ECE 3574: Module and API Design

Changwoo Min

Milestone 0 Release Plan

- **Due: 2/5 by 11:50 pm (hard deadline, no extension)**
- **Don't forget to tag `final`, `final1`, and `final2`**
  - `git tag final`
  - `git push origin final`

Milestone 0 Release Plan

- **From Sunday, our nightly build will test your `final`-tagged code**
  - Monday I will run test and send out emails three times (morning, afternoon, and evening)
- **Extended office hours**
  - My office hours: 2-6 PM Friday (Feb 2nd)
  - SWEL hours: 1-3PM Monday (Feb 5th)

Milestone 1 Plan

- Parsing MIPS assembly code
- **Based on your milestone 0 code!!!**
- Will be release soon (next Tue?)
- Due 2/20?

Meeting 6: Module and API Design

- The goal of today's meeting is to understand what makes a program or library well designed.

Meeting 6: Module and API Design

- These are somewhat subjective criteria but constitute the consensus of a large number of good programmers.
  - DRY Principle
  - Single-Responsibility Principle
  - Orthogonality
  - Coupling and Law of Demeter
  - Principle of Least-Astonishment
  - Naming
  - Some *Code Smells*

Find bad taste here...

> *I think **this is an example of not particularly good taste in code**, and this one is better taste, which one can immediately see. - Linus Torvalds*

```
remove_list_entry(entry)
{
    prev = NULL;
    walk = head;
    /* Walk the list */
    while (walk != entry) {
        prev = walk;
        walk = walk->next;
    }
    /* Remove the entry by updating
     * the head or the previous entry */
    if (!prev)
        head = entry->next;
    else
        prev->next = entry->next;
}
```

Don’t Repeat Yourself (DRY Principle)

- Duplicated code leads to systems that are hard to maintain and understand.
- **Most importantly, you should fix the same bugs several times.**
- Some causes of duplication
  - The environment seems to require duplication
  - The developer does not realize they are duplicating code
  - The developer is in a hurry
  - Poor communication in a team

Comments can be a source of duplication

- Consider the following example of a function definition

```
// function to convert a std::string holding an IP4 address
// as xxx.xxx.xxx.xxx to an unsigned 32 bit integer.
std::uint32_t ip_string_to_int(std::string ipstr);
```

- What happens if I change the return type to a signed and/or larger width integer?
- Don’t put low-level details in comments.
- **Comment intent.**

Comments can be a source of duplication

```
/* Consider a class Foo defined in a header file Foo.h with a custom
   constructor */
    class Foo ...
        // Construct a Foo from an integer such that
        Foo(const int & x);

/* where the implementation file Foo.cpp contains */
    // Construct a Foo from an integer such that
    Foo::Foo(const int & x){ ... }
```

- Don’t duplicate comments between headers and implementation files.
- Put API type comments in the header (how to use the class/method or function)
- Put detailed comments about implementation in the implementation (assumptions, possible issues, etc)

Projects often require documentation and code

- Separate detailed API documentation from descriptive and instructive documentation.
- **Automate the generation of API documentation directly from code**
  - Tools: Doxygen, Sphinx Pydoc, etc.
  - Example: LLVM, Linux Kernel
- Use authoring tools that enable mixing of code and text
- The latter can be taken to an extreme of mixing text and code in the same file, dubbed *literate programming*.

Common code across platforms and languages can cause duplication

Suppose you were writing a webapp that had a backend server component written in C++ and three clients: a desktop app in C++, an Android app in Java, and an iOS app in Objective-C. They will be sharing some data back and forth.

- Should each code base have its own definition of the data?
- How could you prevent duplication?

Duplication in code can be very subtle

- Consider a struct modeling a Line. Obviously lines have two ends and a length.

```
struct Line {
    Point start;
    Point end;
    double length;
};
```

- **But what if the code changes one of the points? Does the length change?**

Duplication in code can be very subtle

- A better approach is to compute the length.

```
struct Line {
    Point start;
    Point end;
    double length() {return distance(start,end); };
};
```

This can always be cached for performance reasons.

```
class Line {
public:
    // set accessors toggle changed, ex
    void setStart(const Point & p){
        start = p;
        changed = true;
    }
    // length can use a cached value
    double length() {
        if(changed){
            length = distance(start,end);
            changed = false;
        }
        return length;
    };
private:
    Point start, end;
    double length;
    bool changed;
};
```

Duplication because of Laziness

We’ve all been there. "I don’t have time to pull this out into a separate class/function." "I’ll just **copy/paste** this for now and clean it up later . . . ."

- In general, make things easy to reuse, so that when there is an opportunity to do so - you will.
- **By reusing, we can find bugs earlier and your one bug fix will apply to many places.**

Single-Responsibility Principle

- Any block of code, Module, class, function, or method should do one well-defined thing, and do it well.
- Another phrasing is that it should have *one, and only one, reason to change*.
    - classes should be small
    - classes should have only a few members (variables and functions)
    - methods that can be implemented outside the class should be split into functions

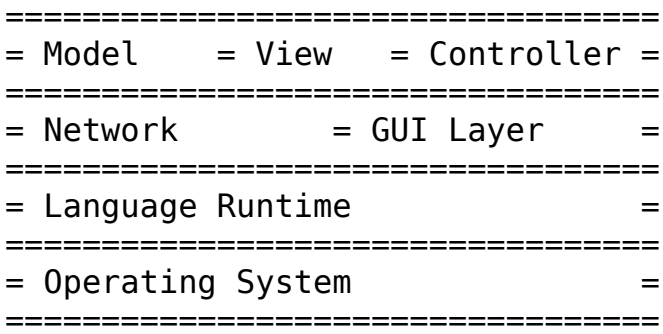
Single-Responsibility Principle

- A code smell that points to violations of the principle are
  - obese classes: more than 5 or so private member variables, more than 10 or so member functions
  - long (member) function implementations

Orthogonal designs reduce complexity

- eliminate effects between unrelated things
- define contracts and test them
- use design patterns that enhance orthogonality
- isolate third party dependencies
- Globals are antithetical to orthogonal designs
- **Write shy code**
  - *Don't reveal anything unnecessary to other modules*
  - *Don't rely on other modules implementation*
- **Inheritance vs. composition?**

Example: orthogonal design using layering



Reduce Coupling

- Organize your code into modules and limit interactions between them.
- Some symptoms of overly-coupled systems:
  - hard to test
  - a change in one module requires changes in many others
  - you are afraid to change code because you don't know who is using it
- The solution: use the *Law of Demeter*

Law of Demeter for methods / member functions

- Any method of an object should call only methods belonging to:
  - itself
  - any parameters that were passed to it
  - any objects it creates
  - any local objects
- In particular no reaching into another class using chaining. Example:

```
int foo = var.someMethod()->doSomething().value;
```

Paper Delivery Example (David Bock)

- A customer usually does not give the wallet to the delivery person to pay.
- Instead, a customer takes out the money from the wallet and gives it to the delivery person.
- Instead of changing `customer.wallet.money` in class DeliveryPerson, use a method `customer.getPayment(..)`.
- This *delegates* the payment retrieval to the customer.
- Proper encapsulation prevents this somewhat. The Customer class should not allow public access to it's wallet.
- E.g., Model-View-Controller, that facilitate this kind of delegation.

Principle of Least-Astonishment

- Minimize the surprise users of your code experience.
  - For end users, don't violate standard practice without good reason.
  - For other programmers, don't do unrelated work or cause unexpected side-effects
- Often *you* are the other programmer, using your own code.
- Suppose you wrote a super fast memory copy:
  - `super_fast_memcpy(void *x, void *y, size_t size)`
  - **Q: What should be the destination address?**

Naming

> *There are only two hard things in Computer Science: cache invalidation and naming things. – attributed to Phil Karlton*

- names should reflect their visible scope, the larger the scope, the longer the name (in general)
- `useCamelCase` or `dont_use_camel_case`, but **be consistent**
- don't use "Hungarian" notation (prefix `p` for pointer, `i` for integer, etc)
- A somewhat acceptable violation of the last rule is prefixing member variables with `m_`, but I don't like it much either.

Code Smells: Functions and Methods

- Function names should say what they do in a specific sense.
- Keep the number of function arguments under three. More than that indicates the code needs to be refactored.
- Functions should be short and do one thing. No function should be larger than you can see on the screen at once and preferably shorter.

Code Smells: Functions and Methods

- Break calculations up into meaningful intermediate expressions and name the variables accordingly. The compiler is good at optimizing this out and it enhances readability.
- When possible do not use output arguments, those passed by reference or pointer should be marked `const`.
- Boolean flags in arguments indicate a violation of the single responsibility principle. Create a separate function instead.
- Remove functions not called (dead functions).

Code Smells: Classes

- Name classes by what they are or what they do. For example, not just `Writer` or even `FileWriter`, but `ConfigurationFileWriter`.
- The public section of a class should be as small as possible. Obese classes indicate a violation of the single responsibility principle.
- Avoid writing accessors (getters and setters) for every private member. If you do it should enforce a constraint.

Further reading

- Chapter 3. Bad Smells in Code

Exercise 06: An Instruction Class

See Website

Next Actions and Reminders

- Read the Catch Tutorial