9
Reviews
& Software Process

Distributed Embedded Systems

Philip Koopman

September 28, 2015

Highly Recommended reading:
Improving Quality Through Software Inspections, Weigers, 1995

Figures & inserted text taken from "Peer Reviews",
Encyclopedia of Software Engineering, D. O'Neill.

Carnegie Mellon

Overview

- **Reviews**
  - How to save money, time, and effort doing reviews
  - Some project-specific review info – checklists to use in course project

- **Software process**
  - What's CMM stuff about?
  - Does the embedded world care (yes!)

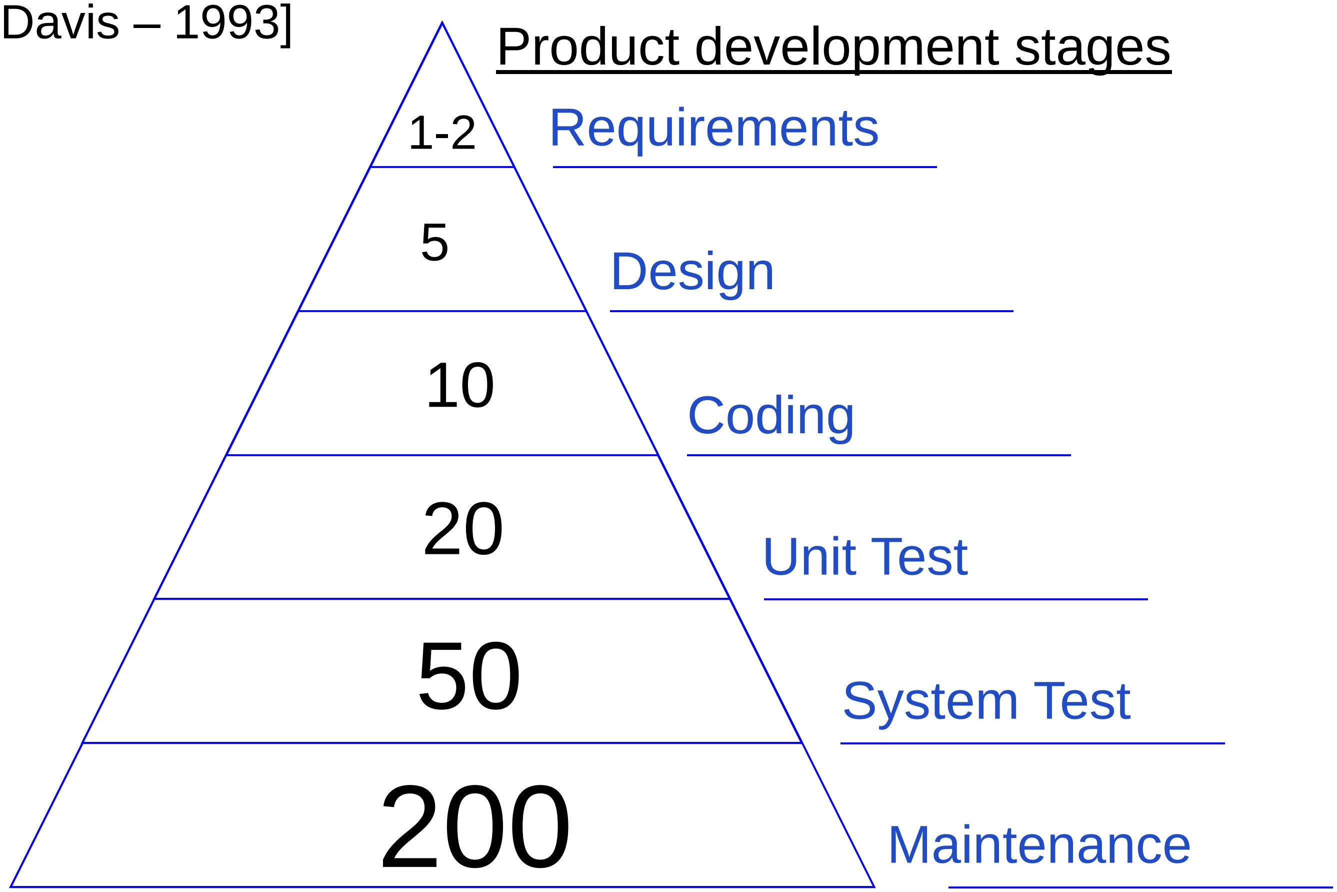
- **Motivation: (From Ganssle required reading)**
  - Software typically ships with 10 to 30 defects per KSLOC
    - (KSLOC = 1000 lines of source code with comments removed)
  - With no reviews at all, might be up to 50 defects per KSLOC
  - With best practice reviews, it might be as low as 7.5 defects per KSLOC
    - (You can get lower, but bring bushels of money; more about that in later lectures)

Early Detection and Removal of Defects -

Peer Reviews - remove defects early and efficiently

Relative Cost to Fix Requirements Errors

[Davis – 1993]



Boehm's Top 10 List On Software Improvement

1. **Fix problems early; it's cheaper to do it then**
2. **Rework is up to 40%-50% of current costs**
3. **80% of avoidable rework comes from 20% of the defects**
4. **80% of defects comes from 20% of modules**
   - "Half the modules are defect-free" [Ed. Note: for narrow view of "correct"]
5. **90% of the downtime comes from 10% of the defects**
6. **Peer review catches 60% of defects -- great bang for buck**
7. **Perspective-based reviews are 35% more effective**
   - Assign roles to reviewers so they are responsible for specific areas
8. **Disciplined design practices help by up to 75% (fewer defects)**
9. **Dependable code costs 50% more per line to write**
   - But it costs less across life cycle if it is not truly a disposable product
   - Of course exception handling increases # lines too
10. **40%-50% of user programs deploy with non-trivial defects**
    - (Spreadsheets, etc.) Are these critical pieces of software?

Most Effective Practices For Embedded Software Quality

Ebert & Jones, "Embedded Software: Facts, Figures, and Future, IEEE Computer, April 2009, pp. 42-52

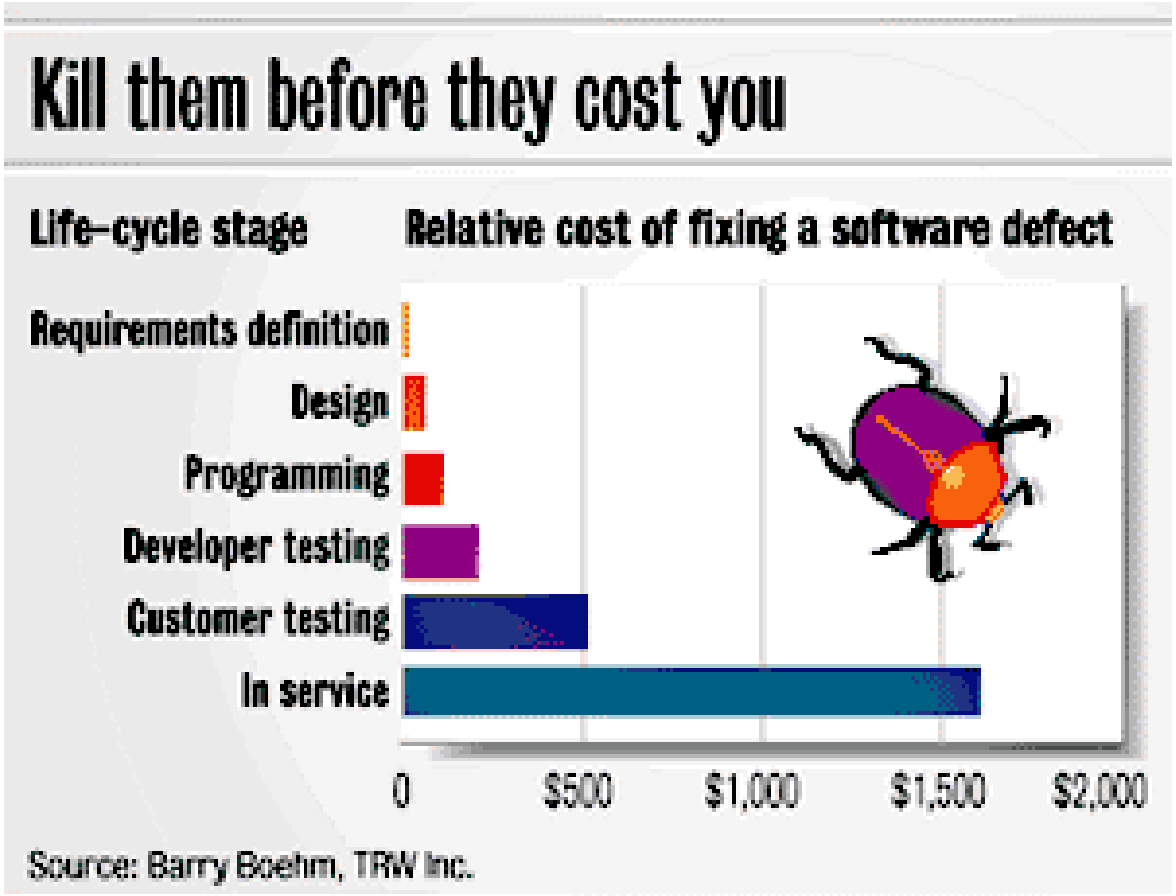
Ranked by defect removal effectiveness in percent defects removed.

"*" means exceptionally productive technique (more than 750+ function points/month)

- * 87% static code analysis ("lint" tools, compiler warnings)
- 85% design inspection
- 85% code inspection
- 82% Quality Function Deployment (requirements analysis used by auto makers)
- 80% test plan inspection
- 78% test script inspection
- * 77% document review (documents that aren't code, design, test plans)
- 75% pair programming (review aspect)
- 70% bug repair inspection
- * 65% usability testing
- 50% subroutine testing
- * 45% SQA (Software Quality Assurance) review
- * 40% acceptance testing

Peer Reviews Help Find Defects Early

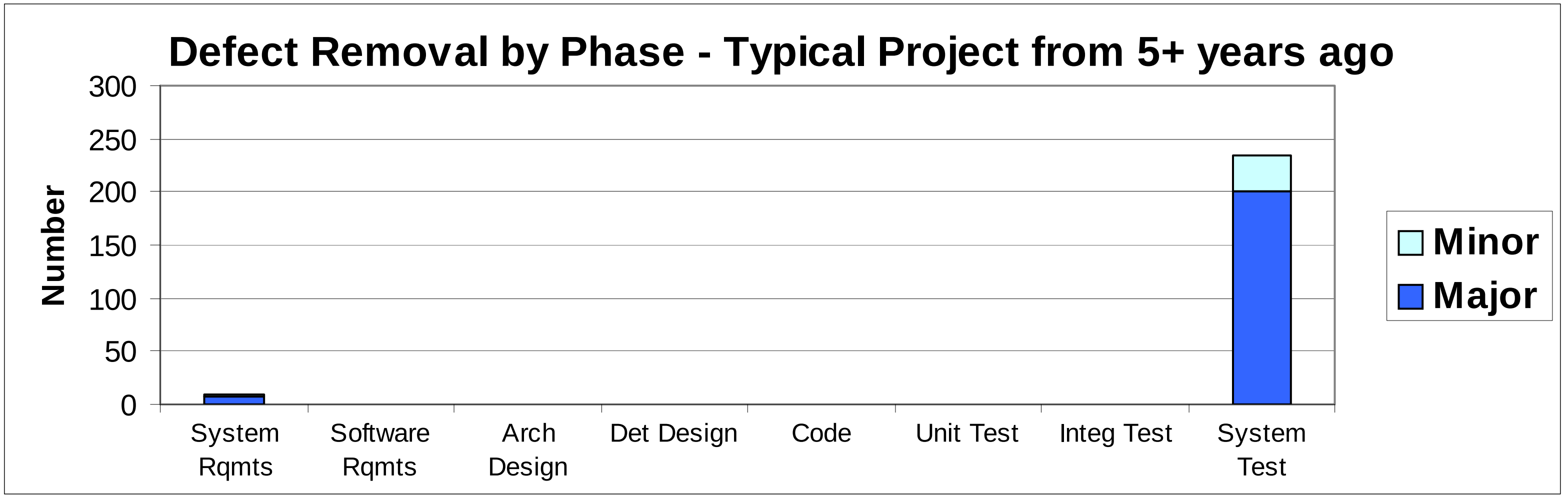
- ***Good*** **Peer Reviews for embedded software can find HALF the defects for about 10% of the total project cost.**
  - This takes into account in-person formal review meetings with 4 attendees
  - Testing (hopefully) finds the other half, but costs up to 50% of project cost

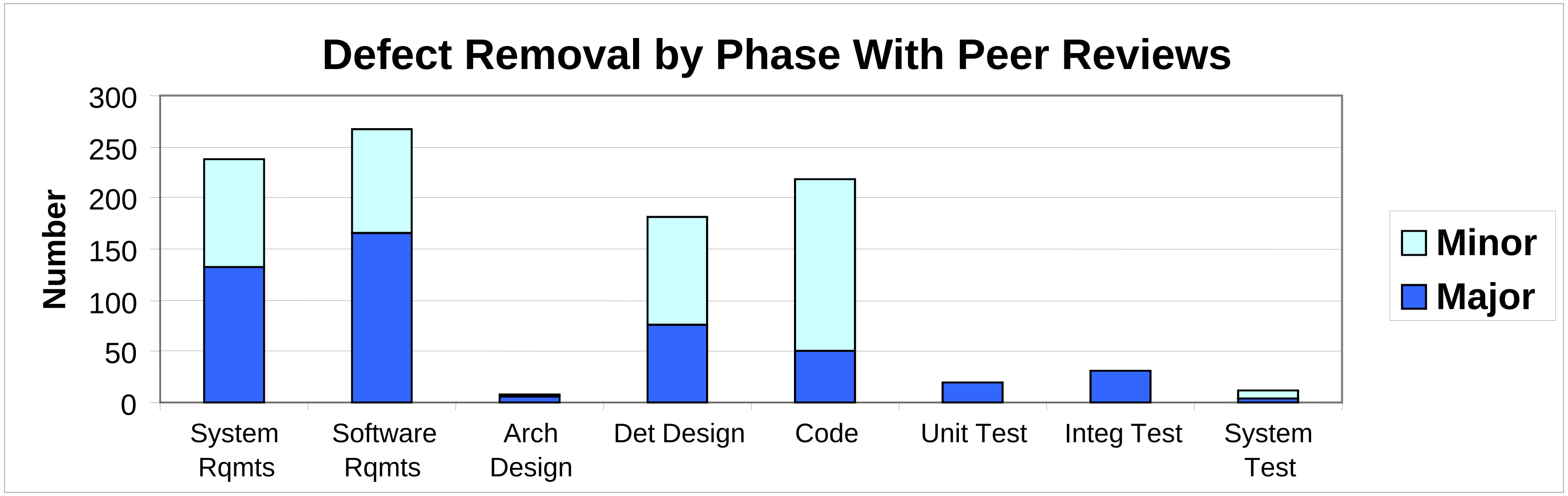


Peer Reviews Really Work

(Real data from embedded industry)

Defects are removed earlier & more defects are removed!



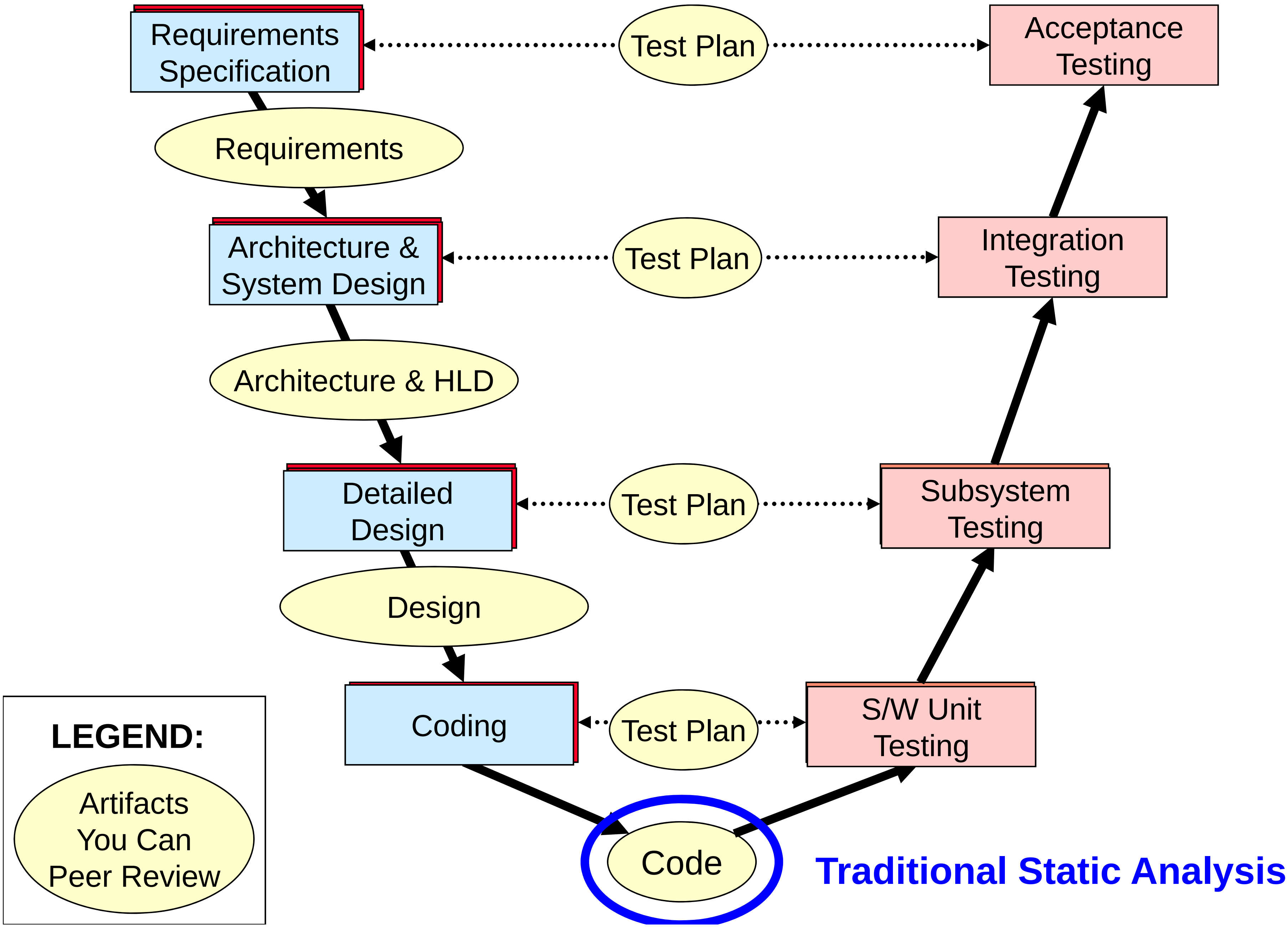


[Source: Roger G., Aug. 2005]

What Can You Review?

- ***Nobody is perfect***
  - We all need help seeing our own mistakes
  - … and to motivate beyond being lazy with our style …
- ***Review anything written down on paper!***
  - Code reviews are just a starting point
- **Examples of things that can and should be reviewed:**
  - Software development plan
  - Software quality assurance plan
  - … and see next slide …

Review Artifacts Across Development Lifecycle



How Formal Should Reviews Be?

- **The more formal the review, the higher the payoff**
  - Formal reviews take more effort; more productive
    - We mean use these: “Fagan style inspections”
  - Formal reviews of absolutely everything should still be less than perhaps 10% of total project cost
    - In return, you find half of your bugs much earlier

- **Gold Standard: Fagan Style Inspection**
  - Pre-review meeting
  - Formal meeting (see next slide)
  - Written review report
  - Follow-up and possible re-inspection

Typical Software Code Inspection

- **Focus on ~200-400 lines of code (probably 1-2 hour session)**
  - Optimum is 100-200 lines of code reviewed per hour
  - Optimum is a 1 to 2 hour session

- **The team:**
  - *producer* explains code for ~20 minutes (and then leaves, or stays only to answer questions)
  - *moderator* keeps the discussion going to cover all the code in < 2 hrs
  - *recorder* takes notes for report
  - *reviewers* go over checklists for each line, raise issues
  - *reader* reads or summarizes code and gives background as needed during the review (rather than having the producer do this)

NOTE: The outcome is a list of defects (issues) to be given to the producer, not the solutions!

- **Rework:** The process of addressing the issues and fixes the code as needed for a re-review.

- **Review – any type of review that does not follow the above formula**
  - Inspection is formal per above formula (and, overall, a more effective use of time)
  - Review is an umbrella term, but often means things that are informal

Typical Topics In Design Reviews

- **Completeness**
  - Are all the pieces required by the process really there?
- **Correctness**
  - Does the design/implementation do everything it should (and nothing it shouldn't)?
  - Is exception handling performed to an appropriate degree?
- **Style**
  - Does the item being reviewed follow style guidelines (internal or external)?
  - Is it understandable to someone other than the author?
- **Rules of construction**
  - Are interfaces correct and consistent with system-level documentation?
  - Are design constraints documented and adhered to?
  - Are modes, global state, and control requirements handled appropriately?
- **Multiple views**
  - Has the design addressed: real time, user interface, memory/CPU capacity, network capacity, I/O, maintenance, and life cycle issues?

Rules for Reviews

- **Inspect the item, not the author**
  - Don't attack the author
- **Don't get defensive**
  - Nobody writes perfect code. Get over it.
- **Find problems – but don't fix them in the meeting**
  - Don't try to fix them; just identify them
- **Limit meetings to two hours**
  - People are less productive after that point
- **Keep a reasonable pace**
  - 150-200 lines per hour.   Faster and slower are both bad
- **Avoid "religious" debates on style**
  - Concentrate on substance. Ideally, use a style guideline and conform to that
  - For lazy people it is easier to argue about style than find real problems

- **Inspect, early, often, and as formally as you can**
  - Expect reviews to take a while to provide value
  - Keep records so you can document the value

Starting Point For Firmware Reviews

- **Design review can be more effective if checking conformance to a checklist**
  - Includes coding standards
  - Includes items to detect defects that occur commonly or have caused big problems in the past (capture of "fly-fix-fly" knowledge)
- **Every project should have a coding standard including:**
  - Naming conventions
  - Formatting conventions
  - Interrupt service routine usage
  - Commenting
  - Tools & compiler compatibility
- **http://www.ganssle.com/misc/fsm.doc**
  - Starting point for non-critical embedded systems
  - But, one of the better starting points; specifically intended for embedded systems

Best Checklist Practices for Code

- **Use the checklist!**
  - Emphasis originated in military organizations; NASA used them extensively
  - Just because you know the steps doesn't mean you will follow them!
- **Make your own local copy of any checklist**
  - Add things to the checklist if they bite you
  - Delete things from the checklist that aren't relevant
  - Keep the checklist length about right – only the stuff that really matters
- **For this particular checklist:**
  - Assign each section below to a specific reviewer, giving two or three sections to each reviewer.
  - Ensure that each question has been considered for every piece of code.
  - Review 100-400 lines of code per 1-2 hour review session. Do the review in person.
- **A word about honesty**
  - Filling out a checklist when you didn't actually do the checks wastes everyone's time. The point is NOT to fill in the checklist. The point is TO FIND BUGS!

Code Review Checklist

- **Based on experience with real projects**
  - Emphasis on actual code and hardware design
  - Intentionally omits architecture, requirements, etc.
    - Those are important! But this list is focused largely on code review

VALIDATION & TEST

- ☐ V-1. Is the code easy to test? (how many paths are there through the code?)
- ☐ V-2. Do unit tests have 100% branch coverage? (code should be written to make this easy)
- ☐ V-3. Are the compilation and/or lint checks 100% warning-free? (are warnings enabled?)
- ☐ V-4. Is special attention given to corner cases, boundaries, and negative test cases?
- ☐ V-5. Does the code provide convenient ways to inject faulty conditions for testing?
- ☐ V-6. Are all interfaces tested, including all exceptions?
- ☐ V-7. Has the worst case resource use been validated? (stack space, memory allocation)
- ☐ V-8. Are run-time assertions being used? Are assertion violations logged?
- ☐ V-9. Is there commented out code (for testing) that should be removed?

HARDWARE

- ☐ H-1. Do I/O operations put the hardware in correct state?
- ☐ H-2. Are min/max timing requirements met for the hardware interface?
- ☐ H-3. Are you sure that multi-byte hardware registers can't change during read/write?

Code Checklist Starting Points

- **There is no point starting from scratch. The "digging deeper" section of the course web page gives some starting points, including:**
  - Ganssle, *A Firmware Development Standard*, Version 1.2, Jan. 2004.
  - Ambler, S., *Writing robust Java code: the Ambysoft coding standards for Java, Ambysoft*, 2000.
  - Baldwin, *An Abbreviated C++ Code Inspection Checklist*, 1992.
  - Madau, D., "Rules for defensive C programming," *Embedded Systems Programming*, December 1999.
  - Wiegers, *Peer Reviews in Software: A Practical Guide*, Addison-Wesley, 2002
  - Khattak & Koopman, Embedded System Code Review Checklist, 2011

- **Following slides are starting checklists for the course project**
  - Be sure to look at the grading rubricks for each assignment!

Architecture Minimal Checklist

- **Architecture Diagram**
  - ❑ All architectural items are in diagram
  - ❑ Each object has replication information
  - ❑ All sensor/actuators send analog information to a controller (or are “smart”)

- **Message Dictionary**
  - ❑ All messages are fully defined:
    - ❑ Replication of transmitters
    - ❑ List of parameters
    - ❑ Range of possible values for each parameter
    - ❑ Value for initialized system
    - ❑ Description of content
  - ❑ Each message has exactly one unique source with ET/TT & repetition rate
  - ❑ Each message appears in at least one Sequence Diagram
  - ❑ “m” prefix notation is used for network messages

Use Case Minimal Checklist

- **Each Use Case:**
  - ❑ Is named with a brief verb phrase
  - ❑ Is numbered
  - ❑ Has one or more actors

- **Traceability:**
  - ❑ Each system level requirement is traced to at least one Use Case
  - ❑ Each Use Case is traceable to at least one Scenario

Scenario Minimal Checklist

- **Each Scenario:**
  - ❑ Is numbered, with that number traceable to a Use Case
  - ❑ Has a one-sentence descrpt
  - ❑ Has numbered pre-conditions
  - ❑ Has numbered steps
  - ❑ Has numbered post-conditions

- **Traceability:**
  - ❑ Each Scenario traces to a Use Case
  - ❑ Each Use Case traces to one or more Scenarios
  - ❑ Scenario to/from Sequence Diagram

Sequence Diagram Minimal Checklist

- **One SD for each "important" nominal & off-nominal behavior**

- **Objects:**
  - Each object is found in Architecture diagram
  - Each object should have replication letter

- **Messages**
  - Each message is found in Message Dictionary
  - Each message has defined replication & parameter values
  - Each message is numbered
  - "m"prefix notation used to indicate "network message"

- **Traceability:**
  - Scenario traces to (at least) one Sequence Diagram
  - Each Scenario step traces to one or more Sequence Diagram arcs
  - Each Sequence Diagram arc traces to one Scenario step
  - Traceable to/from Requirements

Requirements Minimal Checklist

- **Requirement section per object:**
  - ❑ Named and Numbered
  - ❑ Replication
  - ❑ Instantiation
  - ❑ Assumptions
  - ❑ Input Interface
  - ❑ Output Interface
  - ❑ State/variable definitions
  - ❑ Constraints (numbered)
  - ❑ Behaviors (numbered)
- **Behaviors:**
  - ❑ All input values handled
  - ❑ All output values produced
  - ❑ Correct TT/ET formulation
  - ❑ Use of should/shall
- **Traceability:**
  - ❑ Inputs to SD arcs
  - ❑ Outputs to SD arcs
  - ❑ Behaviors to SD arcs
  - ❑ SD arcs to Behaviors
  - ❑ Inputs to LHS of behaviors
  - ❑ Outputs to RHS of behaviors
  - ❑ Requirements to/from Statechart

Statechart Minimal Checklist

- ❑ **At least one (sometimes more) Statechart for each object**

- **Statechart:**
  - ❑ Initialization arc
  - ❑ Named and numbered states
  - ❑ Entry action (if applicable)
  - ❑ Numbered arcs
  - ❑ Per-arc guard condition & action
  - ❑ Notation if event triggered

- **Traceability:**
  - ❑ Statechartarcs to Behavioral Requirements
  - ❑ Behavioral Requirements to Statechartarcs

Code Minimal Checklist

- **Code module for each object**
  - ❑ "Reasonable" and consistent coding style
  - ❑ Enough comments for TA & team members to understand in independent review
  - ❑ Uses straightforward state machine implementation as appropriate
  - ❑ Recompiled for each run (don't get bitten by the "stale .class file" bug!)
  - ❑ Compiles "clean" with NO WARNINGS

- **Traceability**
  - ❑ Subscription information matches input & output sections of requirements
  - ❑ Each Statechart arc traced to a comment on a line of code
  - ❑ Each Statechart state traces to one state machine label/case statement clause

- **Use a consistent implementation style ("coding style")**
  - For example, commenting philosophy and comment headers
  - Chapter 17 of text describes this
    You *should have* heard this all before in programming classes…
    … but read it to make sure you have!

Test Case Minimal Checklist

- **Unit tests –test single object stand-alone**
  - ❑ Each Statechart arc exercised
  - ❑ Each Statechart state exercised
  - ❑ Each conditional branch exercised
  - ❑ All inputs & outputs in Behavioral Requirements exercised
- **Multi-module “integration” tests**
  - ❑ Each Sequence Diagram exercised end-to-end
  - ❑ All Behavioral Requirements exercised
  - ❑ All Constraints checked
- **System-level “acceptance” tests**
  - ❑ Various combinations of single and multiple passengers exercised
  - ❑ All Use Cases exercised
  - ❑ All Scenarios exercised
  - ❑ All High-Level Requirements exercised

Example Light-Weight Review Report

	Peer Review Template for Project X	
Date:	4/17/2011	
Artifact:	Xyzzy.cpp Functions: Foo(), Bar(), Baz()	
Reviewers:	Stella K., Joe B., Sam Q., Trish R.	
Size:	357	SLOC
Time Spent:	112	Minutes
# Issues:	3	
Outcome:	Re-Review of Bug Fixes Required	
Issue#	**Issue Description**	**Status**
1	Issue 1…..	Fixed
2	Issue 2….	Bugzilla
3	Issue 3….	Bugzilla
4	Issue 4….	Not a Bug
5		
6		
7		
8		
Status Key:	Fixed (trivial fix by author; no need to enter in defect list)	
	Bugzilla (entered into project defect system)	
	Not a Bug (false alarm)	

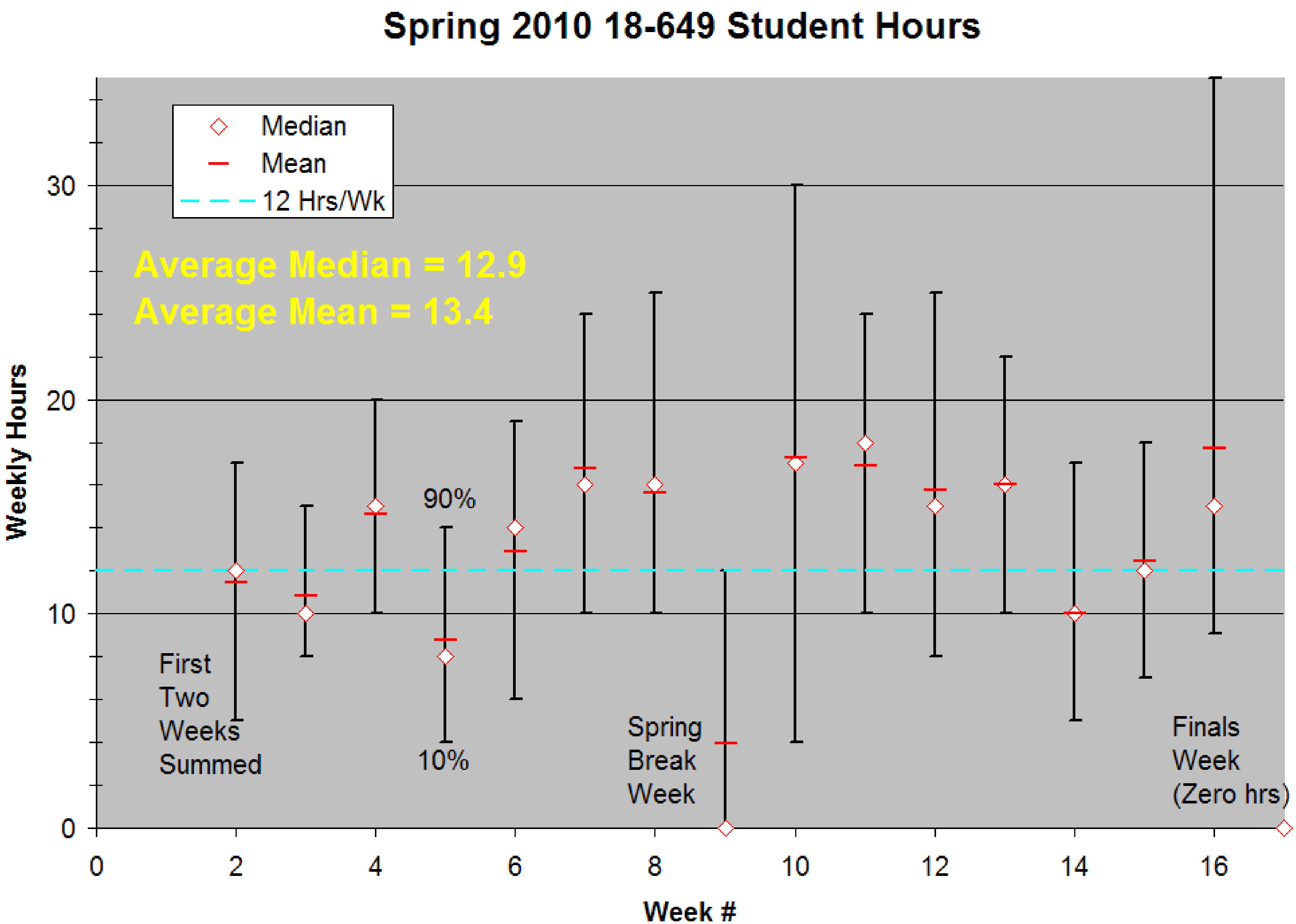
Honesty & Review Reports

- **The point of reviews is to find problems!**
  - If *MOST* reviews that find zero bugs, then we can conclude one or more of:
    - Dishonest
    - Lazy
    - Waste of time
    - … in general broken!

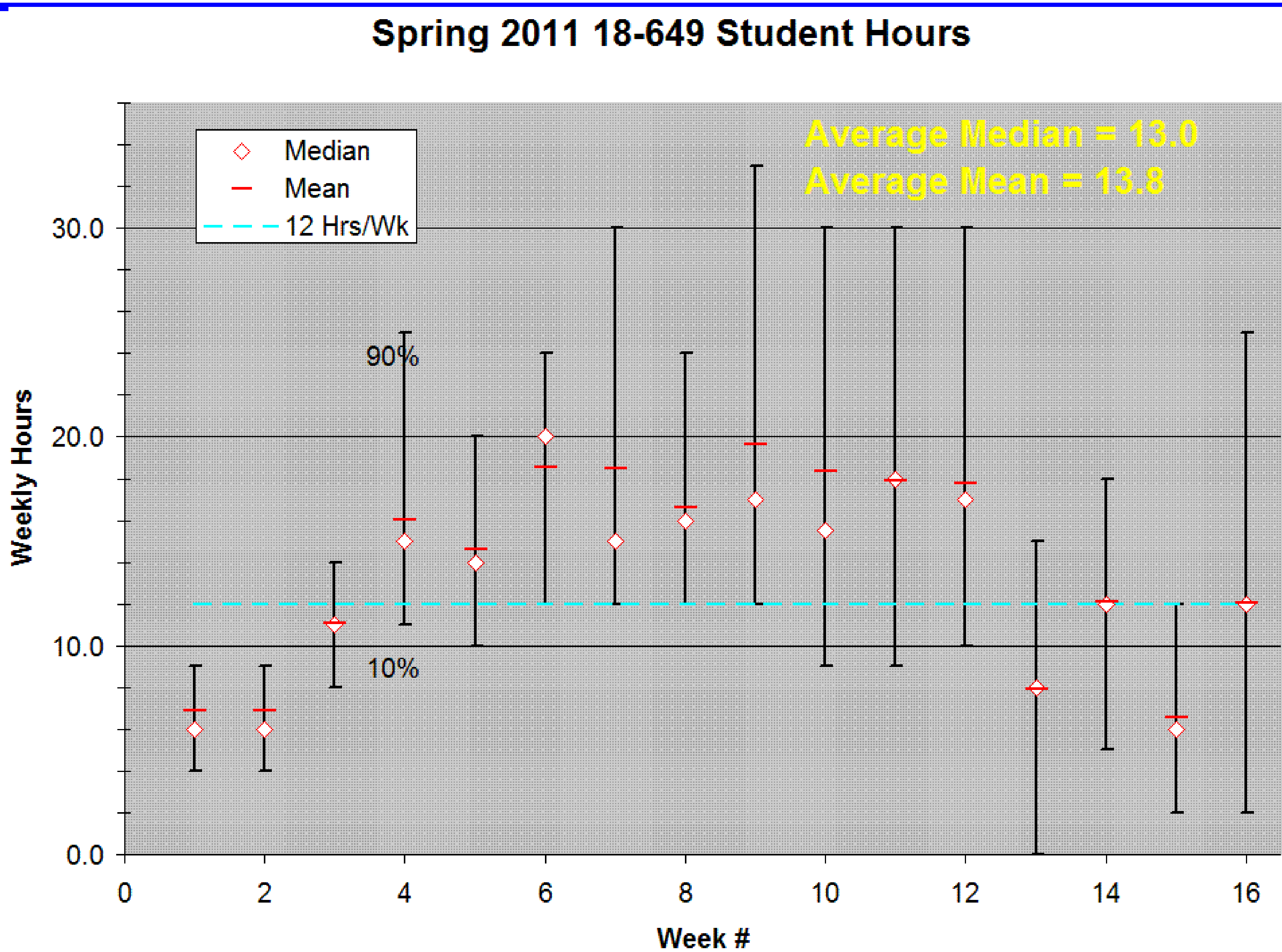
- **Peer reviews should find about HALF the defects**
  - The other half should mostly be found in testing before shipment
  - Peer review range is typically 40%-70% of defects, depending on project
  - If peer reviews find less than 10%, they are BROKEN!

- **Peer reviews take about 10% of project effort** (industry advice below:)
  - That means start scheduling them on first week of project
  - Schedule 2 or 3 per week EVERY week; there is always something to review!
  - If you wait until project end, you won't get effective reviews.

Before Spreadsheet (Generally Ineffective Reviews)



After Spreadsheet & Weekly Defect Reporting



Reviews vs. Inspections Revisited

- **Current data show that inspections are most effective form of review**
  - Informal reviews do not find as many defects overall
  - Informal reviews do not find as many defects per hour of effort
- **Reviews that can be useful, but not a substitute for an inspection:**
  - Coffee room conversations
  - "Hey, take a look at this for me will you?"
  - Sending code around via e-mail
  - External reviews (someone flies in from out of town for a day)
  - Pair programming (extreme programming technique)
- **What about on-line review tools?**
  - For example, Code collaborator – shared work space to do on-line reviews
  - They are probably better than e-mail pass-arounds
  - They may be more convenient than setting up meetings
  - But there is no data to compare effectiveness with inspections
    - Skip inspections at your own risk

The Economics Of Peer Review

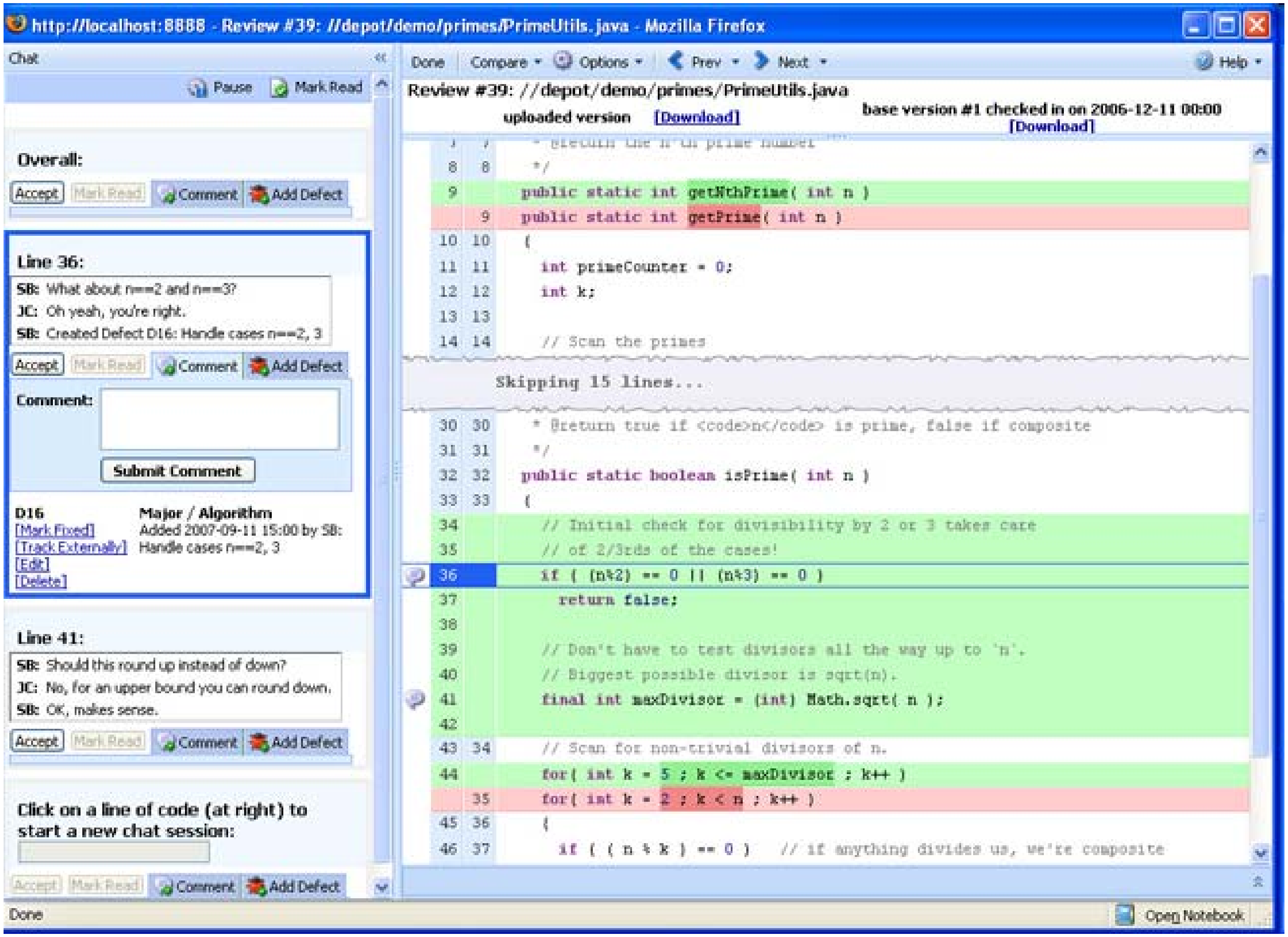
- **Peer reviews are the most cost effective way to find bugs**
  - Automated analysis is only a small part of code reviews
- ***Good*** **embedded coding rate is 1-2 lines of code/person-hr**
  - (Across entire project, including reqts, test, etc.)
  - A person can *review 50-100 times faster* than they can write code
    - If you have 4 people reviewing, that is still more than 10 times faster than writing!
- **How much does peer review cost?**
  - 4 people * 100-200 lines of code reviewed per hour
  - Say 300 lines; 4 people; 2 hrs review + 1 hr prep
    = 25 lines of code reviewed / person-hr
  - Reviews are only about **5%-10% of your project cost**
- **Good peer reviews find about *half the bugs!***
  - *And they find them early, so cost to fix is lower*
- ***Why is it most folks say they don't have time to do peer reviews?***

Peer Review Support Tools

- **There can be value in using these tools…**
  - … but it is important to have realistic expectations
  - The general idea is it provides a virtual, off-line code review meeting
    - Perhaps via using social media style interactions
- **The usual sales pitch**
  - Software developers won't do reviews if they are painful
  - So use tools to make them less painful
    - Off-line instead of in meetings to increase schedule flexibility
    - Easier for geographically dispersed teams (especially if senior/junior folks are split into different sites)
  - May help automate reporting for defect metrics
- **The unspoken parts of the sales pitch**
  - Developers would rather talk to software than to people
  - Developers would *especially* rather not receive criticism in person
  - Implied: if you do peer reviews they will automatically be effective(?)
    - Geographically dispersed tool-based reviews will be *just as effective* as in-person(?)
    - Through the *magic of tools*, you will find *just as many* bugs at lower cost(?)

Code Collaborator (there are other tools too!)

- **SmartBear: $489 named license; $1299 floating license (per seat)**
  - Example webinar: "Can I Really Roll-out a Code Review Process That Doesn't Suck?"
  - Free e-book: smartbear.com/SmartBear/media/pdfs/best-kept-secrets-of-peer-code-review.pdf



Combining Tools With In-Person Reviews

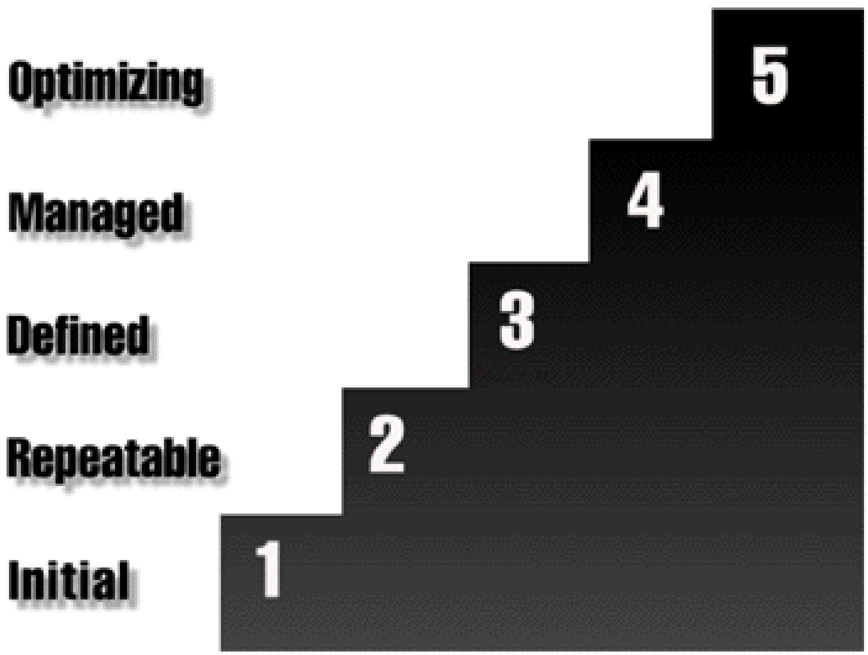
- **Run a static analysis tool before doing peer reviews**
  - Don't waste reviewer time on defects a tool can catch
    - Deciding which warnings you really want to fix can take time
- **Skipping review meetings has serious opportunity costs**
  - **Training:** Exposing newer/younger developers to experienced developers
  - **Synergy:** when one reviewer says "this looks odd" – prompts another to find a bug
  - **Focus**: a meeting forces focus, appropriate pace, and not waiting until the last minute
  - **Consistency:** especially in terms of how deep the review goes
  - **Pride:** people are often more civil in person, and are less able to ignore criticism
- **Peer review defect metrics are essential**
  - Without metrics, you can't know if reviews are broken or working
  - In my experience, reporting number of bugs found *dramatically* increases review quality
- **My suggestions for industry code reviews:**
  - Use tools *in conjunction with* a meeting-based process; for example:
    - Use peer review tool to make comments for pre-meeting review
    - Use tool in review to show comments and record later comments
    - Use tool for author to record fixes
  - Make sure tool you pick supports metrics you want to keep
    - For example, fraction of code base that has been peer reviewed

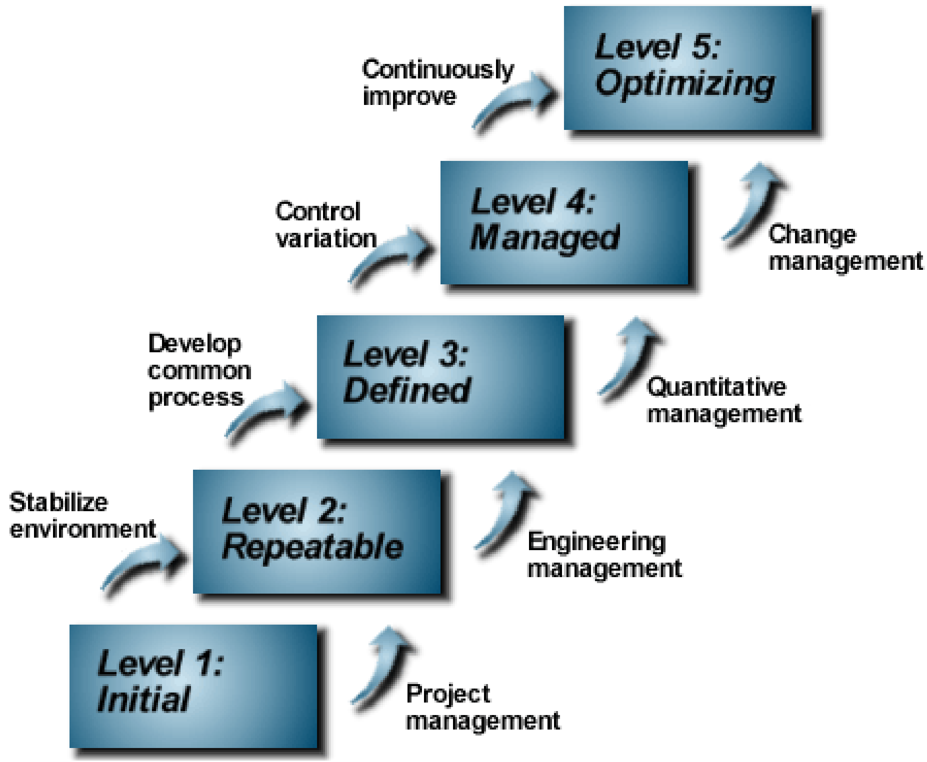
Tool Summary Thoughts

- **If your developers won't do the reviews, then reviews won't find bugs**
  - They need to be motivated
  - They need to eventually be convinced that their life is better with them
  - If a tool helps motivate them, then that's great!
    - If a static analysis tool helps with some of the work, that's great too!
- **If you do ineffective reviews, you're wasting your time**
  - Ineffective reviews might only find 5%-10% of bugs, not 50%
  - Good reviews require process feedback
    - If you aren't finding 50% of you bugs in review, your reviews are broken!
  - Good reviews require training
    - How many of your developers have great skills at running a potentially confrontational meeting?
  - Do you think tool-only reviews will be effective? ("Show me the data!")
- **Combine peer review tools with in-person meetings**
  - It is unrealistic to do 100% review on an entire existing code base instantly
  - Implement regular review meetings and run code through reviews every week
  - It will take time to make your code friendly to static analysis tools

CMM – Capability Maturity Model

- **Five levels of increasing process maturity**
  - Extensive checklist of activities/processes for each level
  - Primarily designed for large-scale software activities
    - Must be tailored to be reasonable for small embedded system projects
  - Growing into a family of models: software/systems/people/…



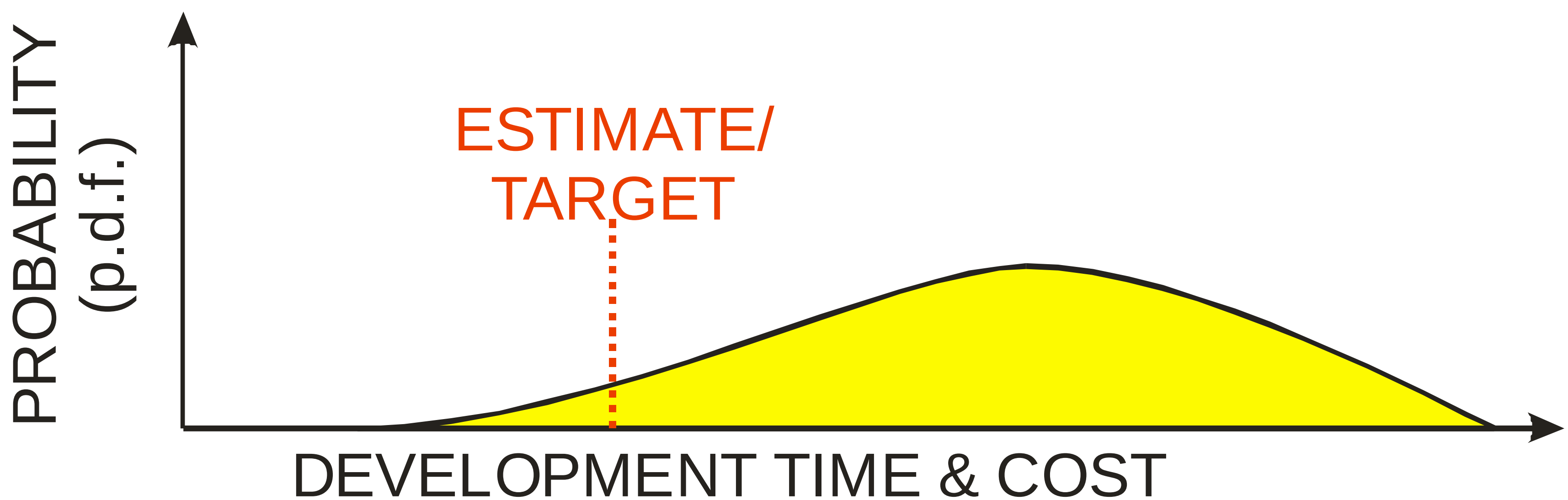


[SEI]

- **[CMM Level 0: "What's the CMM?"]**

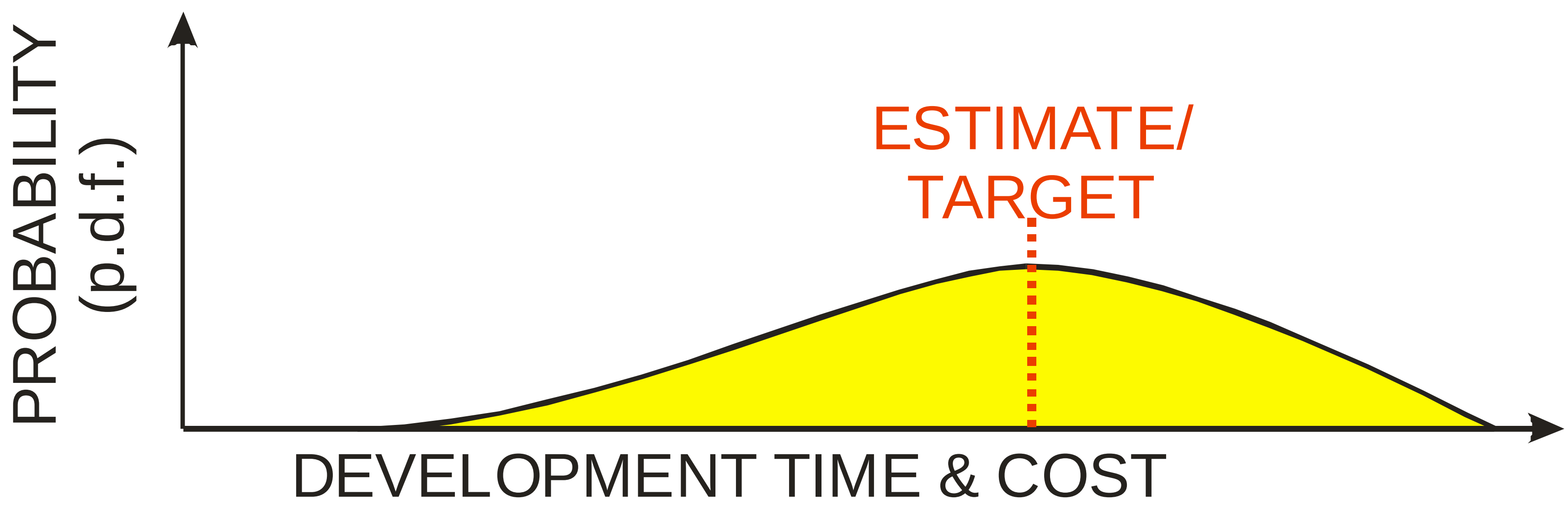
CMM Level 1: Initial

- **CMM Level 1: Initial**
  The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.
  - *Process is informal and unpredictable*
  - Intuitively: You have little idea what's going on with software process



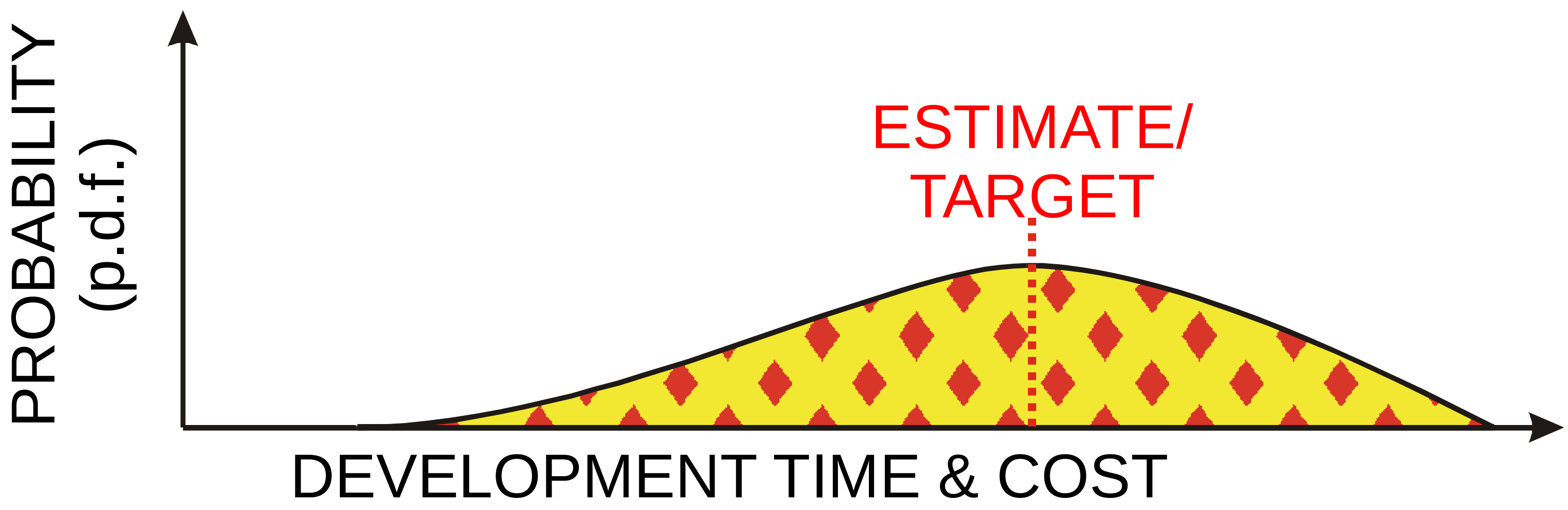
CMM Level 2: Repeatable

- **Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.**
  - Requirements Management, Software Project Planning, Software Project Tracking and Oversight, Software, Subcontract Management, Software Quality Assurance, and Software Configuration Management.
  - *Project management system is in place; performance is repeatable*
  - Intuitively: You know mean productivity



CMM Level 3 - Defined

- **The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.**
  - Organization Process Focus, Organization Process Definition, Training Program, Integrated Software Management, Software, Product Engineering, Intergroup Coordination, and Peer Reviews
  - *Software engineering and management processes are defined and integrated*
  - Intuitively: You know standard deviation of productivity

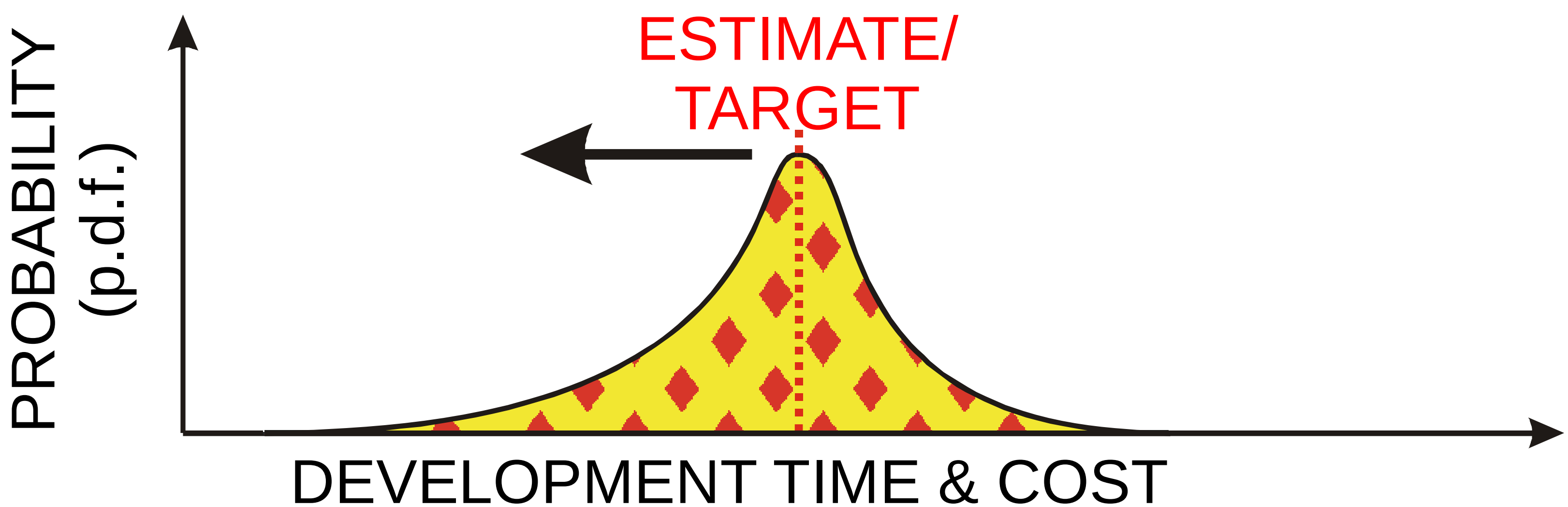


CMM Level 4: Managed

- **Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.**
  - Quantitative Process Management and Software Quality Management
  - *Product and process are quantitatively controlled*
  - Intuitively: You can improve the standard deviation of productivity

CMM Level 5: Optimizing

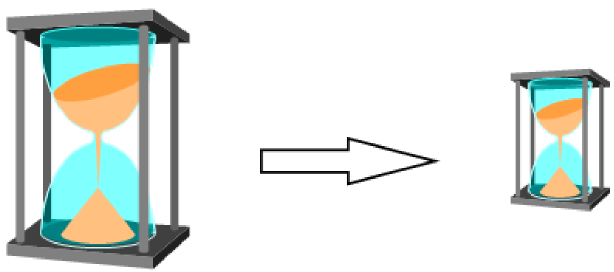
- **Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.**
  - Quantitative Process Management and Software Quality Management.
  - *Process improvement is institutionalized*
  - Intuitively: You can consistently improve mean productivity



Quantified Software Gains From Following SEI's Model

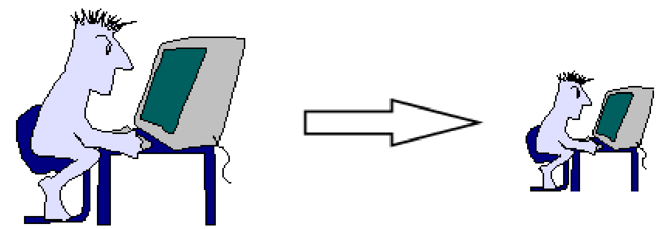
EMERSON

Time-to-Market



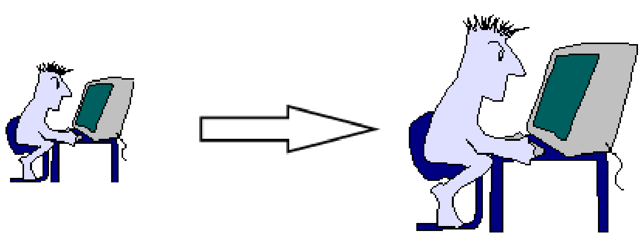
Decreased 15% to 23% per year

Errors Reported Post Release



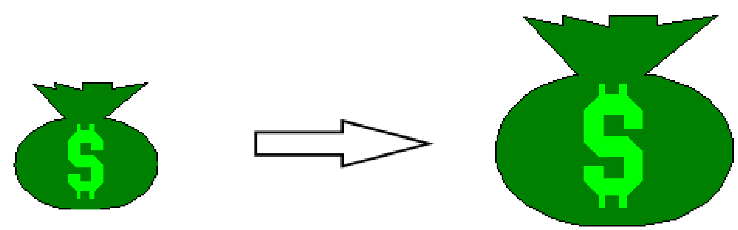
Reduced 10% to 94% per year

Defects Found Prior to Testing



Increased 6% to 25% per year

ROI (savings/cost)



Ranged from 4.0X to 8.8X

Participating Companies: Siemens, Motorola, Lockheed-Martin, Schlumberger, TI, H-P, Northrop, Loral, Hughes, GTE, Bull HN

Slide courtesy of Bill Trosky
Emerson Software Center of Excellence

CMM In Perspective

- **High-quality products can be, and have been, developed by Level 1 organizations.**
  - But, often they rely on personal heroics
  - It is hard to predict how the same organization will do on the next job
- **CMM is complex, hindering use by small projects**
  - 5 levels; 18 key areas; 52 goals; 316 key practices; 2.9 pounds printed
  - CMMI (new version including systems engineering etc.; V1.02 staged):
    5 levels; 24 key areas; 78 goals; 618 key practices; 4.7 pounds printed
- **The CMM itself does not specify a particular process**
  - The CMM does not mandate how the software process should be implemented; it describes what characteristics the software process should have.
  - Ultimately, an organization must recognize that continual improvement (and continual change) are necessary to survive.
- **Good process enables, but does not ensure, good product**
  - Embedded system companies usually find that investing in CMM(I) level 3 pays off
  - Even for very small development teams

What Is Software Quality Assurance?

- **Regular QA for manufacturing involves:**
  - Measuring the manufacturing process to make sure it works properly
  - Sample the manufactured product to see if it meets specifications
  - Often synonymous with product testing
- **But, software design isn't manufacturing**
  - Every new software module is different; it is not the same item every time
  - SQA goes beyond testing; it is monitoring adherence to the process
- **Software Quality Assurance (SQA) – about 5%-6% of project cost**
  - Monitor how well the developers follow their software process
    - Do they follow the steps they are supposed to follow?
    - Is there a written paper trail of every step of the process?
    - Are metrics (e.g., defects per hour found in inspections) within range?
  - Should be perhaps 3% of total software effort
  - Must be an independent person (an auditor) to be effective
    - Includes external audits (these in a sense audit the SQA department)
    - In our course project, the grading TAs are the "SQA department"
  - Another 3% of total software effort for process creation and training
  - Do the math: in general person #20 added to a software team ➔ SQA specialist

Most Effective Practices For Embedded Software Quality

Ebert & Jones, "Embedded Software: Facts, Figures, and Future, IEEE Computer, April 2009, pp. 42-52

Ranked by defect removal effectiveness in percent defects removed.

"*" means exceptionally productive technique (more than 750+ function points/month)

- * 87% static code analysis ("lint" tools, removing compiler warnings)
- 85% design inspection
- 85% code inspection
- 82% Quality Function Deployment (requirements analysis used by auto makers)
- 80% test plan inspection
- 78% test script inspection
- * 77% document review (documents that aren't code, design, test plans)
- 75% pair programming (review aspect)
- 70% bug repair inspection
- * 65% usability testing
- 50% subroutine testing
- * 45% SQA (Software Quality Assurance) review
- * 40% acceptance testing

Industry Good Practices For Better Software Systems

- **Use a defined process**
  - Embedded companies often say CMM(I) level 3 is the sweet spot
    - Benefit to higher levels, but higher variance due to contractors gaming the system
  - Use traceability to avoid stupid mistakes
  - Have SQA to ensure you follow that process
    - SQA answers the question: how do we know we are following the process?

- **Review/inspect early and often – all documents, plans, code, etc.**
  - Formal inspections usually are more effective than informal review
  - Testing should catch things missed in review, *not* be the *only* defect detector

- **Use abstraction**
  - Create and maintain a good architecture
  - Create and maintain a good design
  - ***Self-documenting code isn't***
    - You need a distinct design beyond the code itself
    - Good code comments help in understanding *implementation*, but that isn't a design

Process Pitfalls [Brenner00]

Our project was late,
so we added more people.
The problem got worse

We can't get it right
and still come in on schedule.
Why can't we do both?

When requirements changed,
the schedule did not.
Were we in trouble?

There is no more time,
but the work is unfinished.
Take more time from Test.

I gave estimates.
They cut all of them in half.
Next time I'll pad them.

If a project fails,
but we keep working on it,
has it really failed?