Software Quality and Standards

Dr. James A. Bednar

jbednar@inf.ed.ac.uk

http://homepages.inf.ed.ac.uk/jbednar

Dr. David Robertson

dr@inf.ed.ac.uk

http://www.inf.ed.ac.uk/ssp/members/dave.htm

What is Software Quality?

- High quality software meets the needs of users while being reliable, well supported, maintainable, portable, and easily integrated with other tools.
- Is higher quality better? Is it more expensive? Not always, on both counts.
- We will look at how to achieve quality, the tradeoffs involved, modeling quality improvement, and standards designed to ensure quality.

Cost/Benefit Tradeoff

Making changes to improve software quality requires time and money to:

- Spot the problem
- Isolate its source
- Connect it to the real cause
- Fix the requirements, design, and code
- Test the fix for this problem
- Test the fix has not caused new problems
- Change the documentation

For a given change to make sense, the improvement needs to pay for all these tasks, plus the revenue lost during the delay in the product release.

Feature/Bug Tradeoff

- Meeting the needs of users (not to mention marketing) requires adding features.
- However, given a fix ed amount of development time and money, adding features adds bugs and reduces time for testing.
- Do the features increase user productivity more than the bugs decrease it?
- Difficult to answer this question, because data on users is sparse, and other factors like reputation usually take precedence.

Quality for free?

But is increasing quality always more expensive, in terms of total cost of production and maintenance? No.

In fact, if you focus on quality from the start, then:

- You tend to produce components with fewer defects, so
- You spend less time debugging, so
- You have more time in your schedule for improving other aspects of quality, like usability

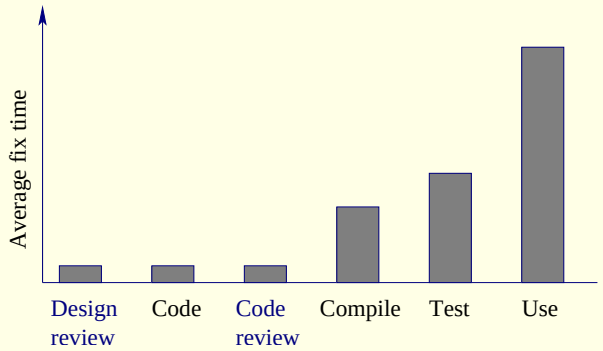
Skimp Now, Pay Later

If you don't focus on product quality then:

- You tend to produce components with more (hidden) defects, so
- You have to spend more time fi xing these (late), so
- You have little time for anything else, so
- You produce poor quality software **even though you put huge amounts of effort into defect checking**.

Thus quality is something that has to be considered throughout the product lifecycle; it cannot be added in later.

Quality Delays are Expensive



Thus it makes sense to focus on improving component quality before testing, to catch difficult defects early.

Better Quality Through Testing?

Humphrey (2002) estimates that experienced software engineers normally inject 100 or more defects per KLOC.

Perhaps half of these are detected automatically (e.g. by the compiler).

So a 50 KLOC program probably contains around 2500 defects to find (semi-)manually.

Suppose we need about fiv e hours to find each of these defects by testing.

That's over 20000 hours for the whole program - **bad news**.

Better Quality Through Inspection?

Code inspection may be able to find up to (say) 70% of these defects in 0.5 hours per defect.

So the first 1750 defects could take 875 hours; then we only have 750 to find in testing at (say) 8 hours each. That's less than 7000 hours in total - **better news**.

Modeling Quality Improvement

$$y(N) = \frac{r(N)}{r(N)+e(N)}$$

where:

- $y(N)$ is fraction of defects removed in step N
- $r(N)$ is the number of defects removed at step N.
- $e(N)$ is the number of defects escaping at step N.

The difficulty with this equation is that we can only estimate $e(N)$ as a function of $e(1), \ldots, e(N-1)$.

Notice that $e(N)$ can *increase* when a change injects defects.

Sensitivity to Inspection Yield (1)

Suppose you have 1000 KLOC with an average of 100 defects per KLOC. That's 100000 defects to find.

Scenario 1:

- You have an inspection process which finds 75% of these, leaving 25000 to find in test.
- You then use 4 levels of test, each trapping 50% of remaining defects. That leaves 1562 defects in the final code.

Sounds good so far...

Sensitivity to Inspection Yield (2)

Scenario 2:

- Your inspection process only finds 50% of defects, leaving 50000 to find in test.
- The same 4 levels of test each trap 50% of remaining defects. That leaves 3125 defects in the final code.

So a 33% drop in yield in inspection caused a doubling in the number of remaining defects. Thus the effectiveness of your inspection process is crucial.

Sensitivity to Defect Injection

Assuming we start with no defects, $P_i = (1-p)^i$, where:

- p is the probability of injecting a defect at a stage.
- i is the number of stages.
- P is the probability of a defect-free product at stage i.

A high probability of fault injection in one step radically drops the overall probability of freedom from defects:

$$(1-0.01)^{10} = 0.904$$
$$(1-0.01)^{9} * (1-0.5)^{1} = 0.4057$$

This is why cleanrooms are so clean.

Sensitivity to Defect Removal

$R_i = N * (1-y)^i$, where:

- N is the initial number of defects.
- y the fraction of defects removed per stage.
- i is the number of stages.
- R_i is the number of defects remaining at stage i.

Dropping a lot lower on one stage of a high quality defect removal process has a small effect on overall yield.

$$100000 * (1-0.8)^{5} = 32$$
$$100000 * (1-0.8)^{4} * (1-0.4) = 96$$

Thus being defect-free is better than relying on fix es.

Yield Management

If we had no resource limitations then an 80-40 test-inspection yield is no different from a 40-80 yield.

But test defect correction typically involves more labour than inspection defect correction, so it costs more and the extra labour means . . . more opportunities for defect injection.

So manage for maximum return for minimum cost and, if in doubt, attempt to maximise on early design stages.

Better Quality via Standards?

Most products have safety standards, and many have usability standards, but computer software rarely has such standards.

Can quality be improved by enforcing standards? Unclear:

- It is very difficult to enforce standards on actual program behavior
- Standardizing the process can help make sure that no steps are skipped, but
- Standardizing to an inappropriate process can reduce productivity, and thus leave less time for quality

Software Engineering Standards

According to the IEEE Comp. Soc. Software Engineering Standards Committee a standard can be:

- An object or measure of comparison that defines or represents the magnitude of a unit
- A characterization that establishes allowable tolerances or constraints for categories of items,
- A degree or level of required excellence or attainment

Why Bother with Standards?

Prevents idiosyncrasy: e.g. Standards for primitives in programming languages)

Repeatability: e.g. Repeating complex inspection processes

Consensus wisdom: e.g. Software metrics

Cross-specialisation: e.g. Software safety standards

Customer protection: e.g. Quality assurance standards

Professional discipline: e.g. V & V standards

Badging: e.g. Capability Maturity Model levels

Legal Implications (1)

Comparatively few software products are forced by law to comply with specific standards, and most have comprehensive non-warranty disclaimers. However:

- For particularly sensitive applications (e.g. safety critical) your software will have to meet certain standards before purchase
- US courts have used voluntary standards to establish a supplier's "duty of care"

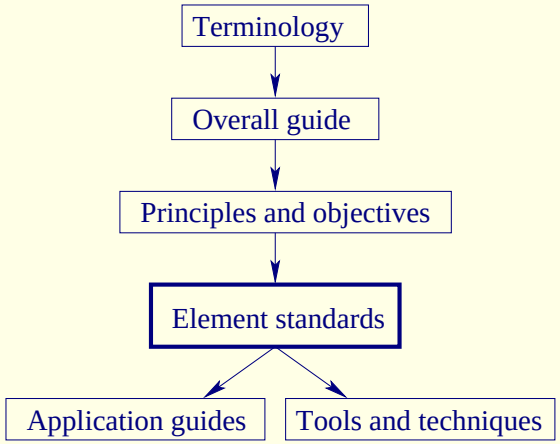
Legal Implications (2)

Adherence to standards is a strong defence against negligence claims (admissible in court in most US states)

There are instances of faults in products being traced back to faults in standards, so

Standards writers must themselves be vigilant against malpractice suits

Levels of Standards



Some Standards Organizations

ANSI: American National Standards Institute. Does not itself make standards but approves them

AIAA: American Institute of Aeronautics and Astronautics (*e.g.* AIAA R-013-1992 Recommended Practice for Software Reliability).

EIA: Electronic Industries Association (e.g. EIA/IS-632 Systems Engineering)

IEC: International Electrotechnical Commission (e.g. IEC 61508 Functional Safety - Safety-Related Systems)

IEEE: Institute of Electrical and Electronics Engineers Computer Society Software Engineering Standards Committee (*e.g.* IEEE Std 1228-1994 Standard for Software Safety Plans)

ISO: International Organization for Standardization (e.g. ISO/IEC 2382-7:1989 Vocabulary-Part 7: Computer Programming)

Computer Science Standards

Surprisingly few CS standards exist, although one could argue this is because CS is pervasive in others.

Examples:

Terminology: IEEE Std 610.12:1990 Standard Glossary of Software Engineering Terminology

Techniques: ISO/IEC 8631:1989 Program Constructs and Conventions for their Representation

Quality Assurance Standards

Differing views of quality standards: taking a systems view (that good management systems yield high quality); and taking an analytical view (that good measurement frameworks yield high quality). Examples:

Quality management: ISO 9000-3 Quality Management and Quality Assurance Standards - Part 3: Guidelines for the application of 9001 to the development, supply, installation and maintenance of computer software

Quality measurement: IEEE Std 1061-1992 Standard for Software Quality Metrics Methodology

Project Management Standards

These are concerned with how general principles of good management are applied to specific areas of software engineering.

Examples:

General project management: IEE Std 1058.1-1987 Standard for Software Project Management Plans

Producing plans: IEEE Std 1059-1993 Guide for Software Verification and Validation Plans

Systems Engineering Standards

Particular application domains develop sophisticated interactions between system and software engineering, so standardizing from a systems point of view can be beneficial. Examples:

Lifecycle: ISO/IEC WD 15288 System Life Cycle Processes

Requirements: IEEE Std 1233-1996 Guide for Developing System Requirements Specifications

Dependability Standards (1)

As hardware dependability has improved, software has received more attention as a dependability risk. Dependability of software isn't just a question of internal measures (e.g. availability, reliability) but also broader issues (e.g. maintainability, system context).

Dependability standards often set integrity levels necessary to maintain system risks within acceptable limits.

Dependability Standards (2)

Examples:

Dependability management: IEC 300-1(1993) Dependability management Part 1: Dependability programme management

Risk analysis: IEC 1025(1990) Fault Tree Analysis

Reliability: AIAA R-013-1992 Recommended Practice for Software Reliability

Safety Standards

These traditionally come out of specific industrial sectors (e.g. American Nuclear Society, UK Ministry of Defence), since safety requires deep analysis of the domain as well as the technology. Examples:

Safety plans: IEEE Std 1228-1994 Standard for Software Safety Plans

Functional safety: IEC 61508 Functional Safety - Safety-Related Systems

Nuclear domain: IEE 603 Criteria for Safety Systems of Nuclear Plants

Resources Standards

Although software engineering is in flux, it is possible to standardize on some forms of resources which are used widely across applications. Examples:

Terminology: IEEE 610,12-1990 Standard Glossary of Software Engineering terminology

Semantics: IEEE P1320.1 Standard Syntax and Semantics for IDEF0

Re-use libraries: AIAA G-010-1993 Guide for Reusable Software: Assessment Criteria for Aerospace Application

Tools: ISO/IEC 14102:1995 Guideline for the Evaluation and Selection of CASE tools

Product Standards

These focus on the products of software engineering, rather than on the processes used to obtain them. Perhaps surprisingly, product standards seem difficult to obtain. Examples:

Product evaluation: ISO/IEC 14598 Software product evaluation

Packaging: ISO/IEC 12119:1994 Software Packages - Quality Requirements and Testing

Process Standards

A popular focus of standardization, partly because product standardization is elusive and partly because much has been gained by refining process. Much of software engineering is in fact the study of process. Examples:

Life cycle: ISO/IEC 12207:1995 Information Technology - Software Life Cycle Processes

Acquisition: ISO/IEC 15026 System and software Integrity Levels

Maintenance: IEEE Std 1219-1992 Standard for Software Maintenance

Productivity: IEE Std 1045-1992 Standard for Software Productivity Metrics

Company Guidelines

Specific companies may develop their own guidelines for system/software design. These define good practice within a company. They often conform to more general standards. Example:

Shell UK Code of Practice: Fire and Gas Detection and Alarm Systems for Offshore Installations. Describes what a fire and gas alarm system must do; prescribes properties of that system; sets goals for achieving those properties; gives examples of typical design solutions.

Trends

- Concern about absence of scientific foundation for standards
- Recognition that standards usually aren't isolated
- Questioning of software (non)warranty agreements

Summary

- It is crucial to think about quality when you start the project
- More quality is not always better, but it is usually is
- Correcting defects is very different at different stages
- Standards can help ensure consistent quality, but primarily for process, not product

References

Humphrey, W. S. (2002). *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley.