Unit Testing

More reading:
C++ text vol2 ch2,
UMD lecture,
Meyer article on Ariane5

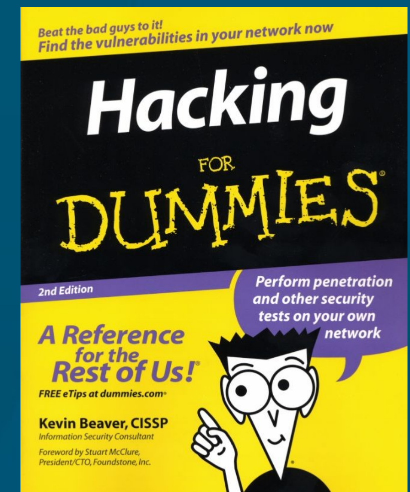
27 Mar 2009
CMPT166
Dr. Sean Ho
Trinity Western University

TRINITY
WESTERN
UNIVERSITY

Review of last time: PThreads

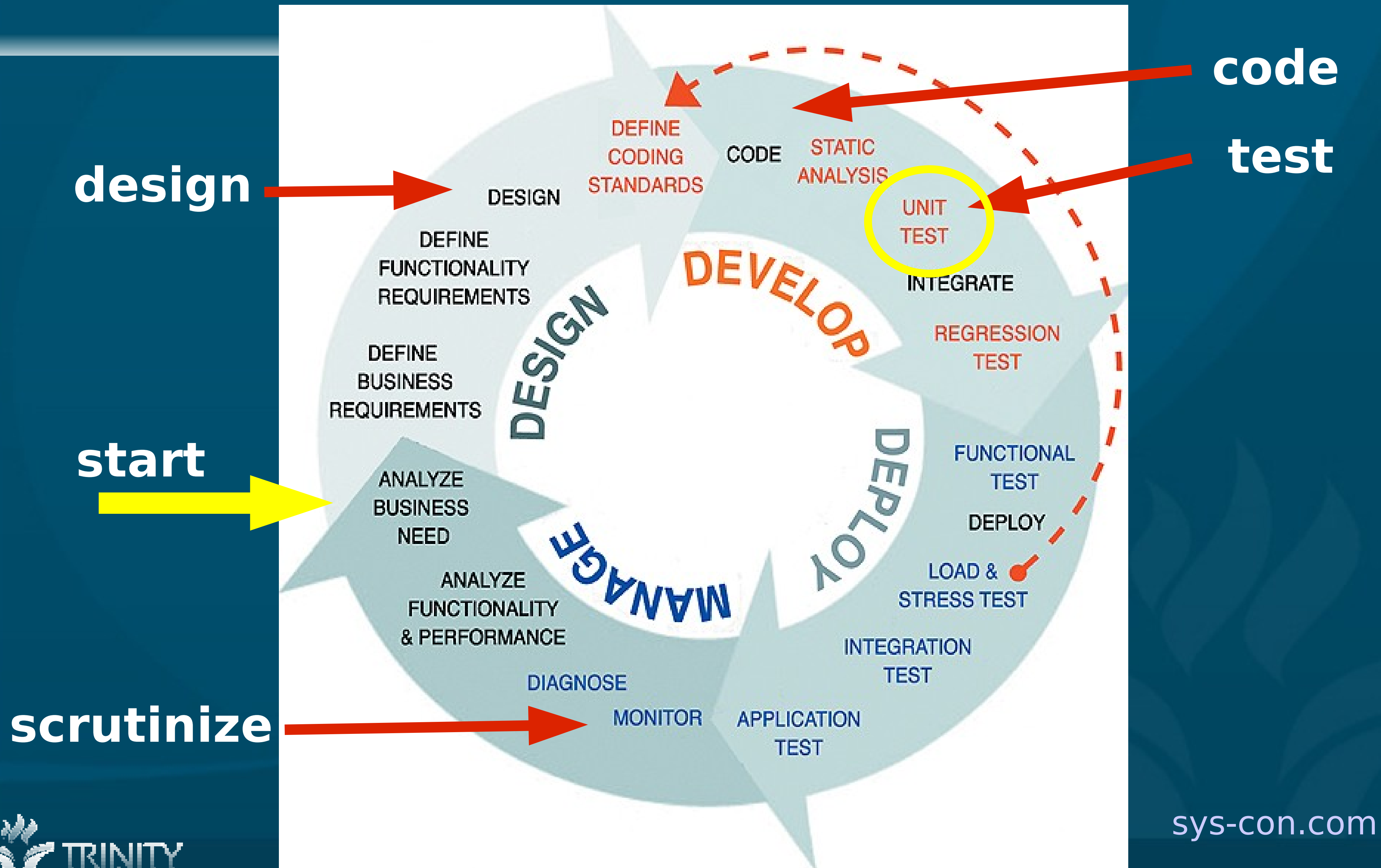
- Thread model: shared memory
- Programming models:
  - Master-worker, coworkers, pipeline
- PThreads: create, exit, join
  - 3 args to create: ID, callback, argument
  - Getting results from a child
- Locks, deadlock
  - with FLTK: Fl::lock/unlock()
- Threads in FLTK: Fl::awake()

Designing, not hacking



- Good, complex software is not easy!
- A little more time spent designing saves a lot more time debugging:
  - Requirements
    - Use-case scenarios
    - Pre/post-conditions
  - Component design
  - Class hierarchy
  - Class design
  - … then fill in the code!

Complete software life-cycle



How to ensure your code works?

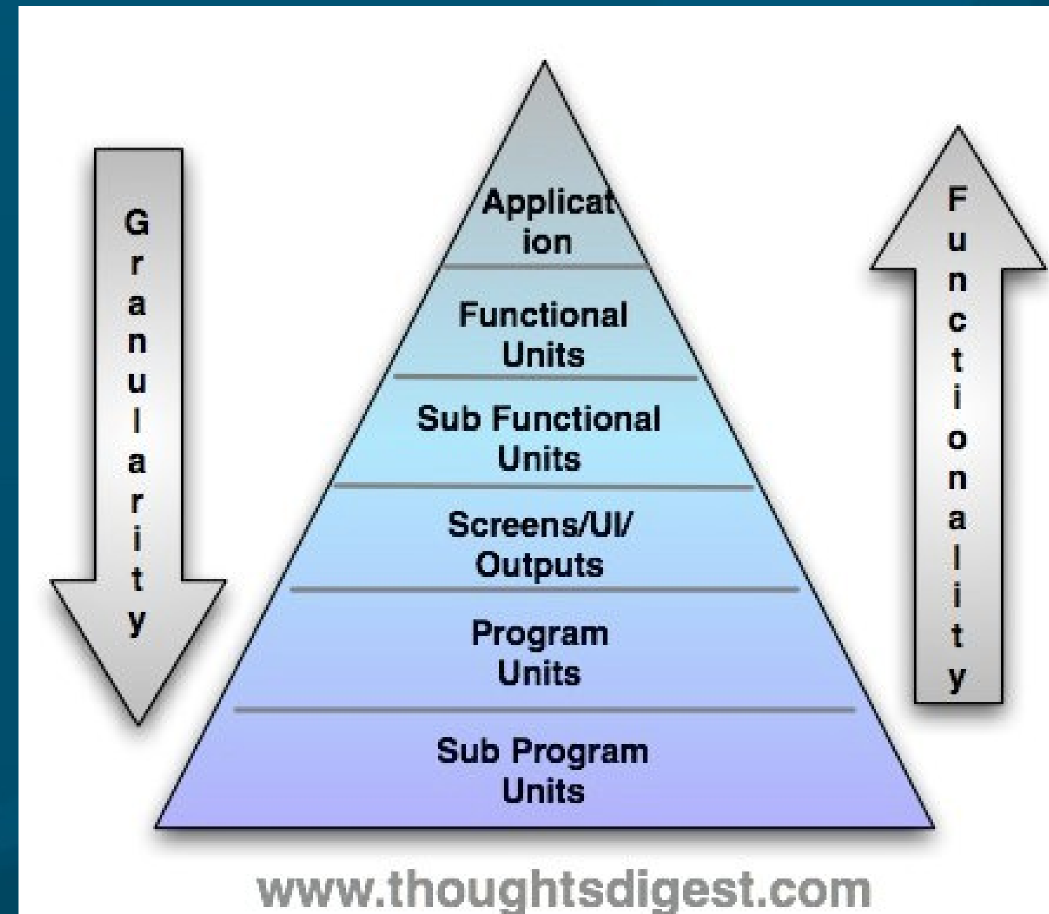
- How do we usually make sure our program is doing the right thing?
- Stare at the code and convince yourself it works
  - Easy to miss errors
  - Easy to be lazy!
  - “Tunnel vision” – same person codes+tests
- Prove that it is correct
  - Very difficult; not always possible
- Ship it first and wait for customers to complain
  - Not very nice!

Ensuring your code works

- Testing!
- Design your software with testing in mind:
  - Catch bugs early on
  - Easier development process
  - Better design, higher-quality code
  - Easier to maintain/upgrade
- Ensures your program does what it's supposed to
- Ensures you know what it's supposed to do!
- Testing + coding is faster than just coding

Unit testing: testing at all levels

- Modular design: break large task down smaller tasks
- Smallest granularity: C++ functions, or even lines of code
- Smaller granularity modules have less functionality, but are easier to test



- Make sure each unit works!

Unit tests vs. integration tests

- Unit testing tests each component in isolation
  - At high levels, units may be whole programs (e.g., client vs. server)
  - At low levels, units may be individual classes or methods (e.g., disconnect())
- Integration testing tests whether all the components interact correctly with each other
  - Very high level, coarse granularity
  - Often harder to design the tests
  - Assumes each component works properly

Coding to a contract

- The requirements for a unit form its contract:
  - Preconditions, postconditions
  - Promise to whoever interacts with it
- Test against the contract:
  - You can write tests before you code!
- Design to the contract (Bertrand Meyer, Airane5)
  - Structure the code to satisfy the contract
- Code to the contract:
  - Test as you code to ensure correctness

Ariane5 case

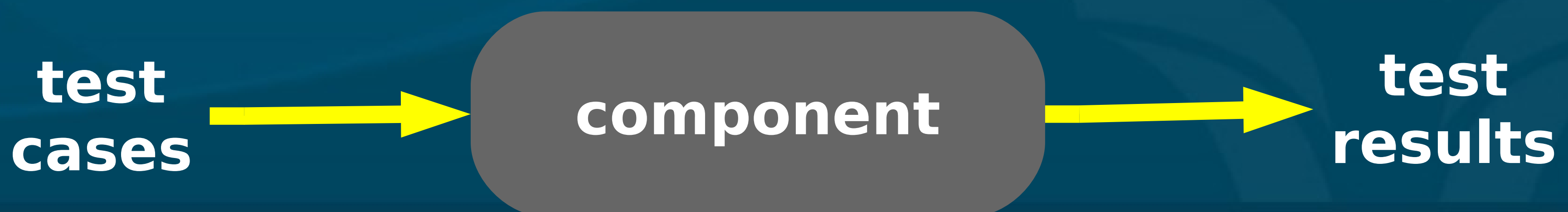
- June 4, 1996: maiden flight of ESA Airane5 space launch vehicle: self-destructs after 40sec
  - Estimated cost: $500 million USD (uninsured)
- Autopilot correcting for illusory severe off-course
  - Both redundant inertial guidance systems had shut down and were spewing error messages
  - Overflow when converting big number from 64-bit long to 16-bit short
- The real error: reusing code from Ariane4 that had no preconditions
  - e.g., “ensure value fits in a 16-bit short”

Example contracts: Stack

- Stack that takes any type object:
  - template <typename Elt> class Stack {
    public:
- push() method with pre/post-conditions:
  - Elt push(Elt item);
  - // pre: none. post: item is at top of stack
- pop() method:
  - Elt pop();
  - // pre: stack has at least one item
  - // post: returns top item from stack;
    top item is removed from stack

Testing against the contract

- Black-box testing:
  don't know/care how the unit is implemented
  - Information hiding: cf. library: *.h vs. *.cpp
- Test cases can be written just from the contract
- Tester and coder may be different people
  - In fact, it's better if they are!
  - Implementation may change; if the contract is same, the tests can be same



When to write tests

- As soon as you have a contract
- Before you start to code (helps clarify design)
- As you code (code a chunk, test a chunk)
- Right after you code (instant gratification)
- Do NOT wait until the whole program is done!
  - You will run out of time or motivation
  - Testing is a design aid –
    it helps you design more modular code

Assertions

- Automated testing: write code to test your code
- One way to express test cases is by assertions:
  - #include <cassert>
- assert(): tests a given Boolean expression
  - assert(size > 0);
    - Prints error message if assertion fails
- Don't change program state in an assertion!
- Set NDEBUG macro to disable asserts
  - #define NDEBUG // disable tests
    - Or compile with: g++ -DNDEBUG

Testing the test code

- Test code may be buggy, too!
- Test it by deliberately breaking your program code and seeing if the tests catch it
- Example: AssertTest.cpp
  - Simple stack using <vector>
  - g++ AssertTest.cpp -o AssertTest.exe