

Software Quality Assurance, Testing, and Metrics

Douglas C. Schmidt

<http://www.cs.wustl.edu/~schmidt/>

schmidt@cs.wustl.edu

Washington University, St. Louis

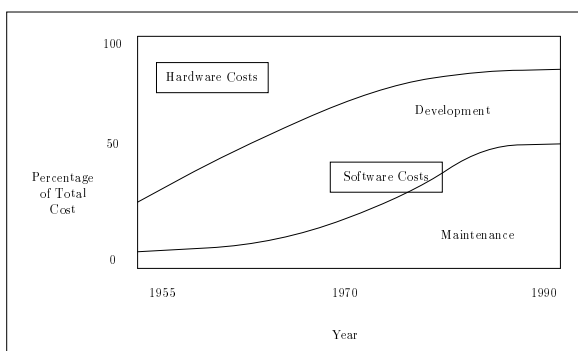
1

Motivation

- The typical state of affairs in system development is termed the “Software Crisis” *i.e.*,
 - Whereas hardware gets smaller, faster, cheaper, software often gets larger, slower, more expensive
- Therefore, we need *methods* and *tools* to create quality software in an orderly, predictable manner
- Design/code reviews, testing techniques, and metrics are an important means of software quality assurance
 - Note that OO software has certain characteristics that make testing and metrics hard

2

Motivation (cont'd)



- Software costs are rising in proportion to hardware costs
- Maintenance costs are rising in proportion to development costs

3

Common Software Development Problems

- Project scheduling is error-prone and unpredictable
 - *e.g.*, mostly done by “educated guessing”
- Systems fail to meet their requirements
 - *e.g.*, AT&T phone failures, Therac 25, etc.
- Execution-time errors are not handled gracefully
 - *e.g.*, bus error, segmentation fault ;-)

4

Common Software Development Problems (cont'd)

- Documentation and source code are difficult to modify and maintain
 - e.g., too much code is “write only”
- Too much duplicated effort exists across projects
 - e.g., most systems are not built with reuse in mind...
- Programs typically do not interact well with other programs
 - e.g., few standard interfaces (cf. with hardware)

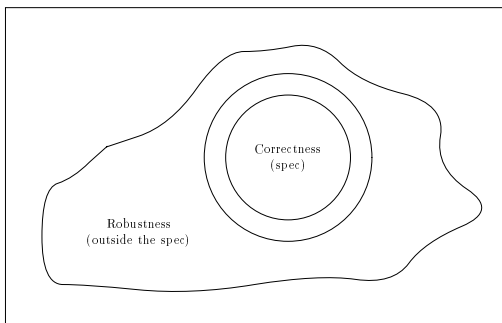
5

Software Quality

- Software quality has several dimensions:
 1. *External Quality Factors*
 - These are detectable by system users and clients
 2. *Internal Quality Factors*
 - These are detectable by software professionals, e.g., designers and implementors
- Object-oriented techniques can help improve both internal and external quality factors

6

Major External Quality Factors



- **Correctness**
 - The ability of software products to exactly perform their tasks, as defined by the system's requirements and specifications
 - Requires some external validation or verification process
- **Robustness**
 - The ability of software systems to function even in abnormal conditions
 - ▷ i.e., outside its specification

7

Major External Quality Factors (cont'd)

- **Extensibility**
 - The ease with which software products may be adapted to changes of their specifications
 - Remember, small/medium-scale systems are often *grown* into large-scale systems
 - ▷ e.g., UNIX, DOS
- **Reusability**
 - The ability of software products to be reused, in whole or in part, for new applications
 - This requires
 1. Standard interfaces
 2. Specification techniques to ensure correctness
 3. Error-handling mechanisms to facilitate robustness

8

Major External Quality Factors (cont'd)

- *Compatibility*
 - The ease with which software products may be combined with others
 - This also requires standard interfaces, and/or some facility for performing transformations between different *vocabularies* and *formats*
 - ▷ e.g., UNIX pipes and filters

9

Additional External Quality Factors

- *Efficiency*
 - The ability of software products to perform their tasks within an acceptable time period, given various hardware, software, or human resource constraints
 - Note, in certain domains this is often regarded as the most important quality factor!!!
 - ▷ e.g., networking vs. large software systems...
- *Ease of Learning and Ease of Use*
 - The degree of difficulty in learning how to use software systems, operating them, preparing input data, interpreting results, recovering from usage errors, etc
 - ▷ A consistent interface is often useful
 - e.g., the Macintosh or Windows 95
 - Easy of learning and use is obviously a *subjective* quality ;-)

10

Additional External Quality Factors (cont'd)

- *Integrity*
 - The ability of software systems to protect their components against unauthorized access and modification
 - Note, this is becoming more problematic with the wide-spread use of distributed systems
- *Safety*
 - The ability of the software to prevent loss of human life and serious property damage
 - This is becoming increasingly important in our litigious society

11

Internal Quality Factors

- *Modularity*
 - The extent to which the various software components comprising the system possess low coupling and high cohesion
- *Maintainability*
 - The ease with which defect removal, performance enhancement, and feature modifications can be added to the original system without unduly compromising system structure and documentation
- *Verifiability*
 - The ease of detecting failures during validation phases using test data, code reviews, and other quality assurance methods
- *Portability*
 - The ease with which software products may be transferred to different hardware and software platforms

12

Quality Factor Trade-Offs

- *Ease of Use vs Safety and Integrity*
 - e.g., INS system on KAL 007
 - passwords and user authentication
- *Efficiency vs Reusability*
 - e.g., performing inline code expansion “by-hand”
- *Efficiency vs Portability*
 - e.g., block-move instructions for optimizing *string* libraries
- *Efficiency vs Modularity*
 - e.g., monolithic vs modular operating systems (micro-kernels) and communications architectures

13

OO Features That Help Improve Quality

- *Correctness and Verifiability*
 - assertions, postconditions, and preconditions
- *Robustness*
 - exception handling (software) and recovery blocks (hardware)
- *Extensibility*
 - inheritance, dynamic binding, and polymorphism
- *Reusability*
 - generics and parameterized types

14

OO Features That Help Improve Quality (cont'd)

- *Compatibility and Modularity*
 - separate compilation, abstract data types, and classes
- *Efficiency*
 - inline functions, macros, register variables
- *Portability*
 - conditional compilation, extern linkage blocks, and information hiding

15

The Challenge of OO Testing

- Testing OO programs is hard since components are often “generic” to increase reuse, e.g.,
 - Parameterized types
 - Abstract base classes
 - Template Method patterns
- Therefore, unit testing becomes hard
 - e.g., it may not be possible to test generic OO components completely in the absence of a particular *instantiation*

16

OO Metrics

- Traditional software complexity metrics focus on simple attributes of programs such as “lines of code”
- These metrics are not very useful for OO systems
- OO metrics tend to focus on relationships among classes, class categories, and subsystems
- *Important Caveat*
 - Metrics are just one of many *tools* developers use to understand their systems
 - However, they are *no* substitute for insight...

17

Weighted Methods per Class

- Give a relative measure of the complexity of individual classes
- If all methods are equally complex this metric measures the number of methods per class
 - However, real classes often have differentially complex methods
- In general, classes with more methods tend to be more complex, more application-specific, and more error-prone

18

Depth of Inheritance Tree

- The depth of the inheritance tree and number of children measures the *shape* and *size* of the class structure
- Well designed OO systems typically have a *forest* of class hierarchies, rather than a single *rooted tree*
- In general, inheritance hierarchies should be no deeper than around 7 classes nor be wider than around 7 classes
 - Note that not being deeper is more important than not being wider...

19

Relational Cohesion

- Represented as the average number of internal relationships per class (H)
- *e.g.*,
 - R is number of class relationships that are *internal* to a class category
 - ▷ *i.e.*, not connected to other classes
 - N is number of classes within the category
- Then, $H = (R + 1)/N$
 - This represents the relationship that the category has to its classes

20

Afferent and Efferent Coupling

- *Afferent Coupling*
 - Calculated as the number of classes from other categories that depend upon the classes within the subject category
- *Efferent Coupling*
 - Calculated as the number of classes in other categories that the classes in the subject category depend upon
- For both, dependencies are class relationships like *inheritance*, *containment*, and *uses*

21

Qualitative vs. Quantitative Reviews

- Software is inherently abstract
 - This makes it hard to measure quality and predict schedules
- Good development processes typically emphasize *qualitative* reviews more than *quantitative* reviews
 - Qualitative reviews focus on inspections of design and code by groups of peers
 - Quantitative reviews use automated complexity metrics and tools
- In general, qualitative reviews are more effective at identifying and correcting strategic problems

22

Motivation for Code Reviews

- *Advantages*
 - Empirical studies show that code reviews catch more bugs than other forms of quality control
- *Disadvantages*
 - Code reviews are time-intensive
 - Make sure to schedule time for code reviews into project planning...
 - Resist the urge to cut reviews as project schedule slips

23

Conducting a Code Review

- Run *one* listing
- Divide it into as many pieces as there are people on the review team
- Assign each person a different colored pen
- Each person reviews a piece, marking comments on the listing
- Pass the pieces around until each person reviews each piece

24

Conducting a Code Review (cont'd)

- Hold a meeting
- Assign one person as moderator
 - The moderator's job is to ensure that all comments are constructive, and no belittling remarks are made
 - One technique is to fine a person a dollar for each personal, rather than code, criticism!
- Give everyone a chance at being moderator
- Ensure an adequate supply of Coca-Cola for the meeting ;-)

25

Conducting a Code Review (cont'd)

- Go through all the marks on the listings
- Action items are agreed upon and marked into the listings
- Assign action items to the team members
- Have a followup meeting to check that all action items were resolved

26