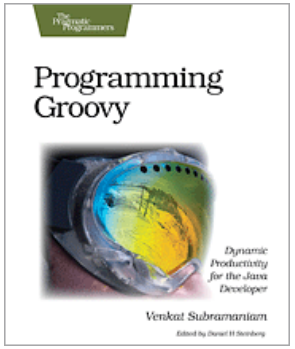
CARING ABOUT CODE QUALITY

```
speaker.identity {
  name        'Venkat Subramaniam'
  company     'Agile Developer, Inc.'
  credentials 'Programmer', 'Author', 'Trainer'
  blog        'http://agiledeveloper.com/blog'
  email       'venkats@agiledeveloper.com'
}
```



Abstract

- We all have seen our share of bad code. We certainly have come across some good code as well. What are the characteristics of good code? How can we identify those? What practices can promote us to write and maintain more of those good quality code. This presentation will focus on this topic that has a major impact on our ability to be agile and succeed.
- Characteristics of quality code
- Metrics to measure quality
- Ways to identify and build quality

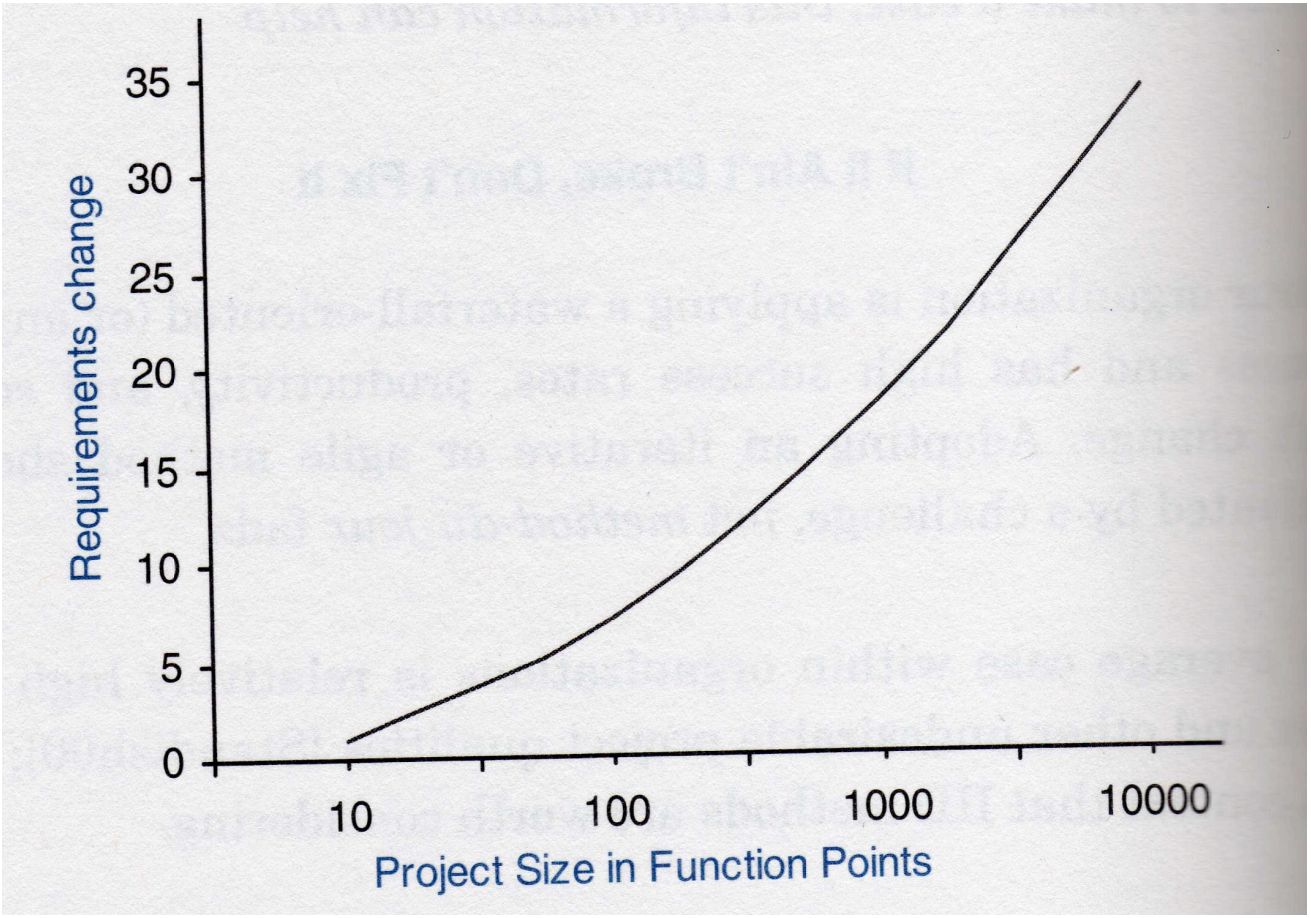
Why care about Code Quality?

You can't be Agile if your Code sucks

Code Quality

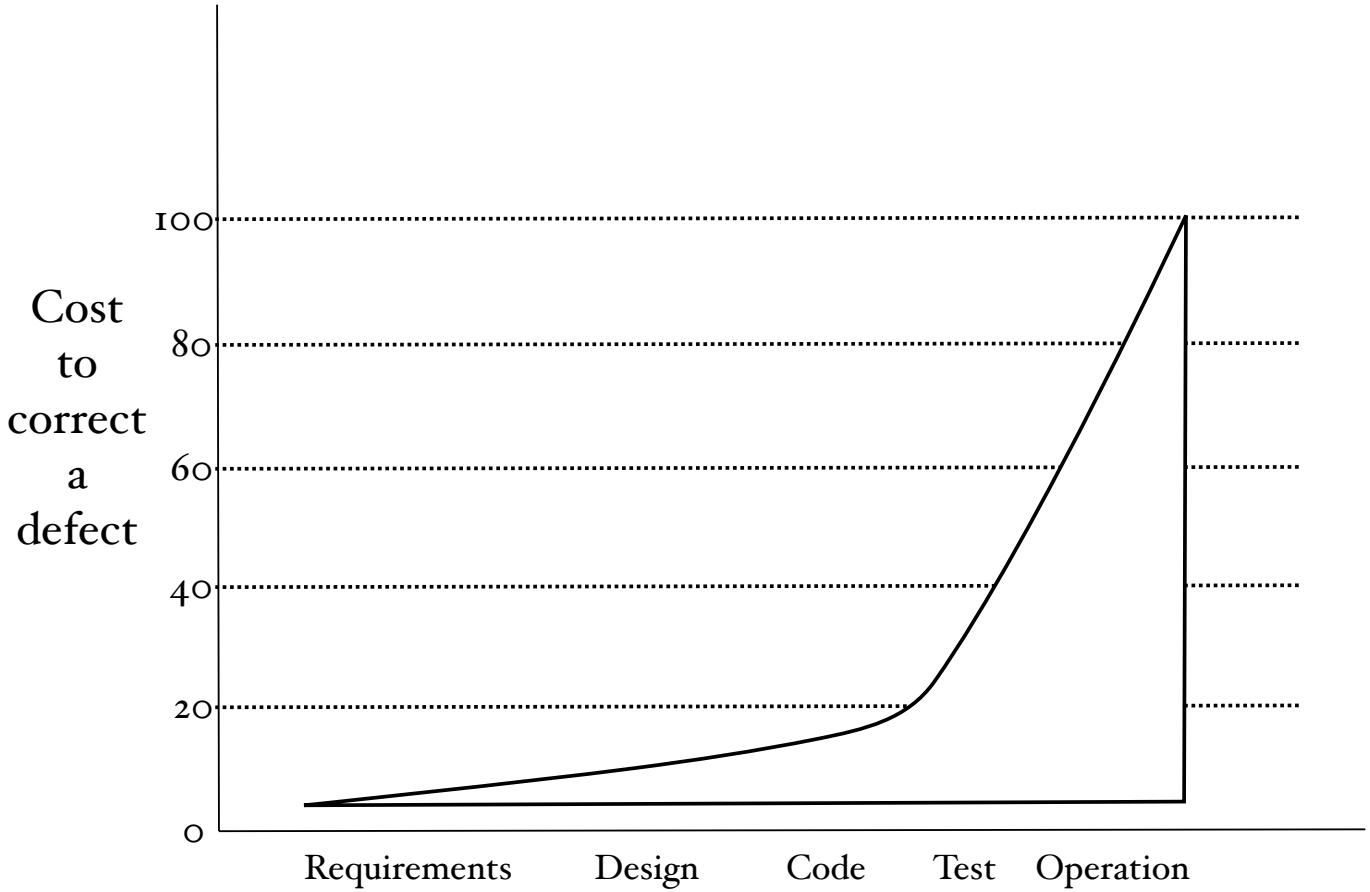
Programs must be written for people to read,
and only incidentally for machines to execute.
-Abelson and Sussman

Change in Requirements



[LARM03]

Cost of Defect



[BOEH01]

Magnitude of Computational Problem

- European Space Agency took 10 years and $8 billion dollars to develop Ariane 5
- On June 4, 1996, it took its first voyage with $500 million cargo
- In 40 seconds its inertial reference system failed
- 64-bit floating point number representing the horizontal velocity of the rocket was converted into 16-bit signed integer—conversion failed because of overflow
- Vehicle was deliberately destroyed for safety

http://www.cnn.com/WORLD/9606/04/rocket.explode/

Major Misfires

- September 1999: Metric mishap causes loss of NASA orbiter [CNN99]
- March 2001: Nike i2 Forecast System found to be inaccurate–Nike takes inventory write-off [CIO03]
- August 2004: NASA: DOS Glitch Nearly Killed Mars Rover–Story on Spirit: "...The flaw, since fixed, was only discovered after days of agonizingly slow tests..." [EXTR04]
- June 2007: United flights grounded by computer glitch [COMP07]
- ...

Software Defect Reduction Top 10 List

- Finding, fixing problem in production is 100 times more expensive than during requirements/design phase.
- 40-50% of effort on projects is on avoidable rework.
- ~80% of avoidable rework comes from 20% of defects.
- ~80% of defects come from 20% of modules; about half the modules are defect free.
- ~90% of downtime comes from at most 10% of defects.

[BOEH01]

Software Defect Reduction Top 10 List

- Peer reviews catch 60% of defects.
- Perspective-based reviews catch 35% more defects than nondirected reviews.
- Disciplined personal practices can ***reduce defect introduction rates*** by up to 75%.
- ...it costs 50% more per source instruction to develop high-dependability software product...
- ~40-50% of user programs have nontrivial defects.

[BOEH01]

But, Still...

- The evidence is overwhelming, but still...
- We never seem to have time to do it, but always seem find time to redo it?!

What's Quality?

- Measure of how well the software is designed and implemented
- Quality is subjective

Why care, there's QA?!

- Can't QA take care of quality, why should developers care?
- QA shouldn't care about quality of design and implementation
- They should care about acceptance, performance, usage, and relevance of the application
- Give them a better quality software so they can really focus on that

"Lowering Quality Lengthens Development Time"
—First Law of Programming http://c2.com/cgi/wiki?FirstLawOfProgramming

More Maintenance...

- You'll do more maintenance if quality is better
- Why? It's easier to accommodate change, so you can be flexible and relevant
- "Maintenance is a solution, not a problem"
- "Better methods lead to *more* maintenance, not less"

[GLAS03]

Pay your Technical Debt

- Technical debt are activities (like refactoring, upgrading a library, conforming to some UI or coding standard, ...) that you've left undone
- These will hamper your progress if left undone for a longer time

Measuring Quality

- Hard to measure
- Need to find useful metrics
- Example of wrong metric: Lines-Of-Code (LOC)
  - Is more code better or worse?
  - You can produce more code?
  - You needed that much code for that?

Measuring Quality...

- Highly subjective
- Highly qualitative
- Is the code readable, understandable?
- Is the code verbose?
- Variable/method names that are meaningful
- Simple code that works
- Does it have tests? What's the coverage?

Ways to Improve Quality

- Start early
- Don't Compromise
- Schedule time to lower your technical debt
- Make it work; make it right (right away)
- Requires monitoring and changing behavior
- Be willing to help and be helped
- Devise lightweight non-bureaucratic measures

Individual Efforts

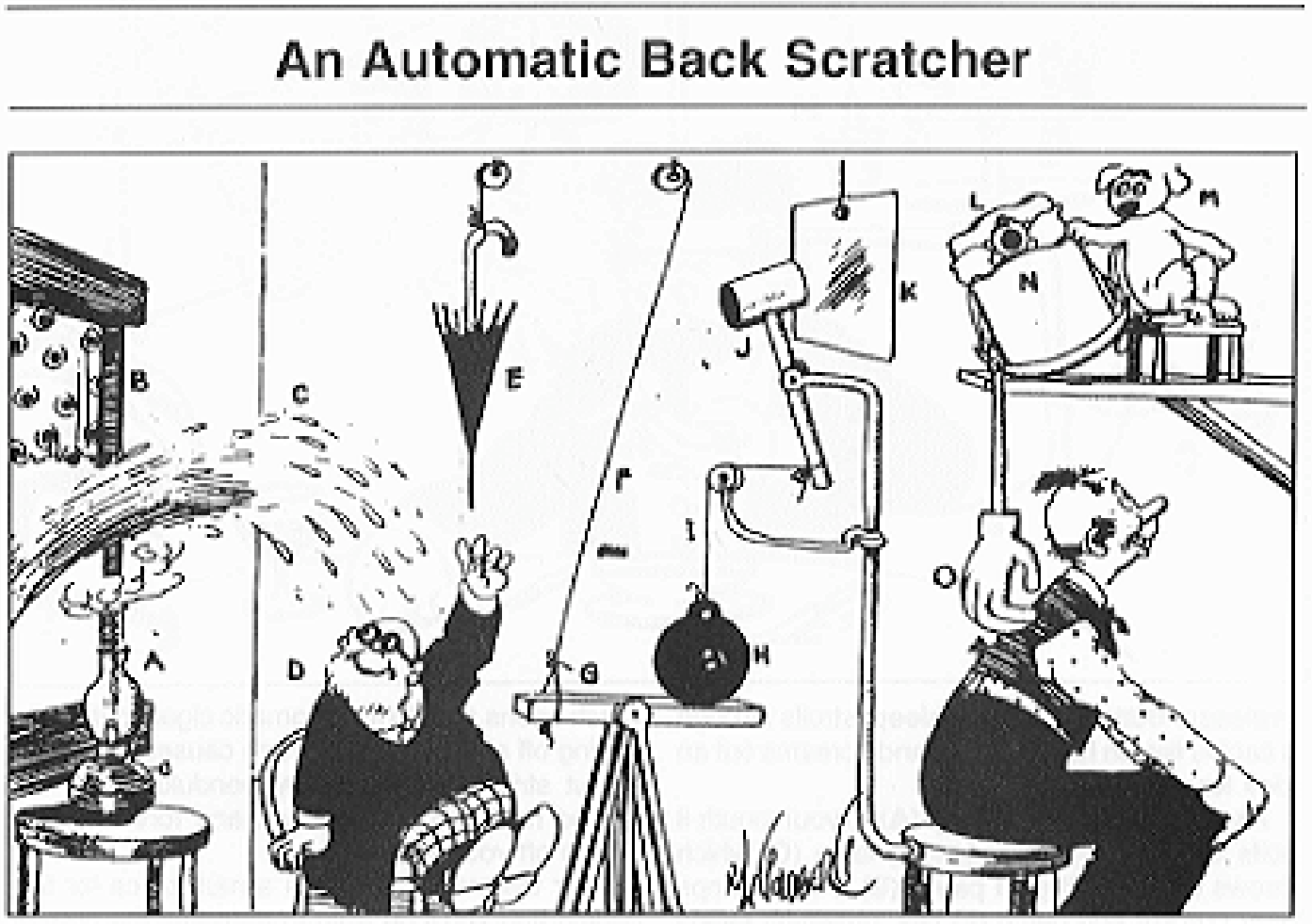
- What can you do?
- Care about design of your code
  - Good names for variables, methods, ...
  - Short methods, smaller classes, ...
  - Learn by reading good code
- Keep it Simple
- Write tests with high coverage
- Run all your tests before checkin
- Checkin Frequently

Individual Efforts

- Learn your language
- If you're switching languages or using multiple languages, know the differences
- Avoid Cargo cult programming–following rituals, styles, principles, or structure that serves no real purpose
- Court feedback and criticism

Keep It Simple!

- Don't build Rube Goldberg Machines–something complex to do simple things



Keep It Simple!

"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies,"–C.A.R. Hoare.

Team Efforts

- Avoid shortcuts
- Take collective ownership–Team should own the code
- Promote positive interaction
- Provide constructive feedback
- Constant code review

You said Code Review?

- Code review is by far the proven way to reduce code defects and improve code quality
- But, code review does not work?
- It depends on how it's done

If code review is...

- Do you get together as a team, project code, and review?
- At least three problems?
  - You hate being critiqued that way
  - You'd much rather write more code in that time
  - Project manager says "last time you guys got together for review, fight ensued and one guy quit, no more code reviews for you..."
- Don't make it an emotionally draining

Seeking and Receiving Feedback

- We've used code review very effectively
- Code reviewed by one developer right after task is complete (or anytime before)
- Rotate reviewer for each review
- Say positive things, what you really like
- Constructively propose changes
- Instead of "that's lousy long method" say, "why don't you split that method..."

Tactical Continuous Review

- Review not only code, but also tests
- Do not get picky on style, instead focus on correctness, readability, design, ...

"Code Review makes me a Hero or makes me Smarter,"– Brian C.

Value of Review

- "Rigorous inspection can remove up to 90 percent of errors before the first test case is run."
- "Reviews are both technical and sociological, and both factors must be accommodated."
- [GLAS03]

Broken Window Problem

- Study shows broken windows lead to vandalism
- Code that no one cares for deteriorates quickly
- Do not tolerate your code being trashed
- Fix code that's not elegant or looks broken
- Keep your code always releasable [SUBR06]

[FIBW, THOM99]

Treat Warning as Errors

- Don't say "that's only a warning"
- Warnings may have hidden problems
- They're annoying and distracting
- Use compiler options to treat warnings as errors
- If unavoidable, suppress (selectively)

Cohesion

- The code is focused, narrow, small
- It does one thing and one thing well
- Single Responsibility Principle
- Higher cohesion -> Lower Cyclomatic Complexity (see later)
- Strive for higher cohesion
  - At method, class, component, subsystem level

Extensibility and Flexibility

- You build abstraction, hierarchy, ... to make your code extensible
- Extensibility is an anticipation
- What if the requirement does not meet what you anticipated?
- You have more code to work with and it is hard to extend in the new direction because of that
- Predicting is hard

Triangulation

- Postpone generalization
- Wait for code to evolve a bit
- You see evidence of what's needed and generalize based on real use
- But, won't that be expensive?
- Changes will require less work since you have lesser code to deal with

[Beck02]

Cohesion and Cost of Change

- Code that's doing too many things is hard to maintain
- Developer who must make change has to understand lots of things
- Code is complex, has higher cyclomatic complexity
- Your change likely will break something else
- Smaller, cohesive code is less expensive to maintain

Code Coverage

- How much (%) of your code is covered by test?
- How about paths through your code
- Is there code that deserve *not* to be tested?
- Instrumentation tools can tell you which and how much code is covered
- Tools: [Java] JCover, Cobertura, ... [.NET] NCover,... [C++] C++ Test Coverage Tool, Bullseye Coverage, CTC++, Visual Studio, ...
- Tools like Guantanamo and Ashcroft delete code that have no test!

Code Coverage

- Code Test Coverage
  - Tells you how much of your code's exercised
  - Does not tell you about test quality, however

Complexity

- Long methods cause pain
- Paths in code
- Unnecessary and stale comments cause confusion
- Large classes are hard to maintain
- ...

Cyclomatic Complexity

- Thomas McCabe's
- Counts distinct paths through code
- Number of decision points + 1
- # of edges - # of nodes + 2
- Cyclomatic Complexity Number (CCN) > 10 is risky
- Strive for lower count
- Consider refactoring and fortify your tests
- Tools: [Java] JavaNCSS, PMD, CheckStyle, ... [.NET] FxCop, Visual Studio Code Analyzer, Resharper, NDepend, ... [C++] Code Counter, CMT++, Cyclo, ...

Cyclomatic Complexity

- For your tests to have reasonable code coverage:
- # of tests > CCN

Cyclomatic Complexity

- Cyclomatic Complexity Number
  - Gives an indication of degree of hardness
  - Does not indicate degree of defect

Code Size

- Addresses problems arising from large, low cohesive code
- Code Size Rules check for the size of code and flags if it exceeds
- How small is small
- Code must fit into a screen (without lowering font size) [about 15 to 20 statements per method]

Code Duplication

- Duplicated code is expensive to maintain
- Hard to fix bugs, hard to make enhancements
- Why do we duplication code then?
- Path of least resistance–you're not breaking existing code, right?
- What to do? Identify and extract methods
- Tools: [Java] PMD, ... [.NET] Simian, ... [C++] ...
- Tools: Simian, StrictDuplicateCode, ...

Assessing Risk

Complexity	Automated Tests	Risk
Low	Low	High
Low	High	Low
High	Low	HIGH
High	High	Medium

C.R.A.P Metric

- Change Risk Analysis and Prediction (CRAP)
  - Experimental metrics and tool
- Measures effort and risk to maintain legacy code
- Uses Cyclomatic Complexity (`comp`) and code coverage (`cov`)
- Created by Bob Evans and Alberto Savoia of Agitar [SAVO07]
- For a method *m*, Version 0.1 of the formula is

```
CRAP(m) = comp(m)^2 * (1-cov(m)/100)^3 + comp(m)
```

- Lower value => low change and maintenance risk
- Lowest value 1. With no tests, risk increases as square of complexity

C.R.A.P Metric

Method's Cyclomatic Complexity	% of coverage required to be below CRAPpy threshold
0 – 5	0%
10	42%
15	57%
20	71%
25	80%
30	100%
31+	No amount of testing will keep methods this complex out of CRAP territory.

C.R.A.P Metric

- 30 = threshold for crappiness
- A complex method (within 30 CCN) can stay below the threshold with adequate tests
- CRAP Load: work estimate to address crappiness
- CRAP Load *N* means indicates the number of tests you need to write to bring your project below the threshold

crap4J is an experimentation tool to measure this metric

Test Quality

- Low coverage indicates inadequate test
- Higher coverage does not mean adequate test, however
- How good is the quality of test?
- Did you cover different conditions, boundaries, ...
- Mutating testers can help determine that
- [Java] Jester
- [.NET] Nester
- [C++] ?

Code Duplication

- Code duplication is common
- Increases maintenance cost
- Makes it hard to fix bugs and make enhancements
- [Java] Simian, PMD (Copy Paste Detector), ...
- [.NET] Simian, ...
- [C++] Simian, ...

Code Analysis

- Analyzing code to find bugs
- Look for logic errors, coding guidelines violations, synchronization problems, data flow analysis, ...
- [Java] PMD, FindBugs, JLint, ...
- [.NET] VS, FxCop, ...
- [C++] VS, Lint, ...

Identifying Problem

"It was on one of my journeys between the EDSAC room and the punching equipment that...the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs,"—Maurice V. Wilkes, pioneer in early machine design and microprogramming.

What is Code Smell?

- It's a feeling or sense that something is not right in the code
- You can't understand it
- Hard to explain
- Does some magic

Code Smells

- Duplication
- Unnecessary complexity
- Useless/misleading comments
- Long classes
- Long methods
- Poor naming
- Code that's not used

- Improper use of inheritance
- Convoluted code
- Tight coupling
- Over abstraction
- Design Pattern overuse
- Trying to be clever
- ...

What?

```
/**
 * This is the common base class of all Java language enumeration types.
 *
 * @author  Josh Bloch
 * @author  Neal Gafter
 * @version 1.12, 06/08/04
 * @since   1.5
 */
public abstract class Enum<E extends Enum<E>>
        implements Comparable<E>, Serializable {
```

Dealing with Code Smell

- Keep an eye on it
- Indicates code that needs either refactoring or some serious redesign
- Technical Debt
- Take effort to clear the air–frequently

From Writing to Coding…

- William Zinsser Wrote "On Writing Well" 25 years ago!
- He gives good principles for writing well
- These principles apply to programming as much as writing non-fiction
  - Simplicity
  - Clarity
  - Brevity
  - Humanity

Clear, not Clever

- Don't be clever, instead be clear

"I never make stupid mistakes. Only very, very clever ones,"—Dr Who

No Rush

- Don't code in a Hurry—"Haste is Waste"
- Take time to read the code and see if is what you meant
- Take the time to write tests—make sure the code does what you meant, not what you typed
- Code defensively

"Act in haste and repent at leisure: Code too soon and debug forever,"—Raymond Kennington.

Commenting and Self-Documenting Code

- Lots of comments are written to coverup bad code
- Comments should say *Why* or *purpose*, not *how*
- Don't comment what a code does—I can read the code for that—keep it DRY
- Don't keep documentation separate from code
  - Use javadoc (Java), doxygen (C++), NDoc (C#),...
  - At least provide a pointer to where it is
- If you copy and paste, check if comments are still relevant

Commenting and Self-Documenting Code

```
int l1, l2, l3, p1, p2, p3;
// God, help me. I have no idea what this means.
...

above was a comment left by a victim, I mean a developer,
who had to maintain this code years later.
```

- Don't use variable names that are cryptic or too brief
- Keep code simple
- Keep code small
- Keep comments minimum and meaningful
- Give names for constants
  - `order(CoffeeSize.LARGE)` instead of `order(3) // large`

Commenting and Self-Documenting Code

- Use Assertions to document assumptions
- Document unique, special, or unexpected conditions
- Keep them short and clear, however
- Don't pour emotions and arguments
- Keep an eye for stale comments

Code for Clarity—self-documenting...

Can someone who does not know English still understand your code?

Can someone who does not speak your language still understand your code?

Some languages and frameworks are being created by experts who don't speak English as their first language

Ruby - Japan

Groovy - European and US collaborators

Comments on Commenting

- Justify violation of good programming practices or styles
  - If you've to use some convoluted logic, unroll a loop, ... drop a comments to say why
  - Will avoid unnecessary refactoring attempt only to discover that it has to stay that way
  - Will help check if the assumptions stated are still valid
- Don't comment clever code, rewrite it
- If everyone stumbles on a particular problem, don't comment to caution, instead fix it

Error In Your Face

- Raise exceptions so they're in your face
- Don't let problems slip by
- Also, if it is hard to locate problems during development, it will only get worse for your support
- Find easy ways to identify what's wrong
- Log is good, but provide a code to easily located the relevant message in it

Summary

- Practice tactical peer code review
- Consider untested code is unfinished code
- Make your code coverage and metrics visible
- Don't tolerate anyone trashing your code
- Write self documenting code and comment whys
- Use tools to check code quality
- Use tools continuously—that is automated
- Treat warnings as errors
- Keep it small
- Keep it simple

References

- [Beck02] "Test Driven Development: By Example," by Kent Beck, Addison Wesley, 2002.
- [BLOC05] "Java Puzzlers: Traps, Pitfalls, and Corner Cases," by Joshua Bloch, Neal Gafter, Addison Wesley, 2005.
- [BOEH01] "Software Defect Reduction Top 10 List," by Barry Boehm and Victor R. Basili, IEEE Computer, January 2001. (http://www.cebase.org/www/resources/reports/usc/usccse2001-515.pdf).
- [GLAS03] "Facts and Fallacies of Software Engineering," by Robert L. Glass, Addison-Wesley, 2003.
- [FIBW] "Fixing Broken Windows," on Wikipedia (http://en.wikipedia.org/wiki/Fixing_Broken_Windows).
- [THOM99] "The Pragmatic Programmer: From Journeyman to Master," by Andy Hunt and Dave Thomas, Addison-Wesley, 2000 (Excerpt on Software Entropy and Broken Window Problem can be found at http://www.pragprog.com/the-pragmatic-programmer/extracts/software-entropy).

You can download examples and slides from

http://www.agiledeveloper.com - download

Thank You!

Please fill in your session evaluations

You can download examples and slides from
http://www.agiledeveloper.com - download