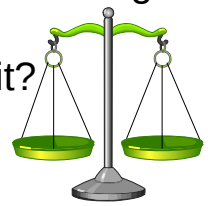
What ever happened to Software Quality?

- What makes one program *better* than another?
- Why is atrocious code still being written?
- What can we do about it?



for ACM Chicago Chapter
May 12, 2010
Conrad Weisert

Agenda

- **Prolog:** 3 examples of atrocious programs.
- **Part 1:** Diagnosis
  - How and why do atrocious programs get written?
- **Part 2:** What is a *good* program?
  - Scope
  - Modular program structure
  - Readability
- **Part 3:** What can be done?

Prolog

3 atrocious short examples

- I am *not* making these up!
  - These examples came from *real* software developed by respected organizations or even published by well-known "experts".
  - They aren't unique or even unusual; we encounter similar and far worse examples regularly

Atrocious example #1

- Found in a production application written by a "senior" **COBOL** programmer:

```
IF CURRENT-MONTH = '01'
 MOVE +1 TO MONTH-NUMBER IN TRANSACTION;
ELSE IF CURRENT-MONTH = '02'
 MOVE +2 TO MONTH-NUMBER IN TRANSACTION;
ELSE IF CURRENT-MONTH = '03'
 MOVE +3 TO MONTH-NUMBER IN TRANSACTION;
ELSE IF CURRENT-MONTH = '04'
 MOVE +4 TO MONTH-NUMBER IN TRANSACTION;
ELSE IF CURRENT-MONTH = '05'
 MOVE +5 TO MONTH-NUMBER IN TRANSACTION;
ELSE IF CURRENT-MONTH = '06'
 MOVE +6 TO MONTH-NUMBER IN TRANSACTION;
ELSE IF CURRENT-MONTH = '07'
         .
         .
```

Atrocious example #1 follow-up

- In a management seminar we explained that that half page of code is equivalent to simply:

```
MOVE CURRENT-MONTH
  TO MONTH-NUMBER IN TRANSACTION;
```

- One manager then objected:

"Why worry about elegance?
The orginal version worked, didn't it?"

A 1960's programming manager's cliché

- *Agree or disagree?*

Atrocious example #1 (conclusion)

```
ELSE IF CURRENT-MONTH = '07'
 MOVE +7 TO MONTH-NUMBER IN TRANSACTION;
 ELSE IF CURRENT-MONTH = '08'
  MOVE +8 TO MONTH-NUMBER IN TRANSACTION;
 ELSE IF CURRENT-MONTH = '08'
  MOVE +9 TO MONTH-NUMBER IN TRANSACTION;
 ELSE IF CURRENT-MONTH = '10'
  MOVE +10 TO MONTH-NUMBER IN TRANSACTION;
 ELSE IF CURRENT-MONTH = '11'
  MOVE +11 TO MONTH-NUMBER IN TRANSACTION;
 ELSE IF CURRENT-MONTH = '12'
  MOVE +12 TO MONTH-NUMBER IN TRANSACTION.
```

How many test cases are needed to give us confidence in the code?

Atrocious example #2

- C++ statement in a program written by an out-sourcing contractor for a major software company:

```
assert(!((!Parm.activation.isPropertyContained)
|| (Parm.activation.isPropertyContained
&& ! Parm.activation.isTransactionIDSame)||
(Parm.activation.isPropertyContained
&& !Help.CheckImplementationsFit())));
```

- *How did it get that way? What could the programmer(s) have been thinking?*
- *What can be done about it?*

Atrocious example #2 follow-up

```
assert(!((!Parm.activation.isPropertyContained)
|| (Parm.activation.isPropertyContained
&& ! Parm.activation.isTransactionIDSame)||
(Parm.activation.isPropertyContained
&& !Help.CheckImplementationsFit())));
```

- First let's just tidy up the layout:

```
assert(
!( (!Parm.activation.isPropertyContained)
|| (Parm.activation.isPropertyContained
     && !Parm.activation.isTransactionIDSame)
||(Parm.activation.isPropertyContained
     && !Help.CheckImplementationsFit() )
);
```

Now what?

Atrocious example #2 (conclusion)

- The final simplified version is:

```
assert(Parm.activation.isPropertyContained)
   &&  Parm.activation.isTransactionIDSame)
   &&  Help.CheckImplementationsFit() );
```

- *How would a programmer know that*
  - *if he or she understood symbolic logic or Boolean algebra?*
  - *if he or she didn't?*

Atrocious example #3

- From an article in a major trade journal:

```
public int[] primeFactors(int n)
   int factor = 2;
   int ctr  = 0;
   int[] factorRegister = new int[100];
   while (factor <= n)
     {while (n % factor == 0)
        {factorRegister[ctr++] = factor;
         n /= factor;
        }
      ++factor;
     }
     .
     .
     .
  return result;
```

How can we fix this one?

Part 1: Diagnosis

- How does such atrocious code come to be written?
- What can we do about it?

What accounts for the first two atrocious examples?

- Incompetent programmers
- Abandonment of design/code reviews *Why?*
- 21st Century managers
- Sloppy programmer education

Incompetent programmers

- We've known (SIGCPR,etc.) for decades that
  - There's a **20-to-1 range** in programmer productivity
  - Usually the most productive programmers also turn out the **highest quality** programs
- That means that one programmer may take **two weeks** to produce what another programmer produces in **an afternoon** (and it won't be as good).
- But many managers still don't believe it!
  - "*A programmer is a programmer.*"

Incompetent programmers

- Even if management understands, they may not be able to identify top performers.
  - recruiting new staff
  - pruning old staff
  - salary range
  - offshore outsourcing
- The creators of the 3 examples probably *shouldn't be employed* as programmers

Design/code reviews: 2 kinds

- **Walkthrough** ("peer") review
  -
- **Quality assurance** review
  -
- *What's the difference?*
- It's unlikely that the atrocious examples were reviewed in either way.

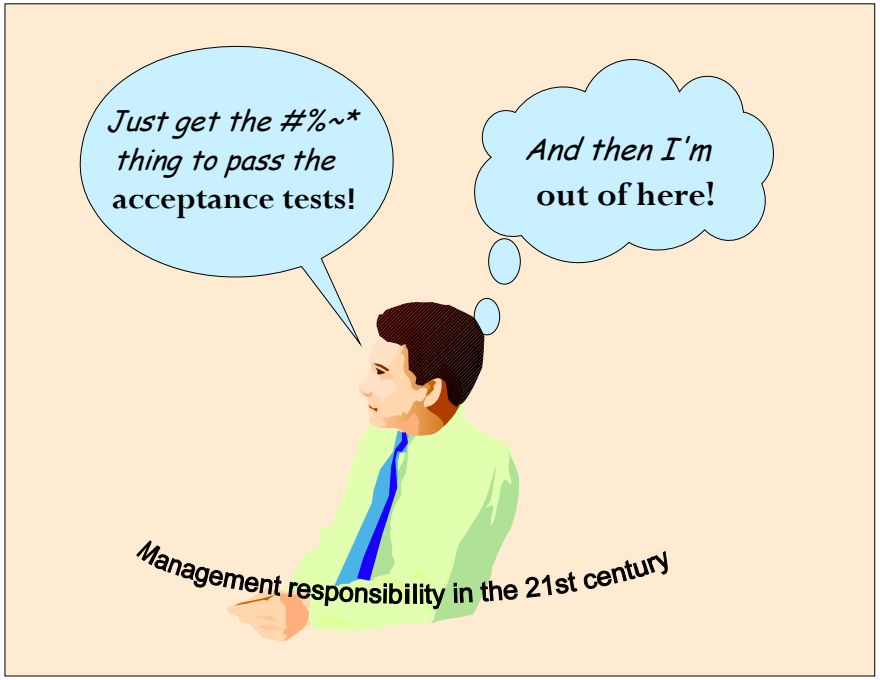
Design/code reviews: 2 kinds

- **Walkthrough** ("peer") review
  - suggestions for improvement; is there a better way?
- **Quality assurance** review
  - flagging deviations from standards / methodology

What about errors?

- Both were essential components of the structured revolution, ca 1976
- But **both** have been widely abandoned by impatient 21st century managers *Why?*

Can pair-programming compensate?



A 1990s trend in American business

- More and more managers are judged on **very short term** performance, sometimes just the current quarter!
- By the time the organization feels the consequences of poor quality, that manager will no longer have functional responsibility
- Therefore, quality is of little or no interest to many managers.

What can be done about that?

Programmer education

- Most ***early*** programmers learned on the job
  - apprenticeship
  - in-house courses
- ***Today's*** programmers learn mostly in academic courses
  - grading emphasis is on *right answer*
  - role of T.A. in grading
  - "points off"
- Are instructors always aware of quality?
  - See end of `www.idinews.com/atrocious.html`

What accounts for the third atrocious example?

- **TFD** (Test-first development)
  a. Write a test case for the module
  b. Write and run just enough code in the module to make that test case yield the expected result (without affecting any previous test cases)
  c. Return to step a until test coverage is complete

Sounds good!

What's wrong with that?

Danger of TFD

- **TFD** may (and often does) distract the programmer(s) from module design
  - appropriate data representations
  - effective algorithms
  - future flexibility
  - module interfaces

"DTSTTCPW" ("Emergent design")!

- Can a **good process** produce a **bad result**?

Part 2: 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*?

What is a "good program"?

- Objective (easy-to-measure) criteria
  -
  -
  -
  -
- Subjective (harder-to-measure) criteria
  -
  -
  -

Which are more important?

What is a "good program"?

- Easy-to-measure criteria
  - **Correctness** -- Gives the right results:
  - **Efficiency** -- cost of operation:
    - space
    - time
    - other resources

Compared with what?

  - **Performance**
    - Reliability, stability
    - Response time, throughput
- Harder-to-measure criteria

What is a "good program"?

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

What do those mean?

  - **Scope**, level of generality
  - **Usability** Intuitiveness, user-friendliness

- **Quality** encompasses ***all*** of the above.

Does today's "QA community" understand that

Software Quality ≠ Absence of defects

- Who is the QA community?
- Unfortunately the **acceptance testing** community has appropriated the term "Quality Assurance"
  - So that some naive managers now believe that's all there is to software quality!

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.

Let's look at these two important quality criteria

- **Modularity**
  - Parameterization
  - Coupling
  - Cohesion
  - Size
  - Generality; reusable components
- **Readability** (or understandability)
  - Commentary
  - Data naming
  - Code layout
  -

These are important with ***all*** programming paradigms.

Modular programming

- Since the 1960's just about everyone agrees that "modular" is good.
- Almost everyone 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):

A common claim

A program that consists of a huge number of subroutines or functions is modular.

Agree or disagree?

- But what if 33 out of 200 modules all know the sequence and alignment of fields in a data structure (or record)?
- What if they all know the internal representation of dates or amounts of money?
- Such a program would be highly ***unmodular***!

Three 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***.

Measures of modularity (continued)

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 defintions
  - An object-oriented **class**
  - A **table**
- Some programming paradigms, languages and tools have their own specialized definition

Common techniques that lead to *high* (poor) coupling

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

Why is that bad?

  - This is very common in:
    - **COBOL** programs and **BASIC** programs because early versions of those languages supported a parameterless subroutine-linkage (**`PERFORM`**, **`GOSUB`**) statement
    - **JAVA** etc. pseudo classes.
- 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

- In beginning programming courses and textbooks, examples often intermix:
  - a function that produces some result with
  - console user dialog to obtain parameters and/or display results or error messages
- Computational functions as well as application-domain object-oriented *methods* (member functions) should rarely if ever conduct dialog with the online user.
- Forms driven (GUI) interfaces may aggravate this problem.

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

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 monoliths?
- 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.

Limiting module size

- The programmer should 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.
  - `//  This function does the first half`
    `//  of the inventory forecast`
- 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 module reuse.

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 well be the original programmer at a later time.

What makes a program readable?

- Simple structure
- Clear presentation
  - indentation
  - alignment
  - white space
- Good commentary
- Appropriate data names

Where do we use comments?

- A ***title*** comment 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 stragegy 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.

Instant disqualifiers

- When I look at a source code listing, I expect to see immediately:
  - What it is; what it does; what it's for
  - How to invoke it
  - i.e. the title and introductory commentary
- A module that fails to convey that information clearly flunks.
  - Surprisingly many actually do.
  - Is the "agile" approach to blame?

When do we write comments?

- 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 as the code is written.
- Block comments can be written before, during, or after the code.

Good line-by-line comments

- Avoid restating what's obvious from the code
- Describe ***what*** is being done or ***why***, 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`

Page width and line length

- Ancient programming languages limited 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.
- **Never** let line wrap mess up indentation

See `www.idinews.com/peeves/tooLong.html`

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.
  - Today's screens are too small to view much code at a time, especially when blank lines are embedded.

What's wrong here?

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 very short scope (e.g. a loop index)

Comments and data names

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

Instead of code

```
cin >> t0; // Set starting temperature
cin >> startTemperature;
```

Instead of code

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

Instead of code

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

A modern phenomenon

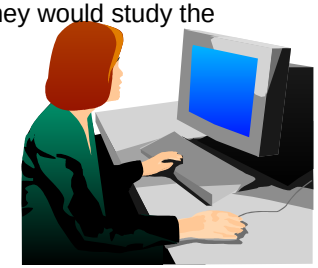
- Presumably today's professional programmer is more aware of enlightened coding techniques than the programmer of 25 years ago.
- Nevertheless, much source code written today is *less* readable than a typical program from 1972!
- What accounts for this surprising result?
  - interactive editing and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

What else?

Impact of online development

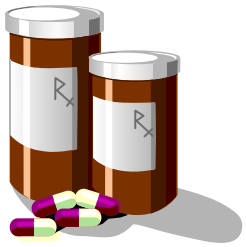
- Programmers used to work on multiple problems at a time
  - They would submit a batch test job
  - While waiting for the results they would work on something else
  - When the job came back they would study the results
- Today we work on one programming task until we think it's done.
  - Why?
  - Because we *can*!

Pros & cons?



Part 3: What can be done?

- Staffing
- Programmer education
- Mentoring
- Design & code reviews
- Setting examples



Building a staff

- If performance ratios are **20-to-1** and salary ranges are **3-to-1**:
  - Don't settle for "above average"
  - Go for the 85th percentile or better
- When recruiting a candidate **always** examine actual work samples
- How should we choose between:
  - a superb programmer who doesn't know the tools we need, and
  - an average programmer who can "hit the ground running"?
- When evaluating performance, **never** rely on measuring ***lines of code***.

Essential infrastructure

- Methodology
  - activity (process) oriented
    vs.
  - results (quality) oriented
- Quality assurance
  - reviews
- Professional development (training)
- Reusable component library

See **`www.idinews.com/methodol.pdf`**

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

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 *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.
  - Is that a plus or a minus?

More information

- Web site with articles, book reviews, etc.
  `www.idinews.com`
  Something new every month!
- Classic books, including:
  - Glenford Myers: *Composite Structured Design*, Van Nostrand Reinhold, ISBN 0-442-80584-5
  - Freedman & Weinberg: *Handbook of Walkthroughs, Inspections, and Technical Reviews*, Little Brown, ISBN 0-316-292826
  - Kernighan & Plauger: *The Elements of Programming Style*, McGraw Hill, ISBN 0-07-034207-5
- Write me at **`cweisert@acm.org`**