Project 1 Q&A
Design & Modularity

15-440 Recitation 2

Vijay Vasudevan

Carnegie Mellon University

Announcements

- Project 1 due September 17, before class
- Project 1 Documentation Updates
  - Range ordering updated to match tester
    - 1234567890 (different than ASCII ordering)
  - Fix to cracker_checker:
    - mkdir testing
    - cp cracker_checker.sh testing

Agenda

- Project 1 Q&A

- Leftover time: Design and Modularity

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?
- Protocol-centric programs
  - How they interact with the rest of the world
  - (Maybe “Interface-centric”)

Design Principles

- Goal: 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

Modularity example

```
void node_mgr::send_put_response(string* key,
                                 uint32_t c) {

   PutResponse pr(myIP.data(), myIP.size(),
                  key->data(), key_size,
                  c);

   string* send_data = pr.to_string();
   if (send_data != NULL) {
       int err_code = 0;
       if ((err_code = send(feSocket,
                            (void *)send_data->data(),
                            pr.size(), 0)) < 0) {
           perror("send");
           cout << "cannot send" << err_code << endl;
       }
       delete send_data;
   }
}
```

Modularity example

```
void node_mgr::send_get_response(string* key,
                                 string* value,
                                 uint32_t c) {

   GetResponse gr(myIP.data(), myIP.size(),
                  key->data(), key->size(),
                  value->data(), value->size(),
                  c);

   string* send_data = gr.to_string();
   if (send_data != NULL) {
       int err_code = 0;
       if ((err_code = send(feSocket,
                            (void *)send_data->data(),
                            gr.size(), 0)) < 0) {
           perror("send");
           cerr << "cannot send:" << err_code << endl;
       }
       delete send_data;
   }

}
```

Breaking up functions

```
void constructMessage(Message *m, string *send_data) {
    int data_size = m->ByteSize() + sizeof(uint32_t);
    send_data->reserve(data_size);
    uint32_t msg_size = htonl(data_size);
    send_data->append((const char*)&msg_size,
                      sizeof(msg_size));
    if (!m->AppendToString(send_data)) {
       ...
    }
}

void constructAndSend(Message *m, int socket, bool cerr) {
    string send_data;
    constructMessage(m, &send_data);

    int err_code = send(socket,
                        (void *)send_data.data(),
                        send_data.size(), 0);
    if (err_code < 0) {
       ...
```

End result

```
void node_mgr::send_put_response(string* key, uint32_t c) {
    FawnKVMesg fm;
    fm.set_type(PUTRSP);
    PutResponse *prp = fm.mutable_prp();
    prp->set_key(*key);
    prp->set_continuation(c);
    constructAndSend(&fm, feSocket, false);
}

void node_mgr::send_get_response(string* key, string* val,
                                 uint32_t c) {
    FawnKVMesg fm;
    fm.set_type(GETRSP);
    GetResponse *grp = fm.mutable_grp();
    grp->set_key(*key);
    grp->set_value(val);
    grp->set_continuation(c);
    constructAndSend(&fm, feSocket, false);
}
```

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: 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 password cracker?

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

440 Testability

- Unit test examples:
  - Any hash, list, etc., classes you write
  - Machinery that buffers input for line-based processing
    - Are you serializing properly?
  - Others?

Bigger tests

- More structured test framework early
  - “Connect” test (does it listen?)
  - “Timeout” test (do timeouts get triggered?)
  - …

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.

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