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

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

Figure 1: Approximating a Square Root
Example: Setting Up Test Harnesses

- Method converges rapidly. Square root of 100:

<table>
<thead>
<tr>
<th>Guess</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>50.5</td>
</tr>
<tr>
<td>#2</td>
<td>26.24009900990099</td>
</tr>
<tr>
<td>#3</td>
<td>15.025530119986813</td>
</tr>
<tr>
<td>#4</td>
<td>10.840434673026925</td>
</tr>
<tr>
<td>#5</td>
<td>10.032578510960604</td>
</tr>
<tr>
<td>#6</td>
<td>10.000052895642693</td>
</tr>
<tr>
<td>#7</td>
<td>10.000000000139897</td>
</tr>
<tr>
<td>#8</td>
<td>10.0</td>
</tr>
</tbody>
</table>
/**
 * Computes approximations to the square root of
 * a number, using Heron's algorithm.
 */

public class RootApproximator {
    /**
     * Constructs a root approximator for a given number.
     * @param aNumber the number from which to extract the
     *                 // square root
     *               (Precondition: aNumber >= 0)
     */
    public RootApproximator(double aNumber) {
        a = aNumber;
        xold = 1;
        xnew = a;
    }
}

Continued…
```java
    /**
     * Computes a better guess from the current guess.
     * @return the next guess
     */
    public double nextGuess()
    {
        xold = xnew;
        if (xold != 0)
            xnew = (xold + a / xold) / 2;
        return xnew;
    }
```
File RootApproximator.java

```java
31:  /**
32:      Computes the root by repeatedly improving the current
33:      guess until two successive guesses are approximately
34:      // equal.
35:      @return the computed value for the square root
36: */
37:  public double getRoot()
38:  {
39:      assert a >= 0;
40:      while (!Numeric.approxEqual(xnew, xold))
41:         nextGuess();
42:      return xnew;
43:  }
44:  private double a; // The number whose square root
45:                          // is computed
46:  private double xnew; // The current guess
47:  private double xold; // The old guess
48:  }
```
/**
 * A class for useful numeric methods.
 */

public class Numeric {

  /**
   * Tests whether two floating-point numbers are equal, except for a roundoff error.
   * @param x a floating-point number
   * @param y a floating-point number
   * @return true if x and y are approximately equal
   */
  public static boolean approxEqual(double x, double y) {
    final double EPSILON = 1E-12;
    return Math.abs(x - y) <= EPSILON;
  }
}
import java.util.Scanner;

/**
 * This program prints ten approximations for a square root.
 */

public class RootApproximatorTester {
    public static void main(String[] args) {
        System.out.print("Enter a number: ");
        Scanner in = new Scanner(System.in);
        double x = in.nextDouble();
        RootApproximator r = new RootApproximator(x);
        final int MAX_TRIES = 10;
        for (int tries = 1; tries <= MAX_TRIES; tries++) {
            double y = r.nextGuess();
            System.out.println("Guess "+ tries + ": "+ y);
        }
        System.out.println("Square root: "+ r.getRoot());
    }
}
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
This program computes square roots of selected input values.

```java
public class RootApproximatorHarness1 {
    public static void main(String[] args) {
        double[] testInputs = {100, 4, 2, 1, 0.25, 0.01};
        for (double x : testInputs) {
            RootApproximator r = new RootApproximator(x);
            double y = r.getRoot();
            System.out.println("square root of "+x +" = " + y);
        }
    }
}
```
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
/**
 * This program computes square roots of input values supplied by a loop.
 */

public class RootApproximatorHarness2 {
    public static void main(String[] args) {
        final double MIN = 1;
        final double MAX = 10;
        final double INCREMENT = 0.5;
        for (double x = MIN; x <= MAX; x = x + INCREMENT) {
            RootApproximator r = new RootApproximator(x);
            double y = r.getRoot();
            System.out.println("square root of " + x + " = " + y);
        }
    }
}
File
RootApproximatorHarness2.java

• Output

```java
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 = 3.0822070014844885
square root of 10.0 = 3.162277660168379
```
Providing Test Input

• Previous test restricted to small subset of values

• Alternative: random generation of test cases
import java.util.Random;

/**
   * This program computes square roots of random inputs.
   */

public class RootApproximatorHarness3 {
    public static void main(String[] args) {
        final double SAMPLES = 100;
        Random generator = new Random();
        for (int i = 1; i <= SAMPLES; i++) {
            // Generate random test value
            double x = 1000 * generator.nextDouble();
            RootApproximator r = new RootApproximator(x);
            double y = r.getRoot();
            System.out.println("square root of " + x + " = " + y);
        }
    }
}
### File

**RootApproximatorHarness3.java**

- **Output**

<table>
<thead>
<tr>
<th>Number</th>
<th>Square Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>810.4079626570873</td>
<td>28.467665212607223</td>
</tr>
<tr>
<td>480.50291114306344</td>
<td>21.9203766195534</td>
</tr>
<tr>
<td>643.5463246844379</td>
<td>25.36821485017103</td>
</tr>
<tr>
<td>506.5708496713842</td>
<td>22.507128863348704</td>
</tr>
<tr>
<td>539.6401504334708</td>
<td>23.230156057019308</td>
</tr>
<tr>
<td>795.0220214851004</td>
<td>28.196134867834285</td>
</tr>
</tbody>
</table>

...
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
```
import java.util.Scanner;

/**
 * This program computes square roots of inputs supplied through System.in.
 */

public class RootApproximatorHarness4 {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        boolean done = false;
        while (in.hasNextDouble()) {
            double x = in.nextDouble();
            RootApproximator r = new RootApproximator(x);
            double y = r.getRoot();
            System.out.println("square root of " + x + " = " + y);
        }
    }
}
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
```
## Reading Test Inputs From a File

- **File** `test.out`:

<table>
<thead>
<tr>
<th></th>
<th>Square Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.0</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>1.414213562373095</td>
</tr>
<tr>
<td>4</td>
<td>1.0</td>
</tr>
<tr>
<td>5</td>
<td>0.5</td>
</tr>
<tr>
<td>6</td>
<td>0.1</td>
</tr>
</tbody>
</table>
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 $x$)
import java.util.Random;

/**
 * This program verifies the computation of square root values by checking a mathematical property of square roots.
 */

public class RootApproximatorHarness5 {
    public static void main(String[] args) {
        final double SAMPLES = 100;
        int passcount = 0;
        int failcount = 0;
        Random generator = new Random();
        for (int i = 1; i <= SAMPLES; i++) {
        }
    }
}
```java
// Generate random test value
double x = 1000 * generator.nextDouble();
RootApproximator r = new RootApproximator(x);
double y = r.getRoot();

// Check that test value fulfills square property
if (Numeric.approxEqual(y * y, x)) {
    System.out.print("Test passed: ");
    passcount++;
} else {
    System.out.print("Test failed: ");
    failcount++;
}
```

Continued…
35: System.out.println("x = " + x
36: + ", root squared = " + y * y);
37: }
38: System.out.println("Pass: " + passcount);
39: System.out.println("Fail: " + failcount);
40: }
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 = 115.4885096006315, root squared = 115.48850960063153
Test passed: x = 384.973238438713, root squared = 384.973238438713
. . .
Pass: 100
Fail: 0
import java.util.Random;

/*
 * This program verifies the computation of square root values
 * by using an oracle.
 */

class RootApproximatorHarness6 {
    public static void main(String[] args) {
        final double SAMPLES = 100;
        int passcount = 0;
        int failcount = 0;
        Random generator = new Random();
        for (int i = 1; i <= SAMPLES; i++) {
            // Generate random test value
    
Continued...
double x = 1000 * generator.nextDouble();
RootApproximator r = new RootApproximator(x);
double y = r.getRoot();

double oracleValue = Math.pow(x, 0.5);

// Check that test value approximately equals
// oracle value
if (Numeric.approxEqual(y, oracleValue))
{
    System.out.print("Test passed: ");
    passcount++;
}
else
{
    System.out.print("Test failed: ");
    failcount++;
}
File

RootApproximatorHarness6.java

```java
System.out.println("square root = " + y + ", oracle = " + oracleValue);
}
System.out.println("Pass: " + passcount);
System.out.println("Fail: " + failcount);
```
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

Fall 2006 Adapted from Java Concepts Companion Slides
Program Trace

- Messages that show the path of execution

```java
if (status == SINGLE) {
    System.out.println("status is SINGLE");
    . . .
}
. . .
```
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 \texttt{Logger.global}

• Log a message

\begin{verbatim}
Logger.global.info("status is SINGLE");
\end{verbatim}
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:

```java
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:

```java
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
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:**

```java
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:**

```java
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:**

```java
public int countSyllables() {
    int 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

```java
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`

```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-letter characters, such as punctuation marks.
08:     * @param s the input string
09:   */
10:   public Word(String s)
11:   {
12:     int i = 0;
13:     while (i < s.length() && !Character.isLetter(s.charAt(i)))
14:       i++;
15:     int j = s.length() - 1;
16:     while (j > i && !Character.isLetter(s.charAt(j)))
17:       j--;
18:     
19:     // Continued...
```
public String getText() {
    return text;
}

/**
 * Counts the syllables in the word.
 * @return the syllable count
 */
Continued...
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:       // An e at the end of the word doesn't count as a vowel
42:       char ch = Character.toLowerCase(text.charAt(end));
43:       if (ch == 'e') end--;
44:       boolean insideVowelGroup = false;
45:       for (int i = 0; i <= end; i++)
46:       {
47:           ch = Character.toLowerCase(text.charAt(i));
48:           String vowels = "aeiouy";
49:           if (vowels.indexOf(ch) >= 0)
50:               {
51:               Continuated..."
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: }
import java.util.Scanner;

/**
 * This program tests the countSyllables method of the Word class.
 */

class WordTester {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter a sentence ending in a period.");
        String input;
        do {
            input = System.out.println("Enter a sentence ending in a period.");
            do
            { 

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'**
Debug the Program

Figure 5: Debugging the CountSyllables Method
Debug the Program

• Check if this works: step to line where check is made and inspect variable ch

• Should contain final letter but contains '1'
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
- \texttt{text} contains "hell", not "hello"
- No wonder \texttt{countSyllables} returns 1
- Culprit is elsewhere
- Can't go back in time
- Restart and set breakpoint in \texttt{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"?
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
  ```java
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
Another Error

- Start debugger
- Erase all old breakpoints and set a breakpoint in `countSyllables` method
- Supply input "hello"
Break in the beginning of `countSyllables`. Then, single-step through loop.

```java
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

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