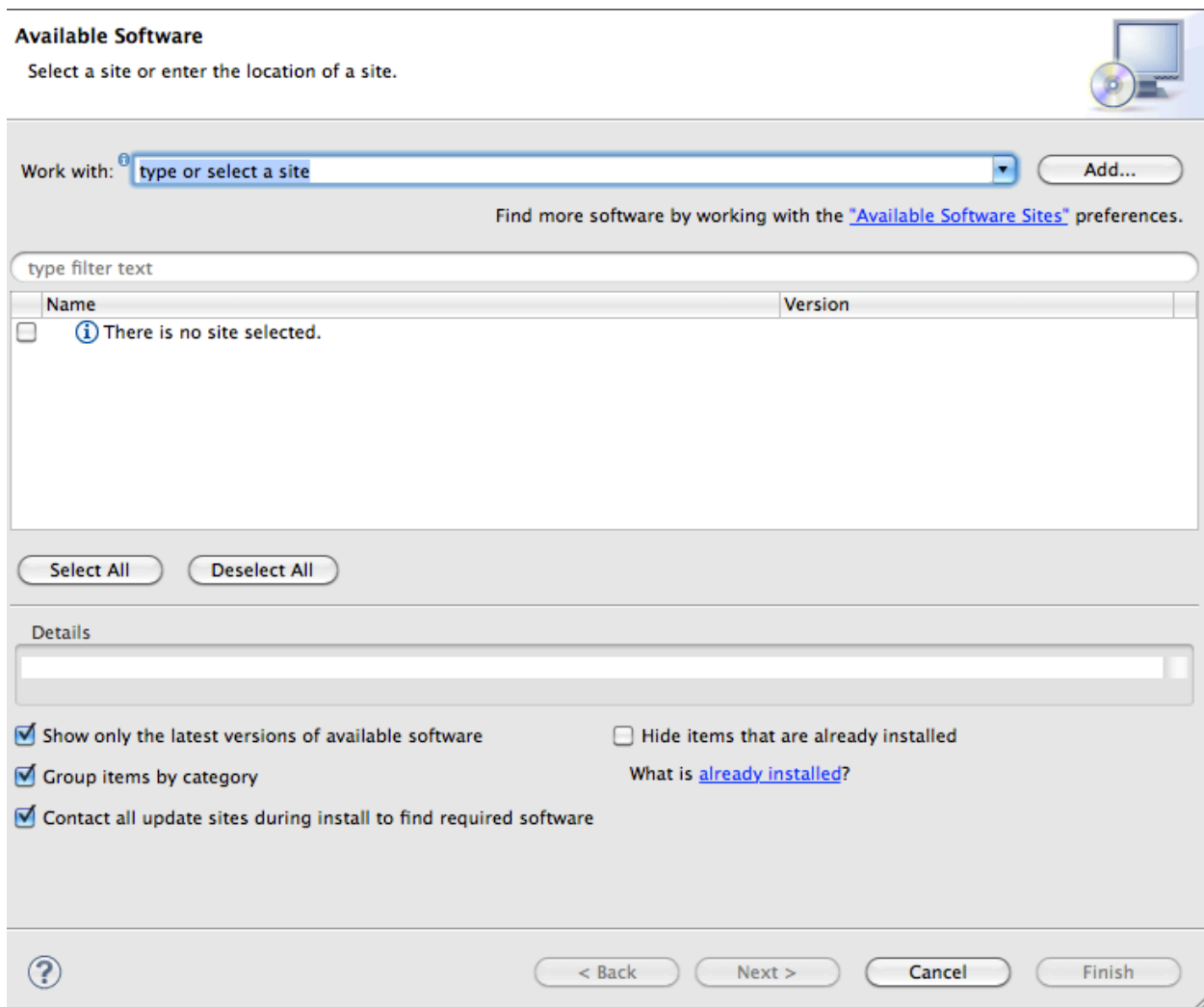


Lab 2: JML and ESC/Java2 Tutorial

For this lab, you will learn how to write method specifications using the Java Modeling Language (JML). In addition, you will learn how to verify your specifications using the ESC/Java2 plugin for Eclipse. ESC/Java2 is an extended static checker that allows you to annotate your Java code with JML specifications and verifies your specifications to see if all preconditions and postconditions are satisfied.

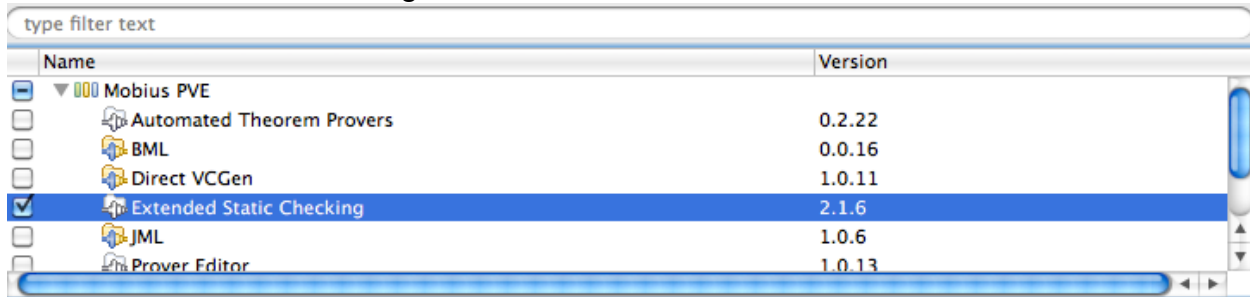
How to Install ESC/Java2

1. Open Eclipse
2. On the top menu, go to Help > Install New Software... You will see the following window pop up.

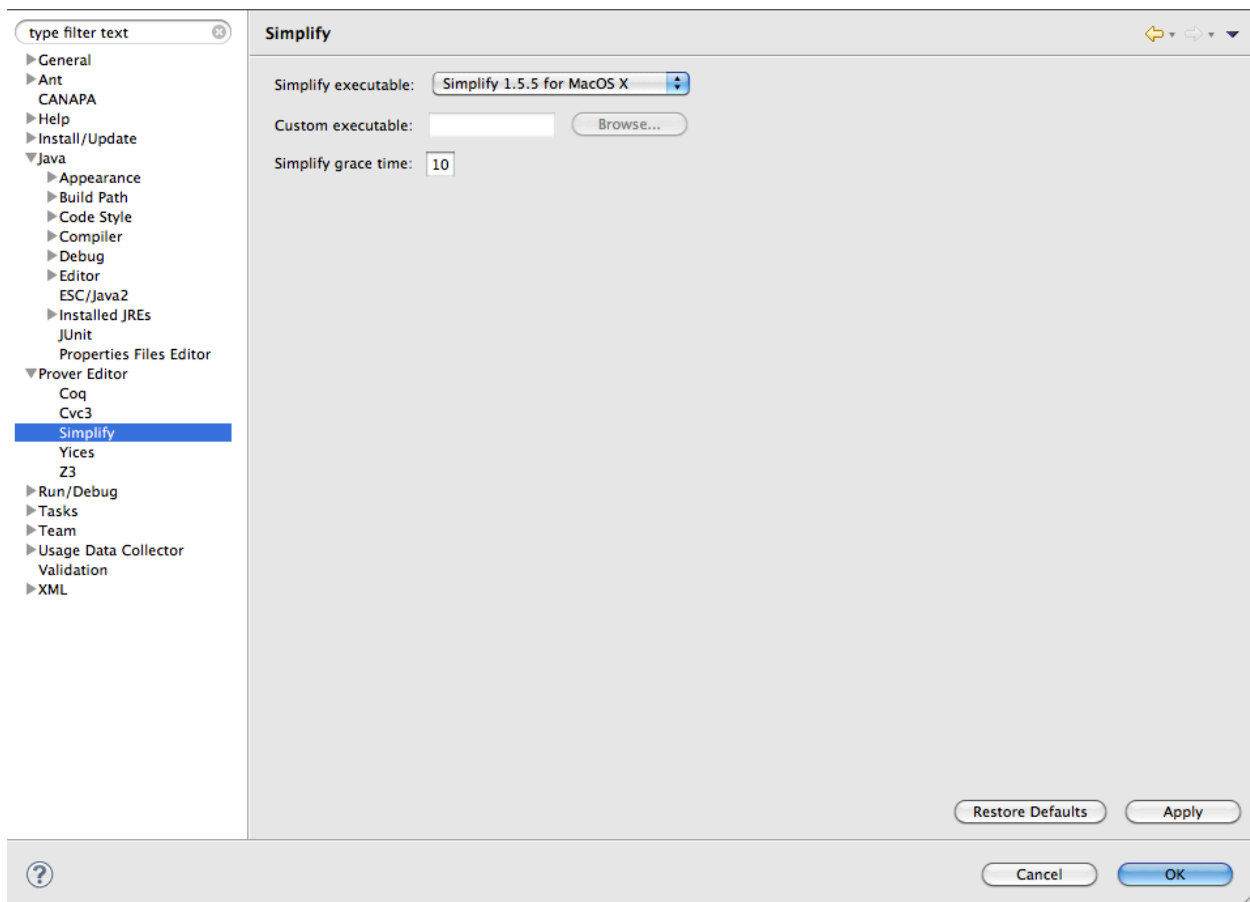


3. On the dropdown area right beside "Works with", type the following site then press enter: <http://kind.ucd.ie/products/opensource/Mobius/updates/>. You will be prompted to give the new site a name. You can name it "Mobius".

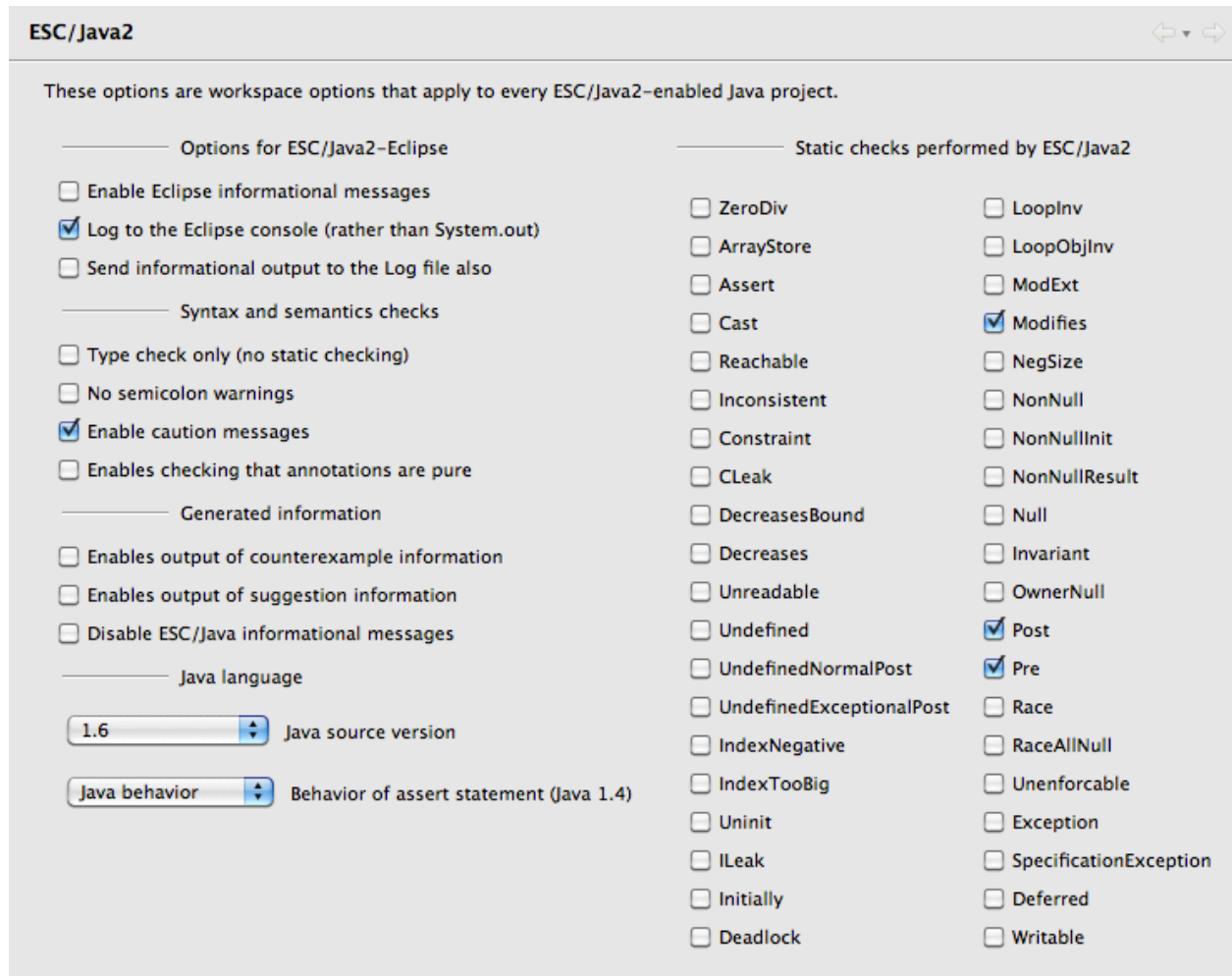
4. Once you've finished Step 3, you will see a list of Eclipse plugins in the pane underneath. Show the contents of "Mobius PVE" and check the box right beside "Extended Static Checking".



5. Press Next and follow the onscreen instructions until ESC/Java2 is installed.
6. To configure ESC/Java2, go to the Eclipse "Preferences..." window. On the left pane, show the contents of "Prover Editor". Finally, on the dropdown beside "Simplify executable", select Simplify 1.5.x for <OS> where OS refers to the operating system you are using, then click Apply. For example, if you are using a MacOS X, the settings would look as follows:



7. Next, show the contents of “Java” on the left pane and select “ESC/Java2”. Make sure your settings look as follows:



8. Press Apply, then press OK. ESC/Java2 should now be configured

How to Write JML Specifications

JML has a very rich set of annotations that can be used to write method specifications. Here, we will focus on three annotations: requires, assignable, and ensures. If you want to go in depth and learn about other annotations, you can check the tutorials found in the following page: <http://sourceforge.net/apps/trac/jmlspecs/wiki/TeachingMaterials>.

1. Create a new Eclipse project and name it “intersect_verify” (see Lab 1 if you’ve forgotten how to create a new project)
2. Under the src folder, create a new package and name it “com”.
3. Under the “com” package, create a new folder called “verify”
4. Create a new file named “IntersectVerify.java” in the “verify” folder.

5. Add the package declaration at the top, and create a public class named IntersectVerify (recall that the class name must match the name of the .java file). This class contains seven int members - m_a, m_b, m_c, m_m, m_y, valAtPoint, and m_absValue - and one boolean member - m_result. The goal is to do the following:

- Determine if a certain integer value (we'll call this iPoint) is a solution to the following equation involving a quadratic and a linear function (m_a is not 0):

$$m_a * x^2 + m_b * x + c = m_m * x + m_y$$

- The approach taken here is to (1) plug in iPoint to x in the left hand side of the above equation (i.e., the quadratic expression) and find the value of that expression, (2) plug in iPoint to the right hand side of the above equation (i.e., the linear expression) and find the value of that expression, and (3) compare the value computed from (1) with the value computed from (2).

- If the values match, then iPoint must be a solution to the above equation, so m_result is set to true, valAtPoint is set to the value of both the right hand side and the left hand side (which should be the same), and m_absValue is set to the absolute value of valAtPoint.

- Otherwise, if the values do **not** match, iPoint is not a solution to the above equation, so m_result is set to false and both m_absValue and valAtPoint are set to -1.

Without the methods, the code should look as follows:

```
package com.verify;

public class IntersectVerify {
    private static int m_a = 1;
    private static int m_b = 2;
    private static int m_c = 1;

    private static int m_m = 2;
    private static int m_y = 5;

    private static boolean m_result = false;
    private static int valAtPoint;
    private static int m_absValue;
}
```

6. We will now add the following methods to the class¹. You can copy and paste the code below to your java file (One of the methods is buggy! Copy and paste it anyway, and we'll try to fix the bug in the next section):

findLinearValue(): Given the slope m , y-intercept y , and an input x , this function returns the value of the linear expression $mx + b$. The code looks as follows:

```
public static int findLinearValue(int m, int y, int x) {  
    int val = 0;  
    val = m*x;  
    val = val + y;  
    return val;  
}
```

findQuadraticValue(): Given the co-efficients a , b , and c , and an input x , this function returns the value of the quadratic expression $ax^2 + bx + c$. This function assumes that a is not 0. The code is shown below:

```
public static int findQuadraticValue(int a, int b, int c, int x)  
{  
    int val = 0;  
    val = a*x*x;  
    val = val + b*x;  
    val = val + c;  
    return val;  
}
```

sameVal(): Given two integers x and y , this function returns 1 if x is equal to y . Otherwise, it returns 0. The implementation is as follows:

```
public static int sameVal(int x, int y) {  
    if (x == y) {  
        return 1;  
    }  
    else {  
        return 0;  
    }  
}
```

abs_value(): Finds the absolute value of input x (so the value it returns must be positive). The code is written as follows:

¹ Technically, we can perform this task easily with only one function and a few lines of code, but the procedure has been split into different methods to better illustrate how JML works.

```

public static int abs_value(int x) {
    if (x >= 0) {
        return (-x);
    } else {
        return x;
    }
}

```

7. To write the main function, copy and paste the code below. The main function assumes that `m_a` is not equal to 0. Also, notice that we will be checking if `iPoint = 2` is a solution to the above equation. (There is also a bug in the main function! Just copy and paste the code below and we'll fix the bug later).

```

public static void main(String[] args) {
    int iPoint = 2;

    int linearVal = findLinearValue(m_m, m_y, iPoint);
    int quadVal = findQuadraticValue(0, m_b, m_c, iPoint);

    int same = sameVal(linearVal, quadVal);

    if (same == 1) {
        m_result = true;
        valAtPoint = linearVal;
        m_absValue = abs_value(linearVal);
    }
    else {
        m_result = false;
        valAtPoint = -1;
        m_absValue = -1;
    }
}

```

8. Now, we have to write the specs for each of the above methods based on their descriptions. However, since our specifications will use the values of the member variables, all of which have been declared private, we need to make sure ESC/Java2 has access to these members. We can do this by adding the `/*@ spec public @*/` annotation to each field declaration, as follows:

```

private /*@ spec_public @*/ static int m_a = 1;
private /*@ spec_public @*/ static int m_b = 2;
private /*@ spec_public @*/ static int m_c = 1;

```

```

private /*@ spec_public @*/ static int m_m = 2;
private /*@ spec_public @*/ static int m_y = 5;

private /*@ spec_public @*/ static boolean m_result = false;
private /*@ spec_public @*/ static int valAtPoint;
private /*@ spec_public @*/ static int m_absValue;

```

9. Next, we will write the specifications for the findLinearValue() method. But before we do this, let us recap the three main parts of a specification:

Requires: This refers to the precondition, which is the condition that **must** be satisfied at function entry. In ESC/Java2, we specify this clause using the “requires” annotation.

Modifies: This refers to a list of all the relevant variables in the class that will be modified. In ESC/Java2, we specify this clause using the “assignable” annotation.

Effects: This refers to the postcondition, which is the condition that **must** hold right after the function finishes its execution. It also refers to what the function must do at the very end. For example, the function sqrt(x) in the Math class of Java must return the square root of x at the end of the function. In ESC/Java2, we specify this clause using the “ensures” annotation.

Now, the findLinearValue() function does not impose any constraints on the values of its parameters or the member variables of the class. Thus, we will leave the “requires” clause blank for this function. In addition, the method does not modify the values of any member variables or the values of any variables on the heap. Finally, the function ensures that the return value is the evaluated value of the expression $mx + y$. The annotated findLinearValue() would then look as follows (note how each annotation is ended by a semi colon and begins with //@):

```

//@ assignable \nothing;
//@ ensures \result == m*x + y;
public static int findLinearValue(int m, int y, int x) {
    int val = 0;
    val = m*x;
    val = val + y;
    return val;
}

```

The special value \result refers to the return value of the function.

Note that in ESC/Java2, methods which do not have the assignable clause default to “//@ assignable \everything”, which means ESC/Java2 assumes that these methods could modify any variable. It is therefore important to annotate variables which do not modify variables with “//@ assignable \nothing”.

10. For `findQuadraticValue()`, the description states that the function assumes that the parameter `a` is non-zero; thus, our requires clause must be `a != 0`. In addition, the method does not modify any member variables. Finally, at the end of the function's execution, the return value must be the evaluated value of the expression $ax^2 + bx + c$. Thus, the annotated function will be:

```
//@ requires a != 0;
//@ assignable \nothing;
//@ ensures \result == a*x*x + b*x + c;
public static int findQuadraticValue(int a, int b, int c, int x)
{
    int val = 0;
    val = a*x*x;
    val = val + b*x;
    val = val + c;
    return val;
}
```

11. The `sameVal()` function modifies no variables. Also, it must return 1 if `x` equals `y`, and must return 0 if `x` is not equal to `y`. Note that in JML, the Boolean expression “if `<expr1>`, then `<expr2>`” is written as “`<expr1> ==> <expr2>`” where `<expr1>` and `<expr2>` are Boolean expressions. Thus, the annotations would look as follows:

```
//@ assignable \nothing;
//@ ensures ((x == y) ==> (\result == 1)) || ((x != y) ==>
(\result == 0));
public static int sameVal(int x, int y) {
    if (x == y) {
        return 1;
    }
    else {
        return 0;
    }
}
```

12. For `abs_value()`, no member variables are modified, and the function returns the absolute value of `x`. The specifications are as follows:

```
//@ assignable \nothing;
//@ ensures (\result >= 0) && (x < 0 ==> \result == -x) && (x >=
0 ==> \result == x);
public static int abs_value(int x) {
    if (x >= 0) {
        return (-x);
    }
}
```



```

    } else {
        return x;
    }
}

```

13. Finally, the main() function assumes that the member variable m_a is not equal to 0. In addition it modifies the variables m_result, valAtPoint, and m_absValue. This function does not return any values; however, after it executes, the value of valAtPoint and m_absValue must be -1 if m_result is false, and the value of m_absValue must be non-negative if m_result is true. Thus, the specification would be:

```

/*@ requires m_a != 0;
   //@ assignable m_result, valAtPoint, m_absValue;
   //@ ensures ((m_result == true) ==> (m_absValue >= 0)) &&
   ((m_result == false) ==> (m_absValue == -1 && valAtPoint == -1));
   public static void main(String[] args) {
       int iPoint = 2;

       int linearVal = findLinearValue(m_m, m_y, iPoint);
       int quadVal = findQuadraticValue(0, m_b, m_c, iPoint);

       int same = sameVal(linearVal, quadVal);


       if (same == 1) {
           m_result = true;
           valAtPoint = linearVal;
           m_absValue = abs_value(linearVal);
       }
       else {
           m_result = false;
           valAtPoint = -1;
           m_absValue = -1;
       }
   }
}

```

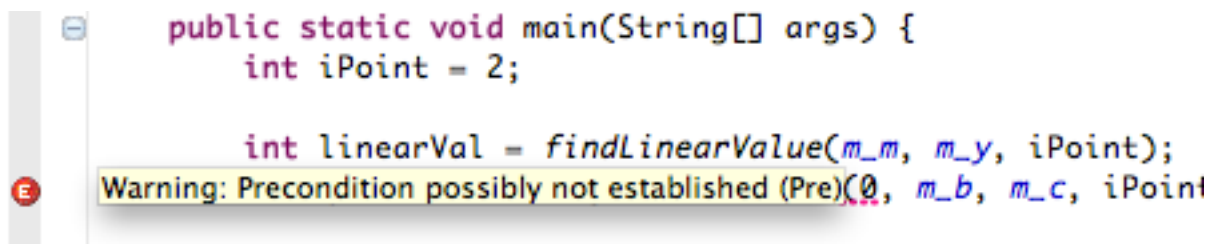
How to Use ESC/Java2

You've now had the chance to write JML specifications. Now, we will verify if the code we've written matches the specifications. This is where ESC/Java2 becomes handy. Before carrying out the following steps, please make sure you've installed and properly configured ESC/Java2 (instructions are shown above).

1. Switch to Verification Perspective by going to Window > Open Perspective > Other... and double clicking "Verification". An additional menu called ESC/Java2 will be added to the top menu.
2. To verify our program against the JML specifications we've written, you should first highlight the intersect_verify project in the Package Explorer. Once you've highlighted the project, on the top menu, click on ESC/Java2 > Run ESC/Java2 with Simplify.

(You can also run ESC/Java2 by clicking on the following symbol: . However, you need to make sure you're running it with Simplify).

3. You will notice two ESC/Java2 warnings appear because of the bugs in the program. The first is in the main() function, and it warns you of a possible precondition violation when the findQuadraticValue() function is called, as shown below (notice the E in the red circle, indicating the offending line):



In our call to findQuadraticValue(), we pass the value 0 to the first parameter, which violates the precondition of that particular function (i.e., a cannot be equal to 0). Of course, the bug here is that we should pass m_a as the first parameter, not 0. If you change the line

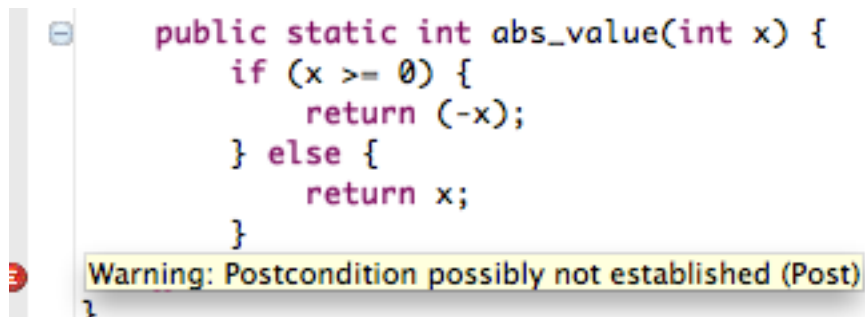
```
int quadVal = findQuadraticValue(0, m_b, m_c, iPoint);
```

to

```
int quadVal = findQuadraticValue(m_a, m_b, m_c, iPoint);
```

and run ESC/Java2 with Simplify again, this warning will disappear.

4. The second warning occurs at the very end of the abs_value() function, which warns of a postcondition violation, as follows:



After debugging the `abs_value()` function, you should notice that the current code returns `-x` when the value of `x` is non-negative, and returns `x` when the value is negative; obviously, this is incorrect, as the function should return `-x` when the value is negative, and `x` when the value is non-negative. Thus, the line

```
if (x >= 0) {
```

should be changed to

```
if (x < 0) {
```

Running ESC/Java2 with Simplify again with this modified code, the postcondition warning should no longer appear.

Additional Notes

- The expression `\old(<expr>)` where `<expr>` is a variable refers to the value of `<expr>` **before** the function began executing. In some cases, you'd want to compare the old value of a variable with the value at function exit for the ensures clause. For example, if you want to ensure that the old value of variable `r` is less than its value at function exit, you should write the following annotation:

```
//@ ensures \old(r) < r;
```

- If the method modifies the values inside an array `A`, the assignable clause should include `A[*]` as one of the variables. If you know that only index 3, for instance, of array `A` will and should be modified by the function, the assignable clause should include the variable `A[3]` instead.