CISC327 - 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

• A Range of Alternatives
  – 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

• 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

• 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
Code Checklists

• 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?
Code Checklists

• 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 semi-colons and do not use semi-colons 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
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
  – 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, 1977

John Lions
The University of New South Wales

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. Duplication or disclosure to unauthorized persons is prohibited. Outside the Bell System, duplication of this document has been licensed to the authors of A Commentary on the UNIX Operating System. Sixth Edition from the McGraw Hill Book Company. Permission to use a portion of this document is subject to the restrictions in the original document issued by 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
May 24 12:35 1976 unix/trap.c Page 1

2650 
2651 #include "../param.h"
2652 #include "../sysm.h"
2653 #include "../user.h"
2654 #include "../proc.h"
2655 #include "../reg.h"
2656 #include "../sys.h"

2657 define EBIT 1  /* user error bit in PS: 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 USER 020  /* user-mode flag added to dev */
2663 define USER 020  /* user-mode flag added to dev */
2664 */
2665 */
2666 */
2667 struct sysent  
2668 int count;  /* argument count */
2669 int (*call)();  /* name of handler */
2670 char sysent[64];
2671 */
2672 */
2673 */
2674 */
2675 */
2676 */
2677 */
2678 */
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 * The order of the words is dictated by the hardware and the details
2687 * of C's calling sequence. They are peculiar in that
2688 * this call is not 'by value' and changes user registers
2689 * set copied back on return.
2690 * dev is the kind of trap that occurred.
2691 */
2692 */
2693 /*
2694 */
2695 */
2696 */
2697 */
2698 */
2699 if ((ps=UMODE) == UMODE)
2700  dev = 1 USER);
2701  u.u_ar0 = &r0;
2702  switch(dev) {
2703  
2704  /*
2705  * Trap not expected.
2706  * Usually a kernel mode bus error.
2707  * The numbers printed are used to
2708  * find the hardware PS/PC as follows,
2709  * (all numbers in octal 10 bits)
2710  * address_of_saved_ps =
2711  * (ka@0100) + ars - 0140000
2712  * address_of_saved_pc =
2713  * address_of_saved_ps - 2
2714  */
2715  default:
2716  printf("ka = %04n", ka);
2717  printf("ars = %04n", ars);
2718  printf("trap type %04n", dev);
2719  panic("trap");
2720  case 0USER: /* bus error */
2721  i = SIGBUS;
2722  break;
2723  
2724  /#
2725  * If illegal instructions are not
2726  * being caught and the offending instruction
2727  * is a SETD, the trap is ignored.
2728  * This is because C produces a SETD at
2729  * the beginnings of every program which
2730  * will trap on CPUs without ii/AS FPU.
2731  */
2732  case 1USER: /* illegal instruction */
2733  if(u.u_word(0) == SETD && u.u.signal(SIGINS) == 0)
2734  return;
2735  /*
2736  Since the floating exception is an
2737  imprecise trap, a user generated
2738  trap may actually come from kernel
2739  mode. In this case, a signal is sent
2740  to the current process to be picked
2741  up later.
2742 */
2743  case 2USER: /* bpt or trace */
2744  i = SIGTRC;
2745  break;
2746  case 3USER: /* int */
2747  i = SIGINT;
2748  break;
2749  case 4USER: /* ext */
2750  i = SIGQUIT;
2751  break;
2752  case 64USER: /* sys call */
2753  u.u_error = 0;
2754  ps = &"EBIT"
2755  callp = &usent[fullword(0xc-2)0777];
2756  if(callp == usent) /* indirect */
2757  a = fullword(pc);
2758  pc += 2;
2759  i = fullword(a);
2760  if(i & "0777"); /* illegal */
2761  i = 0771;
2762  callp = &usent[fullword(0xc-2)0777];
2763  for(i=0; i<callp->count;i++)
2764  if(u.u_arg[i] = fullword(ps))
2765  pc += 2;
2766  case 6USER: /* dir */
2767  u.u_dir[p] = u.u_arg[0];
2768  if(u.u_initfl)
2769  u.u_error = EINTR;
2770  if(u.u_error < 100) (;
2771  if(u.u_error) (;
2772  i = SIGSYS;
2773  break;
2774  return;
2775  else (;
2776  for(i=0; i<callp->count;i++)
2777  if(u.u_arg[i] = fullword(ps))
2778  pc += 2;
2779  case 0USER: /* bpt */
2780  if(u.u_procp == SIGFPE)
2781  return;
2782  i = SIGFPE;
2783  break;
2784  /*
2785  Since the floating exception is an
2786  imprecise trap, a user generated
2787  trap may actually come from kernel
2788  mode. In this case, a signal is sent
2789  to the current process to be picked
2790  up later.
2791 */
2792  case 0: /* floating exception */
2793  if(u.u_procp == SIGFPE)
2794  return;
2795  i = SIGFPE;
2796  break;
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
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 "br?n;" (e.g. line 2616) then the value of "dev" will be n.

2698: "savfps" saves the floating point registers (for the PDP11/40, this is a no-op); 2700: If the previous mode is "user mode", the value of "dev" is modified by the addition of the octal value 028 (2662);
2701: 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[2n-1].")

2702: 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 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.

Traps and System Calls
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