Correctness proofs

- Ideally, we'd enter formal pre- and post-conditions and invariants, and statically prove that our program meets them: *formal verification*
  - Like typechecking
  - Guarantees correct programs!!
- Completely impractical for real programs
  - [*Why, do you think?*]

Testing

- The realistic alternative is testing
- But testing can never guarantee correctness, only that particular runs on particular inputs seem to produce the right answers
  - So let's have lots of test cases!
    - A *test suite*

Good test suites

- A test suite is good if it
  - Exposes bugs *quickly*
  - Exposes *all* bugs
- This is hard!
- Need to get good *coverage* over all the things a program might do
  - All paths through the program's control flow
    - But what about error paths?
  - All "interesting" values of data structures
    - What's interesting?
- Good coverage ≈ slow

Unit tests

- A basic kind of test is a *unit test*
  - Test a single unit of software
    - E.g. a class or a method
- Suitable for a single programmer who's developing the unit
- Manageable to strive for tests that together get good coverage of the interesting cases of the single unit

"Interesting cases"

- Try to exercise each non-"impossible" path through each method
- Try to give crazy inputs
  - Don't violate preconditions, but do everything else
- Think about corner cases
  - 0, negative numbers, empty arrays, empty lists, circular references

Test cases vs. specifications

- A good test suite approximates a specification
  - Each test has a legal input and the expected output
    - input implies a (partial) precondition
    - output implies a (partial) postcondition
- If formal specifications are too unwieldy, a good test suite can be used instead (or in addition)
  - Test suites are machine checkable, but not as complete as real specifications
- Test-Driven Development: write test suite first!
  - Another tenet of Extreme Programming

Running tests

- It can be very tedious to run tests by hand
  - Need to have a test harness that will construct and pass in the right inputs
  - Need to look at the output, and compare it to the expected output
  - Need to handle exceptions, too
- So, let's make tools!

Programming unit tests

- In Java, a simple strategy for unit testing is to define *self-testing* classes
- Each class can define a `static main` method that runs some set of unit tests of the class
  - The `main` method builds arguments, invokes operations, checks results, handles exceptions
  - To run, just invoke the class as if it were the main application
    - `java MyDataStructure`
- Still pretty tedious...

Making unit tests easier

- There exist tools to help in constructing unit test harnesses
- E.g. JUnit, a unit test framework for Java (http://junit.org)
  - Constructs a report of successes & failures
  - Provides some convenient helper functions
  - "Test Infected: Programmers Love Writing Tests"

Regression test suites

- Goal: accumulate a lot of good unit tests
  - Run them frequently after changes
  - Add testing to `make` process
- A good *regression test suite* gives confidence in development
  - Confidence to try big clean-ups without introducing uncaught bugs
  - Confidence to commit changes to rest of team

Beyond unit tests

- Unit tests aren't enough!
- Need to test that the units work together: *integration testing*
- [Why might errors crop up when testing groups of units that weren't caught when unit testing?]

Defensive programming

- The best programmers are defensive
  - They design & implement code that is unlikely to break
  - If there is a problem, the code breaks quickly and clearly
- Some strategies:
  - Minimize preconditions
  - Insert an assertion whenever they mentally expect and rely on something being true

Programming for change

- Expect change:
  - To software's design & requirements
  - To interfaces
  - To data structures
  - To people on the project
- Write code that minimizes reliance on things that might change, & is flexible in face of future changes
  - Fewer bugs introduced when these things change

Other tools

- Programming language choice(s) influence how likely programs are to be correct, how easy programs are to debug
  - E.g. array bounds checking, static type checking
- Programming environment tools can help mechanize much of testing
  - JUnit is a simple example
  - Some advanced static analysis tools can help to find bugs