OOP Testing Lessons So Far

Important themes so far, focusing on how they have appeared in the homeworks.

Unit Test Cases

- Write some obvious, basic ones
- Once the basics are done, subsequence tests must reach to be more "mean" in some way
- However, tests are not exhaustive .. they should sample, and that's good enough
- Or put another way: additional tests give diminishing returns, so try to hit a sweet spot of pretty good test coverage

Call Each Method

- Call it a couple different ways that are different
- The "returns true;" rule

Stack-up Combinations

- If the object builds up state, can test foo() method after the bar() method has run
- Stack up calls 2 or 3 deep, and that's good enough

e.g. Tetris Board

- You can be creative, using a very small board. For example, if the width is 2, even a single piece can yield row clearing.
- The board has many methods -- we'll say that a "check" calls every method 2 or 3 times to check its result. I make a little drawing to figure out the right values, and then write the tests from the drawing.
- Make tests where you stack up 2 or 3 calls, and then check .. trying a few combinations
  - Place a piece, then check
  - Place, then undo, then check
  - Place, then undo, then place, then check
  - Place, row clear, undo, check
  - Place, row clear, place, check
- I suspect you can find all bugs without going deeper than the above examples. Rather than stack up more calls, you just need to check the state carefully. If the state is truly correct, it should support all subsequent calls.
- Can create instance variables in unit test object to contain "fixture" -- common data sitting there ready for unit tests, set up in setUp() method. Tests can be repetitive, so having fixtures ready to go can be a real help. (e.g. BoardTest starter code shows this)

OOP Design 1 -- Encapsulation

- Organize code around nouns
- Encapsulation -- objects keep their state and implementation complexity private as much as possible
- Move the code to the data
  - Avoid pulling data out of an object to do a computation
  - It's fine to add computation to some object later on when the need (from the client side) becomes evident.

API Design

- Driven by client needs -- what do clients want solved, what is the client's natural vocabulary? Common client use cases should work very easily (convenience methods).
- Asymmetry -- the exposed interface is often much simpler than the implementation

Shape Example

- Shape -- both themes: client API and encapsulation. Add methods to shape to perform the client needed operations on the shape's data vs. pulling the data out to operate on it. (e.g. this shape vs. param shape)
- Can add public methods to shape … internal shape methods can "circle back" through those same methods

Tetris Example

- Tetris Piece and Board are working examples of all of the above
  - Encapsulate significant state and complexity
  - Expose relatively simple interface -- solves the problems that JTetris, the brain, etc. need solved to make the game work.

OOP Design 2 -- Inheritance

- Strong "isa" relationship between super and sub classes
- Writing a subclass is hard -- need to "fit in" with superclass design
- Avoid duplicating superclass code in the subclass. The whole point of the structure is avoiding duplication.