

# 7

# Reviews

# & Software Process

**Distributed Embedded Systems**

**Philip Koopman**

**February 2, 2011**

© Copyright 2010, Philip Koopman

**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 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)

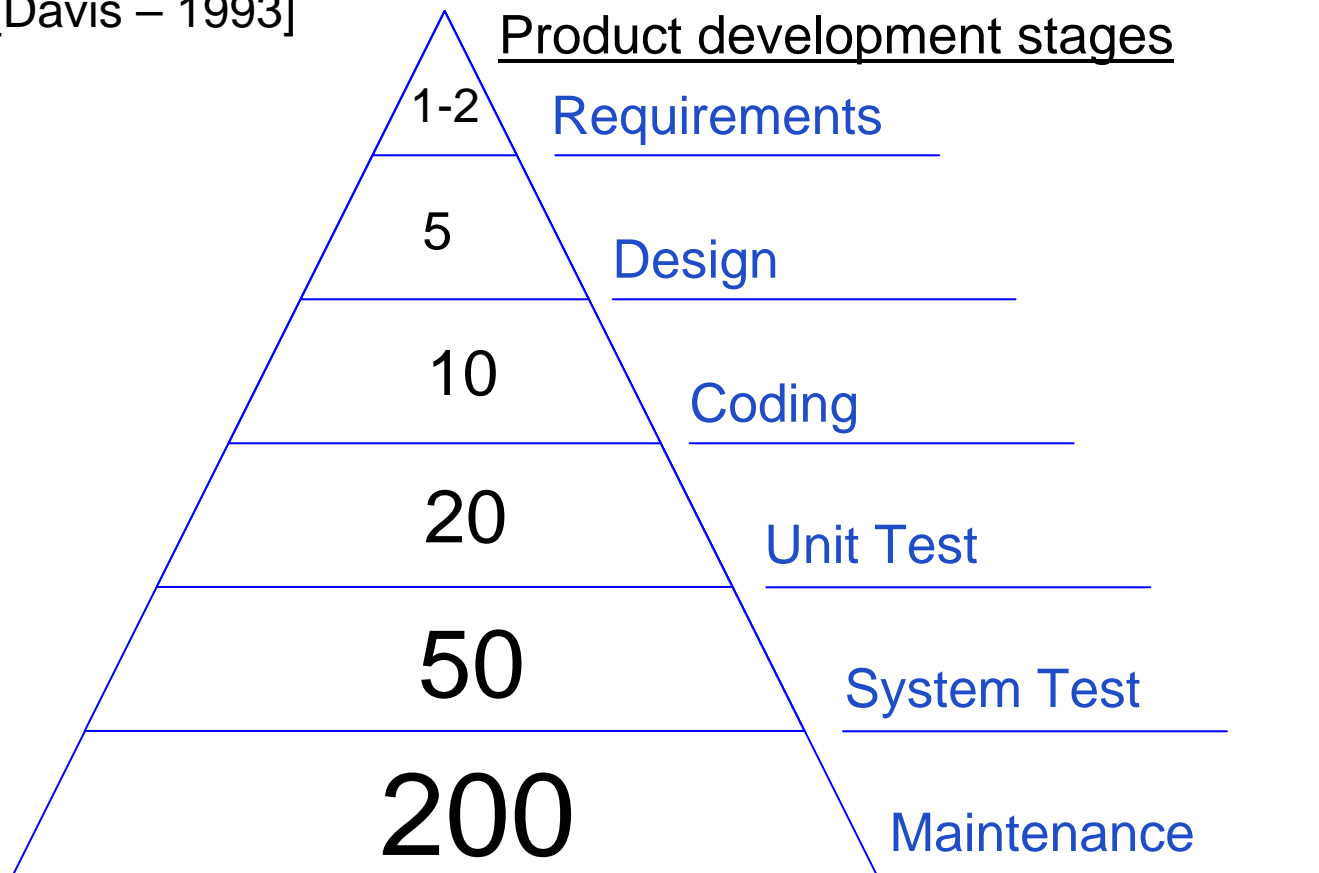
# Early Detection and Removal of Defects -

---

Peer Reviews - remove defects early and efficiently

## Relative Cost to Fix Requirements Errors

[Davis – 1993]



# Boehm's Top 10 List On Software Improvement

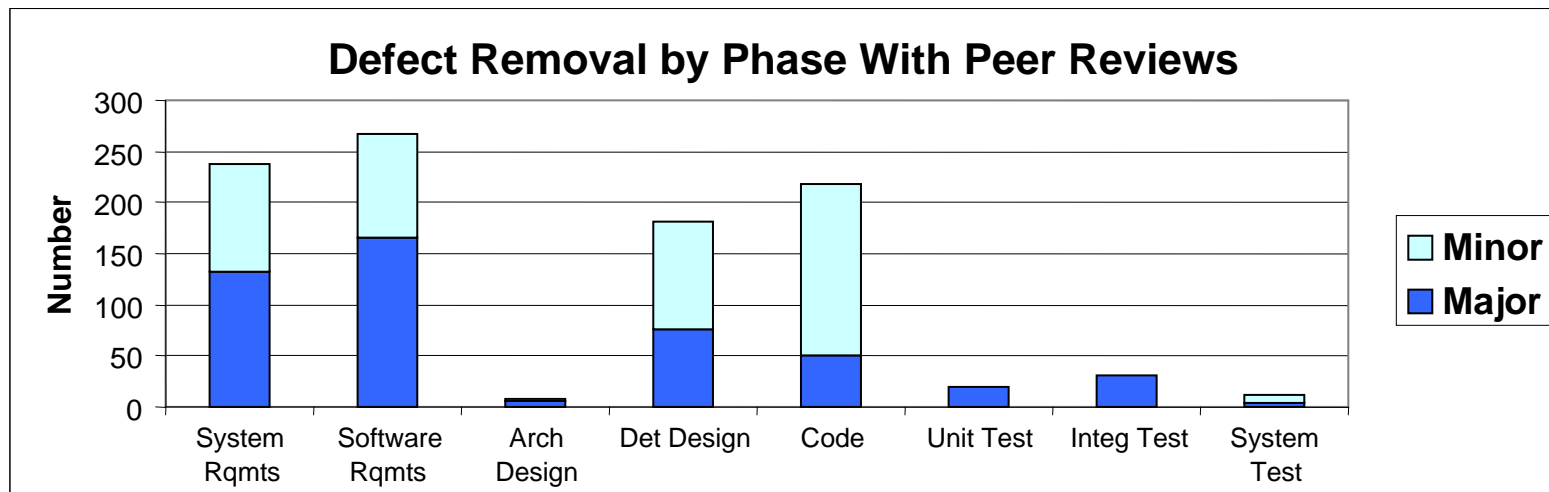
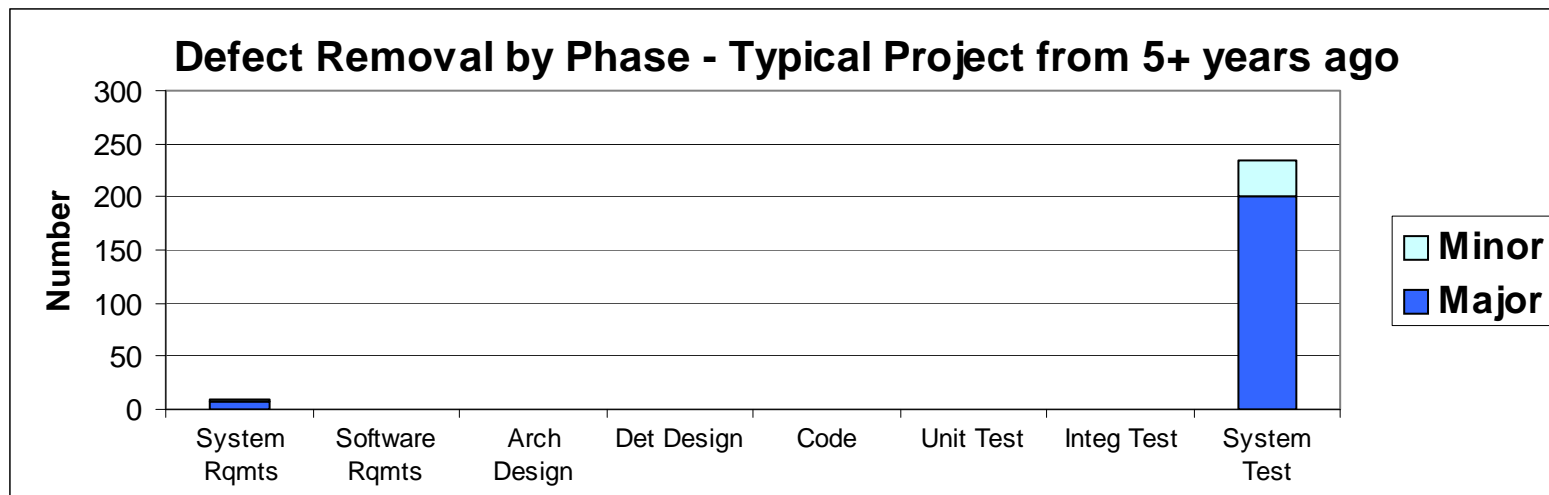
---

- 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?

# Defect Management – Then vs. Now

(Real data from embedded industry)

## Defects are removed earlier



[Source: Roger G., Aug. 2005]

# What Can You Review?

---

- ◆ *Anything written down on paper!*
  - Code reviews are just a starting point
- ◆ **Examples of things that can and should be reviewed:**
  - Requirements
    - Catching problems here can save huge amounts of time/money later
  - Design
  - Test plans
  - Test results
  - Implementation
  - Process plans
  - ...

# Why Not Review Your Own Code?

---

- ◆ **Sometime it's not easy to see your own mistakes!**
  - We get involved in the details and forget the broader implications of our code
  - We are so familiar with our code that we just don't see things
  - We can be too forgiving of ourselves when it comes to breaking the rules for documentation and style guidelines
- ◆ **It's easy to leave unnecessary (and possibly detrimental) “crumbs” as we re-design & debug our code**
  - It is difficult to find motivation to redesign if the hacked-up result “works”
- ◆ **Others remain uninformed about what we've done**
  - Peer review helps others see beyond their own part of the project

# 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



# Inspection Reporting - 2

<b>Report Summary Form</b>						
<b>Defect Types</b>	<b>Major Defects</b>			<b>Minor Defects</b>		
	<b>Missing</b>	<b>Wrong</b>	<b>Extra</b>	<b>Missing</b>	<b>Wrong</b>	<b>Extra</b>
<b>Interface</b>						
<b>Data</b>						
<b>Logic</b>						
<b>I/O</b>						
<b>Performance</b>						
<b>Functionality</b>						
<b>Human Factors</b>						
<b>Standards</b>						
<b>Documentation</b>						
<b>Syntax</b>						
<b>Maintainability</b>						
<b>Other</b>						

"Peer Reviews", *Encyclopedia of Software Engineering*, D. O'Neill.

# Check Lists

## DESIGN AND CODE CHECKLIST

### COMPLETION

1. Has traceability been assessed?
2. Has suitable test evidence been applied in assessing traceability?
3. Have all predecessor requirements been accounted for?
4. Have any product fragments provided not to have traceability to the predecessor requirements?
5. What is the relationship of requirements to product component:
  - 5.1 Can to one single analysis?
  - 5.2 Can to one straightforward?
  - 5.3 Can to many complex?

### CONNECTIONS

1. Are structured programming prime constructs used correctly:
  - 1.1 Sequence: Is the function commentary satisfied for the sequence?
  - 1.2 If-then: If the test is true, is the function commentary satisfied?
  - 1.3 If-then-else: If the test is true, is the function commentary satisfied?
  - 1.4 If the test is false, is the function commentary satisfied?
  - 1.5 While-loop: If the condition is true, is the function commentary satisfied? Does the loop terminate?
  - 1.6 Loop-until: If the condition is false, is the function commentary satisfied? Does the loop terminate? Is a one time loop acceptable?
  - 1.7 For-do: Is the function commentary satisfied? Are there discrete steps through the loop? Is the control variable not modified in the loop? Is the loop inhibited and terminated properly?
  - 1.8 Case: For each leg, is the function commentary satisfied? Is the domain partitioned exhaustively and exhaustively?
2. Are proper programs composed of multiple prime programs linked to single entry and single exit?
3. Are disciplined data structures used to manipulate and transform data?
4. Does the input domain span all legal values?
5. Does the input domain span all possible values, with systematic exception handling for illegal values?
6. Does the output range span all legal values?
7. For modules, does the state data span all legal values?

### STYLE

1. Are style conventions for block structuring defined and followed?
2. Are naming conventions defined and followed?
3. Are the semantics of the product component traceable to the requirements?
4. Are style conventions for commentary defined and followed?
5. Are style conventions for alignment, upper/lower case, and highlighting defined and followed?
6. Are templates used for repeating patterns?

### RULES OF CONSTRUCTION

1. Are guidelines for program unit construction followed?
2. Is the interprocess communication protocol followed?
3. Are data representation conventions followed?
4. Is the system standard time defined and followed?
5. Are encapsulation, localisation, and layering used to achieve object abstraction?
6. Is logical independence achieved through event driven and process driven paradigms, late binding, and implicit binding?
7. Is visibility achieved through uniformity, parameterisation, and portability?
8. Are fault tolerance, high availability, and security achieved?

### MULTIPLE VIEWS

1. Has the logical view of user interfaces and object abstraction considerations been assessed?
2. Has the static view of packaging considerations been assessed including program unit construction, program generation process, and target machine operations?
3. Has the dynamic view of operational considerations been assessed including communications, concurrency, synchronization, and failure recovery?
4. Has the physical view of execution considerations been assessed including timing, memory use, input and output, initialization, and finite word effects?
5. Has the product component been assessed for safety considerations?
6. Has the product component been assessed for Year 2000 compliance?
7. Has the product component been assessed for security considerations?
8. Has the product component been assessed for open system considerations?

"Peer Reviews",  
*Encyclopedia of  
Software Engineering*,  
D. O'Neill.

# Using A Checklist

---

- ◆ **Often you will want to develop your own checklist**
  - Incorporate lessons learned from defects that escape detection
  - Focus on things that matter to your company / your product / your people.
  
- ◆ **But, 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
  
- ◆ **Following slides are starting checklists for the course project**
  - Be sure to look at the grading rubrics 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 descript
- 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

# Reviews vs. Inspections Revisited

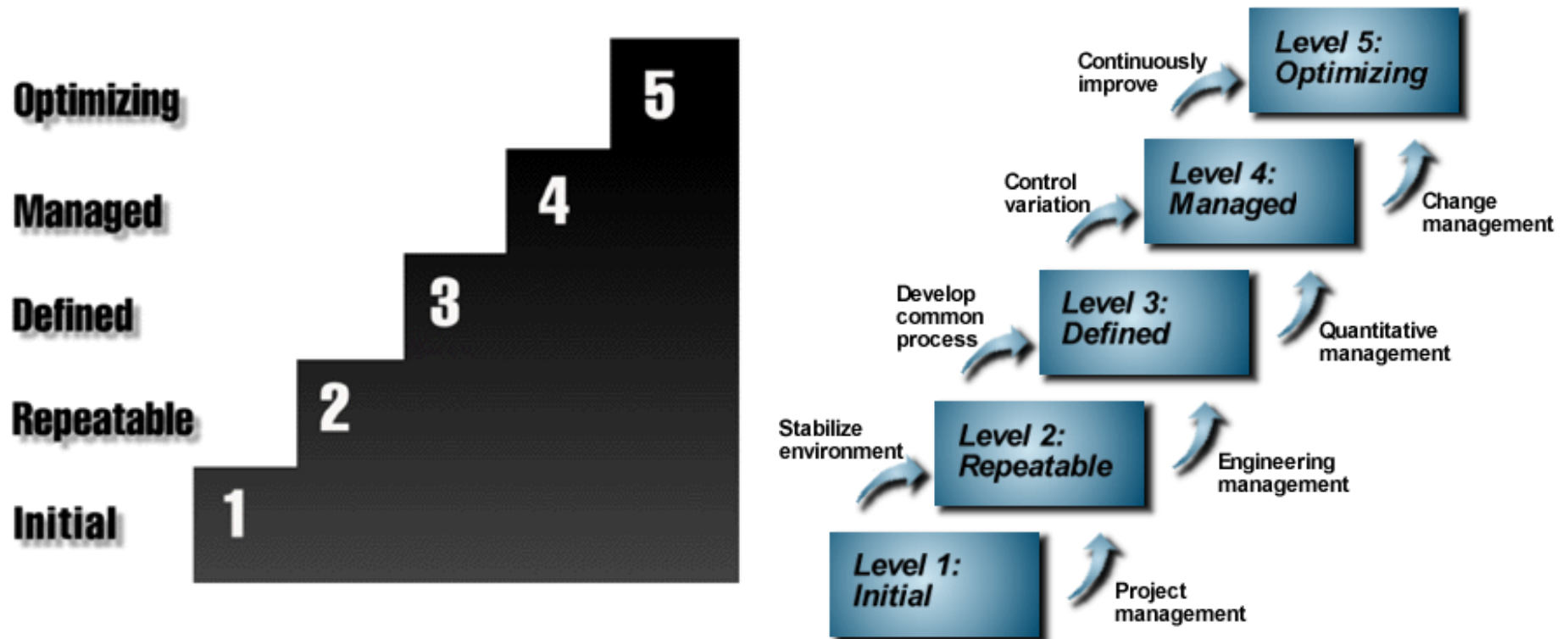
---

- ◆ **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

# 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/...



## ◆ [CMM Level 0: “What’s the CMM?”]

[SEI]

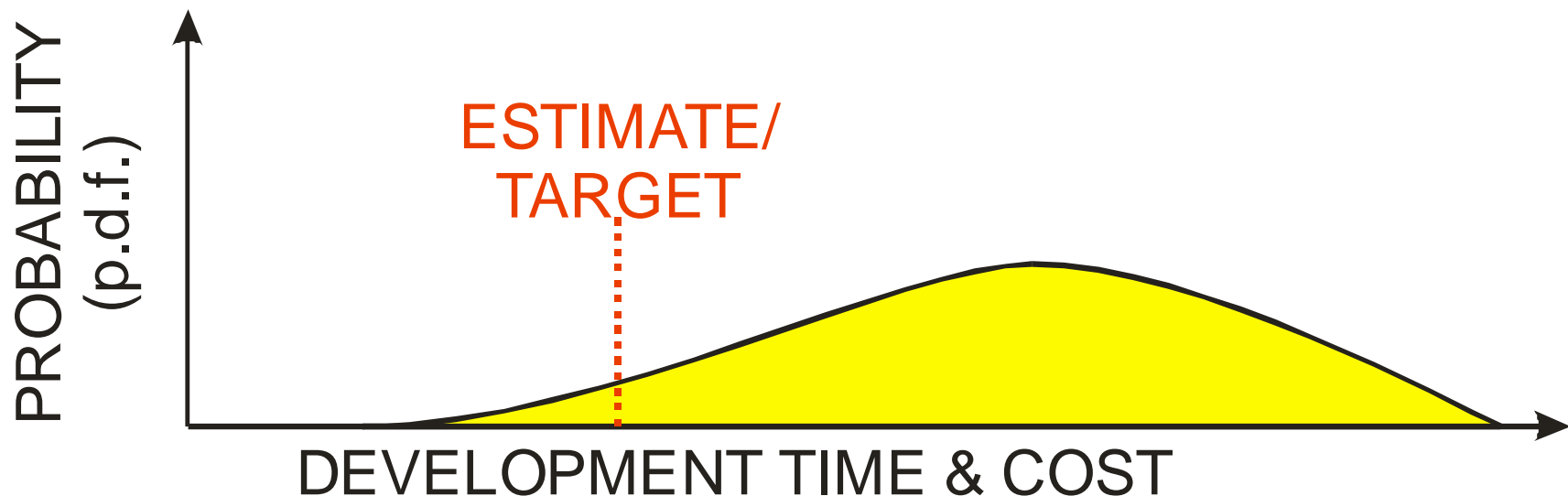
# CMM Level 1: Initial

---

## ◆ 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

---

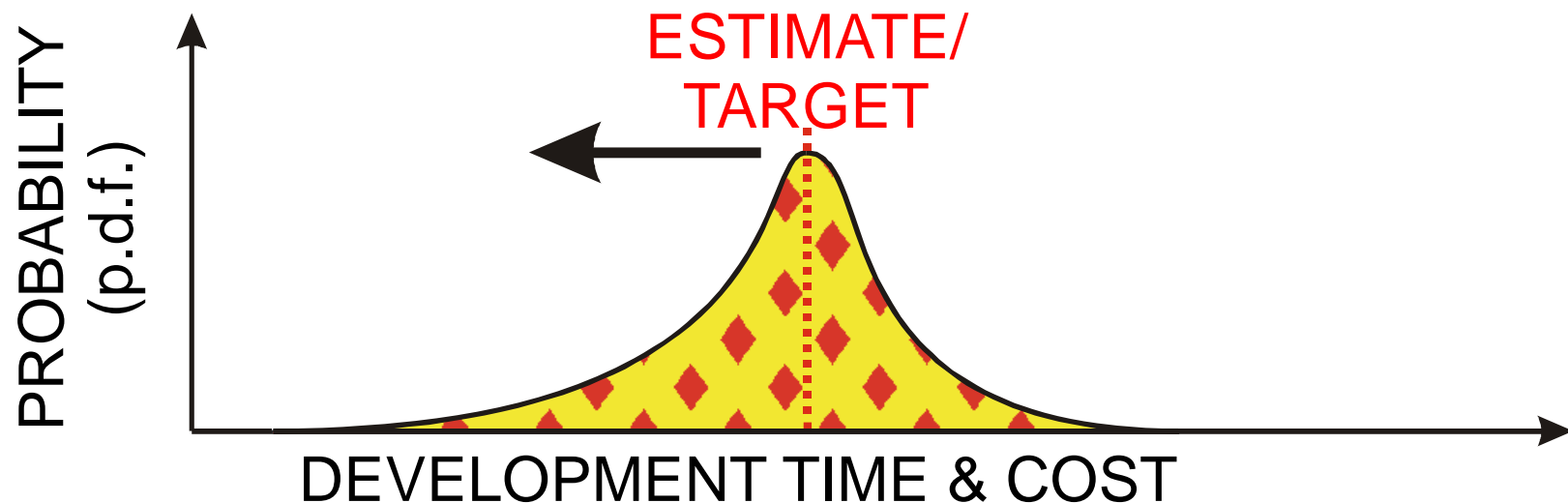
- ◆ Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.
  - Quantitative Process Management and Software Quality Management
  - *Product and process are quantitatively controlled*
  - Intuitively: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



# CMM Level 5: Optimizing

---

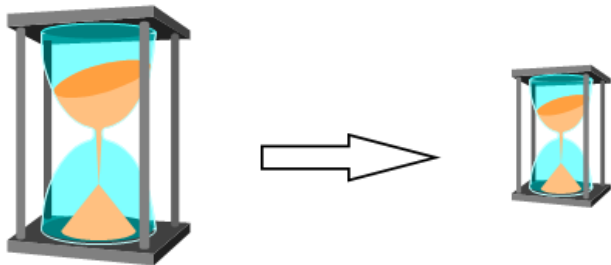
- ◆ Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.
  - Quantitative Process Management and Software Quality Management.
  - *Process improvement is institutionalized*
  - Intuitively: You can consistently improve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



# Quantified Software Gains From Following SEI's Model

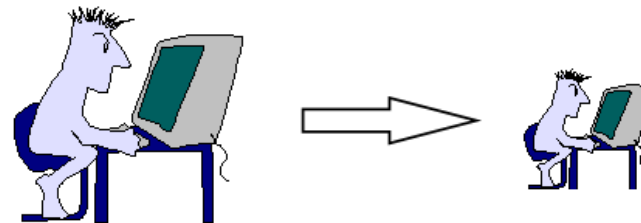


## Time-to-Market



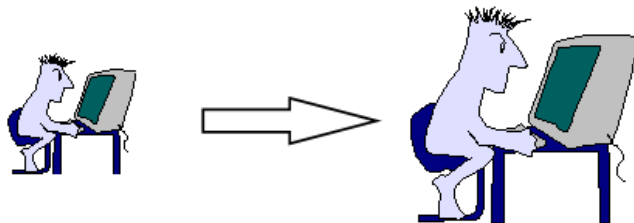
Decreased 15% to 23% per year

## Errors Reported Post Release



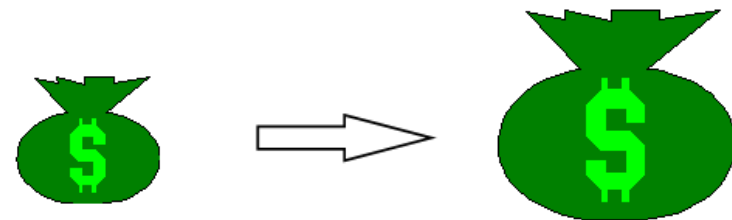
Reduced 10% to 94% per year

## 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**)

- 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”

# 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
- 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?**