CISC 327 - Software Quality Assurance

Lecture 22
Code Inspections
Code Inspections

• Outline
  – Today we look at the actual practice of code inspections
    • Checklist-directed code reviews
    • Code paraphrasing
    • Structured code walkthroughs
    • Lightweight code review practices
    • Heavyweight code review practices
Code Inspection Practices

• Even if we have a highly formalized inspection process such as the generic one we looked at last time, there is still a range of actual practices that can be applied to implement the actual review of the artifact

• When the artifact is the actual code, we can use:
  – Checklists
  – Paraphrasing
  – Structured walkthroughs
  – Axiomatic formal proofs
Code Checklists

• Code checklists give a concrete list of properties of the code to check for
• Checklists may be general properties for any program, or specific properties for the specific program or kind of program
• Both desired properties (ones we want the code to have) and undesired properties (ones the code should not have) may appear in the list
Code Checklists

• Checklist items can range from simple surface properties such as code format to deep semantic properties such as termination

• The idea is that the inspector should look through the code to check for the presence or absence of each individual property, and check it off the list

• Checklists are only part of the inspection - the correctness of the code must also be checked
An Example: Generic Java Code Inspection Checklist

– 1. Variable and Constant Declaration Defects
   • 1.1 Are descriptive variable and constant names used in accord with naming conventions?
   • 1.2 Are there variables with confusingly similar names?
   • 1.3 Is every variable properly initialized?
   • 1.4 Can any non-local variables be made local?
   • 1.5 Are there literal constants that should be named constants?
   • 1.6 Are there variables that should be constants?

– 2. Method Definition Defects
   • 2.1 Are descriptive method names used in accord with naming conventions?
   • 2.2 Is every parameter value checked before being used?
   • 2.3 Does every method return a correct value at every return point?
An Example: Generic Java Code Inspection Checklist

...

– 4. Computation Defects
  • 4.1 Is underflow or overflow possible in any computation?
  • 4.2 Does any expression depend on order of evaluation of operators? Are parentheses used to avoid ambiguity?

...

– 6. Control Flow Defects
  • 6.1 Will all loops terminate in all cases?

...
Code Checklists

• Google Python Style Rules (partial)
  – **Semicolons**: Do not terminate your lines with semicolons and do not use semicolons to put two commands on the same line.
  – **Line length**: Maximum line length is 80 characters.
  – **Parentheses**: Use parentheses sparingly.
  – **Indentation**: Indent your code blocks with 4 spaces.
  – **Blank Lines**: Two blank lines between top-level definitions, one blank line between method definitions.
  – **Strings**: Use the % operator for formatting strings, even when the parameters are all strings. Use your best judgement to decide between + and % though.
Code Paraphrasing

• **Reading the Code in English**
  – Code *paraphrasing* is the original method of review described by Fagan for use in code inspections
  – Consists of reading the lines of code for their meaning in the *problem domain*, not in the programming language
  – The object is to ensure that the code really does implement what we want to have done
Code Paraphrasing

• **Reading the Code in English**
  – Paraphrasing should avoid mentioning variables or control flow, rather it should be phrased in terms of the abstract meaning of the *concepts* and *processes* being implemented by them
  – Discussion is seeded by *scenarios*, potential situations that may have to be handled
  – Paraphrasing is often coupled with *checklists* - the checklist addresses low level properties of the code itself, while the paraphrasing addresses its high-level meaning
def binarySearch(list_obj, key, start, end):
    middle = (start + end) // 2

    if list_obj[middle] < key:
        return binarySearch(list_obj, key, middle, end)
    elif list_obj[middle] > key:
        return binarySearch(list_obj, key, start, middle)
    else:
        return middle

def find(list_obj, key):
    return binarySearch(list_obj, key, 0, len(list_obj))

Path testing found no problems in this code
# Code Paraphrasing Example

# binarySearch(list_obj, key, start, end)
#
# Search for key within range list_obj[start:end].
# Return index of key.
def binarySearch(list_obj, key, start, end):
    middle = (start + end) // 2

    if list_obj[middle] < key:
        return binarySearch(list_obj, key, middle, end)
    elif list_obj[middle] > key:
        return binarySearch(list_obj, key, start, middle)
    else:
        return middle

def find(list_obj, key):
    return binarySearch(list_obj, key, 0, len(list_obj))
Structured Code Walkthroughs

• A Guided Tour
  – Code walkthroughs are a sort of guided tour of the program, led by a "tour guide" who describes the points of interest, signposts, family stories, anecdotes, etc. as we "walk" through the paths of the code
  – When used in code inspections, walkthroughs are conducted by the code reviewers in the meeting as a sort of exploration of the code's workings - this is a dynamic walkthrough (like we just did)
  – Code walkthroughs can also be static - documented as an annotated version of the source code with the tour guide's comments shown alongside the code itself
Structured Code Walkthroughs

• A Guided Tour
  – Both kinds of walkthroughs have been found to be very effective in training, to introduce new programmers to the code
  – Walkthroughs are less effective as an inspection method (although they are still much better than testing)
A Famous Static Walkthrough

• The Lions Commentary
  – Perhaps the most famous code walkthrough in history is the 1977 "Source Code and Commentary on UNIX Level 6" by John Lions of U. New South Wales
  – The walkthrough is organized as two parts: a line-numbered commented listing of the Unix kernel source code, and a detailed commentary organized in parallel with the source code structure, indexed by line number in the source code listing
A COMMENTARY ON THE UNIX OPERATING SYSTEM

JOHN LIONS
Department of Computer Science
The University of New South Wales
June, 1979

This document contains proprietary information of the Bell System. Use is restricted to authorized employees of the Bell System who have a job-related need-to-know. Distribution or disclosure to unauthorized persons is prohibited. Outside the Bell System, distribution of this document has been limited to licensed vendors of the UNIX® Operating System. Bell Edition from the Western Electric Company. This document is subject to the restrictions of the Bell System and Western Electric Company.

UNIX is a trademark of Bell Laboratories.
A Famous Static Walkthrough

• The Lions Commentary
  – The commentary pointed out the meaning and effect of each section of code on the whole system, reminded the reader of what each mentioned variable represents and where it is defined in the ten thousand lines of code, and so on


```c
2650 #
2651 #include "../param.h"
2652 #include "../sysm.h"
2653 #include "../user.h"
2654 #include "../proc.h"
2655 #include "../red.h"
2656 #include "../ses.h"
2657
define EBIT 1 /* user error bit in PS1 C-bit */
2658 define UMODE 0170000 /* user-mode bits in PS word */
2659 define SETD 0170011 /* SETD instruction */
2660 define SYS 0104000 /* sys (trap) instruction */
2661 define USER 020 /* user-mode flag added to dev */
2662
define TBIT 020 /* PS trace bit */
2663
2664 /*
2665 * structure of the system entry table (sysent.c)
2666 */
2667
2668 struct sysent {     /* argument count */
2669 int count;
2670 (call)(); /* name of handler */
2671#--------------------------#
2672#
2673#
2674 * Offsets of the user’s registers relative to R0. See sys.h
2675 * the saved R0. See red.h
2676 */
2677 char realoc[64]
2678
t R0, R1, R2, R3, R4, R5, R6, R7, RPS
2679#--------------------------#
2680#
2681#
2682#
2683#
2684 * Called from 140.s or 145.s when a processor trap occurs.
2685 * The arguments are the words saved on the system stack
2686 * by the hardware and software during the trap processing.
2687 * Their order is dictated by the hardware and the details
2688 * of C’s calling sequence. They are peculiar in that
2689 * this call is not ‘by value’ and changes user registers
2690 * set copied back on return.
2691 * dev is the kind of trap that occurred.
2692 */
2693 trap(dev, sp, r1, rps, ro, pc, ps)
2694 {
2695     register i, a;
2696     register struct sysent *call;  
2697     savfr();
2698     if ((ps=UMODE) == UMODE)
```

Sheet 26
case 64USER: /* sys call */
    u.u_error = 0;
    ps = &"EBIT64";
    callp = &sysent[fulword(ps-2)];
    if (callp == sysent) { /* indirect */
        a = fulword(pc);
        pc += 2;
        i = fulword(a);
        if (i == 0) !SIGSYS;
        callp = &sysent[fulword(pc) + 1];
        for(i=0; i<callp->count; i++)
            u.u_arg[1] = fulword(a + i);
    } else {
        for(i=0; i<callp->count; i++)
            u.u_arg[1] = fulword(pc + i);
    }
    u.u_dirp = u.u_arg(0);  
    trap1(callp->call);
    if(u.u_init1) 
        u.u_error = EINTR;
    if(u.u_error < 100) 
        if(u.u_error) 
            ps = &"EBIT64";
    r0 = u.u_error;
    sotoup;
    i = SIGSYS;
    break;

/ * Since the floating exception is an
  imprecise trap, a user generated
  trap may actually cause from kernel
  mode. In this case, a signal is sent
  to the current process to be picked
  up later. */

case 0: /* floating exception */
    signal(u.u_proc, SIGFPE);
    return;

case 80: /* SIGFPE */
    psignal1(u.u_proc, SIGFPE);
    return;

case 4USER: /* int */
    i = SIGTERM;
    break;

case 5USER: /* int */
    i = SIGINT;
    break;

case 6USER: /* int */
    i = SIGALRM;
    break;

# Make sure our device is 64 bits.

dev = USER;

u.u_ar0 = 0;
switch(dev) {

/* Trap not expected. */

# Usually a kernel mode bus error.

# The numbers printed are used to

# find the hardware PS/PC as follows,

# (all numbers in octal 10 bits)

# address of saved ms =

# (k4) & 01000)

# address of saved ms =

# address of saved ms - 2

/*

print("k4 = \%x\n", k4);
print("n1 = \%x\n", n1);
print("n2 = \%x\n", n2);
panic("trap");

*/

case 0USER: /* bus error */
    i = SIGBUS;
    break;

/*

If illegal instructions are not

being caught and the offending instruction

is a SETD, the trap is ignored.

This is because C produces a SETD at

the beginnings of every program which

will trap on CPUS without 11/45 FU.

*/

case 1USER: /* illegal instruction */
    if(fulword(pc-2) = SETD && u.u_signal(SIGINS) == 0)
        sotoup;

i = SIGINS;
break;

/*

Since the floating exception is an

imprecise trap, a user generated

trap may actually cause from kernel

mode. In this case, a signal is sent

... continued... */

/ *

return;

*/
assembler "trap" routine carries out certain fundamental housekeeping tasks to set up the kernel stack, so that when this procedure is called, everything appears to be kosher.

The "trap" procedure can operate as though it had been called by another C procedure in the normal way with seven parameters:

\[ \text{dev, sp, rl, npa, r0, pc, ps.} \]

(There is a special consideration which should be mentioned here in passing. Normally all parameters passed to C procedures are passed by value. If the procedure subsequently changes the values of the parameters, this will not affect the calling procedure directly.

However if "trap" or the interrupt handlers change the values of their parameters, the new values will be picked up and reflected back when the previous mode registers are restored.)

The value of "dev" was obtained by capturing the value of the processor status word immediately after the trap and masking out all but the lower five bits. Immediately before this, the processor status word had been set using the prototype contained in the appropriate vector location.

Thus if the second word of the vector location was "br7*n:" (e.g. line 018) then the value of "dev" will be n.

2698: "savfps" saves the floating point registers (for the PDP11/40, this is a no-op).

2788: If the previous mode is "user mode", the value of "dev" is modified by the addition of the octal value 028 (2662);

2789: The stack address where r0 is stored is noted in "u.u_ar0" for future reference. (Subsequently the various register values can be referenced as "u.u_ar0[Rn].")

2782: There is now a multi-way "switch" depending on the value of "dev".

At this point we can observe that UNIX divides traps into three classes, depending on the prior processor mode and the source of the trap:

(A) kernel mode;
(B) user mode, not due to a "trap" instruction;
(C) user mode, due to a "trap" instruction.

Kernel Mode Traps

The trap is unexpected and with one exception, the reaction is to "panic". The code executed is the "default" of the "switch" statement:

2716: Print:

the current value of the seventh kernel segment address register (i.e. the address of the current per process data area);
the address of "ps" (which is in the kernel stack); and
the trap type number;

2719: "panic", with no return.

Floating point operations are only used by programs, and not by the operating system. Since such operations on the PDP11/45 and 11/70 are handled asynchronously, it is possible that when a floating point exception occurs, the processor may have already switched to kernel mode to handle an interrupt.

Thus a kernel mode floating point exception trap can be expected occasionally and is the concern of the current user program.
A Famous Static Walkthrough

• The Lions Commentary
  – This book was quashed and never published because Western Electric (now AT&T) considered Unix a trade secret
  – Underground photocopies made it out to researchers all over the world, and it was used as training material for students who later made FreeBSD, Linux, and all other Unix derivatives
Lightweight Code Inspection Practices

• **Learning from Success**
  – Formal inspections are very successful at finding defects, but many find the formal inspection process too cumbersome
  – As a result, many practices have been developed that can gain some of the advantages of formal inspection without the overhead
Lightweight Code Inspection Practices

- **Four Eyes Principle**
  - One such practice is the *four eyes principle*, in which programmers work in loose pairs, where each module or class authored by one programmer is informally inspected by the other.
  - Both programmers are *equally responsible* for the quality and correctness of the code, and both must “sign off” on it before it is released for testing and integration.
  - In small teams, this method has been shown to produce code with very few defects.
Lightweight Code Inspection Practices

• Chief Programmer Teams
  – The four eyes principle is part of the chief programmer team software development method, an early predecessor of XP
  – CPT development involves weekly meetings in which the whole team informally reviews the system design, architecture, interfaces and schedule once each week, continuously updating each of these as the project proceeds
Lightweight Code Inspection Practices

• Extreme Programming – Continuous Inspection
  – The four eyes principle is the predecessor of XP’s pair programming, another lightweight inspection method
  – XP’s code refactoring is a continuous inspection method descended from CPT’s weekly design reviews
Lightweight Code Inspection Practices

• **White Box Testing (By Hand)**
  – When applied manually, most white box testing methods force the test author to examine the code in detail to create the tests - in practice this is how most defects are found
  – This too is essentially a lightweight inspection process
Heavyweight Inspection Practices

• Doing It Right in the First Place
  – If you carry formal inspection to the opposite extreme from lightweight informal methods, you wind up at **formal verification**
  – This extreme is best represented by the **Cleanroom** software development method
Cleanroom Software Development

• Cleanroom (Mills 1987)
  – Essentially the "ideal" inspection process
  – Based on static verification to ensure error-free development
    • defects avoided rather than detected and corrected
    • defects avoided by developing in an "ultra-clean" environment (analogous to semiconductor fabrication)
    • formal inspections augmented with formal correctness proofs as code review
  – Software components are formally specified and verified instead of usual development and unit/module testing
  – Predecessor of modern Model-Driven Engineering, where code is derived from formal mathematical models of desired behaviour
Cleanroom Software Development

• **Cleanroom Method**
  – *Formal specification*: software to be developed is formally specified
  – *Incremental development*: software is partitioned into increments which are developed separately using the Cleanroom approach
  – *Structured programming*: only a limited number of control and data constructs are allowed to be used in the code
  – *Stepwise refinement*: code is developed directly as a refinement of the formal specification
  – *Static verification*: developed software components are not tested but statically verified using mathematically based correctness arguments
  – Correctness proof used as code review in inspection process
  – *Statistical testing*: integrated software is tested statistically to determine its reliability
Summary

• **Code Inspections**
  – When the inspection process is applied to code, there are a range of techniques we can use to inspect the code, including checklists, paraphrasing, walkthroughs
  – **Lightweight** inspection practices attempt to gain the advantages of code inspections without the formal process, as in XP
  – **Heavyweight** inspection practices take the other extreme, formalizing code inspection all the way to formal verification

• **Reminders**
  – Assign #3 due **Monday, Nov. 6th**
  – Mini-Exam #3; date to be determined (mutation testing, regression testing, inspections)