Chapter 11: A design example

- 75 pages of source code and discussion
- Simulates tank monitoring system
- Actually works: runs under DOS and µC/OS
  - Hardware-independent part is reasonably realistic
  - Scaffold code replaces hardware-dependent code
  - Simple user interface
- Invitation: look through it to get better feel for RTOS application code
  - Ask yourself: would I have organized it in the same way?

Author's afterword

We never got a chance to design the tank monitoring system discussed at length in this book. It was brought to us at Probitas, the consulting firm where I work, already specified, designed, coded, and (supposedly) tested. The client brought it to us for some minor hardware and software enhancements.

We made the hardware enhancements, fixing a few miscellaneous problems along the way, without too much difficulty. Then we delved into the software. It was written with a polling loop and some interrupt routines; it did not use an RTOS. To get any kind of response, the software that calculated the levels in the tanks periodically saved its intermediate results and returned to the polling loop to check if the user had pressed any buttons. The software was written in interpreted BASIC. It was spaghetti.

I leave it to your imagination to visualize the difficulties that we encountered trying to add features to this software without breaking it and without spoiling its response.

This was a number of years ago now, and it would stretch the truth to say that I wrote this book in reaction to what I saw in that system. It gives me great satisfaction, however, to hope that this book will prevent at least a few similar horrors in the future.

David E. Simon

Supplemental material

Software development issues

- We've seen and discussed some challenges of developing reliable code:
  - Can't get error free code by testing.
  - Better to start with careful design: 425 labs hopefully illustrate this.
  - Much harder for complex systems, with changing requirements.
- Let's look at some real-world issues pertaining to software development.

issue #1

Software productivity

- Customers expect software to be delivered in increasingly shorter cycles.
  - The longest most are willing to wait: ~12 months
- How much software can a typical IT shop create in 12 months?
  - Clearly scales with number of programmers, right?
- Not quite so simple:
  - Study of 281 SW development projects completed in 2000-2001
  - Only 10% of projects built more than 75,000 SLOC in one year
    - SLOC ≡ Source Lines of Code
  - Typical team size producing < 75,000 SLOC: 5-10 people
  - Typical team size producing > 75,000 SLOC: 20-100 people

Source: www.qsm.com

Software productivity

- Study: consider engineer-months per SLOC
  - Significantly more effort required to produce large software fast
  - For projects that produced over 75,000 SLOC in 12 months:
    - Effort 1.17 to 4.19x above average of all projects
- Conclusions:
  - If you want it fast, you'll pay much more.
  - Team of 3-10 people can produce ≤ 75,000 SLOC in a year.
    - High end more feasible if software has relatively low level of complexity.
    - Shooting for high end likely to result in delays, reduced reliability.
  - Practical upper limit: 180,000 SLOC per year with 70-100 people.
    - Cost: 2x to 4x more than same work done by fewer people over more time.
  - Software is complex, and managing development is tricky.

So what is productivity per person per day?

Software productivity

WALLY, HAVE YOU MADE ANY PROGRESS CODING YOUR MODULE?
PROGRESS IS DIFFICULT TO MEASURE IN THE SOFTWARE REALM.
YOU COULD MEASURE THE LINES OF CODE I PRODUCE, BUT THAT WOULD REWARD INEFFICIENCY.
THE ART OF THIS JOB IS BINDING THE RARE MOMENTS OF INSPIRATION TO KNOWLEDGE AND MACHINES.
IN FACT, JUST A MINUTE AGO I COULD FEEL THE INSPIRATION WELLING UP INSIDE ME.
BUT THEN YOU INTERRUPTED ME WITH YOUR NAIVE QUESTION AND THE MOMENT WAS LOST.
MAYBE YOU SHOULD GO BACK TO YOUR OFFICE AND REFLECT ON THE DAMAGE YOU'VE DONE HERE TODAY.
THERE GOES THE ONE PERSON WHO HAS LESS OF A REAL JOB THAN I DO.
DilbertCartoonist@gmail.com
© 2010 Scott Adams, Inc./Dist. by UFS, Inc.
www.dilbert.com

issue #2

Two heads are better than one: Pair programming

- One approach for generating better software
- Characteristics:
  - Two programmers work side-by-side at one computer, continuously collaborating on design, coding, and testing.
  - "Drivers" take turns; observers actively and continuously review
    - Strategic thinking: Where will this approach lead? Is there a better way?
  - Team is "like a coherent, intelligent organism working with one mind, responsible for every aspect of this artifact."
  - Participants equal: Not "a problem in <u>your</u> code" – it's <u>our</u> code.

Source: "All I really need to know about programming I learned in kindergarten", Williams and Kessler, CACM, May 2000

Pair programming

- Benefits claimed by proponents:
  - Helps keep both coders on task: neither feels they can slack off.
  - Continual exchange of ideas makes programmers better.
  - Pair can solve problems together that they can't solve alone.
  - Observer often spots defects; less animosity than formal code review.
  - Programmers in shared space often overhear something that matters.
    - "Programmers need contact with other programmers."
  - Productivity and enjoyment both increase.
- Challenges:
  - Getting everyone to buy in: programmers are used to working alone.
  - Fine balance between too much and too little ego.
    - Healthy disagreement and debate is best.

Keeping your ego out of the way

On a particularly bad programming day, an individual egolessly laughed because his reviewer found 17 bugs in 13 statements. After fixing those defects, however, the code performed flawlessly during testing and in production. How different this outcome might have been had the programmer been too proud to accept the input of others or had viewed this input as an indication of his inadequacies. Having another [person] review design and coding continuously and objectively is an extremely beneficial aspect of pair programming. "The human eye has an almost infinite capacity for not seeing what it does not want to see… Programmers, if left to their own devices, will ignore the most glaring errors in their output – errors that anyone else can see in an instant."

Williams and Kessler

Pair programming

- Recommendations:
  - Workspace layout critical: "slide the keyboard, don't move the chairs"
  - Take a break periodically
    - Pair programming is intense and mentally exhausting
    - Go check email, make phone calls
  - It is acceptable to work alone 10% - 50% of time
    - Experimental prototyping, thinking through hard problems
  - Avoid
    - competition
    - blaming individuals for errors

Pair programming: results

- In 1999 study, 96% of programmers said they enjoyed their jobs more when pair programming.
- Data suggests that two together are more than twice as fast
  - Also, pair suggests > 2x possible solutions than two independent individuals
- One careful experiment with 15 expert programmers:
  - Assigned challenging problem for 45 minutes
  - 5 worked individually, 10 in pairs; conditions + materials were the same
  - All teams outperformed individuals, enjoyed it more, and had higher confidence in their solution
  - Outcome surprised managers and even participants

Pair programming anecdotes

Having adopted this approach, they were delivering finished and tested code faster than ever … The code that came out the back of the two-programmer terminals was nearly 100% bug free… It was better code, tighter and more efficient, having benefited from the thinking of two bright minds and the steady dialogue between two trusted terminal mates… Two programmers in tandem is not redundancy; it's a direct route to greater efficiency and better quality.

Larry Constantine
Describing visit to Whitesmiths, Ltd., a software company

I strongly feel pair programming is the primary reason our team has been successful. It has given us a very high level of code quality (almost to the point of zero defects). The only code we have ever had errors in was code that wasn't pair programmed.

(Anonymous respondent to survey)

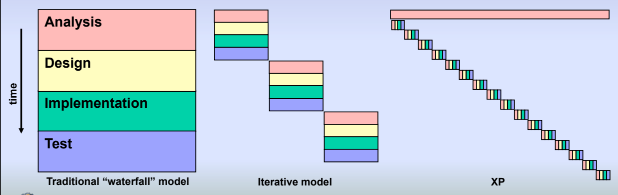
425: pair programming?

- How many of you worked on the labs in this way?
- How would you describe your experiences?

issue #3

XP: Extreme programming

- Turns conventional development process *sideways*

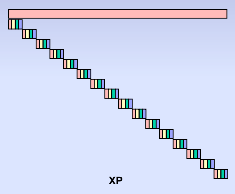


Comparing development models

- Waterfall:
  - Users specify exactly what they want up front
  - Problem: users don't know, are inconsistent, change their minds
  - Problem: programmers dramatically underestimate required effort
- Iterative model:
  - Development cycle shortened to accommodate design changes
  - Entire system not specified in advance: done in chunks
- Extreme programming:
  - Includes simultaneous planning, analysis, and testing on small pieces throughout entire development cycle
  - Currently viewed as popular flavor of Agile software development

XP development cycle

- Up front: big-picture analysis and design
- Customers
  - identify desired features (called *stories* in XP), one per index card
  - pick *either* desired stories for next iteration *or* date of next release
- Programmers
  - estimate cost of each story
  - separate selected stories into iterations and smaller-grained tasks
  - individually accept responsibility for each task



XP: task implementation

- Programmer first finds a partner for pair programming.
- At outset, team creates a set of test cases that will demonstrate that the task is complete.
- Programmers pick a single test case, write code to pass it, and run test.
  - If test is passed, they go on to next test case.
  - If test is failed, they figure out why and do the cleanest redesign possible.
- Technique at heart of XP: unit testing.

XP: testing

- Every programmer does testing every day.
- Tests are written before code is written.
- Tests are added to (large) permanent test set.
  - Reruns automatically to verify *every* subsequent code change.
  - If your change breaks something, you know immediately.
- Each story created must be *testable* and *estimatable*.
- Programmers create unit tests; customers create functional tests for the stories in each iteration.
  - Customer: "I'll know it works when it can do X."

XP: guidelines and philosophy

- Programmers implement only the functionality required by stories selected for current iteration.
- At every moment, the design:
  - runs all tests,
  - contains no duplicate code, and
  - has fewest possible classes and methods.
- Design evolves through changes, keeping all tests running.
- New code is integrated after no more than a few hours.
  - At each point, system rebuilt from scratch: if any test fails, changes are discarded.
- For large projects, customer representative is on site full-time.
- 40-hour weeks: no one can work two consecutive weeks of overtime.
- Everyone follows the rules, but team can agree to change rules.
  - Must agree on how to assess the effects of rule change.

XP developers: quotes

Refactoring [transforming existing design] is a major part of our development effort. It was evident to us that if we were afraid to change some code because we did not know what it did, we were not good developers. We were letting the code control us. If we don't know what the code does now, we break it and find out. It is better to implement a solid piece of code than it is to let a piece of code control the application.

The key to XP is setting developer and team expectations. We have found that all developers on the team must buy into Extreme or it doesn't work. We tell prospective developers if they do not want to follow our development style, this is not a good team for them. One person not buying into the approach will bring down the whole team. XP focuses on the team working together to come up with new ideas to develop the system.

XP developers: quotes

When we started with XP, some of the developers did not want to follow it. They felt that it would hurt their development style and that they would not be as productive. What happened was that their pieces of the application were producing the most problem reports. Since they were not programming in pairs, two people had not designed the subsystem, and their skills were falling behind the other developers who were learning from each other. Two well-trained developers working together and with the rest of the team will always outperform one "intelligent" developer working alone.

A misconception about XP is that it stifles your creativity and individual growth. It's actually quite the contrary. XP stimulates growth and creativity and encourages team members to take chances. The key is to decide the direction of the corporation and stand behind the hard decisions.

Early XP success: DaimlerChrysler

The C3 project began in January 1995 under a fixed-price contract that called for a joint team of Chrysler and contract partner employees. Most of the development work had been completed by early 1996. Our contract partners had used a very GUI-centered development methodology, which had ignored automated testing. As a result we had a payroll system that had a lot of very cool GUIs, calculated most employees' pay incorrectly, and would need about 100 days to generate the monthly payroll. Most of us knew the program we had written would never go into production.

We sought Kent Beck to help with performance tuning. He found what he had often found when brought in to do performance tuning: poorly factored code, no repeatable tests, and a management that had lost confidence in the project. He went to Chrysler Information Services management and told them what he had found, and that he knew how to fix it. Throw all the existing code away! The first full XP project was born.

C3 = Chrysler Comprehensive Compensation

DaimlerChrysler, cont.

We brought Kent in as head coach; he would spend about a week per month with us. Ron Jeffries was brought in as Kent's full-time eyes and ears. The fixed-price contract was cancelled, and about one-half of the Chrysler developers were reassigned. Martin Fowler, who had been advising the Chrysler side of the project all along and clashing with the fixed-price contractor, came in to help the customers develop user stories. From there, we followed Kent as he made up the rules of XP. A commitment schedule was developed, iterations were laid out, rules for testing were established, and paired programming was tried and accepted as the standard. At the end of 33 weeks, we had a system that was ready to begin performance tuning and parallel testing. Ready to begin tuning because it was well factored and backed up by a full battery of unit tests. And, ready to begin parallel testing because a suite of functional tests had shown the customers that the required functionality was present.

DaimlerChrysler, cont.

That increment of C3 launched in May 1997... Since the launch of the monthly system, we've added several new features, and we have enhanced the system to pay the biweekly paid population. We have been paying a pilot group since August 1998 and will roll out the rest before the Y2K code freeze in November 1999.

Looking back on this long development experience, I can say that when we have fallen short of keeping our promises to our management and our customers, it is because we have strayed from the principles of XP. When we have driven our development with tests, when we have written code in pairs, when we have done the simplest thing that could possibly work, we have been the best software development team on the face of the earth.

Chet Hendrickson, DaimlerChrysler

Should you XP?

- Recognize:
  - It is not a polished, one-size-fits all technique.
  - Probably most appropriate for small- to medium-sized systems where requirements are vague, likely to change.

If you want to try XP, for goodness sake don't try to swallow it all at once. Pick the worst problem in your current process and try solving it the XP way. When it isn't your worst problem, rinse and repeat. As you go along, if you find that any of your old practices aren't helping, stop doing them.
Kent Beck

Another view of XP:

"Latest eruption in hostilities between two opposing camps"

Programmers (scruffy hackers)	**Software engineers (tweedy CS types)**
Beliefs/characteristics:	Beliefs/characteristics:
Code is easy to change	Code is expensive to change
Likes verbal communication	Likes written specification
The code is the design	Code is poor design artifact
Good designs emerge	Good design comes up front
Programmers collaborate	Programmers can't communicate
Codes with peers	Reviews code for defects
Informal requirements suffice	Formal specs and change control

"They resemble Republicans and Democrats battling ideologues caught up in the divisive dualism of either-or positions on hot button issues…."

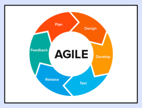
Source: "Programming Extremism", McCormick, CACM June 2001

Another view, cont.

- Statements from paper:
  - "A one-size-fits-all development process does not exist."
  - Author "watched helplessly while a high ceremony heavyweight process brought an organization of talented, formerly productive software engineers to a dead stop. Crimes were committed in the name of SEI CMM and ISO 9001."
  - "On the other hand, I once had the privilege to observe an organization achieve CMM maturity Level 4 certification without the baggage of a productivity-killing, paperwork-clogged high ceremony methodology."
  - "What's needed is not a single software methodology, but a rich toolkit" of options, from which developers pick the best matches for a given project.

Agile

- Agile is a time-boxed iterative approach to software delivery that builds software incrementally from the start of the project, instead of trying to deliver it all at once near the end.
- The three most popular flavors of Agile are
  - XP
  - Scrum
  - Kanban
- Differences between the three approaches:
  - Teams organization, required meetings and practices, who picks features for next iteration, iteration length, allowing changes during an iteration, using pair programming, etc.



Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions	over	processes and tools
Working software	over	comprehensive documentation
Customer collaboration	over	contract negotiation
Responding to change	over	following a plan

That is, while there is value in the items on the right, we value the items on the left more.

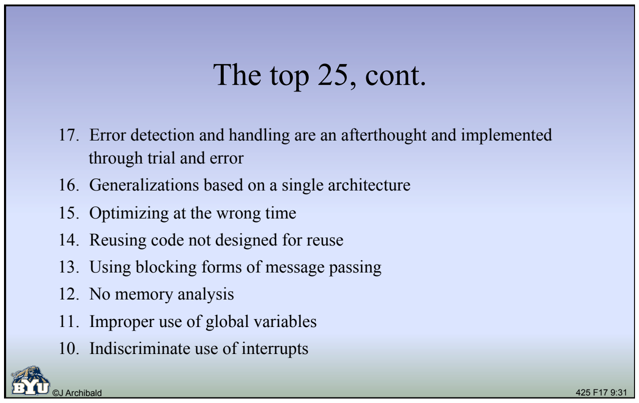
<signed by 17 individuals>

issue #4

The top 25 mistakes (with real-time software development)

25. My problem is different
24. Delays implemented as empty loops
23. Tools choice driven by marketing hype, not by evaluation of technical needs
22. Large if-then-else and case statements
21. Documentation was written after implementation
20. Interactive and incomplete test programs
19. Software engineers not participating in hardware design
18. No emulators of target application

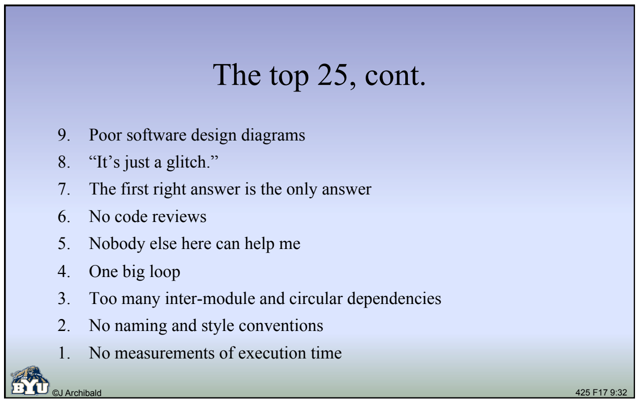
David Stewart, Embedded Systems Conference, Boston, Sept. 2006



The top 25, cont.

17. Error detection and handling are an afterthought and implemented through trial and error
16. Generalizations based on a single architecture
15. Optimizing at the wrong time
14. Reusing code not designed for reuse
13. Using blocking forms of message passing
12. No memory analysis
11. Improper use of global variables
10. Indiscriminate use of interrupts

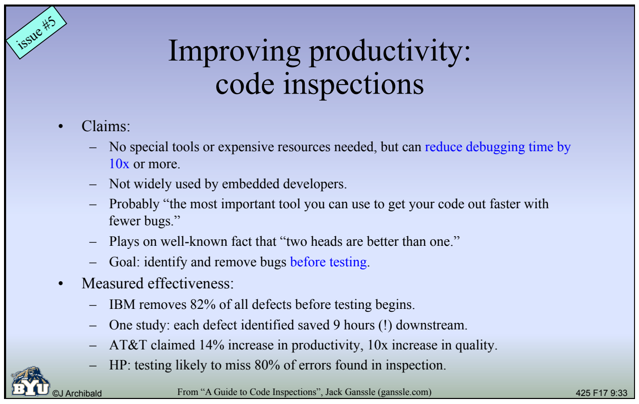
©J Archibald 425 F17 9:31



The top 25, cont.

9. Poor software design diagrams
8. "It's just a glitch."
7. The first right answer is the only answer
6. No code reviews
5. Nobody else here can help me
4. One big loop
3. Too many inter-module and circular dependencies
2. No naming and style conventions
1. No measurements of execution time

©J Archibald 425 F17 9:32

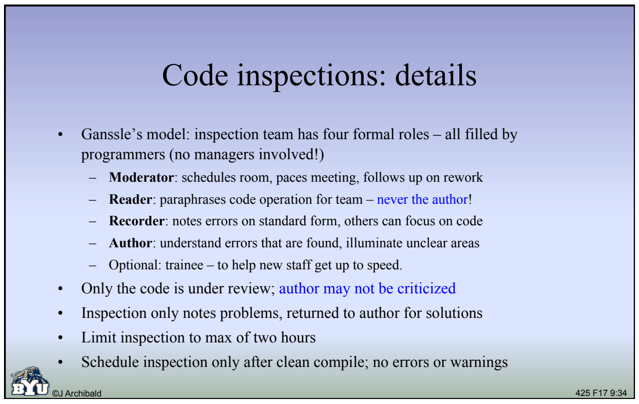


issue #5

Improving productivity: code inspections

- Claims:
  - No special tools or expensive resources needed, but can reduce debugging time by 10x or more.
  - Not widely used by embedded developers.
  - Probably "the most important tool you can use to get your code out faster with fewer bugs."
  - Plays on well-known fact that "two heads are better than one."
  - Goal: identify and remove bugs before testing.
- Measured effectiveness:
  - IBM removes 82% of all defects before testing begins.
  - One study: each defect identified saved 9 hours (!) downstream.
  - AT&T claimed 14% increase in productivity, 10x increase in quality.
  - HP: testing likely to miss 80% of errors found in inspection.

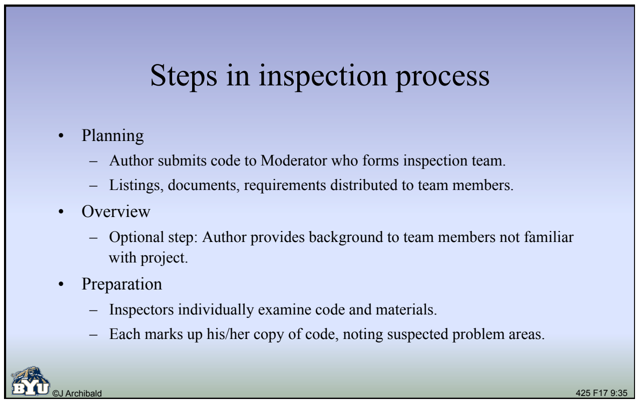
©J Archibald From "A Guide to Code Inspections", Jack Ganssle (ganssle.com) 425 F17 9:33



Code inspections: details

- Ganssle's model: inspection team has four formal roles – all filled by programmers (no managers involved!)
  - **Moderator**: schedules room, paces meeting, follows up on rework
  - **Reader**: paraphrases code operation for team – never the author!
  - **Recorder**: notes errors on standard form, others can focus on code
  - **Author**: understand errors that are found, illuminate unclear areas
  - Optional: trainee – to help new staff get up to speed.
- Only the code is under review; author may not be criticized
- Inspection only notes problems, returned to author for solutions
- Limit inspection to max of two hours
- Schedule inspection only after clean compile; no errors or warnings

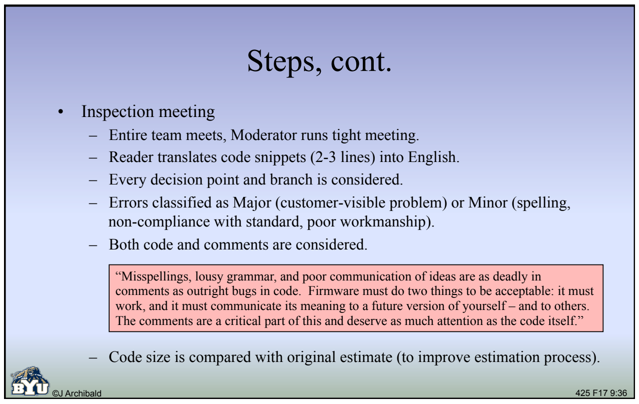
©J Archibald 425 F17 9:34



Steps in inspection process

- Planning
  - Author submits code to Moderator who forms inspection team.
  - Listings, documents, requirements distributed to team members.
- Overview
  - Optional step: Author provides background to team members not familiar with project.
- Preparation
  - Inspectors individually examine code and materials.
  - Each marks up his/her copy of code, noting suspected problem areas.

©J Archibald 425 F17 9:35



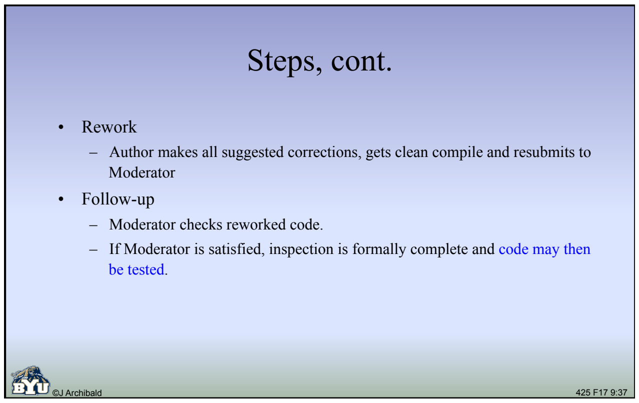
Steps, cont.

- Inspection meeting
  - Entire team meets, Moderator runs tight meeting.
  - Reader translates code snippets (2-3 lines) into English.
  - Every decision point and branch is considered.
  - Errors classified as Major (customer-visible problem) or Minor (spelling, non-compliance with standard, poor workmanship).
  - Both code and comments are considered.

"Misspellings, lousy grammar, and poor communication of ideas are as deadly in comments as outright bugs in code. Firmware must do two things to be acceptable: it must work, and it must communicate its meaning to a future version of yourself – and to others. The comments are a critical part of this and deserve as much attention as the code itself."

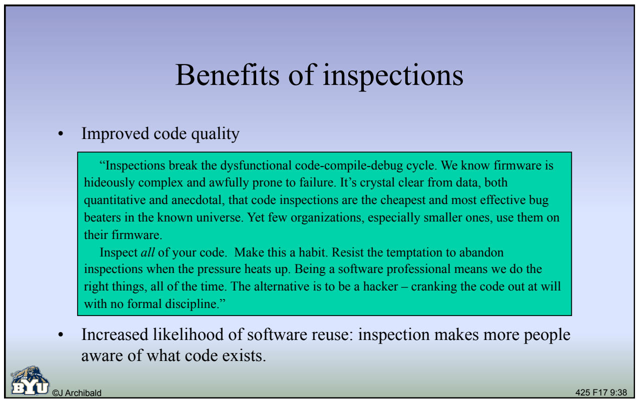
  - Code size is compared with original estimate (to improve estimation process).

©J Archibald 425 F17 9:36



Steps, cont.

- Rework
  - Author makes all suggested corrections, gets clean compile and resubmits to Moderator
- Follow-up
  - Moderator checks reworked code.
  - If Moderator is satisfied, inspection is formally complete and code may then be tested.



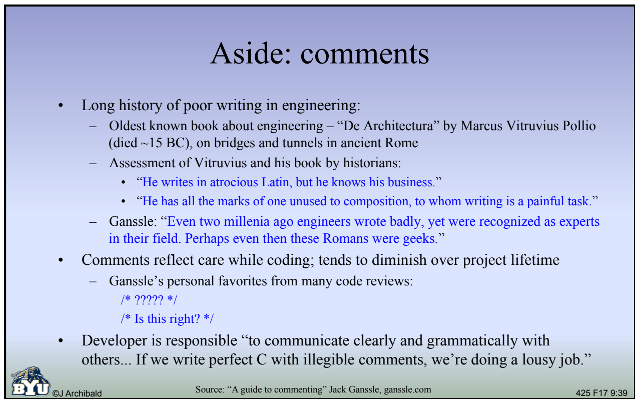
Benefits of inspections

- Improved code quality

"Inspections break the dysfunctional code-compile-debug cycle. We know firmware is hideously complex and awfully prone to failure. It's crystal clear from data, both quantitative and anecdotal, that code inspections are the cheapest and most effective bug beaters in the known universe. Yet few organizations, especially smaller ones, use them on their firmware.

Inspect *all* of your code. Make this a habit. Resist the temptation to abandon inspections when the pressure heats up. Being a software professional means we do the right things, all of the time. The alternative is to be a hacker – cranking the code out at will with no formal discipline."

- Increased likelihood of software reuse: inspection makes more people aware of what code exists.



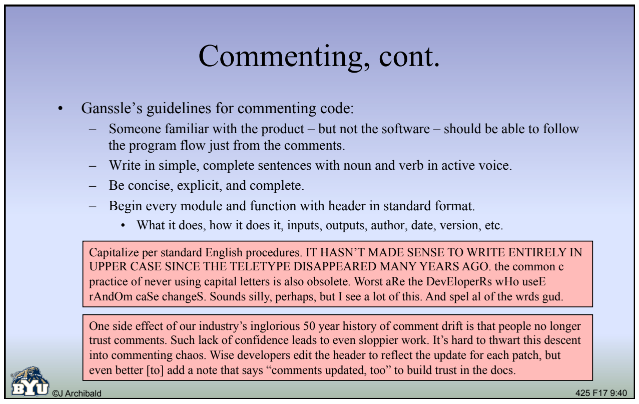
Aside: comments

- Long history of poor writing in engineering:
  - Oldest known book about engineering – "De Architectura" by Marcus Vitruvius Pollio (died ~15 BC), on bridges and tunnels in ancient Rome
  - Assessment of Vitruvius and his book by historians:
    - "He writes in atrocious Latin, but he knows his business."
    - "He has all the marks of one unused to composition, to whom writing is a painful task."
  - Ganssle: "Even two millenia ago engineers wrote badly, yet were recognized as experts in their field. Perhaps even then these Romans were geeks."
- Comments reflect care while coding; tends to diminish over project lifetime
  - Ganssle's personal favorites from many code reviews:

    /* ????? */

    /* Is this right? */
- Developer is responsible "to communicate clearly and grammatically with others... If we write perfect C with illegible comments, we're doing a lousy job."

Source: "A guide to commenting" Jack Ganssle, ganssle.com

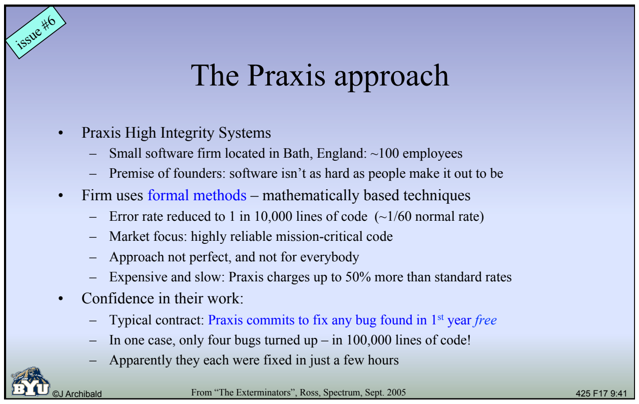


Commenting, cont.

- Ganssle's guidelines for commenting code:
  - Someone familiar with the product – but not the software – should be able to follow the program flow just from the comments.
  - Write in simple, complete sentences with noun and verb in active voice.
  - Be concise, explicit, and complete.
  - Begin every module and function with header in standard format.
    - What it does, how it does it, inputs, outputs, author, date, version, etc.

Capitalize per standard English procedures. IT HASN'T MADE SENSE TO WRITE ENTIRELY IN UPPER CASE SINCE THE TELETYPE DISAPPEARED MANY YEARS AGO. the common c practice of never using capital letters is also obsolete. Worst aRe the DevEloperRs wHo useE rAndOm caSe changeS. Sounds silly, perhaps, but I see a lot of this. And spel al of the wrds gud.

One side effect of our industry's inglorious 50 year history of comment drift is that people no longer trust comments. Such lack of confidence leads to even sloppier work. It's hard to thwart this descent into commenting chaos. Wise developers edit the header to reflect the update for each patch, but even better [to] add a note that says "comments updated, too" to build trust in the docs.

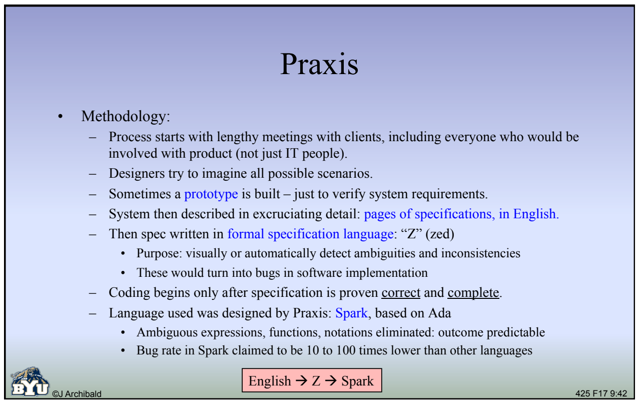


issue #6

The Praxis approach

- Praxis High Integrity Systems
  - Small software firm located in Bath, England: ~100 employees
  - Premise of founders: software isn't as hard as people make it out to be
- Firm uses formal methods – mathematically based techniques
  - Error rate reduced to 1 in 10,000 lines of code (~1/60 normal rate)
  - Market focus: highly reliable mission-critical code
  - Approach not perfect, and not for everybody
  - Expensive and slow: Praxis charges up to 50% more than standard rates
- Confidence in their work:
  - Typical contract: Praxis commits to fix any bug found in 1st year *free*
  - In one case, only four bugs turned up – in 100,000 lines of code!
  - Apparently they each were fixed in just a few hours

From "The Exterminators", Ross, Spectrum, Sept. 2005



Praxis

- Methodology:
  - Process starts with lengthy meetings with clients, including everyone who would be involved with product (not just IT people).
  - Designers try to imagine all possible scenarios.
  - Sometimes a prototype is built – just to verify system requirements.
  - System then described in excruciating detail: pages of specifications, in English.
  - Then spec written in formal specification language: "Z" (zed)
    - Purpose: visually or automatically detect ambiguities and inconsistencies
    - These would turn into bugs in software implementation
  - Coding begins only after specification is proven correct and complete.
  - Language used was designed by Praxis: Spark, based on Ada
    - Ambiguous expressions, functions, notations eliminated: outcome predictable
    - Bug rate in Spark claimed to be 10 to 100 times lower than other languages

English → Z → Spark

Praxis

- Will approach scale?
  - Praxis thinks so, but biggest tried at time was ~200,000 lines
  - Reference points:
    - Windows XP: 45 million lines of code
    - Debian Linux 2.2: 55 million lines of code
    - IRS tax return software: 100s of millions of lines of code

Classic quotes re-quoted in paper:

In most projects, the first system built is barely usable. It may be too slow, too big, awkward to use, or all three. There is no alternative but to start again, smarting but smarter and build a redesigned version in which these problems are solved. The discard and redesign may be done in one lump, or it may be done piece-by-piece. But all large-system experience shows that it will be done…. Hence plan to throw one away; you will, anyhow.
Frederick Brooks

Program testing can be used to show the presence of bugs, but never their absence.
Edsger Dijkstra

issue #7

System reliability: a case study

- True story:
  - Location: secret Serpukhov-15 early-warning facility, near Moscow
  - Center responsibility: validate warnings of US launch to Soviet high command
  - Lieutenant Colonel Stanislav Petrov is in command
  - Just after midnight, 26 September 1983
  - Siren howls, computers indicate launch of an ICBM in the US
  - 200 computer operators stop work, leap from their seats, look to Petrov
- Situation and background:
  - High political tensions between Soviets and US
  - US openly planning European deployment of long-range Pershing II ballistic missiles and ground-launched cruise missiles
  - US deployment was a response to Soviet deployments of intermediate range SS-20 ballistic missiles

From: "False alarm, nuclear danger", Spectrum, March 2000

World War III?

- Tension:
  - US and NATO were organizing a military exercise later that fall focusing on the use of tactical nuclear weapons in Europe
  - Soviet leaders feared exercise was a cover for an actual invasion
- The technology:
  - The Soviets had long had ground-based radars ringing the country
  - Would give the leaders ~15 minutes warning in case of nuclear attack
  - Soviets had just added space-based early warning system to extend warning to ~30 minutes
  - Nine Oko satellites in highly elliptical orbits took turns scanning skies above US missile fields

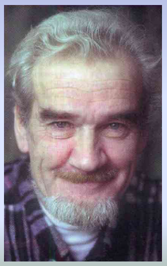
World War III?

- The incident:
  - Alarm given by Cosmos 1382, just reaching the high point of its orbit, directly above northern Europe
  - From its perspective, US was on horizon
  - Line from satellite to Malmstrom AFB in Montana extended directly into setting sun
  - Apparently scattered high-altitude clouds above Malmstrom reflected sunlight into infrared sensors aboard Cosmos 1382
  - This was mistaken for bright light given off by hot gases in missile plume
  - Normally infrared light reflects diffusely, but near the equinox co-linear sun can cause specular reflections; clouds act as mirrors
  - Designers had tried to avoid this by choosing grazing viewing angle to increase atmospheric absorption; this kind of reflection was unanticipated

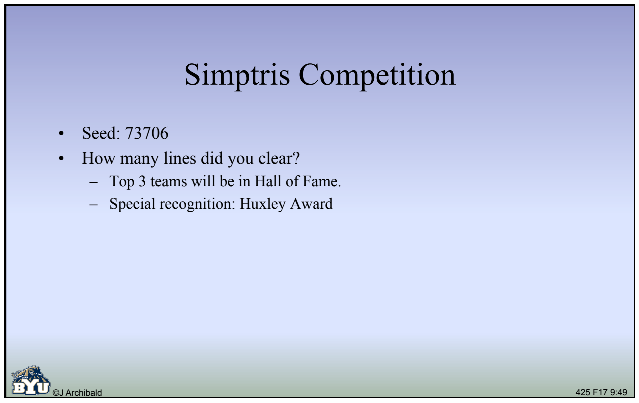
World War III?

- So what happened?
  - The instruments showed one launch, then a second, eventually five total (apparently as different clouds started reflecting light): "probability of attack, 100%"
  - This clearly wasn't the all-out attack they were expecting – but what was happening?
  - He was scared, his hands were shaking, but he yelled at the others to get back to work
  - He notified his superiors that it was a false alarm – later called this a 50-50 guess
  - 15 unbearable minutes later, it was clear that he had made the right call
  - Military leaders were horribly embarrassed by what he had done, showing them up
  - Old Russian rule: subordinate must never be cleverer than the boss
  - They cited him for failing to fill out the operations log that night
  - He left the army a few months later to take job as research engineer
  - Within one year, Soviets started using separate satellites in geostationary orbit to give different viewing angles to confirm warning data

Cold war hero



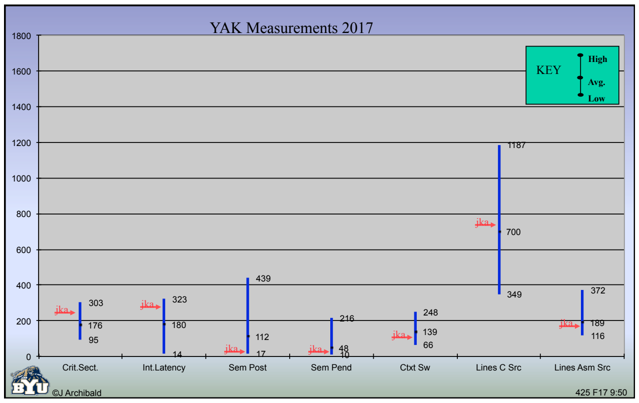
Lieutenant Colonel Stanislav Petrov, died on May 19th, 2017. For the last several years of his life, he resided in the outskirts of Moscow, living on a small military pension. His story stayed secret until 1998.



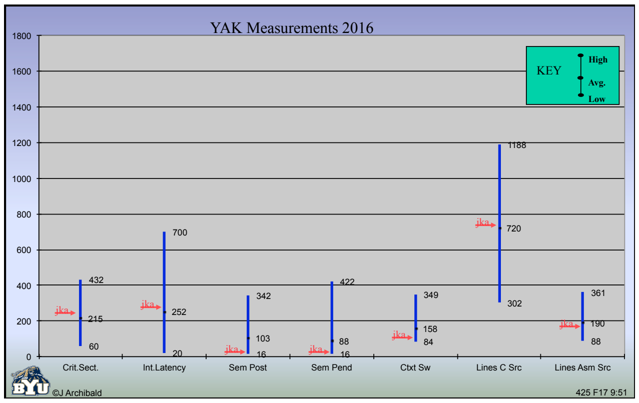
Simptris Competition

- Seed: 73706
- How many lines did you clear?
  - Top 3 teams will be in Hall of Fame.
  - Special recognition: Huxley Award

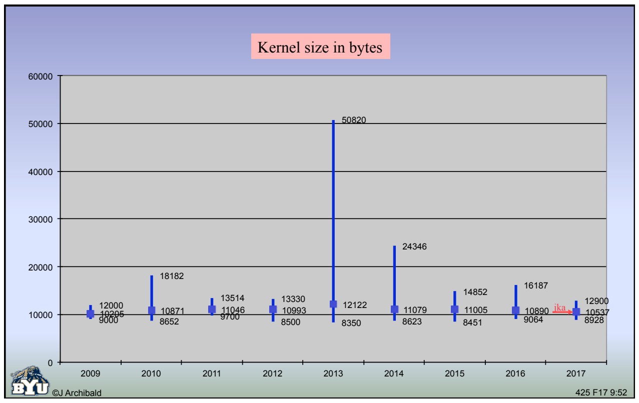
©J Archibald 425 F17 9:49



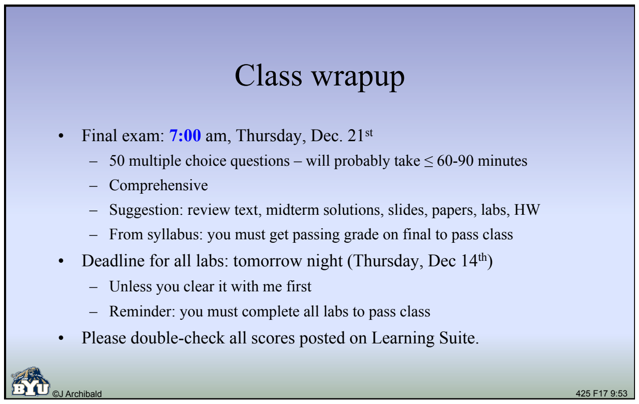
©J Archibald 425 F17 9:50



©J Archibald 425 F17 9:51



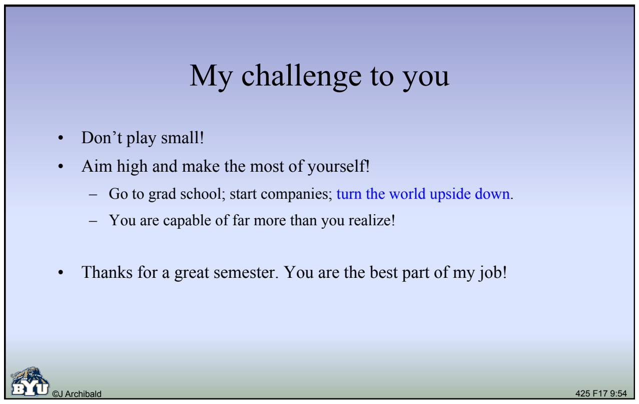
©J Archibald 425 F17 9:52



Class wrapup

- Final exam: **7:00** am, Thursday, Dec. 21st
  - 50 multiple choice questions – will probably take ≤ 60-90 minutes
  - Comprehensive
  - Suggestion: review text, midterm solutions, slides, papers, labs, HW
  - From syllabus: you must get passing grade on final to pass class
- Deadline for all labs: tomorrow night (Thursday, Dec 14th)
  - Unless you clear it with me first
  - Reminder: you must complete all labs to pass class
- Please double-check all scores posted on Learning Suite.

©J Archibald 425 F17 9:53



My challenge to you

- Don't play small!
- Aim high and make the most of yourself!
  - Go to grad school; start companies; turn the world upside down.
  - You are capable of far more than you realize!
- Thanks for a great semester. You are the best part of my job!

©J Archibald 425 F17 9:54