CSE 303
Concepts and Tools for Software Development

Magdalena Balazinska
Winter 2007
Lecture 21 – Unit testing, stubs, and specifications

Where We Are

- Starting to learn basic software engineering
  - In hw4: break system into components
  - Golden rule: w*rite as little code as possible and test!*

- Today: software development process
- In particular
  - Minimal specifications
  - Unit testing and stubs

Motivation

- If you are writing a tiny, simple piece of software for yourself... you don't really need any process. You can just start throwing some code together
- But what if you were in charge of writing the software for a nuclear power plant?
  - You have 20 software developers to help you
  - How would you manage the overall project?
  - How would you go about figuring out what you are supposed to develop?
  - How would you ensure that everyone knows what they are supposed to do?
  - How would you organize everyone's efforts?

Software Development Process

- The software dev. process is there to guide you
- Main steps involved in building a system
  - Requirements analysis
  - Specification
  - Design (high-level then detailed)
  - Implementation
  - Testing
  - Documentation
  - Maintenance

Software Development Process

- Requirements analysis
  - What are we supposed to build? What do our customers need?
- Specification
  - Precise description of provided functionality
  - How precise? Depends on what we are building
- Design (high-level then detailed)
  - Define the internal software architecture
  - Break system into components
    - Modules, interfaces, classes, etc.
    - Need to write specifications for each component

Software Development Process

- Implementation
  - Write the code and perform simple tests
- Testing
  - Extensive testing of components & whole system
- Documentation
  - All steps in the process must be documented
  - User guide, developer's guide, etc.
- Maintenance
  - Basically that means fixing bugs and working on release 1256 of the same product

Software Development Process

- Main steps involved in building a system
  - Requirements analysis
  - Specification
  - Design
  - Implementation
  - Testing
  - Documentation
  - Maintenance
- Order of steps varies, cycles are possible
- How formal? Depends on what you're building

Remember: the software process
- Guides your efforts
- Helps you clarify your thoughts
- Helps you **communicate** your ideas
- It is there to help you!
- You can view it as kind of tool

Specification

- You need to write specs for entire software system but also **for each module**
  - **Man pages are basically specifications**
- Writing a complete specification is often as difficult as writing code (even worse when trying to be formal)
- But, partial specification is better than none
- Clear specification
  - Guides implementation, tests, integration, code reuse
  - Acts as a contract between client and implementor
- Iterating is normal: going back and fixing specs

Function Specification

- We will focus on function specifications
- Specification acts as a contract
  - If client meets its obligations (**precondition**)
  - Implementor meets its obligation (**postcondition**)
- Specification helps **decoupling**
  - Client need not know implementation details
  - Implementor can change implementation details
  - Implementor need not know details of how the function will be used
  - Specifications should thus be **declarative**
    - Describe what a function does but not how it does it

Specification Example

- Something simple like a linked list of strings
- Let's write an informal specification for

```
void insert(Node** head, char* val);
```

Specification First Attempt

```
/**
* Inserts a value into the list
* @param head address of pointer to
*        the first element in the list
* @param val new string to insert
* @return nothing
*/
void insert(Node** head, char* val);
```

A Better Specification

```
/**
* Short description: Inserts a value into a list.
* Precondition:
*   head must be valid address of pointer to beginning of list.
*   List is sorted in alphabetical order.
* Postcondition:
*   Modifies (*head).
*   Inserts val into list pointed to by (*head)
*   Does not check for duplicates.
*   If val is NULL, does nothing
*   Makes a copy of the inserted string.
*   Output list is sorted in alphabetical order.
* @throw nothing (C++ only)
* @param head address of pointer to the first element in the list
* @param value string to insert into the list
* @return nothing
*/
void insert(Node** head, char* val);
```

Minimum Function Specification

- Short description: one line
- State precondition
  - Assumptions about the state of the system in which the function can be called
    - Ex: units are inches, list has no cycles, ...
  - In your code: never trust caller, **check preconditions**
    - Sometimes, it does not make sense to check preconditions (e.g., cannot test that units are inches)
- State postcondition
  - What the function does when the precondition holds

Precondition

- Precondition is an obligation on the client (i.e., the caller of the function)
  - If precondition is violated, the function is allowed to do anything including setting the computer on fire
- Note: for invalid inputs, better to specify what the function does in the postcondition rather than use preconditions
  - Example: when val is NULL, insert does nothing
  - Use the precondition only as a last resort
  - When it does not make sense to handle invalid inputs
    - Ex: assume head holds a valid address
  - Sometimes, use precondition for performance too
    - Ex: assumes input list is sorted

Postcondition

- Describe all input parameters (not really postcondition)
- Identify all objects that can potentially be modified
  - Gobal vars, data members, arguments
  - Sometimes this is called the “frame condition”
- Describe what the function does
  - Describe what the function returns
    - Through return value or by modifying arguments
    - Include any thrown exceptions (C++ only)
  - Describe all side effects
    - Condition that will hold true after function execution
    - Ex: how it modifies data members, what it writes to a file

Testing

- Goal: Verification and validation
  - Does the system work?
  - Does it do what it is supposed to do?
  - Increase our confidence in the system
- How do we know when we are done?
  - Standard coverage metrics
    - Execute each statement at least once
    - Execute each branch or path at least once
  - **Rule of thumb**: there are as many bugs left in the system as you are still finding... never done

Two Basic Types of Tests

- Black box tests: very useful in practice!
  - Test without looking at implementation
  - Someone else than implementor shoud write them
  - Design test cases in terms of specification
    - All tests **must satisfy preconditions**
    - Divide inputs into equivalence classes
      - Need at least one test for each equivalence class
      - Also test boundaries of equivalence classes

Black Box Test Example

```
/**
* Precondition: none
* Postcondition:
* If x is greater than zero, returns the square
  root of x. Otherwise, returns -1
* @param x the number for which to compute sqrt
* @return the square root of x or -1
*/
double sqrt(double x);
```

Some good tests: -20, -1, 0, 1, +20

Other tests: case where sqrt(x) < x, sqrt(x) > x, perfect squares, others

Two Basic Types of Tests

- White box tests
  - Take implementation into account
  - Easier to ensure good coverage
    - All statements at least once (statement coverege)
    - All branches at least once (decision coverage)
    - All possible paths at least once (path coverage)
  - Common sense
    - Try to test all branches at least once

More Types of Tests

- Unit testing
  - Test one or a few functions at the time
  - This is what you will do in hw6
- Integration testing
  - Combining units together
- System testing
  - The whole thing
- Perform them all as your develop the system

Hugely Important in Practice

- Regression tests
  - Whole battery of tests that exercise as many features of the system as possible
  - Rerun all tests automatically
    - Every time you add a feature
    - Every time you fix a bug
- They help verify that everything still works

Stubs

- How to test a “unit” when the other code
  - Does not exist yet
  - Is buggy
  - Is large and slow
- Answer: create a “fake implementation” of the missing pieces
  - Just good enough for the tests
  - As small as possible, so often called stub

Summary

- Software dev. involves a certain number of steps
  - Carefully think what you need to build
  - Carefully think how to build it
  - Prepare tests based on your specs
  - Implement, test, and document
- In assignement 6
  - Your partner and you will agree on a spec
  - One person writes the code
  - Other person prepares black-box tests
  - And then you switch

Readings

- No readings