Testing
Limitations

- Unit tests only test “visible” (public) methods.
- It is challenging to write code that thoroughly tests every piece of code. That is, to get good coverage.
- It may not help you as you are composing code.
  - How do I test little pieces of functionality that are only parts of the outward facing behaviors?
  - How do I test private/helper methods?
Testing Benefits

1. Problem-first is the right focus!
2. Constraints help you drive the design. If you have adopted problem-first, this should drive good design.
3. Worst case, tests will be thrown away. But they helped you build a solution and deeper understanding of the problem.
4. At best, they prevent future regression, and provide help and documentation for future developers.

One misconception: tests ensure correctness. They do not. Your tests document behaviors that you have verified and some assumptions that you have made about those behaviors.
What is expected of us in CS126 in regards to testing?

(How do I get the best score?)

- As always, we expect you to think about what you are doing and what the limitations are of your approach.
- You should endeavor to write meaningful tests and use them to drive your design.
- Once you have understood the responsibilities and limitations of testing, you will be expected to exhibit that understanding on all subsequent assignments.
  - Meaningful test cases
  - No redundancy
  - No obvious “holes”
What do we expect for “flexibility”? 

In a nutshell, “how hard would it be to modify your code to accommodate m x n?”

So if you…

1. use literals (3, 9). That’s bad.
2. assume square dimensions. That’s better than assuming 3x3 but not best.
3. use named constants. That’s better.
4. have a configuration that allows recompiling to a different dimension but is still fixed. That’s better.
5. could easily accommodate a different dimension for each board. That’s best.
What do we expect for “flexibility”?  

Where does the grading fall?

Somewhere between 4 and 5 in this list.

1. use literals (3, 9). That’s bad.  
2. assume square dimensions. That’s better than assuming 3x3 but not best.  
3. use named constants. That’s better.  
4. have a configuration that allows recompiling to a different dimension but is still fixed. That’s better.  
5. could easily accommodate a different dimension for each board. That’s best.
Testing

- helps refine our designs
- documents PUBLIC* behaviors

*do not test private methods with your public tests
Things to consider:

Should your method “defend against bad input”?

If an input satisfies more than one case, are you getting the right result?

Numerical methods: Overflow, underflow, useless results

Do not waste time with multiple tests that do not verify anything new.
Things to consider:

Should your method “defend against bad input”?
- It may be unnecessary.
- This depends on the specific method.
- Private methods should be able to be freed from this burden.

If an input satisfies more than one case, are you getting the right result?

Numerical methods: Overflow, underflow, useless results

Do not waste time with multiple tests that do not verify anything new.
Things to consider:

If a test input satisfies more than one case, are you getting the right result?

Fail fast.

Both X and O satisfy a winning state.

This is not a valid game state.

Can you detect what went wrong?

Numerical methods: Overflow, underflow, useless results

Some tests may not be straightforward. Sum of two unsigned integers.

This should not apply to this assignment but underflow WILL come into play later in the semester.

Do not waste time with multiple tests that do not verify anything new.

Variations on (string.length <> m*n) are unnecessary.
Create a table of behaviors

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Input</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board String must be &lt;=9</td>
<td>0123456789</td>
<td>STRING_TOO_LONG</td>
</tr>
<tr>
<td>Unreachable State</td>
<td>XXXXXXXiii</td>
<td>UNREACHABLE_STATE</td>
</tr>
<tr>
<td>“Other classes of unreachable states”</td>
<td></td>
<td>UNREACHABLE_STATE</td>
</tr>
<tr>
<td>X Wins (diag, vert, horz)</td>
<td>X_O_X_O_X</td>
<td>X_WINS</td>
</tr>
<tr>
<td>O Wins (diag, vert, horz)</td>
<td>X_OXO_O_X</td>
<td>O_WINS</td>
</tr>
</tbody>
</table>
Private Methods

In the case of a Tic-Tac-Toe board string, we could normalize or rectify the input before evaluating the string.

xOxoXo would become XOXOXO after conditioning but we would not return this to the user. The user of this class would only be interested in knowing who won or if it could not be evaluated. The reason for the inability to evaluate may also be of interest.

How do we verify that our private methods work?
Reading
(understanding why)

PrairieLearn (Testing)
Software Testing Stuff (types of testing)
http://www.softwaretestingstuff.com/2008/12/software-testing-techniques-and-levels.html

Unit Testing: You’re doing it wrong
https://medium.com/@Cyrdup/unit-testing-youre-doing-it-wrong-407a07692989
Testing Misconceptions

Unit Testing: You’re doing it wrong

https://medium.com/@Cyrdup/unit-testing-youre-doing-it-wrong-407a07692989

1. TDD is all about unit tests.

2. Automated testing is all about unit tests

3. 100% code coverage requires extensive unit testing
Testing Misconceptions

4. You have to make private methods public to reach 100% coverage

Again, no!; private methods will be tested through public entry points. Once again, unit testing is not about testing every method in isolation.

5. Some code does not need be tested

6. You need to use a mocking framework
Testing Misconceptions

7. Tests are expensive to write

8. The ‘testing pyramid’ is the ultimate testing strategy
Some Truths

1. Unit tests are not about testing a method in isolation

2. 100% coverage does not mean your code is bug free

3. There is a tooling problem

4. It is difficult
5. Tests require maintenance

To sum it up: **tests are part of your codebase and must be treated as such.** Which leads to the next truth:
Some Truths

6. Having too many tests is a problem

7. Throwing away tests is a hygienic move
8. Automated tests are useful

An interesting and important 2014 study analyzed 198 user reported issues on distributed systems (incl HBase, Cassandra, Zookeeper). Among several important findings, it concluded that 77% of the analyzed production issues could have been reproduced by a unit test.

Another key finding was that almost all catastrophic failures were the result of incorrect error handling.