Testing and Debugging

Advanced Programming

ICOM 4015

Lecture 9

Reading: Java Concepts Chapter 10

Chapter Goals

- To learn how to carry out unit tests
- To understand the principles of test case selection and evaluation
- To learn how to use logging
- To become familiar with using a debugger
- To learn strategies for effective debugging

Unit Tests

- The single most important testing tool
- Checks a single method or a set of cooperating methods
- You don't test the complete program that you are developing; you test the classes in isolation
- For each test, you provide a simple class called a *test harness*
- Test harness feeds parameters to the methods being tested

Example: Setting Up Test Harnesses

- **To compute the square root of *a* use a common algorithm:**
  1. Guess a value *x* that might be somewhat close to the desired square root (*x* = *a* is ok)
  2. Actual square root lies between *x* and *a/x*
  3. Take midpoint (*x* + *a/x*) / 2 as a better guess

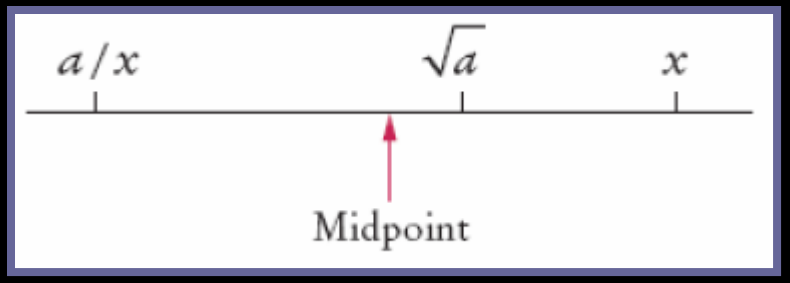

Figure 1: Approximating a Square Root

  4. Repeat the procedure. Stop when two successive approximations are very close to each other

Continued…

Example: Setting Up Test Harnesses

- **Method converges rapidly. Square root of 100:**

```
Guess #1: 50.5
Guess #2: 26.24009900990099
Guess #3: 15.025530119986813
Guess #4: 10.840434673026925
Guess #5: 10.032578510960604
Guess #6: 10.000052895642693
Guess #7: 10.000000000139897
Guess #8: 10.0
```

File RootApproximator.java

```
01: /**
02:    Computes approximations to the square root of
03:    a number, using Heron's algorithm.
04: */
05: public class RootApproximator
06: {
07:    /**
08:       Constructs a root approximator for a given number.
09:       @param aNumber the number from which to extract the
             // square root
10:       (Precondition: aNumber >= 0)
11:    */
12:    public RootApproximator(double aNumber)
13:    {
14:       a = aNumber;
15:       xold = 1;
16:       xnew = a;
17:    }
```

Continued…

File RootApproximator.java

```
18:
19:    /**
20:       Computes a better guess from the current guess.
21:       @return the next guess
22:    */
23:    public double nextGuess()
24:    {
25:       xold = xnew;
26:       if (xold != 0)
27:          xnew = (xold + a / xold) / 2;
28:       return xnew;
29:    }
30:
```

Continued...

File RootApproximator.java

```
31:    /**
32:       Computes the root by repeatedly improving the current
33:       guess until two successive guesses are approximately
             // equal.
34:       @return the computed value for the square root
35:    */
36:    public double getRoot()
37:    {
38:       assert a >= 0;
39:       while (!Numeric.approxEqual(xnew, xold))
40:          nextGuess();
41:       return xnew;
42:    }
43:
44:    private double a; // The number whose square root
                        // is computed
45:    private double xnew; // The current guess
46:    private double xold; // The old guess
47: }
```

File Numeric.java

```
01: /**
02:    A class for useful numeric methods.
03: */
04: public class Numeric
05: {
06:    /**
07:       Tests whether two floating-point numbers are.
08:       equal, except for a roundoff error
09:       @param x a floating-point number
10:       @param y a floating-point number
11:       @return true if x and y are approximately equal
12:    */
13:    public static boolean approxEqual(double x, double y)
14:    {
15:       final double EPSILON = 1E-12;
16:       return Math.abs(x - y) <= EPSILON;
17:    }
18: }
```

File: RootApproximatorTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:    This program prints ten approximations for a square root.
05: */
06: public class RootApproximatorTester
07: {
08:    public static void main(String[] args)
09:    {
10:       System.out.print("Enter a number: ");
11:       Scanner in = new Scanner(System.in);
12:       double x = in.nextDouble();
13:       RootApproximator r = new RootApproximator(x);
14:       final int MAX_TRIES = 10;
15:       for (int tries = 1; tries <= MAX_TRIES; tries++)
16:       {
17:          double y = r.nextGuess();
18:          System.out.println("Guess #" + tries + ": " + y);
19:       }
20:       System.out.println("Square root: " + r.getRoot());
21:    }
22: }
```

Testing the Program

- **Output**

```
Enter a number: 100
Guess #1: 50.5
Guess #2: 26.24009900990099
Guess #3: 15.025530119986813
Guess #4: 10.840434673026925
Guess #5: 10.032578510960604
Guess #6: 10.000052895642693
Guess #7: 10.000000000139897
Guess #8: 10.0
Guess #9: 10.0
Guess #10: 10.0
Square root: 10.0
```


Continued…

Testing the Program

- Does the `RootApproximator` class work correctly for all inputs?
  It needs to be tested with more values
- Re-testing with other values repetitively is not a good idea; the tests are not repeatable
- If a problem is fixed and re-testing is needed, you would need to remember your inputs
- Solution: Write test harnesses that make it easy to repeat unit tests

Self Check

1. What is the advantage of unit testing?
2. Why should a test harness be repeatable?

Answers

1. It is easier to test methods and classes in isolation than it is to understand failures in a complex program.
2. It should be easy and painless to repeat a test after fixing a bug.

Providing Test Input

- There are various mechanisms for providing test cases
- One mechanism is to hardwire test inputs into the test harness
- Simply execute the test harness whenever you fix a bug in the class that is being tested
- Alternative: place inputs on a file instead

File RootApproximatorHarness1.java

```
01: /**
02:    This program computes square roots of selected input
          // values.
03: */
04: public class RootApproximatorHarness1
05: {
06:    public static void main(String[] args)
07:    {
08:       double[] testInputs = { 100, 4, 2, 1, 0.25, 0.01 };
09:       for (double x : testInputs)
10:       {
11:          RootApproximator r = new RootApproximator(x);
12:          double y = r.getRoot();
13:          System.out.println("square root of " + x
14:                + " = " + y);
15:       }
16:    }
17: }
```

File RootApproximatorHarness1.java

- **Output**

```
square root of 100.0 = 10.0
square root of 4.0 = 2.0
square root of 2.0 = 1.414213562373095
square root of 1.0 = 1.0
square root of 0.25 = 0.5
square root of 0.01 = 0.1
```

Providing Test Input

- You can also generate test cases automatically
- For few possible inputs, feasible to run through (representative) number of them with a loop

File RootApproximatorHarness2.java

```
01: /**
02:    This program computes square roots of input values
03:    supplied by a loop.
04: */
05: public class RootApproximatorHarness2
06: {
07:    public static void main(String[] args)
08:    {
09:       final double MIN = 1;
10:       final double MAX = 10;
11:       final double INCREMENT = 0.5;
12:       for (double x = MIN; x <= MAX; x = x + INCREMENT)
13:       {
14:          RootApproximator r = new RootApproximator(x);
15:          double y = r.getRoot();
16:          System.out.println("square root of " + x
17:                + " = " + y);
18:       }
19:    }
20: }
```

Continued…

File RootApproximatorHarness2.java

- **Output**

```
square root of 1.0 = 1.0
square root of 1.5 = 1.224744871391589
square root of 2.0 = 1.414213562373095
. . .
square root of 9.0 = 3.0
square root of 9.5 = 3.082207001484885
square root of 10.0 = 3.162277660168379
```

Providing Test Input

- Previous test restricted to small subset of values
- Alternative: random generation of test cases

File RootApproximatorHarness3.java

```
01: import java.util.Random;
03: /**
04:    This program computes square roots of random inputs.
05: */
06: public class RootApproximatorHarness3
07: {
08:    public static void main(String[] args)
09:    {
10:       final double SAMPLES = 100;
11:       Random generator = new Random();
12:       for (int i = 1; i <= SAMPLES; i++)
13:       {
14:          // Generate random test value
15:
16:          double x = 1000 * generator.nextDouble();
17:          RootApproximator r = new RootApproximator(x);
18:          double y = r.getRoot();
19:          System.out.println("square root of " + x
20:                + " = " + y);
21:       }
22:    }
23: }
```

File RootApproximatorHarness3.java

- **Output**

```
square root of 810.4079626570873 = 28.467665212607223
square root of 480.50291114306344 = 21.9203766195534
square root of 643.5463246844379 = 25.36821485017103
square root of 506.5708496713842 = 22.507128863348704
square root of 539.6401504334708 = 23.230156057019308
square root of 795.0220214851004 = 28.196134867834285
. . .
```

Providing Test Input

- Selecting good test cases is an important skill for debugging programs
- Test all features of the methods that you are testing
- Test typical test cases
  100, 1/4, 0.01, 2, 10E12, for the `SquareRootApproximator`
- Test boundary test cases: test cases that are at the boundary of acceptable inputs
  0, for the `SquareRootApproximator`

Providing Test Input

- Programmers often make mistakes dealing with boundary conditions

  Division by zero, extracting characters from empty strings, and accessing null pointers

- Gather negative test cases: inputs that you expect program to reject

  Example: square root of -2. Test passes if harness terminates with assertion failure (if assertion checking is enabled)

Reading Test Inputs From a File

- More elegant to place test values in a file
- Input redirection:

```
java Program < data.txt
```

- Some IDEs do not support input redirection. Then, use command window (shell).
- Output redirection:

```
java Program > output.txt
```

File RootApproximatorHarness4.java

```
01: import java.util.Scanner;
03: /**
04:    This program computes square roots of inputs supplied
05:    through System.in.
06: */
07: public class RootApproximatorHarness4
08: {
09:    public static void main(String[] args)
10:    {
11:       Scanner in = new Scanner(System.in);
12:       boolean done = false;
13:       while (in.hasNextDouble())
14:       {
15:          double x = in.nextDouble();
16:          RootApproximator r = new RootApproximator(x);
17:          double y = r.getRoot();
18:
19:          System.out.println("square root of " + x
20:                + " = " + y);
21:       }
22:    }
23: }
```

Reading Test Inputs From a File

- **File test.in:**

```
1 100
2 4
3 2
4 1
5 0.25
6 0.01
```

Run the program:

```
java RootApproximatorHarness4 < test.in > test.out
```


Continued…

Reading Test Inputs From a File

- File `test.out`:

```
1 square root of 100.0 = 10.0
2 square root of 4.0 = 2.0
3 square root of 2.0 = 1.414213562373095
4 square root of 1.0 = 1.0
5 square root of 0.25 = 0.5
6 square root of 0.01 = 0.1
```

Self Test

1. How can you repeat a unit test without having to retype input values?
2. Why is it important to test boundary cases?

Answers

1. By putting the values in a file, or by generating them programmatically.
2. Programmers commonly make mistakes when dealing with boundary conditions.

Test Case Evaluation

- How do you know whether the output is correct?
- Calculate correct values by hand
  E.g., for a payroll program, compute taxes manually
- Supply test inputs for which you know the answer
  E.g., square root of 4 is 2 and square root of 100 is 10

Continued…

Test Case Evaluation

- Verify that the output values fulfill certain properties
  E.g., square root squared = original value
- Use an Oracle: a slow but reliable method to compute a result for testing purposes
  E.g., use `Math.pow` to slower calculate $x^{1/2}$ (equivalent to the square root of x)

File RootApproximatorHarness5.java

```
01: import java.util.Random;
02:
03: /**
04:    This program verifies the computation of square root
       // values
05:    by checking a mathematical property of square roots.
06: */
07: public class RootApproximatorHarness5
08: {
09:    public static void main(String[] args)
10:    {
11:       final double SAMPLES = 100;
12:       int passcount = 0;
13:       int failcount = 0;
14:       Random generator = new Random();
15:       for (int i = 1; i <= SAMPLES; i++)
16:       {
```

Continued…

File RootApproximatorHarness5.java

```
17:          // Generate random test value
18:
19:          double x = 1000 * generator.nextDouble();
20:          RootApproximator r = new RootApproximator(x);
21:          double y = r.getRoot();
22:
23:          // Check that test value fulfills square property
24:
25:          if (Numeric.approxEqual(y * y, x))
26:          {
27:             System.out.print("Test passed: ");
28:             passcount++;
29:          }
30:          else
31:          {
32:             System.out.print("Test failed: ");
33:             failcount++;
34:          }
```

Continued…

File RootApproximatorHarness5.java

```
35:
36:          System.out.println("x = " + x
37:                + ", root squared = " + y * y);
38:       }
39:       System.out.println("Pass: " + passcount);
40:       System.out.println("Fail: " + failcount);
41:    }
42: }
```

File RootApproximatorHarness5.java

- **Output**

```
Test passed: x = 913.6505141736327, root squared = 913.6505141736328
Test passed: x = 810.4959723987972, root squared = 810.4959723987972
Test passed: x = 503.84630929985883, root squared = 503.8463092998589
Test passed: x = 115.48850960063153, root squared = 115.48850960063153
Test passed: x = 384.973238438713, root squared = 384.973238438713
. . .
Pass: 100
Fail: 0
```

File RootApproximatorHarness6.java

```
01: import java.util.Random;
02:
03: /**
04:    This program verifies the computation of square root
          // values
05:    by using an oracle.
06: */
07: public class RootApproximatorHarness6
08: {
09:    public static void main(String[] args)
10:    {
11:       final double SAMPLES = 100;
12:       int passcount = 0;
13:       int failcount = 0;
14:       Random generator = new Random();
15:       for (int i = 1; i <= SAMPLES; i++)
16:       {
17:          // Generate random test value
18:
```

Continued...

File RootApproximatorHarness6.java

```
19:          double x = 1000 * generator.nextDouble();
20:          RootApproximator r = new RootApproximator(x);
21:          double y = r.getRoot();
22:
23:          double oracleValue = Math.pow(x, 0.5);
24:
25:          // Check that test value approximately equals
             // oracle value
26:
27:          if (Numeric.approxEqual(y, oracleValue))
28:          {
29:             System.out.print("Test passed: ");
30:             passcount++;
31:          }
32:          else
33:          {
34:             System.out.print("Test failed: ");
35:             failcount++;
36:          }
```

Continued…

File RootApproximatorHarness6.java

```
37:             System.out.println("square root = " + y
38:                   + ", oracle = " + oracleValue);
39:          }
40:          System.out.println("Pass: " + passcount);
41:          System.out.println("Fail: " + failcount);
42:       }
43: }
```

File RootApproximatorHarness5.java

- **Output**

```
Test passed: square root = 718.3849112194539, oracle = 718.3849112194538
Test passed: square root = 641.2739466673618, oracle = 641.2739466673619
Test passed: square root = 896.3559528159169, oracle = 896.3559528159169
Test passed: square root = 591.4264541724909, oracle = 591.4264541724909
Test passed: square root = 721.029957736384, oracle = 721.029957736384
. . .
Pass: 100
Fail: 0
```

Self Test

1. Your task is to test a class that computes sales taxes for an Internet shopping site. Can you use an oracle?
2. Your task is to test a method that computes the area of an arbitrary polygon. Which polygons with known areas can you use as test inputs?

Answers

1. Probably not–there is no easily accessible but slow mechanism to compute sales taxes. You will probably need to verify the calculations by hand.
2. There are well-known formulas for the areas of triangles, rectangles, and regular *n*-gons.

Regression Testing

- Save test cases
- Use saved test cases in subsequent versions
- A test suite is a set of tests for repeated testing
- Cycling = bug that is fixed but reappears in later versions
- Regression testing: repeating previous tests to ensure that known failures of prior versions do not appear in new versions

Test Coverage

- Black-box testing: test functionality without consideration of internal structure of implementation
- White-box testing: take internal structure into account when designing tests
- Test coverage: measure of how many parts of a program have been tested
- Make sure that each part of your program is exercised at least once by one test case
  E.g., make sure to execute each branch in at least one test case

Test Coverage

- **Tip: write first test cases before program is written completely → gives insight into what program should do**
- **Modern programs can be challenging to test**
  - Graphical user interfaces (use of mouse)
  - Network connections (delay and failures)
  - There are tools to automate testing in this scenarios
  - Basic principles of regression testing and complete coverage still hold

Self Test

1. Suppose you modified the code for a method. Why do you want to repeat tests that already passed with the previous version of the code?
2. Suppose a customer of your program finds an error. What action should you take beyond fixing the error?

Answers

1. It is possible to introduce errors when modifying code.
2. Add a test case to the test suite that verifies that the error is fixed.

Unit Testing With JUnit

- http://junit.org
- Built into some IDEs like BlueJ and Eclipse
- Philosophy: whenever you implement a class, also make a companion test class

Continued…

Unit Testing With JUnit

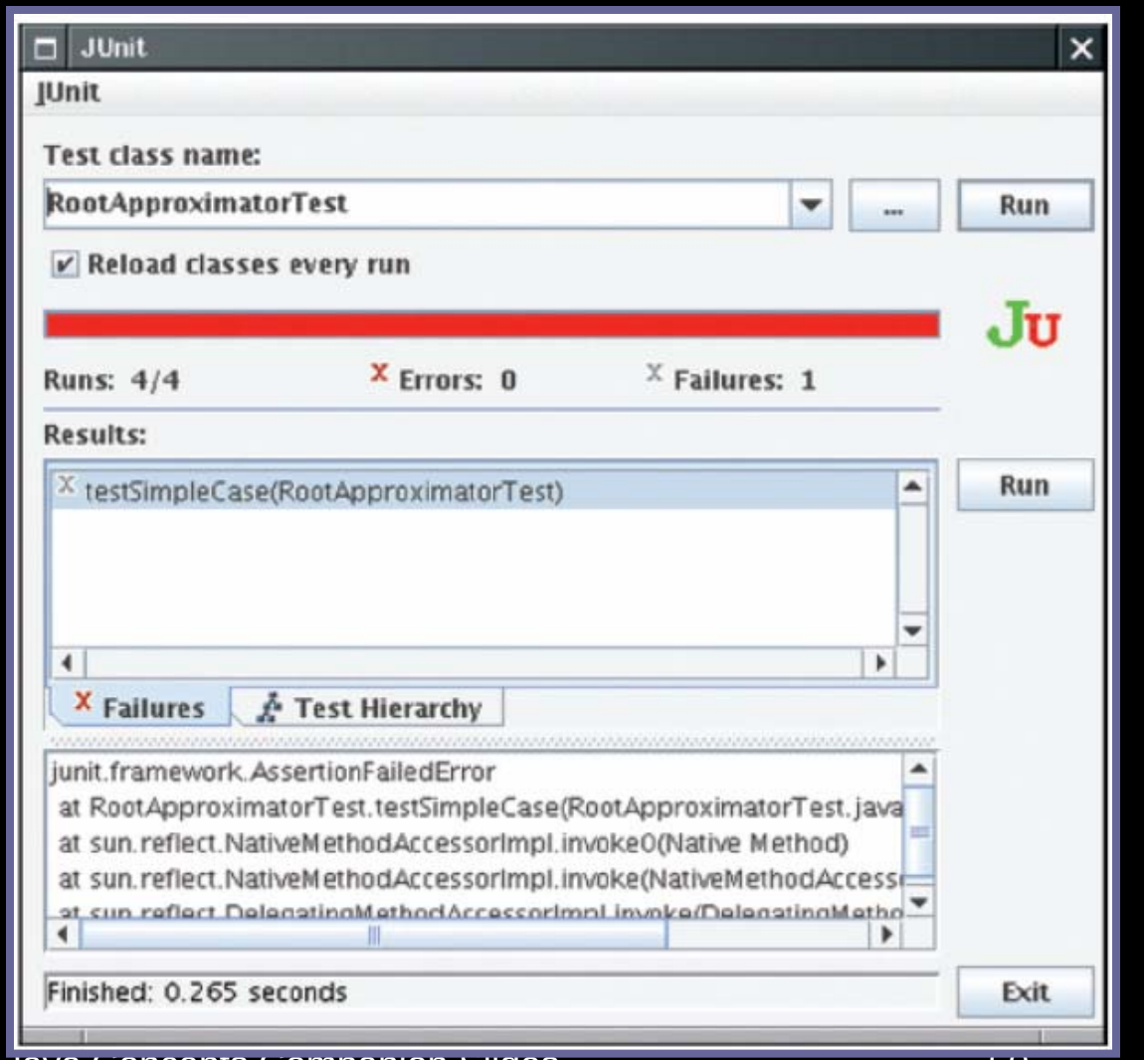

Figure 2:
Unit Testing with JUnit

Program Trace

- **Messages that show the path of execution**

```
if (status == SINGLE)
{
System.out.println("status is SINGLE");
. . .
}
. . .
```


Continued…

Program Trace

- Drawback: Need to remove them when testing is complete, stick them back in when another error is found
- Solution: use the `Logger` class to turn off the trace messages without removing them from the program

Logging

- Logging messages can be deactivated when testing is complete
- Use global object Logger.global
- Log a message

```
Logger.global.info("status is SINGLE");
```


Continued…

Logging

- By default, logged messages are printed. Turn them off with

```
Logger.global.setLevel(Level.OFF);
```

- Logging can be a hassle (should not log too much nor too little)
- Some programmers prefer debugging (next section) to logging

Logging

- When tracing execution flow, the most important events are entering and exiting a method
- At the beginning of a method, print out the parameters:

```
public TaxReturn(double anIncome, int aStatus)
{
   Logger.global.info("Parameters: anIncome = " + anIncome
      + " aStatus = " + aStatus);
   . . .
}
```

Logging

- **At the end of a method, print out the return value:**

```
public double getTax()
{
   . . .
   Logger.global.info("Return value = " + tax);
   return tax;
}
```

Self Check

1. Should logging be activated during testing or when a program is used by its customers?
2. Why is it better to send trace messages to `Logger.global` than to `System.out`?

Answers

1. Logging messages report on the internal workings of your program–your customers would not want to see them. They are intended for testing only.
2. It is easy to deactivate `Logger.global` when you no longer want to see the trace messages, and to reactivate it when you need to see them again.

Using a Debugger

- Debugger = program to run your program and analyze its run-time behavior
- A debugger lets you stop and restart your program, see contents of variables, and step through it
- The larger your programs, the harder to debug them simply by logging

Continued…

Using a Debugger

- **Debuggers can be part of your IDE (Eclipse, BlueJ) or separate programs (JSwat)**
- **Three key concepts:**
  - Breakpoints
  - Single-stepping
  - Inspecting variables

The Debugger Stopping at a Breakpoint

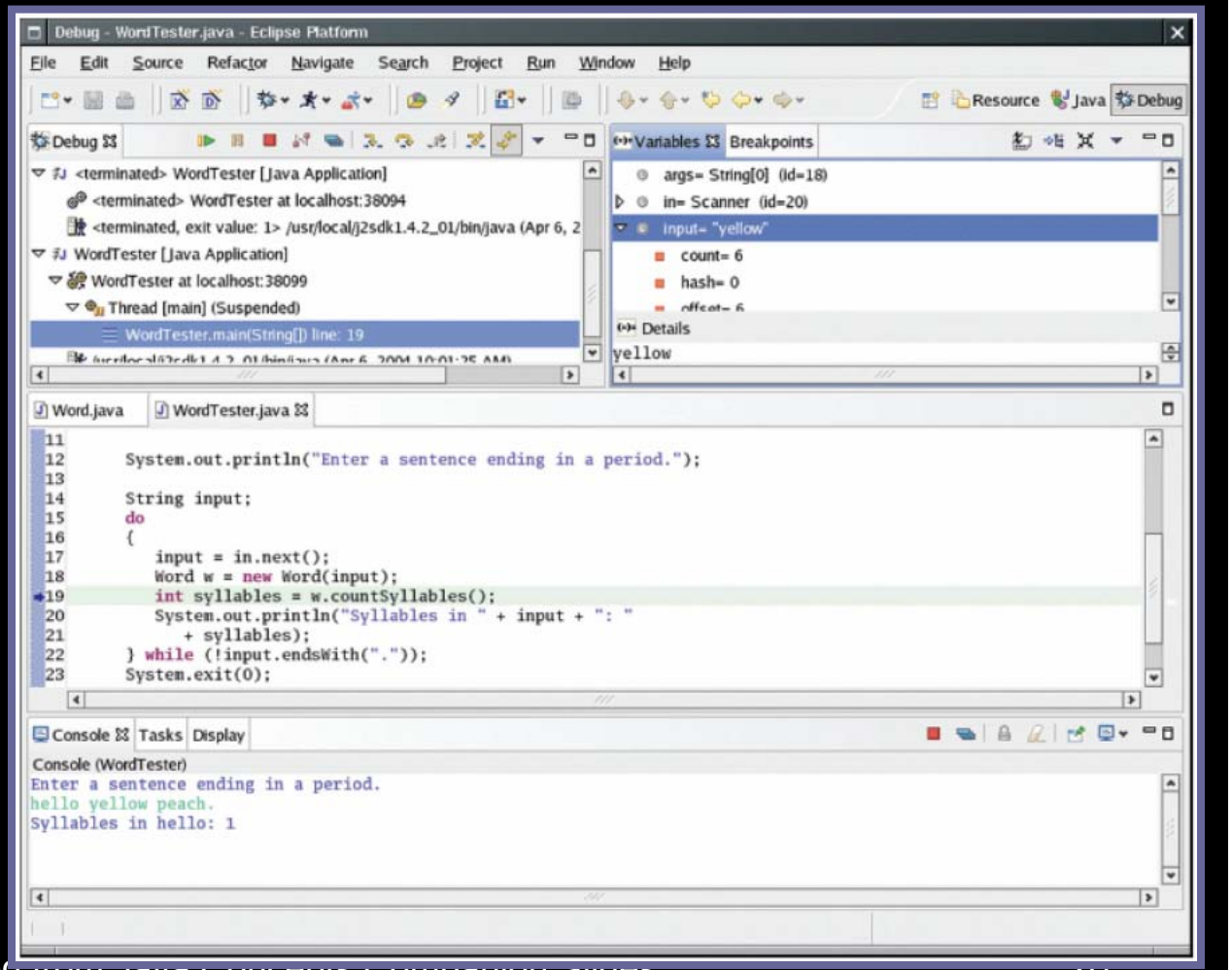

Figure 3: Stopping at a Breakpoint

Inspecting Variables

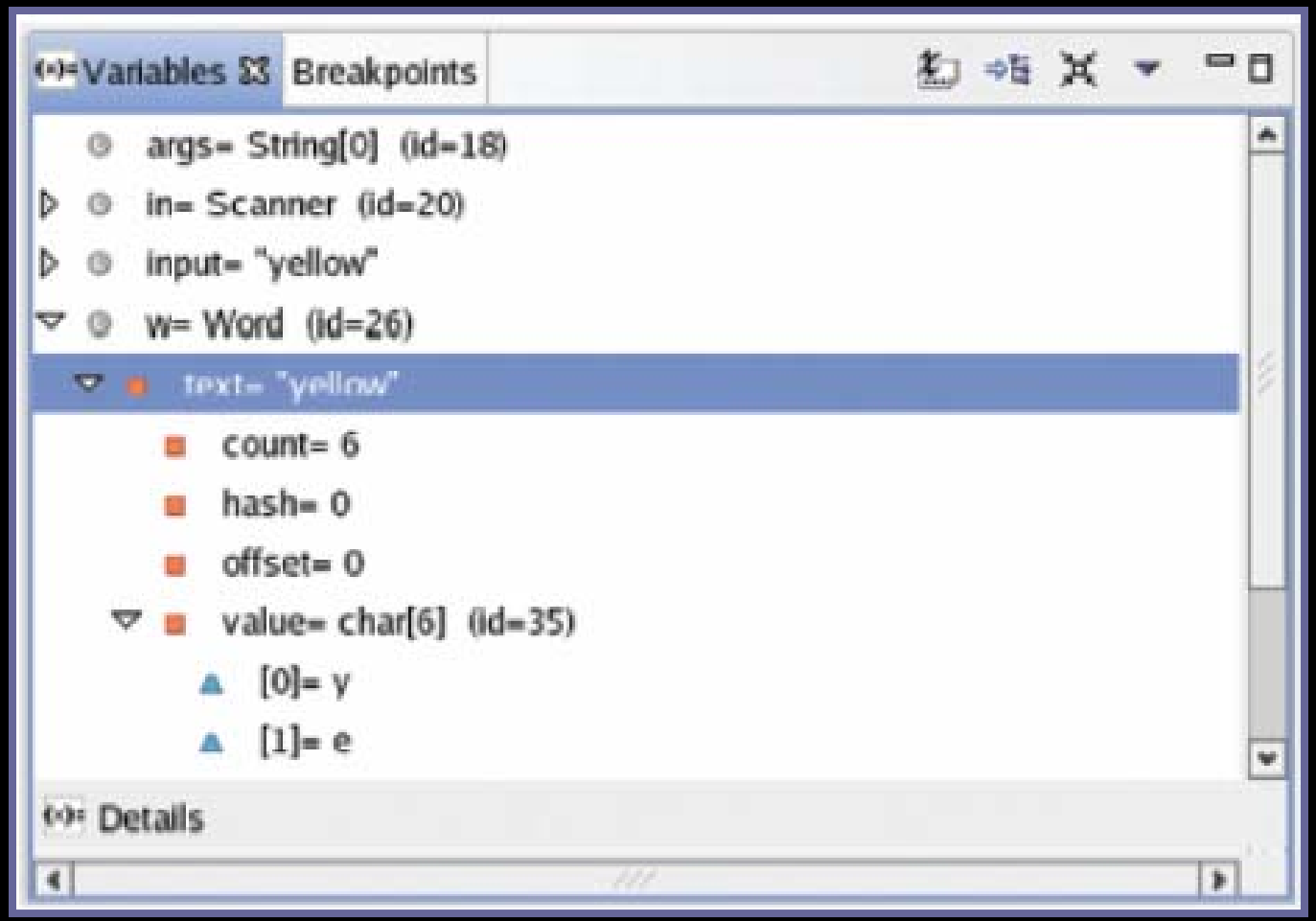

Figure 4: Inspecting Variables

Debugging

- **Execution is suspended whenever a breakpoint is reached**
- **In a debugger, a program runs at full speed until it reaches a breakpoint**
- **When execution stops you can:**
  - Inspect variables
  - Step through the program a line at a time
  - Or, continue running the program at full speed until it reaches the next breakpoint

Continued…

Debugging

- **When program terminates, debugger stops as well**
- **Breakpoints stay active until you remove them**
- **Two variations of single-step command:**
  - Step Over: skips method calls
  - Step Into: steps inside method calls

Single-Step Example

- **Current line:**

```
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " + syllables);
```

- **When you step over method calls, you get to the next line:**

```
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " + syllables);
```

- **However, if you step into method calls, you enter the first line of the countSyllables method:**

```
public int countSyllables() {
 nt count = 0; int end = text.length() - 1;

  . . .
}
```

Single-Step Example

- However, if you step into method calls, you enter the first line of the `countSyllables` method

```
public int countSyllables()
{
   int count = 0;
   int end = text.length() - 1;
   . . .
}
```

Self Check

1. In the debugger, you are reaching a call to `System.out.println`. Should you step into the method or step over it?
2. In the debugger, you are reaching the beginning of a long method with a couple of loops inside. You want to find out the return value that is computed at the end of the method. Should you set a breakpoint, or should you step through the method?

Answers

1. You should step over it because you are not interested in debugging the internals of the `println` method.
2. You should set a breakpoint. Stepping through loops can be tedious.

Sample Debugging Session

- `Word` class counts syllables in a word
- Each group of adjacent vowels (a, e, i, o, u, y) counts as one syllable
- However, an e at the end of a word doesn't count as a syllable
- If algorithm gives count of 0, increment to 1
- Constructor removes non-letters at beginning and end

File Word.java

```
01: /**
02:    This class describes words in a document.
03: */
04: public class Word
05: {
06:    /**
07:       Constructs a word by removing leading and trailing non-
08:       letter characters, such as punctuation marks.
09:       @param s the input string
10:    */
11:    public Word(String s)
12:    {
13:       int i = 0;
14:       while (i < s.length() && !Character.isLetter(s.charAt(i)))
15:          i++;
16:       int j = s.length() - 1;
17:       while (j > i && !Character.isLetter(s.charAt(j)))
18:          j--;
```

Continued...

File Word.java

```
19:          text = s.substring(i, j);
20:       }
21:
22:       /**
23:          Returns the text of the word, after removal of the
24:          leading and trailing non-letter characters.
25:          @return the text of the word
26:       */
27:       public String getText()
28:       {
29:          return text;
30:       }
31:
32:       /**
33:          Counts the syllables in the word.
34:          @return the syllable count
35:       */
```

Continued…

File Word.java

```
36:    public int countSyllables()
37:    {
38:       int count = 0;
39:       int end = text.length() - 1;
40:       if (end < 0) return 0; // The empty string has no
          // syllables
41:
42:       // An e at the end of the word doesn't count as a vowel
43:       char ch = Character.toLowerCase(text.charAt(end));
44:       if (ch == 'e') end--;
45:
46:       boolean insideVowelGroup = false;
47:       for (int i = 0; i <= end; i++)
48:       {
49:          ch = Character.toLowerCase(text.charAt(i));
50:          String vowels = "aeiouy";
51:          if (vowels.indexOf(ch) >= 0)
52:          {
```

Continued…

File Word.java

```
53:             // ch is a vowel
54:             if (!insideVowelGroup)
55:             {
56:                // Start of new vowel group
57:                count++;
58:                insideVowelGroup = true;
59:             }
60:          }
61:       }
62:
63:       // Every word has at least one syllable
64:       if (count == 0)
65:          count = 1;
66:
67:       return count;
68:    }
69:
70:    private String text;
71: }
```

File WordTester.java

```
01: import java.util.Scanner;
02:
03: /**
04:    This program tests the countSyllables method of the Word
          // class.
05: */
06: public class WordTester
07: {
08:    public static void main(String[] args)
09:    {
10:       Scanner in = new Scanner(System.in);
11:
12:       System.out.println("Enter a sentence ending in a
                // period.");
13:
14:       String input;
15:       do
16:       {
```

Continued…

File WordTester.java

```
17:             input = in.next();
18:             Word w = new Word(input);
19:             int syllables = w.countSyllables();
20:             System.out.println("Syllables in " + input + ": "
21:                + syllables);
22:          }
23:          while (!input.endsWith("."));
24:       }
25: }
```

Debug the Program

- Buggy output (for input `"hello yellow peach"`):

```
Syllables in hello: 1
Syllables in yellow: 1
Syllables in peach: 1
```

- Set breakpoint in first line of `countSyllables` of `Word` class
- Start program, supply input. Program stops at breakpoint
- Method checks if final letter is `'e'`

Continued…

Debug the Program

Figure 5:
Debugging the `CountSyllables` Method

Continued…

Debug the Program

- Check if this works: step to line where check is made and inspect variable `ch`
- Should contain final letter but contains `'l'`

More Problems Found

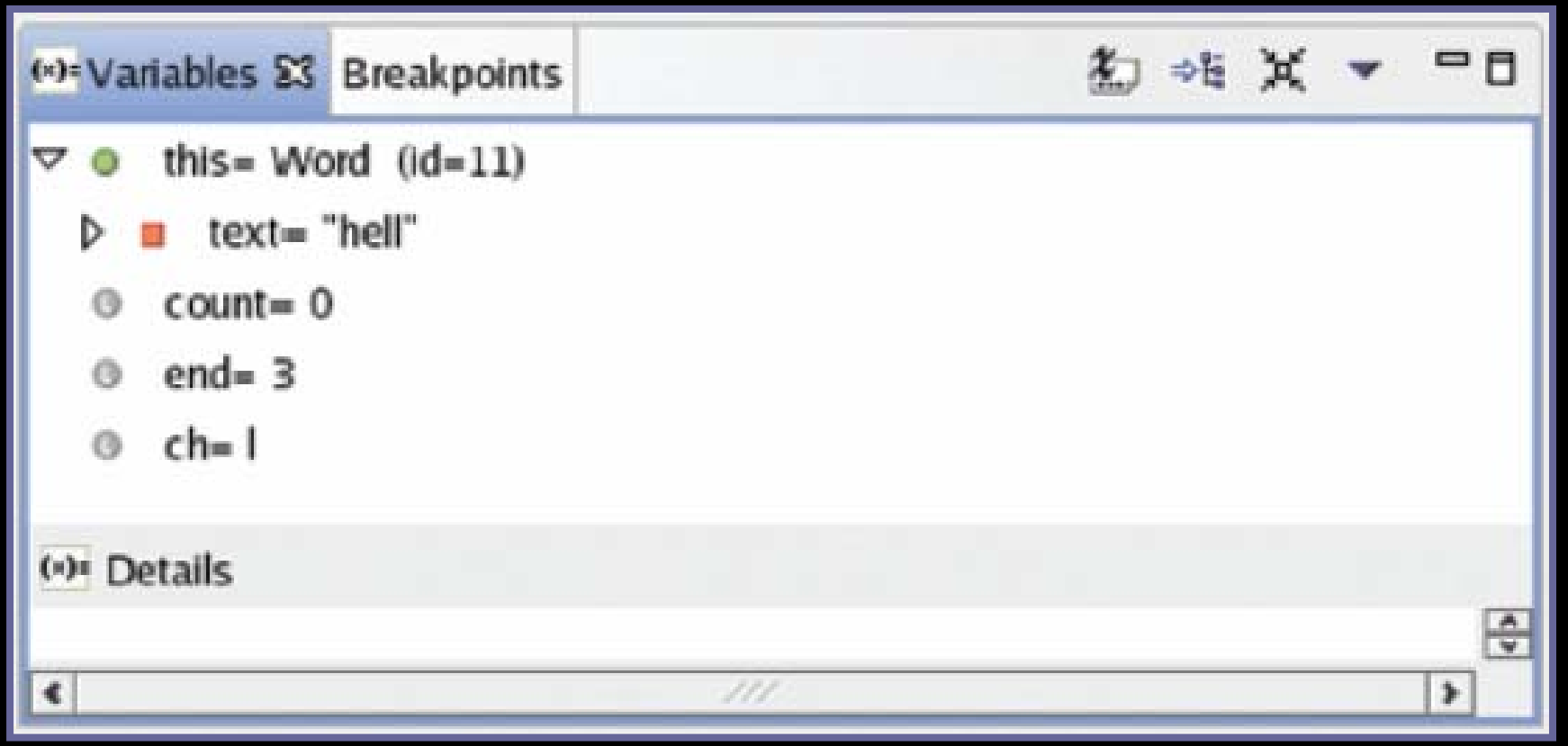

Figure 6:
The Current Values of the Local and Instance Variables

Continued…

More Problems Found

- end is set to 3, not 4
- `text` contains `"hell"`, not `"hello"`
- No wonder `countSyllables` returns 1
- Culprit is elsewhere
- Can't go back in time
- Restart and set breakpoint in `Word` constructor

Debugging the Word Constructor

- Supply "`hello`" input again
- Break past the end of second loop in constructor
- Inspect `i` and `j`
- They are 0 and 4–makes sense since the input consists of letters
- Why is `text` set to "`hell`"?

Continued…

Debugging the Word Constructor

- Off-by-one error: Second parameter of `substring` is the first position *not* to include
- `text = substring(i, j);`
  should be
  `text = substring(i, j + 1);`

Debugging the Word Constructor

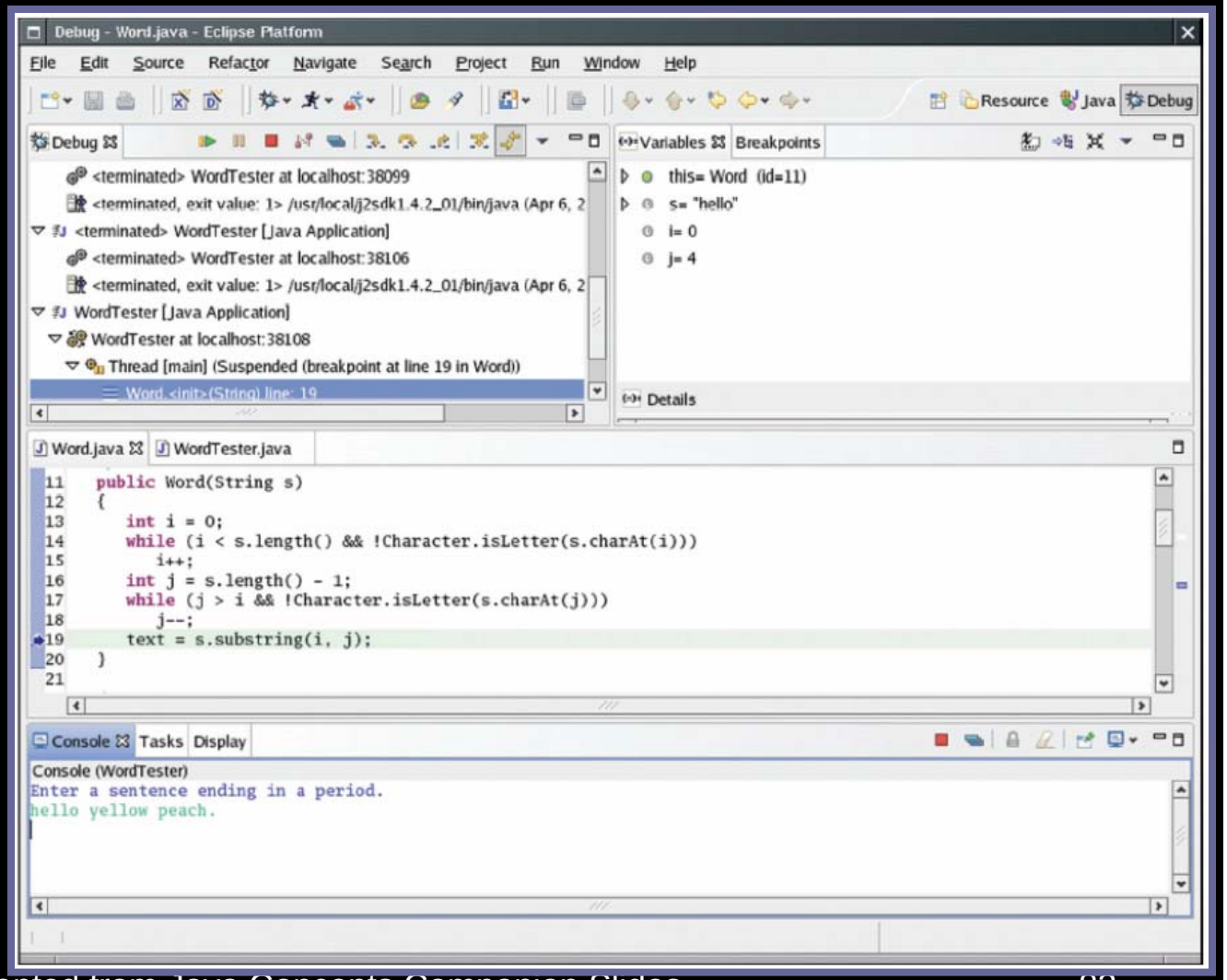

Figure 7: Debugging the Word Constructor

Another Error

- Fix the error
- Recompile
- Test again:

```
Syllables in hello: 1
Syllables in yellow: 1
Syllables in peach: 1
```

- Oh no, it's still not right

Continued…

Another Error

- Start debugger
- Erase all old breakpoints and set a breakpoint in `countSyllables` method
- Supply input `"hello"`

Debugging `CountSyllables` (again)

- **Break in the beginning of `countSyllables`. Then, single-step through loop**

```
boolean insideVowelGroup = false;
for (int i = 0; i <= end; i++)
{
   ch = Character.toLowerCase(text.charAt(i));
   if ("aeiouy".indexOf(ch) >= 0)
   {
      // ch is a vowel
      if (!insideVowelGroup)
      {
         // Start of new vowel group
         count++;
         insideVowelGroup = true;
      }
   }
}
```

Debugging `CountSyllables` (again)

- First iteration ('`h`'): skips test for vowel
- Second iteration ('`e`'): passes test, increments `count`
- Third iteration ('`l`'): skips test
- Fifth iteration ('`o`'): passes test, but second `if` is skipped, and `count` is not incremented

Fixing the Bug

- `insideVowelGroup` was never reset to false
- Fix

```
if ("aeiouy".indexOf(ch) >= 0)
{
   . . .
}
else insideVowelGroup = false;
```


Continued…

Fixing the Bug

- **Retest: All test cases pass**

```
Syllables in hello: 2
Syllables in yellow: 2
Syllables in peach.: 1
```

- **Is the program now bug-free? The debugger can't answer that.**

Self Check

1. What caused the first error that was found in this debugging session?
2. What caused the second error? How was it detected?

Answers

1. The programmer misunderstood the second parameter of the substring method–it is the index of the first character not to be included in the substring.
2. The second error was caused by failing to reset `insideVowelGroup` to false at the end of a vowel group. It was detected by tracing through the loop and noticing that the loop didn't enter the conditional statement that increments the vowel count.

The First Bug

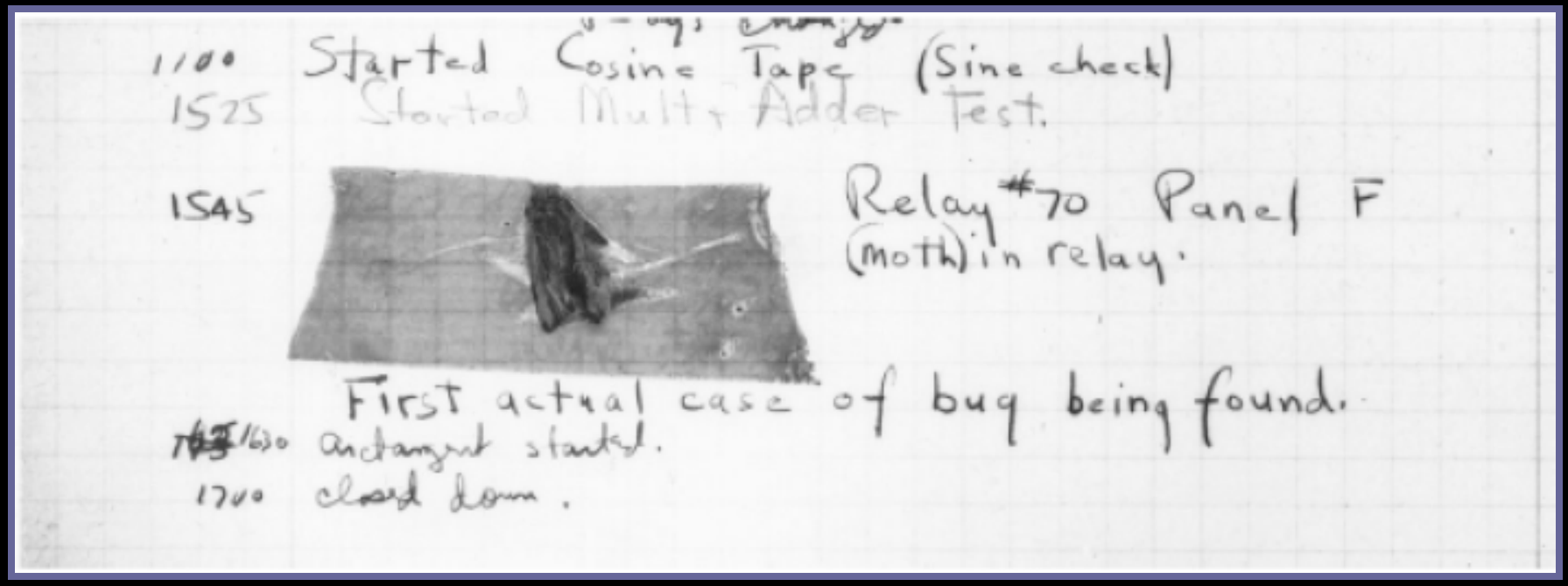

Figure 8:
The First Bug

Therac-25 Facility

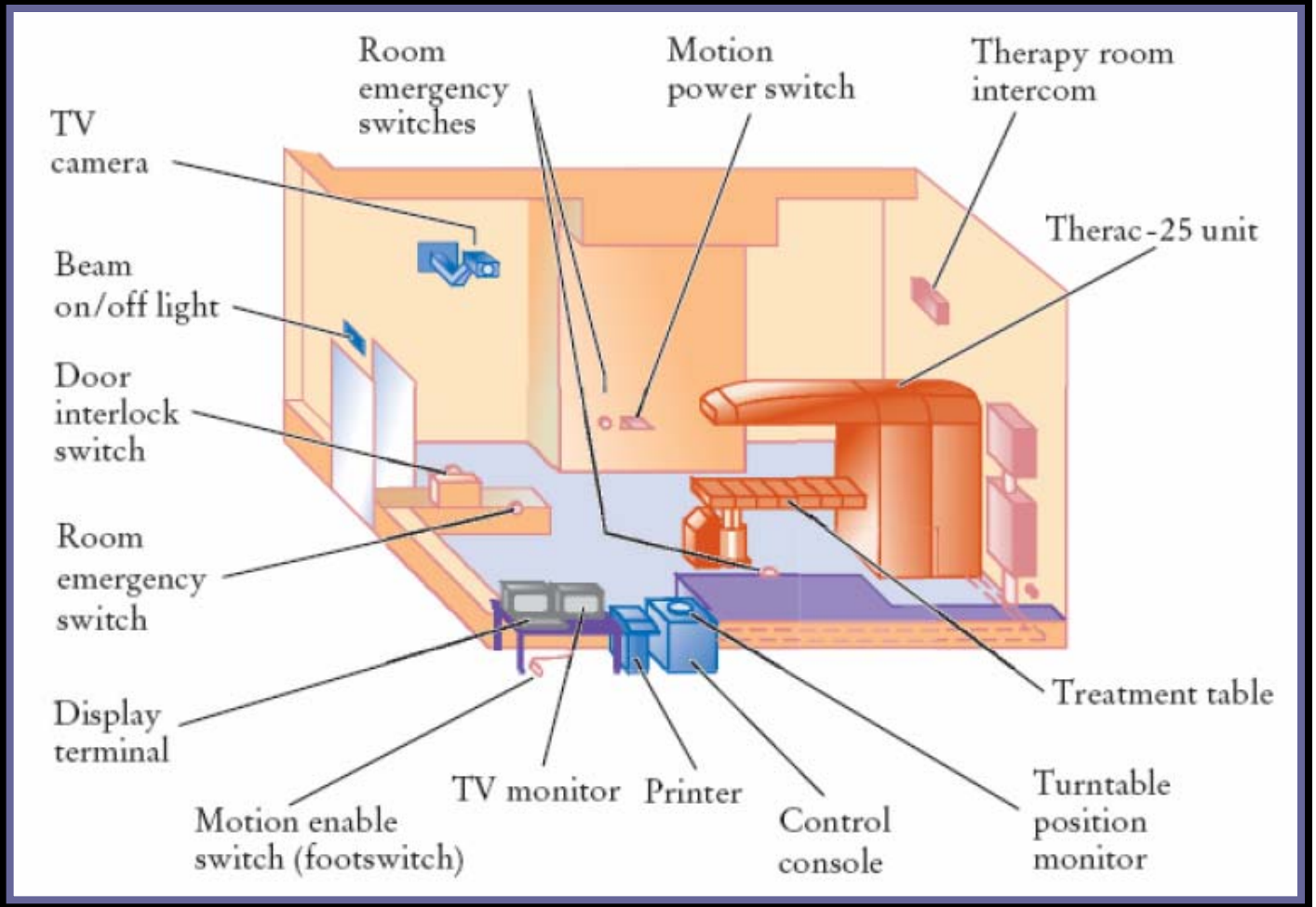

Figure 9:
Typical Therac-25 Facility