeforeUse.get(1) + "\n";
       for (int i = 2; i < assignBeforeUse.size(); i++) {
          returnString += "in loop " + assignBeforeUse.get(i) + "\n";
       }
       returnString += "and not used outside of assign loop\n";
       return returnString;
    }

    /**
     * State that variable is used outside of loop in which it is
     * assigned
     *
     * @return String to that effect
     */
    public String usedOutsideAssign() {
       return "variable " + variable +
             " is used outside of loop in which it is assigned";
    }

    /**
     * State that variable is not assigned before use in a loop
     *
     * @return String to that effect
     */
    public String assignBeforeUse() {
       return "variable " + variable +
             " is not assigned value before use in loop";
    }

    /**
     * State that variable is only assigned directly with value of other
     * variable
     *
     * @return String to that effect
     */
    public String notMethodAssign() {
       return "variable " + variable +
             " only assigned with value of other variable";
    }

    /**
     * State that variable appears to be array filled in loop
     *
     * @return String to that effect
     */
    public String arrayAssignInLoop() {
       return "variable " + variable +
             " is array that is filled in loop";
    }

    /**
     * State that variable is used directly in program
     *
     * @return String to that effect
     */
    public String directUse() {
       return "variable " + variable + " is used directly";
    }

    /**
     * State that variable is not used for condition of assignment loop
     *
     * @return String to that effect
     */
    public String notUsedForAssignLoopCondition() {
       return "variable " + variable +
             " is not used for condition of loop in which it is" +
             "assigned";
    }

    /**
     * State that variable is used for loop condition outside of loop in
     * which it is assigned
     *
     * @return String to that effect
     */
    public String loopUseNoAssign() {
       return "variable " + variable + " is used for loop condition " +
             "outside of loop in which it is assigned";
    }
}
```

A5.7 main.utils

A5.7.1 InfoPrinter.java

```
/*
 * Created on 24-Jun-2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package main.utils;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Set;
import java.util.Stack;

import main.progAnal.ProgramAnalyser;
import main.progAnal.RoleHolder;

/**
 * @author cbishop
 *
 * Class to print out information for debug etc.
 */
public class InfoPrinter {

   private ProgramAnalyser programAnalyser;

   /**
    * Constructor for InfoPrinter
    *
    */
   public InfoPrinter() {
      programAnalyser = new ProgramAnalyser();
   }

   /**
    * Print roles for each variable
    *
    * @param roleHolder
    *           RoleHolder containing variables and their roles
    */
   public void printRoles(RoleHolder roleHolder) {
      String roleSet = roleHolder.toString();
      System.out.println(roleSet);
   }

   /**
    * Print formatted source code from sorted source LinkedHashMap
    *
    * @param brokenSource
    *           LinkedHashMap containing hierarchy of source code
    * statements
    */
   public void printSource(LinkedHashMap brokenSource) {
      Set keySet = (Set) brokenSource.keySet();
      Stack iterators = new Stack();
      Stack maps = new Stack();
      Iterator it = keySet.iterator();
      String indent = "";
      while (it.hasNext()) {
         String key = (String) it.next();
         if (key.endsWith("{")) {
            System.out.println(indent + key);
            indent += "    ";
            maps.push(brokenSource);
            iterators.push(it);
            brokenSource = (LinkedHashMap) brokenSource.get(key);
            keySet = brokenSource.keySet();
            it = keySet.iterator();
        } else if (key.endsWith("}")) {
            indent = indent.substring(0, indent.length() - 4);
            System.out.println(indent + key);
            brokenSource = (LinkedHashMap) maps.pop();
            it = (Iterator) iterators.pop();
        } else {
            System.out.println(indent + key);
        }
    }
    System.out.println();
}

/**
 * Print ArrayLists of statements for a given variable, together will
 * all preceding control constructs
 *
 * @param vars
 * @param statements
 */
public void printStatementArrays(Set vars,
        LinkedHashMap statements) {
    Iterator iter = vars.iterator();
    while (iter.hasNext()) {
        String keyString = (String) iter.next();
        ArrayList tempArray = (ArrayList) statements.get(keyString);
        System.out.println(keyString + ": " + tempArray.size());
        for (int i = 0; i < tempArray.size(); i++) {
            ArrayList printArray = (ArrayList) tempArray.get(i);
            for (int count = 0; count < printArray.size(); count++) {
                String printString = (String) (printArray.get(count));
                System.out.println(printString);
            }
        }
        //System.out.println();
    }
}

/**
 * Print program slices for each variable
 *
 * @param vars
 *            Set of variable names
 * @param statements
 *            HashMap of statement for each variable
 */
public void printStatements(Set vars, HashMap statements) {
    Iterator iter = vars.iterator();
    while (iter.hasNext()) {
        String keyString = (String) iter.next();
        ArrayList statementArray =
                (ArrayList) statements.get(keyString);
        System.out.println(keyString + ": " + statementArray.size());
        String indent = "";
        for (int i = 0; i < statementArray.size(); i++) {
            String statementString = (String) (statementArray.get(i));
            if (statementString.endsWith("}")) {
                indent = indent.substring(0, indent.length() - 4);
            }
            //System.out.println(statementString);
            System.out.println(indent + statementString);
            if (statementString.endsWith("{")) {
                indent += "    ";
            }
        }
        System.out.println();
    }
}

/**
 * Print analysis of slices for each variable e.g. whether assignemt,
 * usage, conditional or other statements, and their positions within
 * the source.
 *
 * @param analysedStatements
 *            HashMap of analysed statements
 */
public void printAnalysedStatements(HashMap analysedStatements) {
    Set keySet = analysedStatements.keySet();
    Iterator it = keySet.iterator();
    while (it.hasNext()) {
        String keyString = (String) it.next();
        LinkedHashMap analysedMap = (LinkedHashMap) analysedStatements
                .get(keyString);
        printWhereaboutsInfo(keyString, analysedMap);
        System.out.println();
    }
}

/**
 * Print specific category statements for a given variable, e.g.
 * usage statements
 *
 * @param inputArray
 *            ArrayList containing all relevant statements
 * @param whatArray
 *            String specifying what sort of statements to print
 * @param variable
 *            String variable for which statements are to be printed
 */
public void printArray(ArrayList inputArray, String whatArray,
        String variable) {
    System.out.println(whatArray + " statements for " + variable);
    for (int i = 0; i < inputArray.size(); i++) {
        System.out.println(inputArray.get(i));
    }
}

/*
 * Used by print whereabouts infor to print specific map
 */
private void printMap(HashMap whereaboutsMap) {
    Set keySet = whereaboutsMap.keySet();
    Iterator it = keySet.iterator();
    while (it.hasNext()) {
        String statement = (String) it.next();
        System.out.println("Statement: " + statement); //debug
        ArrayList tempArray =
                (ArrayList) whereaboutsMap.get(statement);
        ArrayList indexCard = (ArrayList) tempArray.get(0);
        for (int index = 0; index < indexCard.size() - 1; index++) {
            Integer tempInt = (Integer) indexCard.get(index);
            Integer statementLine = (Integer) tempArray.get(tempInt
                    .intValue());
            System.out.println("found at line " +
                    statementLine.intValue());
            Integer nextInt = (Integer) indexCard.get(index + 1);
            for (int i = tempInt.intValue() + 1;
                    i < nextInt.intValue(); i += 2) {
                String statementSignature =
                        (String) tempArray.get(i + 1);
                if (programAnalyser.loop(statementSignature)) {
                    System.out.print("in loop: " + statementSignature);
                } else if (programAnalyser.isIf(statementSignature)
                        || programAnalyser.isElse(statementSignature)) {
                    System.out.print("in branch: " + statementSignature);
                } else if (programAnalyser.isClass(
                        statementSignature)) {
                    System.out.print("in class: " + statementSignature);
                } else if (programAnalyser
                        .methodStatement(statementSignature)) {
                    System.out.print("in method: " + statementSignature);
                } else if (programAnalyser
                        .tryCatchStatement(statementSignature)) {
                    System.out.print("in block: " + statementSignature);
                }
                Integer loopBranchLine = (Integer) tempArray.get(i);
                System.out.println(" (at line " +
                        loopBranchLine.intValue() + ")");
            }
        }
    }
}

/*
 * Used by printAnalysedStatements to print whereabouts info
 */
private void printWhereaboutsInfo(String variable,
        HashMap inputMap) {
    Set keySet = inputMap.keySet();
    System.out.println("Statements for variable: " + variable);
    Iterator it = keySet.iterator();
    while (it.hasNext()) {
        String mapType = (String) it.next();
        HashMap whereaboutsMap = (HashMap) inputMap.get(mapType);
        System.out.println("Analysed map for " + variable + ": " +
                mapType + " statements");
        printMap(whereaboutsMap);
    }
}

/**
 * Print simply results following roles checking
 *
 * @param checkResults
 *            ArrayList containg results for a given variable
 * @param variable
 *            String specifying variable for which to print results
 * @param role
 *            String specifying role played by variable
 */
public void printResults(ArrayList checkResults, String variable,
        String role) {
    Boolean isOk = (Boolean) checkResults.get(0);
    if (!isOk.booleanValue()) {
        System.out.println("Possible incorrect role declaration of '"
                + role + "' for variable: " + variable);
        for (int i = 1; i < checkResults.size(); i++) {
            System.out.println(checkResults.get(i));
        }
    } else
        System.out.println("Role declaration OK for variable: " +
                variable);
    System.out.println();
}

/**
 * Print results in format output by BlueJ
 *
 * @param checkResults
 *            ArrayList containg results for a given variable
 * @param variable
 *            String specifying variable for which to print results
 * @param role
 *            String specifying role played by variable
 */
```

```
    public void printTestResults(ArrayList checkResults, String variable,
            String role) {
        Boolean isOk = (Boolean) checkResults.get(0);
        if (!isOk.booleanValue()) {
            System.out.println("Possible incorrect role declaration of '"
                    + role + "' for variable: " + variable);
            System.out.println("offending statement: " +
                    checkResults.get(1));
            System.out.println("reason: " + checkResults.get(2));
            System.out.println("possible role: " + checkResults.get(3));
        } else
            System.out.println("Role declaration OK for variable: " +
                    variable);
        System.out.println();
    }

    /**
     * Print all methods name detected in source code
     *
     * @param methods
     *            ArrayList containing names of all methods in source
     * code
     */
    public void printMethods(ArrayList methods) {
        Iterator it = methods.iterator();
        System.out.println("method statements:");
        while (it.hasNext()) {
            System.out.println(it.next() + "\n");
        }
    }
}
```

A5.7.2 SliceWriter.java

```
/*
 * Created on 24-Jun-2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package main.utils;

import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Set;

/**
 * @author cbishop
 *
 * @version June 2005
 *
 * Class to output sliced programs to appropriate files for analysis
 */
public class SliceWriter {

   /**
    * Constructor for SliceWriter
    *
    * @param roles
    *            LinkedHashMap containg variables as keys and their
    *            associated roles as values
    * @param statements
    *            HashMap contaning a sliced program for each variable
    */
   public void writeSlice(LinkedHashMap roles, HashMap statements) {
      Set vars = roles.keySet();
      Iterator iter = vars.iterator();
      while (iter.hasNext()) {
         String variable = (String) iter.next();
         Integer roleValue = (Integer) roles.get(variable);
         switch (roleValue.intValue()) {
         case 1:
            writeRole("../../Sliced files/Fixed Value.txt", variable,
                  statements);
            break;
         case 2:
            writeRole("../../Sliced files/Organizer.txt", variable,
                  statements);
            break;
         case 3:
            writeRole("../../Sliced files/Stepper.txt", variable,
                  statements);
            break;
         case 4:
            writeRole("../../Sliced files/Most Recent Holder.txt",
                  variable, statements);
            break;
         case 5:
            writeRole("../../Sliced files/Gatherer.txt", variable,
                  statements);
            break;
         case 6:
            writeRole("../../Sliced files/Most Wanted Holder.txt",
                  variable, statements);
            break;
         case 7:
            writeRole("../../Sliced files/One Way Flag.txt", variable,
                  statements);
```

```
                break;
            case 8:
              writeRole("../../Sliced files/Transformation.txt", variable,
                        statements);
                break;
            case 9:
                writeRole("../../Sliced files/Follower.txt", variable,
                        statements);
                break;
            case 10:
                writeRole("../../Sliced files/Temporary.txt", variable,
                        statements);
                break;
            }
        }
    }

    /*
     * Write slice for each variable to specified file
     */
    private void writeRole(String fileName, String variableName,
            HashMap statements) {
        ArrayList statementArray =
                (ArrayList) statements.get(variableName);
        try {
            FileWriter dataOutput = new FileWriter(fileName, true);
            System.out.println(variableName);
            dataOutput.write(variableName + "\n");
            String indent = "";
            for (int i = 0; i < statementArray.size(); i++) {
                String statementString = (String) (statementArray.get(i));
                if (statementString.endsWith("}")) {
                    indent = indent.substring(0, indent.length() - 4);
                }
                dataOutput.write(indent + statementString + "\n");
                System.out.println(indent + statementString);
                if (statementString.endsWith("{")) {
                    indent += "    ";
                }
            }
            dataOutput.write("\n");
            System.out.println();
            dataOutput.close();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```