Software Defects: are inevitable in a complex software system.
- In industry: 10-50 bugs per 1000 lines of code!
- Defects can be obvious or remain hidden.

Software Reliability: What is the probability of failure of a software package over time:
- Measurements: mean time between failures, crash statistics, uptime versus downtime.

Software Testing is a systematic attempt to reveal errors in software by running test programs or scripts (interactively or automated).
- FAILING TEST: an error was demonstrated in the software under test.
- PASSING TEST: no error was found, at least for this particular situation.
- Theory of testing says you cannot prove absence of all defects in software.
Software Testing

- **Methods**
  - Black-box, white-box

- **Levels**
  - Unit (Method), Module (Class), Integration, System

- **Types**

- **Processes**
  - Regression Testing, Test Automation, *Test-Driven Development*, Code Coverage, ...

---

Unit Testing

- JUnit is a simple, open source framework to write and run repeatable tests. JUnit is commonly used in industry for unit testing.

  Features include:
  - Assertions for testing expected results
  - Test fixtures for sharing common test data
  - Test runners for running tests

---

JUnit Testing

Citation: JUnit testing framework (http://www.junit.org/)
Exhaustive Testing?

- We consider a program to be correct if it produces the expected output for all inputs.
- Domain of input values can be very large, e.g. $2^{32}$ values for an integer or float:
  ```java
  int divide (int operand1, int operand2);
  ```
  $2^{32} \times 2^{32} = 2^{64}$, a large number, so we clearly cannot test exhaustively!
- And that is just for one method, in one class, in one package, and relatively simple.

Boundary Testing (1)

- Test with the boundaries of the domain of parameter values:
  ```java
  // Boundary testing of Math.floor
  System.out.println(Math.floor(Double.MIN_VALUE));
  System.out.println(Math.floor(Double.MAX_VALUE));
  System.out.println(Math.floor(-9.87654321L));
  System.out.println(Math.floor(1.9999999L));
  System.out.println(Math.floor(-1.0000001L));
  System.out.println(Math.floor(1.0000001L));
  System.out.println(Math.floor(0.9999999L));
  System.out.println(Math.floor(0.87654321L));
  ```

Regression Testing

- Scenario: company with engineers continually writing or modifying code.
  - Question: How to find new defects introduced while enhancing code or fixing old defects?
  - Answer: Regression test the code with a build and verification cycle on a regular basis, using a growing test suite, with known correct outputs.
  - Assumes test automation otherwise not practical.
- Code won’t remain stable without a consistent and thorough effort to maintain quality.

Code Coverage

- Code coverage improves software quality by illumination: it shines light on executed code, and reveals dark corners that are untested or never used. This software metric can enhance many projects, from standard business apps to those with ultra-low tolerance for error, for example medical devices.

Citation: Emma code coverage tool (http://emma.sourceforge.net/)
Pre and Post Conditions

- **Pre-Condition**: a statement of what must be true about the input or variable values at the beginning of the program segment:
  - e.g. Math.sqrt requires the input parameter to be $\geq 0$

- **Post-Condition**: a statement of what must be true about the output or variable values at the end of the program segment:
  - e.g. Math.sqrt return value squared $\leq$ input parameter

- Java Assertions: can be used to check pre and post conditions.

What is an assertion?

- **An assertion** is a statement in Java that enables you to test your assumptions about your program.
  - Each assertion contains a boolean expression that you believe will be true when the assertion executes.
  - Confirms your assumptions about the behavior of your program, increasing your confidence that the program is error free.
Why use assertions?

- Programming by contract
  - Defines responsibilities of producer and user
- Pre-conditions
  - Assert precondition as requirement for method input
- Post-conditions
  - Assert post-condition as requirement for method output

Simple Assertion Form

The assertion statement has two forms:

- The first form is:
  ```
  assert Expression;
  ```
- Where Expression is a boolean expression
- When the system runs the assertion, it evaluates Expression and if it is false throws an AssertionError with no details.

Complex Assertion Form

- The second form is:
  ```
  assert Expr1 : Expr2 ;
  ```
  where:
  - Expr1 is a boolean expression;
  - Expr2 is an expression (e.g. a String)
    if(!expr1) throw exception(expr2);

Enabling Assertions

- For the virtual machine to enable assertions, use `-ea, -enableassertions`
- In Eclipse, this resides in the VM arguments in the Run Configuration
Performance Problems

- Assertions may slow down execution - why?
- So, assertions may be enabled and disabled
- Assertions, by default, disabled at run-time
- In this case, the assertion is empty statement
- Don’t use for user errors - why not?

Pre-Condition Example

```
public ArrayList<String> readFile(String filename) {
    ArrayList<String> contents = new ArrayList<>();
    try {
        File file = new File(filename);
        assert (file.exists()); // file: file does not exist: " + filename;
        assert (file.canRead()); // file: file is not readable: " + filename;
        assert (file.length() > 0); // "readFile: file is empty: " + filename;
        Scanner reader = new Scanner(file);
        // Code to read file
        reader.close();
        } catch (FileNotFoundException e) {
            System.out.println("getErrorMessage");
        }
        return contents;
    }
```

Post-Condition Example

```
public void sortNames(ArrayList<String> names) {
    Collections.sort(names);
    // Check sorting
    for (int i = 1; i < names.size(); i++) {
        assert (names.get(i).compareTo(names.get(i-1)) > 0); // names sorted incorrectly?
    }
}
```

Loop invariants

- We use predicates (logical expressions) in assertions to reason about our programs.
- A loop invariant is a predicate
  - that is true directly before the loop executes
  - that is true before and after the loop body executes
  - and therefore true directly after the loop has executed

i.e., it is kept invariant by the loop.
Loop invariants cont'

Combined with the loop condition, the loop invariant allows us to reason about the behavior of the loop:

```plaintext
<loop invariant>
while(test){
  <test AND loop invariant>
  S;
  <loop invariant>
}
<not test AND loop invariant>
```

What does it mean...

- **If we can prove**
  - the loop invariant holds before the loop and that
  - the loop body keeps the loop invariant true i.e. `<test AND loop invariant> S <loop invariant>`
  **then we can infer**
  - not test AND loop invariant holds after the loop terminates

Example summing

```plaintext
int total (int[] elements){
  // pre: elements.length > 0
  int sum = 0,i = 0, n = elements.length;
  // invariant?
  while (i < n){
    // i<n and invariant?
    sum += elements [i];
    i++;
    // invariant?
  }
  // i==n (previous example) AND invariant
  // sum == sum of int[] elements
  return sum;
}
```

```plaintext
Example summing
```

```plaintext
int total (int[] elements){
  // pre: elements.length > 0
  int sum = 0,i = 0, n = elements.length;
  // invariant?
  while (i < n){
    // i<n and invariant?
    sum += elements [i];
    i++;
    // invariant?
  }
  // i==n (previous example) AND invariant
  // sum == sum of int[] elements
  return sum;
}
```
Example: Egyptian multiplication

\[
\begin{array}{cc}
A & B \\
19 & 5 \\
/2 & 9 10 \times 2 \\
/2 & 4 20 \times 2 \\
/2 & 2 40 \times 2 \\
/2 & 1 80 \times 2 \\
\end{array}
\]

throw away all rows with even A:

\[
\begin{array}{cc}
A & B \\
19 & 5 \\
9 & 10 \\
1 & 80 \\
\end{array}
\]

add B’s  
95

\rightarrow the product !!

Egyptian multiplication

\[
\begin{array}{cc}
A & B \\
19 & 5 \\
/2 & 9 10 \times 2 \\
/2 & 4 20 \times 2 \\
/2 & 2 40 \times 2 \\
/2 & 1 80 \times 2 \\
\end{array}
\]

throw away all rows with even A:

\[
\begin{array}{cc}
A & B \\
19 & 5 \\
9 & 10 \\
1 & 80 \\
\end{array}
\]

add B’s  
95

\rightarrow the product !!

Can we show it works? Loop invariants!!

```c
// pre: left >=0 AND right >=0
int a=left, b=right, p=0;
// p+(a*b) == left * right loop invariant
while (a!=0) {
    // a!=0 and p+a*b == left*right loop condition and loop invariant
    if (odd(a)) p+=b;
    a/=2;
    b*=2;
    // p+(a*b) == left*right
}
// a==0 and p+a*b == left*right \rightarrow p == left*right
```

Try it on 7 * 8

<table>
<thead>
<tr>
<th></th>
<th>left</th>
<th>right</th>
<th>a</th>
<th>b</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>+=b: 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>32</td>
<td>+=b: 24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>+=b: 56</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Try it on 8*7

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>a</th>
<th>b</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>56</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>118</td>
<td>+=b: 56</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Relation to int representation 19*5

\[
\begin{array}{c}
00101 \\
10011 \\
\hline
101 \\
1010 \\
\hline
00000 \\
000000 \\
1010000 \\
\hline
1011111 \\
\end{array} = 64 + 31
\]

Summary: Loop Invariant Reasoning

```java
//loop invariant true before loop
while (b){
    // b AND loop invariant
    S;
    // loop invariant
}
// not b AND loop invariant
```

not b helps you make a stronger observation than loop invariant alone.

Software Debugging

- Related topic - you cannot test code until you get it working.
- Possible methods for debugging:
  - Examining code (weak)
  - Print debugging (better)
  - Using debugger (strong)
- Computer Science department has print debugging package – Debug.java
Print Debugging

```java
public static void readFile (String filename) {
    try {
        Scanner reader = new Scanner(new File(filename));
        while (reader.hasNextLine()) {
            String line = reader.nextLine();
            System.out.println(line); // debug print
            contents.add(line);
        }
        reader.close(); // code defect
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

Debugging Tools