Week 2: Review of Internal Program Quality

What makes one program "better" than another?

What is a good program?

Suppose we give the same assignment to two programmers (or to two teams of programmers).

Suppose each of them produces a program that:
- produces the correct output.
- has almost the same user-interface.

How can we judge which program is better?

Why do we care?

What is a "good program"?

Objective (easy-to-measure) criteria
- Correctness -- Gives the right results:
- Efficiency -- cost of operation:
  - space
  - time
  - other resources
- Performance
  - Reliability, stability
  - Response time, throughput

Subjective (harder-to-measure) criteria
- Compared with what?

Which are more important?
**What is a "good program"?**

- **Easy-to-measure criteria**
  - Maintainability -- cost of future enhancement
  - Modularity
  - Readability

- **Harder-to-measure criteria**
  - Scope, level of generality
  - Usability  Intuitiveness, user-friendliness

**Which are most important?**

---

**A good program component:**

- Solves the right problem:
  - Its scope isn't so specific that it can't be used in other similar contexts.
  - But it's not so general that it's unduly complicated to use.

- Is easy and economical to maintain:
  - It's easy to change.
  - It's easy to understand.
  - It's not unnecessarily prone to bugs.

*Modular organization is essential here.*

---

**Part 1 -- Principles of modular program organization**

- Measures of modularity
  - Parameterization
  - Coupling
  - Cohesion
  - Size
  - Generality
    - Reusable components

---

**Modular programming**

- Since the 1960's just about everyone agrees that "modular" is good.
- Almost every programmer claims to do modular programming.
- But only a minority of programs really are modular.
- One cause is that many people don't agree on exactly what "modular" means
- Here's one definition (3 characteristics):
Measures of modularity:

1. Each program attribute (value, pattern, data representation, etc.) is known in only a single place.
   - Such a program is said to be highly parameterized, and the attributes are localized.
   - That one place should be easy to find -- the logical place one would think to look.
   - When related program attributes are packaged together, they're said to be encapsulated.
   - Repetition is the enemy!

2. Each module performs one well-defined function at a single level of detail.
   - Such a module is said to have high cohesion or strength.
   - But what's a module?

3. Each module depends on other modules only through explicit, well-defined interfaces.
   - Such a module is said to have low coupling or low interdependency.

What is a module?

- We used to equate module with subroutine or executable function, but the term is now used more generally.
- A module can be any piece of code (or collection of closely related pieces), such as:
  - A subroutine or function
  - A macro definition or package of related macro definitions
  - An object-oriented class
  - A table
- Some programming languages and tools have their own specialized definition

A common myth

- If a complete program consists of a very large number of subroutines or functions, then it's modular.

- Agree or disagree?
A common myth

- If a program consists of a large number of subroutines or functions, then it's modular.

- But what if 35 out of 200 modules all know the sequence of fields in a data structure?

- What if they all know the internal representation of dates or amounts of money?

- Such a program would be highly unmodular!

Advantage of low coupling

A 2000 statement monolithic program

- Anything inside can potentially interact with anything else

- Nothing in here . . .

- Anything in here . . .

25 modules of about 80 statements each

How many potential interactions in each?

Common techniques that lead to high (poor) coupling

- Using global (or external) data instead of explicit parameters to communicate between functions

  - That's very common in: Why is that bad?
    - COBOL programs
    - BASIC programs
    - because early versions of those languages supported a parameterless subroutine-linkage (PERFORM, GOSUB) statement

- Repeating knowledge in multiple modules:
  - Structure definitions
  - Constant values

Common techniques that lead to low (poor) cohesion

- Packaging unrelated functions together just because they're done at about the same time.

  // This module computes deductions for
  // an employee and prints the paycheck.

- Using "function codes" or "option switches" to force multiple functions into a single routine:

  long numFunc(long m, long n, int code)
  // If code=1, returns greatest common
  // divisor of m and n
  // If code=2, returns least common
  // multiple of m and n
  // If code=3, returns absolute value
  // of m - n
A common beginner's cohesion violation
- Beginning programmers often intermix:
  - a function that produces some result with
  - console user dialog (or I-O) to obtain parameters and/or display results or error messages
- Computational functions as well as object-oriented methods (member functions) should rarely if ever conduct dialog with the online user.
- Unfortunately, some textbooks and "experts" violate these criteria!

Constants: 3 kinds:
- An absolute constant occurs naturally in mathematics.
  Examples?
- A fundamental constant is a property of the application or the environment, determined by measurement, by policy, or by tradition.
  Examples?
- A derived constant is a function of one or more fundamental constants
  Examples?

Binding time of a constant
- A constant is said to be bound:
  - at coding time if you have to look through the code to change all of its occurrences (also called a "hard coded" constant).
  - at compile time if you can change it by changing a single occurrence that's in a logical and easy-to-find place in the program code. (also called a "parameterized" or "localized" constant).
  - at run time if you can change it without modifying the program at all, i.e. via some kind of input.

Coding-time binding (hard-coding) is OK for
- Absolute constants if they're short:
  - Most instances of 0 or 1, etc. in formulas.
  - The blank character ' ', or delimiter punctuation '
  
  
  What about PI?
Impact of object-oriented programming
- OOP is neither necessary nor sufficient for achieving highly modular structure and thus easily maintained programs.
- Nevertheless, OOP:
  - encourages highly modular structure
  - helps greatly in packaging highly modular programs
  How?

Size criterion: How big should a module be?
- Everyone agrees that a 5000 statement function is much too big. What's a reasonable limit?
  Does anyone actually write 5000 line monolithic programs?
- Readability is greatest when we can view an entire module on a printed listing, without turning pages.
- Two pages of printed source code = about 100 lines including commentary.
  - Future screen displays may soon accommodate that much source code text.
  - Some do already!

How can a programmer limit module size
- Consider dividing a module into two or more smaller modules whenever:
  - The logic is getting too complicated to keep track of, e.g. deeply nested loops or conditionals.
  - A non-trivial pattern of code appears two or more times.
  - The code is more than about 100 lines long, including commentary.
- But division mustn't be arbitrary.
  - Comment: // This function performs the first half of the inventory forecasting algorithm
- Can a module be too small?

Generality and code reuse
- It has been estimated that up to 80% of the program code that's written is redundant. The same thing has been coded before:
  - Somewhere in the world
  - In the same company
  - Even on the same project!
- If we could reduce that even to 50%, software development and maintenance would be far less expensive and more predictable.
- Modular program organization is essential before we can have easy reuse. Why?
  - That's why most organizations that used COBOL as their main language failed to achieve high reuse.
A central module library

- Every organization doing serious software development should establish a repository of reusable components. *What's a component?*
  - Every non-trivial project should contribute several new components.
  - As the library matures, the ratio of custom code to library code in new programs will continue to diminish.
- Object-oriented programming has renewed interest in reuse, because it provides a natural way of packaging components.
  - But you don't need OOP to achieve high reuse.
  - Look for opportunities to generalize (extend the scope of) a module.

Program Quality Part 2: Review of Program readability

- Commentary
- Choosing data names
- Code layout
- Commentary
- Pitfalls

Reading a program

- A program is not only something to be run on a computer, but also a document for people to read.
- We can assume the reader is an experienced programmer.
- The reader may is often the original programmer at a later time.

What makes a program readable?

- Simple structure
- Clear presentation, layout
  - indentation
  - alignment
  - white space
- Good commentary
- Appropriate data names
**Program commentary**

- All programming languages permit comments in some form.
- Some languages provide a *line-oriented* comment delimiter. Any source text to the right of the delimiter is a comment.
- Other languages provide a *bracketing pair*. Source text between a left delimiter and the next right delimiter is a comment, and may extend over multiple lines.
- C++, Java & C# provide both! *more useful?*
  - C uses /* and */ (originally the PL/I convention)
  - C++, Java, & C# also allow //

**Where do we write comments?**

- A *title* line introduces a function or other kind of module or an entire source-code file.
- *Introductory* comments describe the purpose and usage of a function or module.
- *Block* comments describe the purpose and strategy of a group of program statements.
- *Line-by-line* comments explain an individual statement or even part of a statement. They are needed more in assembly language than in high-level language programs.

*What about Javadoc?*

**When do we write comments?**

- Internal program documentation is an *integral part* of programming not a separate activity.
- Title and introductory comments are best written *before* the code. This helps the programmer to clarify his or her thoughts, and can actually save time.
- Line by line comments should be written when the code is written.
- Block comments can be written before, during, or after the code.

**What about Javadoc?**

- Javadoc was an attempt to:
  - standardize usage documentation
  - integrate usage documentation with internal (maintenance) documentation

*Was that a good idea?*

- Many Java textbooks use Javadoc
  - They often omit title comments.

*What does Javadoc do?*

- C# equivalents based on XML are available
Good line-by-line comments

- Avoid restating what's obvious from the code
- Describe what is being done, not how, i.e. emphasize the intent or effect.

Examples

Not  ++posn; // advance the position
but  ++posn; // skip over the comma

Not  weight *= 2.2 // multiply by conversion factor
but  weight *= 2.2 // convert to pounds

Not  while(count>0) // Loop until count exhausted
but  while(count>0) // Examine all work orders

Another use for bracketing (/* . */) comment delimiters

- In addition to documentation, comment delimiters can be used for temporarily removing ("commenting out") a block of code from the program.
  - What use is that?
  - What about comments inside the commented-out block?
  - Is there a better way?

Indenting flow logic

- In most programming languages, the programmer can control horizontal spacing.
- We exploit that capability to show levels of flow structure.

The compiler considers this:

```
while (lbound <= hbound)
    {midp = (lbound + hbound) / 2;
     if (arg == tbl[midp])
        return midp;
     if (arg < tbl[midp])
        hbound = midp - 1;
     else lbound = midp + 1;
    }
return -midp;
```

to be equivalent to:

```
while (lbound<=hbound){
    midp = (lbound+hbound)/2;
    if (arg == tbl[midp])
        return midp;
    if (arg < tbl[midp])
        hbound=midp-1;
    else lbound=midp+1;
} return -midp;
```

- but a human reader doesn't.

Page width and line length

- Old programming languages limited program statements to 72 or 80 characters. Why?
- Later languages allowed free-form statements on multiple lines, but still limited the line size.
- Many current editors, and compilers including the C family, allow lines of arbitrary length!
  - Is that good or bad?
  - Never ever force the reader to use horizontal scrolling.
White space

- Occasional blank lines help the reader to separate logically distinct sections of code.
- Multiple blank lines or skipping to the top of a fresh page help to introduce a new function.
- But unfortunately:
  - Standard C++ and Java provide no built-in listing-control facilities, for example to force a page break.
  - Today's screens are too small to view much code at a time, especially when multiple blank lines are embedded.

Aligning corresponding elements

- A sequence of similar expressions (or other statements) is greatly clarified by aligning corresponding parts. Compare:

  ```cpp
  INT Black   = 0;
  INT Blue    = 1;
  INT Green   = 2;
  INT Red     = 4;
  INT Cyan    = Blue + Green;
  INT Magenta = Blue + Red;
  INT Amber   = Green + Red;
  INT White   = Blue + Green + Red;
  ```

  with

  ```cpp
  INT Black = 0;
  INT Blue = 1;
  INT Green = 2;
  INT Red = 4;
  INT Cyan = Blue + Green;
  INT Magenta = Blue + Red;
  INT Amber = Green + Red;
  INT White = Blue + Green + Red;
  ```

  Some compilers/editors change the first version into the second!

What's wrong here?

- The program is finished. All we have to do is to go back and document it.

Choosing data names

- Names should be mnemonic, suggesting the purpose or usage of the data item from the point of view of the module.
- Names should be long enough to be mnemonic (or self-documenting) but not so long as to force typical statements to span multiple lines.
- Single character variable names are sometimes appropriate for abstract mathematical quantities or for bound variables having a short scope (e.g. a loop index)
Comments and data names

- By choosing a meaningful data name, we may avoid the need for a line-by-line comment.
- Examples:

  Instead of `t0=sc.nextInt(); // Set starting temp.
  code
  startTemperature = sc.nextInt();`

  Instead of `weight *= 2.2; // convert to pounds
  code
  weightInPounds = weight * kgToPound;`

  Instead of `while(count>0)// Examine all work orders
  code
  while(workOrderCtr > 0)`

A modern phenomenon

- Presumably today’s professional programmer is more aware of enlightened coding techniques than the programmer of 25 years ago.
- Nevertheless, much source code written today is less readable than a typical program from 1972!
- What accounts for this surprising result?
  - interactive editing and on-line time pressure
  - the screen-size limitation
  - compilers with poor support for producing readable printed listings

Egoless programming (Gerald Weinberg)

- Many old-time programmers viewed programming as a private activity.
  - No one but the original programmer was expected to look at source code.
  - Programmers kept their source code in their own files.
  - Managers seldom rated a programmer’s job performance by the internal quality of his or her programs.
- A minority of programmers and managers still hold that view!

Egoless programming (continued)

- Today’s enlightened approach emphasizes writing code for an audience, subjecting one’s work to peer review (“structured walkthroughs”), and filing the end products for anyone to look at.
- A secure professional likes to show his or her work to colleagues.
- One recent fad, “Extreme Programming” (XP) calls for programmers always to work in pairs.

What else?
What does all this have to do with software quality?

- This lecture was about **internal quality**

- But as we noted earlier:
  - Our textbook is mainly about **external quality**.
  - and poor internal quality also undermines external quality.