Testing
Importance of Testing

• Testing is a ubiquitous and expensive software engineering activity
  – It is not unusual to spend 30-40% of total project effort on testing
  – For big and/or life-critical systems (e.g., flight control), testing cost can be several times the cost of all other software engineering activities combined
How Big is Big?

- The method bodies we have been writing average maybe a dozen lines of code
- Claim: Boeing 787 Dreamliner avionics (flight control) software has about ...
How Big is Big?

- The method bodies we have been writing average maybe a dozen lines of code.
- Claim: Boeing 787 Dreamliner avionics (flight control) software has about 6.5 million lines of code.
- Claim: Microsoft Windows 10 has about ...
How Big is Big?

• The method bodies we have been writing average maybe a dozen lines of code

• Claim: Boeing 787 Dreamliner avionics (flight control) software has about 6.5 million lines of code

• Claim: Microsoft Windows 10 has about 50 million lines of code

• Claim: a modern car has about ...
How Big is Big?

• The method bodies we have been writing average maybe a dozen lines of code

• Claim: Boeing 787 Dreamliner avionics (flight control) software has about 6.5 million lines of code

• Claim: Microsoft Windows 10 has about 50 million lines of code

• Claim: a modern car has about 100 million lines of code (though this figure is highly dubious)
Unit Testing: Dealing with Scale

• **Best practice** is to test individual *units* or *components* of software (one class, one method at a time)
  – This is known as *unit testing*
  – Testing what happens when multiple components are put together into a larger system is known as *integration testing*
  – Testing a whole end-user system is known as *system testing*
Unit Testing: Dealing with Scale

• **Best practice** is to test individual *units* or *components* of software (one class, one method at a time)
  – This is known as *unit testing*
  – Testing what happens when multiple components are put together into a larger system is known as *integration testing*
  – Testing a whole end-user system is known as *system testing*
Unit Testing:

• **Best practice** is to test individual *units* or *components* of software (one class, one method at a time)
  
  – This is known as *unit testing*
  
  – Testing what happens when multiple components are put together into a larger system is known as *integration testing*
  
  – Testing a whole end-user system is known as *system testing*
Testing Functional Correctness

• What does it mean for a program unit (let’s say a method) to be correct?
  • It does what it is supposed to do.
  • It doesn’t do what it is not supposed to do.
“Supposed To Do”?

• How do we know what a method is supposed to do, and what it is not supposed to do?
  – We look at its *contract*, which is a *specification* of its *intended behavior*
Behaviors

Allowed behaviors of the method (see contract)

Actual behaviors of the method (see body)
Each point in this space is a *legal input* with a corresponding *allowable result*.

**Allowed behaviors of the method (see contract)**

**Actual behaviors of the method (see body)**
Example Method Contract

/**
 * Reports some factor of a number.
 * ...
 * @requires
 * n > 0
 * @ensures
 * aFactor > 0 and
 * n mod aFactor = 0
 */

private static int aFactor(int n) {...}
Example Method Contract

/**
 * Reports some factor of a number.
 * ...
 * @requires
 * n > 0
 * @ensures
 * aFactor > 0 and
 * n \text{ mod } aFactor = 0
 */

private static int aFactor(int n) {...}
private static int aFactor(int n) {
    return 1;
}
private static int aFactor(int n) {
    return 1;
}
Behaviors

Allowed behaviors of the method (see contract)

Actual behaviors of the method (see body)
Behaviors

Contract for `aFactor` allows:

\[ n = 12 \]
\[ aFactor = 4 \]

Allowed behaviors of the method (see contract)

(12,4)

Actual behaviors of the method (see body)
Contract for \texttt{aFactor} 	extit{forbids}: 
\begin{align*}
  n &= 12 \\
  \texttt{aFactor} &= 5
\end{align*}

Allowed behaviors of the method (see contract) 

Actual behaviors of the method (see body)
Behaviors

Contract for \texttt{aFactor} allows:
\[
  n = 12 \\
  \texttt{aFactor} = 6
\]

Allowed behaviors of the method (see contract)

Actual behaviors of the method (see body)

(12,4) (12,6) (12,5)
Behaviors

Contract for \texttt{aFactor} allows:

\[
\begin{align*}
n &= 12 \\
aFactor &= 1
\end{align*}
\]

Allowed behaviors of the method (see contract)

Actual behaviors of the method (see body)
Body for `aFactor` gives:

\[ n = 12 \]

\[ aFactor = 1 \]

Allowed behaviors of the method (see contract)

Actual behaviors of the method (see body)
Definition of Correctness

• Body is *correct* if *actual* is a subset of *allowed*. 

- **Allowed behaviors of the method** (see contract)
- **Actual behaviors of the method** (see body)
“Implements” Revisited

• If you write `class C implements I`, the Java compiler checks that for each method in `I` there is some method body for it in `C`.

• We really care about much more: that for each method in `I` the method body for it in `C` is `correct` in the sense just defined.
“Implements” Revisited

• If you write `class C implements I`, the Java compiler checks that for each method in `I` there is some method body for it in `C`.

• We really care about much more: that for each method in `I` the method body for it in `C` is `correct` in the sense just defined.

How can you decide whether this is the case for a given method body?
Testing

• *Testing* is a technique for trying to *refute the claim* that a method body is correct for the method contract

• In other words, the *goal* of testing is to show that the method body does *not* correctly implement the contract, i.e., that it is *defective*
  – As a tester, you really want to think this way!
Psychology of Testing

• Design and coding are *creative* activities
• Testing is a *destructive* activity
  – The primary goal is to “break” the software, i.e., to show that it has defects
• Very often the same person does both coding and testing (*not* a *best practice*)
  – You need a “split personality”: when you start testing, become paranoid and malicious
  – It’s surprisingly hard to do: people don’t like finding out that they made mistakes
Testing vs. Debugging

• Goal of **testing**: given some code, show by executing it that it has a defect (i.e., there is at least one situation where the code’s actual behavior is not an allowed behavior)

• Goal of **debugging**: given some source code that has a defect, find the defect and repair it
Incorrect (Defective) Code

- If actual behaviors are not a subset of allowed...
Incorrect (Defective) Code

• ... and we start trying some inputs and observing results ...
Incorrect (Defective) Code

- ... one might lie outside the allowed behaviors!
Incorrect (Defective) Code

• ... one might lie outside the allowed behaviors!

If this happens, testing has succeeded (in revealing a defect in the method body).
Test Cases

• Each input value and corresponding allowed/expected result is a **test case**
• Test cases that do *not* reveal a defect in the code do not help us refute a claim of correctness
• Test cases like that last one should be cherished!
Test Plan/Test Fixture

• A set of test cases for a given unit is called a *test plan* or a *test fixture* for that unit
Correct Code

- If actual behaviors are a subset of allowed...
Correct Code

• ... and we start trying some inputs and observing results ...

Allowed behaviors of the method (see contract)

Actual behaviors of the method (see body)
Correct Code

• ... then we will never find a defect.
Severe Limitation of Testing

• “Program testing can be used to show the presence of bugs, but never to show their absence!”

— Edsger W. Dijkstra (1972)
Designing a Test Plan

• To make testing most likely to succeed in revealing defects, **best practices** include:
  – Test **boundary** cases: “smallest”, “largest”, “special” values based on the contract
  – Test **routine** cases
  – Test **challenging** cases, i.e., ones that, if you were writing the code (maybe you didn’t write the code being tested!), you might find difficult or error-prone
Example Method Contract #1

/**
 * Returns some factor of a number.
 * ...
 * @requires
 * n > 0
 * @ensures
 * aFactor > 0 and
 * n mod aFactor = 0
 */

private static int aFactor(int n) {...}
## Partial Test Plan

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Results</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 1$</td>
<td>$a\text{Factor} = 1$</td>
<td>boundary</td>
</tr>
<tr>
<td></td>
<td>$a\text{Factor} = 2$</td>
<td></td>
</tr>
<tr>
<td>$n = 2$</td>
<td>$a\text{Factor} = 1$</td>
<td>routine</td>
</tr>
<tr>
<td></td>
<td>$a\text{Factor} = 2$</td>
<td>challenging? (prime)</td>
</tr>
<tr>
<td>$n = 4$</td>
<td>$a\text{Factor} = 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a\text{Factor} = 2$</td>
<td>challenging? (square)</td>
</tr>
<tr>
<td></td>
<td>$a\text{Factor} = 4$</td>
<td></td>
</tr>
<tr>
<td>$n = 12$</td>
<td>$a\text{Factor} = 1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a\text{Factor} = 2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a\text{Factor} = 3$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a\text{Factor} = 4$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a\text{Factor} = 6$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a\text{Factor} = 12$</td>
<td>routine</td>
</tr>
</tbody>
</table>
Example Method Contract #2

/**
 * Decrement the given NaturalNumber.
 * ...
 * @updates n
 * @requires
 * n > 0
 * @ensures
 * n = #n - 1
 */

private static void decrement(NaturalNumber n) {...}
## Partial Test Plan

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Results</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>#n = 1</td>
<td>n = 0</td>
<td>boundary</td>
</tr>
<tr>
<td>#n = 2</td>
<td>n = 1</td>
<td>routine</td>
</tr>
<tr>
<td>#n = 10</td>
<td>n = 9</td>
<td>challenging? (borrow)</td>
</tr>
<tr>
<td>#n = 42</td>
<td>n = 41</td>
<td>routine</td>
</tr>
</tbody>
</table>
## Partial Test Plan

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Results</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>#n = 1</td>
<td>n = 0</td>
<td>boundary</td>
</tr>
<tr>
<td>#n = 2</td>
<td>n = 1</td>
<td>routine</td>
</tr>
<tr>
<td>#n = 10</td>
<td>n = 9</td>
<td>challenging? (borrow)</td>
</tr>
<tr>
<td>#n = 42</td>
<td>n = 41</td>
<td>routine</td>
</tr>
<tr>
<td>#n = 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What about this “boundary” case, which is on the illegal side of the “boundary” between legal and illegal inputs?
## Partial Test Plan

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Results</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n = 1)</td>
<td>(n = 0)</td>
<td>boundary</td>
</tr>
<tr>
<td>(n = 2)</td>
<td>(n = 1)</td>
<td>routine</td>
</tr>
<tr>
<td>(n = 10)</td>
<td>(n = 9)</td>
<td>challenging? (borrow)</td>
</tr>
<tr>
<td>(n = 42)</td>
<td>(n = 41)</td>
<td>routine</td>
</tr>
<tr>
<td>(n = 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This test case is worthless: it violates the requires clause, so it cannot possibly reveal a defect in the method body. Why not?
Resources

  - [https://library.ohio-state.edu/record=b8532947~S7](https://library.ohio-state.edu/record=b8532947~S7)