Design & Modularity

15-441 Recitation 3
Dave Andersen
Carnegie Mellon University

Thinking about Design

- How do you start thinking about how a program should work?
- Data-centric programs:
  - What data does it operate on?
  - How does it store it?
  - Examples?
- Protocol-centric programs
  - How they interact with the rest of the world
  - (Maybe "Interface-centric")
- (Not exclusive! Think about IRC server)

Design Principles

- Goal: once again, pain management

- Be able to develop independently
- Avoid the big brick end-of-semester wall
- Stay motivated

P1: Don't Repeat Yourself

- Aka "DRY"
- Like factoring out common terms…
- If you're copy/pasting code or writing "similar feeling" code, perhaps it should be extracted into its own chunk.

- Small set of *orthogonal* interfaces to modules

P2: Hide Unnecessary Details

- aka, "write shy code"
  - Doesn't expose itself to others
  - Doesn't stare at others' privates
  - Doesn't have too many close friends
- Benefit:
  - Can change those details later without worrying about who cares about them

Example 1:

```
• int send_message_to_user(
           struct user *u,
           char *message)

• int send_message_to_user(
          int user_num,
          int user_sock,
          char *message)
```

Example 2

```
int send_to_user(char *uname, char *msg){
  …
  struct user *u;
  for (u = userlist; u != NULL; u = u->next) {
    if (!strcmp(u->username, uname)
      …
```

Consider factoring into:

```
   struct user *find_user(char *username)
```

- Hides detail that users are in a list
  - Could re-implement as hash lookup if bottleneck
- Reduces size of code / duplication / bug count
  - Code is more self-explanatory ("find_user" obvious), easier to read, easier to test

P3: Keep it Simple

- We covered in previous recitation, but
  - Don't prematurely optimize
    - Even in "optimization contest", program speed is rarely a bottleneck
    - Robustness is worth more points than speed!
  - Don't add unnecessary features
    - (Perhaps less pertinent in 441)

P3.1: Make a few bits good

- Some components you'll use again
  - Lists, containers, algorithms, etc.
- Spend the time to make these a bit more reusable
  - Spend 20% more time on component during project 1
  - Save 80% time on project 2…

P4: Be consistent

- Naming, style, etc.
  - Doesn't matter too much what you choose
  - But choose some way and stick to it
  - `printf(str, args)`   `fprintf(file, str, args)`
  - `bcopy(src, dst, len)`   `memcpy(dst, src, len)`
- Resources: Free where you allocate
  - Consistency helps avoid memory leaks

Error handling

- Detect at low level, handle high
  - Bad:
    malloc() { … if (NULL) abort(); }
  - Appropriate action depends on program
  - Be consistent in return codes and consistent about who handles errors

Incremental Happiness

- Not going to write program in one sitting
- Cycle to go for:
  - Write a bit
  - Compile; fix compilation errors
  - Test run; fix bugs found in testing
- Implies frequent points of "kinda-working-ness"

Development Chunks

- Identify building blocks (structures, algos)
  - Classical modules with clear functions
  - Should be able to implement some with rough sketch of program design
- Identify "feature" milestones
  - Pare down to bare minimum and go from there
  - Try to identify points where testable
  - Helps keep momentum up!
- Examples from IRC server?

Testability

- Test at all levels
  - Recall goal: reduced pain!
  - Bugs easiest to find/correct early and in small scope. Ergo:
    - Unit tests only test component (easier to locate)
    - Early tests get code while fresh in mind
    - Write tests *concurrently* with code. Or before!
  - Also need to test higher level functions
    - Scripting languages work well here

441 Testability

- Unit test examples:
  - Your hash, list, etc., classes
  - Machinery that buffers input for line-based processing
  - Command parser
  - Routing table insert/lookup/etc.
  - Others?

Bigger tests

- More structured test framework early
  - "Connect" test (does it listen?)
  - Alternate port # test (cmd line + listen)
  - …

Testing Mindset

- Much like security: *Be Adversarial*
- Your code is the enemy. *Break it!*
  - Goal of testing is not to quickly say "phew, it passes test 1, it must work!"
  - It's to ensure that 5 days later, you don't spend 5 hours tracking down a bug in it
- Think about the code and then write tests that exercise it. Hit border cases.

Testing a Hash Table

- Insert an item and retrieve it
  - Why?
- Insert two items and retrieve both
  - Why?

[help me fill in this list!]

Note ordering: Simple to complex…

Design & Debugging

- Covering more next week, but…
- Strongly, strongly encourage people to use a consistent DEBUG()-like macro for debugging
- Leave your debugging output in
- Make it so you can turn it on/off