Software Engineering

Lecture 08 – Code Quality

© 2015-20 Dr. Florian Echtler
Bauhaus-Universität Weimar
<florian.echtler@uni-weimar.de>

Today's topics

- Software quality metrics
- “Code smells”
- Antipatterns

Software quality attributes

Source (FU): Sommerville, Software Engineering, Chapter 24

- Safety, Security
- Reliability, Resilience, Robustness
- Understandability, Learnability, Usability
- Reusability, Adaptability, Portability
- Modularity, Complexity, Maintainability
- Efficiency, Testability

Code quality metrics

- Functional quality:
  - Compliance to functional requirements/ specifications
  - Usually determined by automated tests (= *dynamic analysis*)
- Structural quality
  - Compliance to non-functional requirements (robustness, maintainability)
  - Determined by “lint” checkers, code review (= *static analysis*)

Management issues

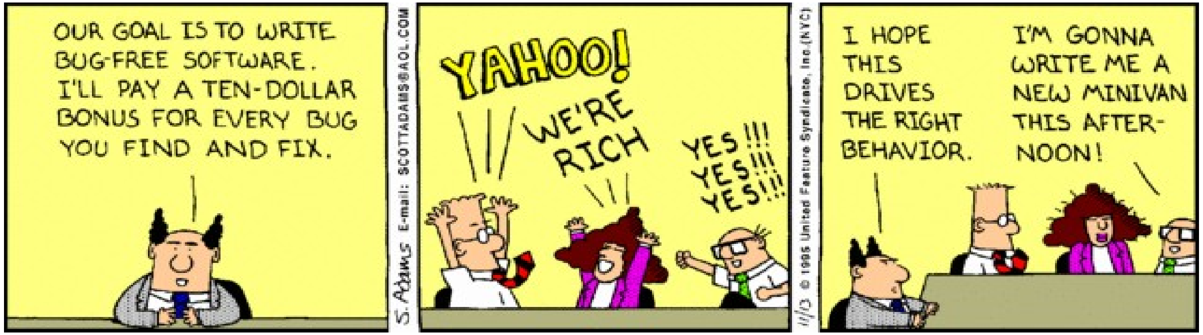
Source (FU): http://blog.codinghorror.com/a-visit-from-the-metrics-maid/

- "You can't manage it if you can't measure it."

  → overuse of metrics

  → metrics-based incentives

- May lead to results which fit the metrics but are not actually better


OUR GOAL IS TO WRITE BUG-FREE SOFTWARE. I'LL PAY A TEN-DOLLAR BONUS FOR EVERY BUG YOU FIND AND FIX.
S. Adams E-mail: SCOTTADAMS@AOL.COM
YAHOO!
WE'RE RICH
YES!!! YES!!! YES!!!
© 1995 United Feature Syndicate, Inc.(NYC)
I HOPE THIS DRIVES THE RIGHT BEHAVIOR.
I'M GONNA WRITE ME A NEW MINIVAN THIS AFTER-NOON!

Metrics: basics

- Simple metric: source lines of code (SLOC)
- "Measuring software productivity by lines of code is like measuring progress on an airplane by how much it weighs." - Bill Gates

- Useful as base for other metrics
- “Raw” SLOC include whitespace, comments, …
- Often replaced by logical lines of code
  (LLOC = 1 statement per line)

Useful metrics

- Bugs per 1000 LOC (= kLOC)
- Fan-in/fan-out
- Code (test) coverage
- Cyclomatic complexity

Bugs per kLOC

Source (FU): Coverity, Open Source Integrity Report 2012

- NASA Software Assurance Technology Center: 0.1 bugs per kLOC (not applicable for everyday use)
- Open-source software (45 projects, 37 million lines of code): 0.45 bugs per kLOC
- Commercial software (41 projects, 300 million lines of code): 0.64 bugs per kLOC
- *Note:* code size of commercial projects larger by factor ~ 10

Fan-In/Fan-Out

- Fan-In for method X: number of functions/ methods that call X
- Fan-Out for method X: number of functions/ methods called by X
- High fan-in → changes to X may cause extensive secondary changes
- High fan-out → X may be overly complex

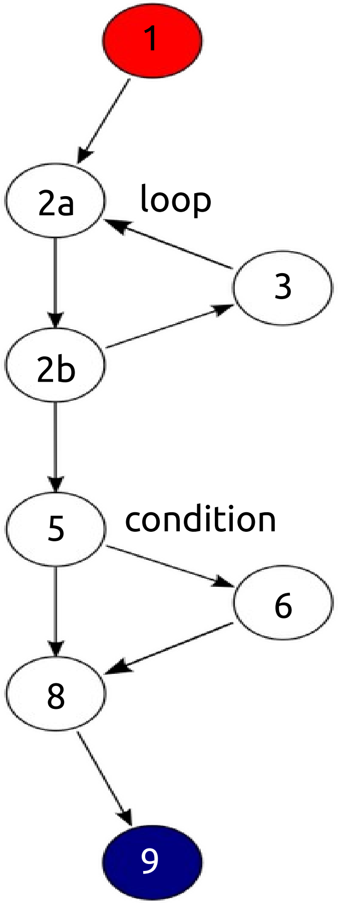
Code coverage

- Hybrid of static and dynamic metrics
- Relates to unit/component/system tests
- Different variants (in increasing order of complexity): tests cover percentage of …
  - Functions (called at least once)
  - Statements (executed at least once)
  - Branches (executed at least once)
  - Conditions (evaluated as true and false)
  - Execution paths (executed at least once)

Cyclomatic complexity

Image source (PD): https://en.wikipedia.org/wiki/Cyclomatic_complexity

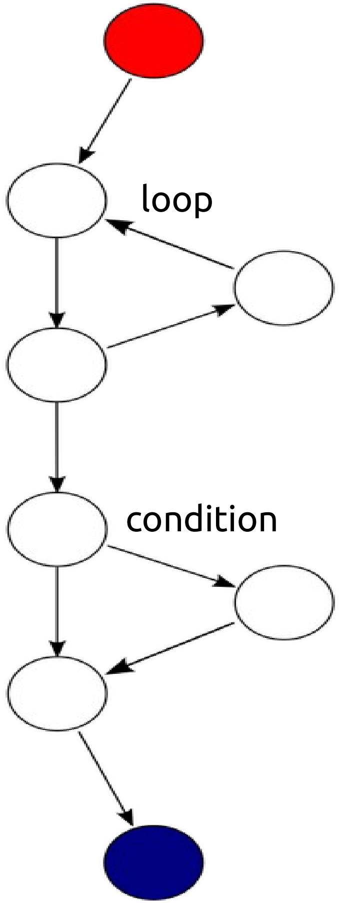
```
1: void func(int a) {
2:   for (int i = 0; i < a; i++) {
3:     process(i);
4:   }
5:   if (a == 42) {
6:     answer();
7:   }
8:   cleanup();
9: }
```



Cyclomatic complexity

Image source (PD): https://en.wikipedia.org/wiki/Cyclomatic_complexity

- Number of linearly independent paths through code
- Can be calculated from control flow graph
- Complexity M = E – N + 2P (edges E, nodes N, graph components P)
- Example: M = 9 – 8 + 2*1 = 3
- Rule-of-thumb: split a module if M > ~ 10



“Code smells”

- Syntactically and functionally correct code
- However: “smells” indicate structural problems
- May lead to future bugs/maintenance issues
- Also known as “lint”, “fuzz”
- (Partial) purpose of compiler warnings, e.g. “-Wall” switch in gcc (= enable all warnings)

“Code smells” - Examples

- General impact on readability/ understandability:
  - Unused variables/code (also increase binary size)
  - Long method (longer than ~ 50 SLOC/1 screen)
  - Excessive use of literals instead of named constants (so-called “magic values”)
  - Depth of conditional nesting

“Code smells” - Examples (2)

- General impact on readability/ understandability:
  - Excessively short identifiers:
    ```
    int a,b,c,e,x;
    ```
  - Excessively long identifiers:
    ```
    for (int loopVariable = 0; loopVariable <
     loopMaximum; loopVariable++) { … }
    ```
  - Lack/overuse of comments:
    ```
    int x = 0; // set integer variable x to zero
    ```

“Code smells” - Examples (3)

- Impact on code maintenance:
  - Duplicated code:
    `a; b; c; … a; b; c;`
  - Redundant code:
    `if (x) { … if (x) { … } … }`
  - Empty statements:
    `if (…) { }`
  - Side effects in conditions:
    `if (a = b) { … }` instead of `if (a == b) { … }`

“Code smells” - Examples (4)

- Too many parameters (more than ~5)
  - related to maximum number of items in human short-term memory = 7 ± 2
- Too many local variables in method
- Too many member variables in class
- Overly large class (also known as *God Object*)

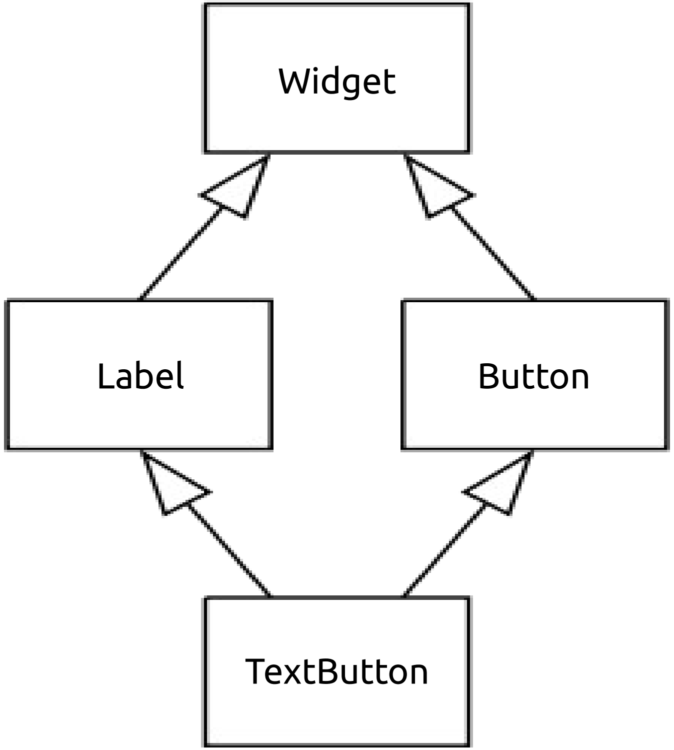
OOP “Code smells”

- Excessively deep inheritance structure
- “Feature envy”: excessive use of another class
  - May happen intentionally in some patterns (which?)
- Violation of substitution principle by method overriding: subclass can no longer replace BC
- Contrived complexity: overuse of patterns/ templates etc.

Diamond Problem

Source (PD): https://en.wikipedia.org/wiki/Multiple_inheritance

- Multiple inheritance with shared base class (C++ only)
- Assume overridden method `draw` in Label and Button
- If `draw` is called in TextButton: is it `Label::draw` or `Button::draw`?
- Solvable, but may point to overly complex design



Law of Demeter

Source (CC): https://en.wikipedia.org/wiki/Law_of_Demeter

- Goal: decrease coupling between components
- “Only talk to your direct friends.” → call only …
  - Methods of class itself
  - … of parameter objects
  - … of objects in instance variables
  - … of objects created by class
- When disregarded: requires knowledge about internals of other classes

Law of Demeter (2)

Source (CC): https://en.wikipedia.org/wiki/Law_of_Demeter

```
class Motor {
  public void start() { … }
}

class Car {
  public Motor motor;
  public Car() {
    motor = new Motor();
  }
}

class Driver {
  public void drive() {
    Car carToDrive = new Car();
    carToDrive.motor.start();
  }
}
```

```
class Motor {
  public void start() { … }
}

class Car {
  private Motor motor;
  public Car() {
    motor = new Motor();
  }
  public void getReady() {
    motor.start();
  }
}

class Driver {
  public void drive() {
    Car carToDrive = new Car();
    carToDrive.getReady();
  }
}
```

In Java → “Use only one dot.”

AntiPatterns

- Similar to design patterns: simple and widely used solutions to common problems …
- … which cause other issues down the road.
- http://c2.com/cgi/wiki?AntiPatternsCatalog
  - StringWithoutLength
  - ParsingHTMLWithRegex
  - ZeroMeansNull
  - FloatingPointCurrency
  - ExceptionFunnel

StringWithoutLength

Source (FU): http://c2.com/cgi/wiki?StringWithoutLength

- Store a string without explicit length
- Use “marker” (NULL byte) instead
- Unfortunately embedded in C Standard Library
- Often also used for other types of arrays
- Requires constant recalculation of length (e.g. for copy, concatenation, …)
- “Proper” solution: store length as separate int (e.g. in std::string)

ParsingHTMLWithRegex

Source (FU): http://c2.com/cgi/wiki?ParsingHtmlWithRegex

- Goal: extract information from web page
- Use regular expressions to extract data from HTML
- Will usually break if page is changed at all
- Parsed result may still contain HTML tags
- “Proper” solution:
  - Use a dedicated data source
  - If not possible: use an XML parser

ZeroMeansNull

Source (FU): http://c2.com/cgi/wiki?ZeroMeansNull

- Goal: implement an optional field
- Use 0 (*zero*) to represent NULL (*empty*)

  → Field can never be actually set to zero
- Variants: use string “NULL” (there are people who have that name), use value -1 (may lead to overflow errors), …
- “Proper” solution:
  - Use additional boolean flag
  - Use pointer to data object

FloatingPointCurrency

Source (FU): http://wiki.c2.com/?FloatingPointCurrency

- Goal: store an amount of money
- Use a float (or double), e.g. 1.23f = 1 € 23 cents
- Problem: decimal fractions can **not** be 100% accurately represented in a float/double

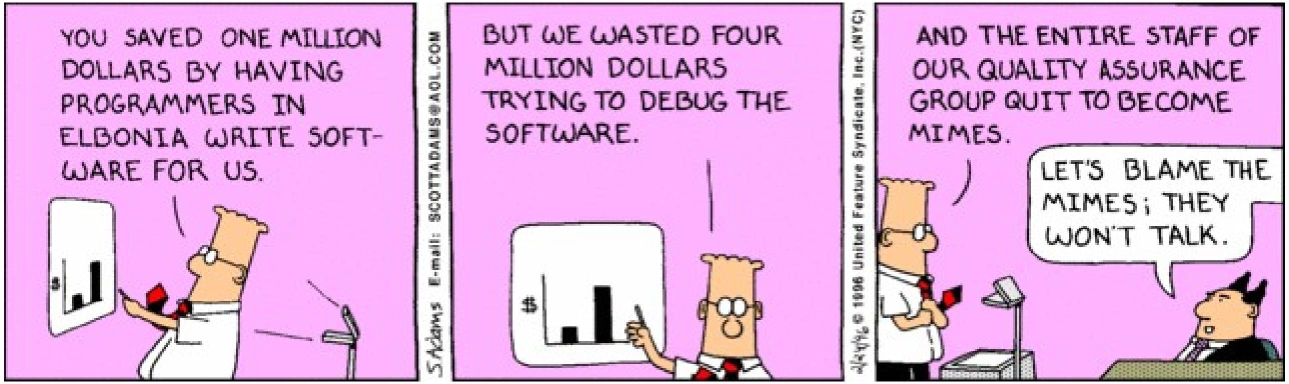
  → Rounding errors can accumulate over time
- “Proper” solution:
  - Use fixed-point math
  - Use separate integers

ExceptionFunnel

Source (FU): http://wiki.c2.com/?ExceptionFunnel

- Goal: handle errors, but don’t confuse users
- Few catch-all blocks that may even throw exceptions away (“`catch(Exception e){}`”)
- Problems:
  - No useful debug output at all
  - Errors may go unhandled, cause issues later
- “Proper” solution:
  - Use descriptive exceptions
  - Catch and handle them separately

Questions/Comments?


YOU SAVED ONE MILLION DOLLARS BY HAVING PROGRAMMERS IN ELBONIA WRITE SOFTWARE FOR US.
$
S.Adams E-mail: SCOTTADAMS@AOL.COM
BUT WE WASTED FOUR MILLION DOLLARS TRYING TO DEBUG THE SOFTWARE.
$
© 1996 United Feature Syndicate, Inc.(NYC)
AND THE ENTIRE STAFF OF OUR QUALITY ASSURANCE GROUP QUIT TO BECOME MIMES.
LET'S BLAME THE MIMES; THEY WON'T TALK.