Software Engineering: Theory and Practice

NMA – National Student Academy
Summer Session 2008, Nida, Lithuania

Lecture 1

Tom Verhoeff

Technische Universiteit Eindhoven
Department of Mathematics & Computer Science
Software Engineering & Technology

Feedback to `T.Verhoeff@TUE.NL`

Topics

Welcome!

You have just started a software company:

***N**ew **M**ind **A**pplications*

and now you should think about

- Software
- Engineering

Engineering

What are the major ingredients and concerns of engineering?

Make your own list.

Compare with your neighbor and discuss.

Engineering: The World Is Not Ideal

- Design and produce products/artifacts (incl. tools) for others.
  Resources needed: materials, energy, labor, intellect
- Apply scientific knowledge and scientific method (incl. math).
  Use theories, models to predict, analyse, compare.
  Do experiments, prototyping, problems solving (not: exercises).
  Also inspires new science.
- Ethical aspects (society, environment, life, death)
- Economical aspects (limited resources: time, effort, cost)
- Human limits: productivity, memory, accuracy, communication

Engineering: The World Is Not Ideal

- Variability, risks (stochastic aspects)

  Account for variability in drill hole location when fixing a plate:

No freedom after drilling in wall Two degrees of freedom

- Quality (satisfy expectations: not black & white)

Software

In what ways does software differ from hardware?

Make your own list.

Compare with your neighbor and discuss.

Software: Its Special Character

- Immaterial, intangible (like holes in a punchcard)
- Malleable (reshapable, flexible, "soft": is also a danger)
- Intellectually intensive
- Incomplete product (requires hardware to run)
- Easy/cheap to reproduce
- Stable, no wear
- Easy to create unmaintainable products

Software

What kinds of software can you distinguish?

Make your own list.

Compare with your neighbor and discuss.

Software: Its Diversity

- Consumer software for "standard" computer (word processor)
- Consumer software for "special" hardware (video games)
- Embedded software in consumer products (TV, shaver)
- Embedded software in industrial systems (baggage handling)
- Administrational/information systems (library, tax department)
- Software tools (to develop software, or hardware)
- Operating systems (to extend capabilities of hardware)

Software

In what markets is software relevant?

Make your own list.

Compare with your neighbor and discuss.

Software: Its Omnipresence

Where not?

- Science, engineering
- Industry, business
- Medical
- Government, education
- Transportation (air, road, water, space)
- Consumer products

Software Engineering: IEEE Definition

The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software

Also: the scientific study of such approaches

IEEE = Institute of Electrical and Electronics Engineers

The IEEE develops and maintains numerous internationally-accepted standards for Software Engineering.

History

Evolution from

- Customer, developer, and user of software are the same amateur

to

- Software products developed by professionals for external non-expert customers and used by yet other non-experts

Goals and Challenges of Software Engineering

Make quality software products, on time, within budget that

- are (part of) large and complex systems,
- are built by teams,
- are requested and used by non-experts,
- exist in many revisions and variants,
- last for many years, and
- undergo frequent changes.

Welcome to **N***ew* **M***ind* **A***pplications* and the world of software!

References

- IEEE — IEEE computer society
- acm Association for Computing Machinery — Advancing Computing as a Science & Profession
- *Software Engineering Code of Ethics and Professional Practice*

  Copy at IEEE Computer Society

  Copy at ACM
- *SWEBOK – Software Engineering Body of Knowledge*

Software Engineering: Theory and Practice

NMA – National Student Academy
Summer Session 2008, Nida, Lithuania

Lecture 2

Tom Verhoeff

Technische Universiteit Eindhoven
Department of Mathematics & Computer Science
Software Engineering & Technology

Feedback to `T.Verhoeff@TUE.NL`

Topics

- (Software) Project Organization
- Requirements Engineering
- Configuration Management

Product and Process

What is important in a software project?

Historic learning path, becoming aware of the relevance of

- Product (ultimate deliverable)
- Product documentation, other intermediate artifacts, verification
- Process (how work is organized and done)
- Process documentation, verification of process

Project Management

Cycle of activities:

1. Plan: Make/change a plan

   Who does what when; write it down

2. Do: Execute the plan
3. Check: Monitor the plan
4. Act: Analyze and decide, go to 1

"Failing to plan is planning to fail."

General Problem Solving (left) and Waterfall Process (right)

1. Admit you have a problem	Business Case
2. Define the problem clearly	User Requirements
3. Understand and analyze the problem	Software Requirements
4. Outline a solution approach (blueprint)	Architectural Design
5. Construct an actual solution	Coding/Production
6. Teach the solution to others	Release/Transfer
7. Apply/adjust the solution	Operation and Maintenance
8. (Not applicable?)	Retirement

When doing work, always verify the work as soon as possible.

Software Development Process

There are many alternative ways of organizing activities.

There is not one right way.

Any clear process is better than no process.

Is more than project management (who does what when).

Answers questions: Why do it? Do what? Make what? Do it how?

Process quality affects product quality:

Process → Product Internals → Product Externals → User Experience

Why Requirements Engineering?

Important potential problem:

Building a good system but not the right system

Requirements Engineering: Concepts

- Requirement: capability or constraint that must be met in order to satisfy a contract or specification
- Quality characteristic/dimension/factor: . . . , e.g. Functionality, Reliability, Usability, Efficiency, Maintainability, Portability

How To Obtain Requirements: Elicitation and Analysis

- Interview, survey
- Study existing products
- Analyze requirements of system that contains the software
- Consolidate and structure raw requirements
- Follow-up, discuss draft requirements
- Prototype (could be a paper mock-up)
- Model (e.g. state machine)

General Requirements

- Name, purpose
- Overview of capabilities, constraints, rationale
- User categories, typical usage
- Operational environment, context diagram
- External interfaces
- Expected future changes

Specific Requirements

- Atomic (single sentence; avoid passive voice)
- Identifier (ensure traceability), priority
- Capability requirements: include capacity, speed, accuracy
- Constraint requirements: product and process qualities
- Redundancy: good and bad, cross reference
- Concern the problem domain, *not* the solution domain.
  N.B. Software engineers often lack problem domain knowledge
- Must be verifiable

Why Configuration Management?

Important potential problems:

- Ambiguity as to what document or file is meant
- Inconsistent combinations of files
- Lost files, lost changes
- Undocumented, unapproved changes
- Not knowing the composition of released software
- Not being able to go back to a previous version

Configuration Management (CM) Terminology

- Configuration Item (CI): (document, software, hardware) entity treated as a unit for CM; atomic or composite
- Version and relations between versions:
  - revision replaces obsolete/incorrect version
  - variant exists parallel to other version
- Baseline: formally reviewed and approved CI serving as a basis for further development

Configuration Management Goals

- Identify and define the CIs of the project: all relevant artifacts
- Control the release and change of CIs throughout the project
- Record and report the status of CIs and of change requests
- Verify the completeness and correctness of CIs

Configuration Management Tools

- Tools are not a complete solution: also need to use them well.

  Establish clear and effective procedures and rules. Train users.
- CVS, Subversion, Bazaar, Mercurial, Git
- Bugzilla
- Trac: an enhanced wiki and issue tracking system for software development projects
- Also consider tools for building and releasing (not treated here).

Configuration Management with Subversion

- One central repository with current and past versions
  - Trunk holds main line of development
  - Tags mark baselines, releases, . . .
  - Branches hold variants, experiments, . . .
- Multiple local working copies
  - checkout, update, commit

Copying in the repository is cheap, both in time and in memory usage.

Tagging and branching are done by copying. The name of the copy identifies the tag or branch.

Subversion: Create Working Copy from Repository

Subversion: Create Working Copy from Repository

Subversion: Commit Changes in Working Copy to Repository

Configuration Management: Good Practices

- Only commit code that compiles, runs, and passes all unit tests.
- When committing, always include a log message explaining *why* the change was made (**svn diff** can tell you *what* changed).

References

- Example: *Anagrams User Requirements Document*
- Checklist for User Requirements
- "Configuration Management", Ch. 17 from *Software Engineering: Planning for Change* by D. A. Lamb. Prentice-Hall, 1988.
- *Common CM Tasks in Subversion* by T. Verhoeff
- *The Subversion Book*
- Turtoise SVN: GUI client for Subversion on Windows
- svnX: GUI client for Subversion on Mac OS X

Software Engineering: Theory and Practice

NMA – National Student Academy
Summer Session 2008, Nida, Lithuania

Lecture 3

Tom Verhoeff

Technische Universiteit Eindhoven
Department of Mathematics & Computer Science
Software Engineering & Technology

Feedback to `T.Verhoeff@TUE.NL`

Topics

- Engineering and Errors
- Verification by Review
- Architecture

What Is an Error?

- What is the most impressive error that you have made?

  Watch video of Ariane 501 flight.
- Beautiful versus ugly: an opinion, not an error

Terminology: IEEE Classification

- Failure: product deviates from requirements during use/operation
- Defect, fault: anomaly in a product that can somehow (eventually) lead to a failure
- Mistake: human action ("slip") causing a fault
- Error: difference between actual and specified/expected result

Assumes requirements/specification/contract (establish in advance)

Economy of Defects

- The later a defect is discovered, the higher the costs: grows exponentially in amount of time between injection and discovery.
- Defects decrease the predictability of a project: cost (time) of defect localization and repair are extremely variable.
- Defects concern risks, i.e. uncertainty; product could be defect-free at once, but defects are likely.

Dealing with Defecs

1. Admit that people make mistakes and inject defects
2. Prevent them as much as possible
3. Detect their presence as early as possible
4. Localize them
5. Repair them
6. Trace them: root causes and consequences
7. Learn from them: improve the process

Preventing Defects (or instant detection and repair)

- Impossible to do for 100%, but prevention offers the biggest gains
- Every defect not prevented increases the cost
- Always work neatly, also on prototypes, test software, . . .
- Use checklists and standards
- Work in pairs
- "Think before you act"

Detecting Defects

- Reviewing:
  - Examine an artifact *with the intent of finding defects*.
  - Can be done early in the development process.
  - Often localizes the defects as well.
  - Can and should also be applied to code.
- Testing:
  - Use a product systematically *with the intent of finding defects*.
  - Works through failures; need not localize underlying defects.
  - Requires a working product (part).

Limits of Testing

Edsger W. Dijkstra (CACM, 1972):

> "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

Testing in itself does not create quality.

Dijkstra's advice: Prove mathematically that an artifact has required properties. Ideally: let proof development drive the design.
See: NMA Summer Session 2006.

Removing Defects

Debugging: localize, diagnose, and correct detected defects

Time consuming and unpredictable process

Coding Standards

- Restrict what program code "looks" like
- Layout: indentation, spacing, blank lines, line length; at most one definition/declaration/statement per line
- Naming: constant, variable, method, class, attribute
- Comments: file header, contract (assume, effect), explain variable declaration or statement
- Structure: how to order things; maximum size of code blocks

Why Use Coding Standards?

- You make fewer mistakes.
- If you make them, they are found easier and quicker.
- If you cannot find them yourself, then others can help you more effectively.
- In case of law suits, you are in a better position to defend yourself.

Costs of Dealing with Defects Responsibly

- Costs extra effort and time (mostly initially).
- Consider this to be a small pre-paid insurance fee.
- Cost applies equally to standardization, reviewing, testing, ...
- Not using these techniques increases risks and unpredictability, and hence costs, often considerably.

Why Do Reviewing?

- Early detection of deficiencies and risks:
  - Humans make mistakes, no matter what.
  - Late detection is (very) costly.
  - Testing cannot find all (kinds of) defects.
- Communication of knowledge
- Monitoring of status and progress

Types of Reviewing

- Management review: of a process, on behalf of managment
- Technical review: of a product (not of its creator)
- Inspection: visual examination, by peers
- Walk-through: creator explains artifact to others
- Audit: independent examination, also of an organization

How to Do Reviewing

1. Determine type, purpose, and timing of review in advance.
2. Check initial fitness (e.g.: is document spell checked).
3. Select and inform reviewers (and train them, if needed).
4. Distribute material on time.
5. Prepare individually (read material, make notes).
6. Hold meeting (pre-determined roles), decide on recommendations.
7. Write and present a review report.

General Advice on Reviewing

- Take reviews seriously and spend time well.

  Do not waste time on trivialities.

- Use checklists and standard.

- Stick to the purpose (e.g. do not criticize creators).

- Do not try to solve problems while reviewing.

  But: do recommend changes, also to the development process.

- React on review outcome (do rework, adjust the process).

What is Software Architecture?

- The fundamental organization of a system

- embodied in its components,

- their relationships to each other and

- to the environment, and

- principles guiding its design and evolution.

Why Software Architecture?

- Facilitates parallel construction by a team.

- Improves ability to make work plans.

- Improves verifiability (makes it easier to get it to work):

  – Allows unit testing of separate components.

  – Allows stepwise integration (no "big bang").

- Improves maintainability: most changes affect few components.

Architecture Description: Ingredients

- Stakeholders

- Viewpoints

- Architectural views

- Inconsistencies and conflicts among views

- Rationale

Compare to architectural description of buildings: spaces and doors, water supplies and drains, electricity, heating/cooling, fire safety, . . .

Kruchten's 4 + 1 Views

	Static (Structure)	Dynamic (Behavior)
Abstract	Logical	Process
Concrete	Development concerns code in files	Deployment concerns processors

+ Use case scenarios traced through the architecture

Example of Logical View

How to Design an Architecture

Almost any architecture can be made to work, that is, can be made to provide required *functionality*.

Extra-functional requirements should drive the architectural design: understandability, verifiability, efficiency, maintainability, . . .

Approaches: Top down, bottom up, yo-yo, functional decomposition, data distribution

KISS: Keep It Simple, Stupid

Consider alternatives and compare them: on paper, by experiment.

References

- *Ariane 5 Failure: Full Report* by ESA
- "The $100,000 Keying Error", *IEEE Computer*, pp.106–108, April 2008.
- *Code Conventions for the Java Programming Language* by SUN
- *Java Coding Standards* by ESA
- Example: *Anagrams Architectural Design Document*

Software Engineering: Theory and Practice

NMA – National Student Academy
Summer Session 2008, Nida, Lithuania

Lecture 4

Tom Verhoeff

Technische Universiteit Eindhoven
Department of Mathematics & Computer Science
Software Engineering & Technology

Feedback to `T.Verhoeff@TUE.NL`

Topics

- Verification by Testing
- Test Case Design

Do Not Confuse Testing and Debugging

Testing = The process of executing software with the intent of detecting the presence of defects.

Testing determines a measure for quality.

Testing is only one of many verification activities.

Debugging = The act of fault diagnosis and correction.

Debugging concerns rework.

Debugging is time consuming and unpredictable.

Self-Assessment Test

The problem is the **testing** of the following program:

The program reads three integer values from a card.

The three values are interpreted as representing the lengths of the sides of a triangle.

The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

Write a set of **test cases** that you feel would *adequately* test this program.

Glenford J. Myers. *The Art of Software Testing*. Wiley, 1979.

Self-Assessment Test Scoring

1. Valid scalene triangle included?
   OK $(3, 4, 5)$. **NO** $(1, 2, 3)$ **or** $(2, 5, 10)$.

2. Valid equilateral triangle included?
   OK $(3, 3, 3)$. **NO** $(0, 0, 0)$.

3. Valid isosceles triangle included?
   OK $(3, 3, 1)$. **NO** $(2, 2, 4)$.

Self-Assessment Test Scoring

4. All three permutations of valid isosceles triangle?
   OK $(3, 3, 1)$ **and** $(3, 1, 3)$ **and** $(1, 3, 3)$.

5. One side equal zero?
   OK $(0, 4, 5)$.

6. One side negative?
   OK $(-3, 4, 5)$.

Self-Assessment Test Scoring

7. Degenerate triangle $(a + b = c)$?
   OK $(1, 2, 3)$.

8. All three permutations of degenerate triangle?
   OK $(1, 2, 3)$ **and** $(2, 3, 1)$ **and** $(3, 1, 2)$.

9. Non-triangle with positive sides $(a + b < c)$?
   OK $(1, 2, 4)$.

10. All three permutations of non-triangle?
    OK $(1, 2, 4)$ **and** $(2, 4, 1)$ **and** $(4, 1, 2)$.

11. All sides zero?
    OK $(0, 0, 0)$.

Self-Assessment Test Scoring

12. Non-integer values?
    OK (`'A','B','C'`).

13. Wrong number of values?
    OK $(3, 4)$ **or** $(3, 4, 5, 6)$.

14. Expected output for each case included?

Some Testing Principles

- A necessary part of a test case is a definition of the expected output or result.
- Thoroughly inspect the result of each test.
- Avoid throw-away test cases unless the program is truly a throw-way program.
- Do not plan a testing effort under the tacit assumption that no faults will be found.
- Testing is an extremely creative and intellectually challenging task.

Levels of Testing in V-Model (from ESA SE Std)

What Qualities to Test

- Utility: To what extent is required functionality provided?
- Reliability: To what extent does the product fail? How frequently, how critical?
- Robustness: What happens in unexpected situations?
- Efficiency: How much is used of resources? Time, memory, disk, network,
- Usability: How easy is the product to use?

Approaches to Test Case Design

Black-box, or test-to-specifications, or functional:

Checks the functionality of the software.

Consider specification/requirements only. Ignore code.

Glass-box, or test-to-code, or structural:

Checks the internal logic of the software.

Consider code only. Ignore specification/requirements.

Techniques for Constructing Test Cases

- Boundary analysis
- Equivalence classes
- Statement, branch, and path coverage

Coverage: Example

if C **then** $v := 1$
; **if** D **then** $w := 2$
else $w := 3$

5 (!) statements, $2 + 2$ branches, $2 * 2$ paths

Test Cases				Coverage		
1	2	3	4	Statement	Branch	Path
$\neg C, \neg D$				60%	50%	25%
C, D				80%	50%	25%
C, D	$C, \neg D$			100%	75%	50%
C, D	$\neg C, \neg D$			100%	100%	50%
C, D	$C, \neg D$	$\neg C, D$	$\neg C, \neg D$	100%	100%	100%

Testing Advice

- Develop test cases before coding (Test-Driven Development).
- Test incrementally (not everything together at once).
- Test simple parts first.
- Use assertions (built-in tests; "fail early"): Test pre- and post-conditions, and 'can't-happen' cases.
- Automate testing.
- Keep test software, data, and results (commit in repository).
- Re-test after making changes (regression testing).

Testing Terminology

Test case: control activation and input; observe response and output; decide on pass/fail.

JUnit Automated Testing Framework

JUnit: organizes code for test cases, runs them, reports results

See NetBeans IDE sample program Anagrams (via New Project).

Help > Javadoc References > JUnit API

Test case: method named `test...`

Facilities: `fail`, `assertTrue`, `assertEqual`, ...

Right-click Java file in NetBeans project: Tools > Create JUnit Tests

Can also test for required exceptions: no/wrong exception → failure

References

- *Code Complete*, 2nd Ed. by Steve McConnell. Microsoft Press, 2004.
- JUnit Testing Framework (integrated into the NetBEans IDE)

Software Engineering: Theory and Practice

NMA – National Student Academy
Summer Session 2008, Nida, Lithuania

Lecture 5

Tom Verhoeff

Technische Universiteit Eindhoven
Department of Mathematics & Computer Science
Software Engineering & Technology

Feedback to `T.Verhoeff@TUE.NL`

Topics

- Metrics
- Future Developments

GQM – TQM

- Goals (high-level, ultimate)
- Questions (specific)
- Metrics (how to)
- Total Quality Management (address quality everywhere)

Goals: What Do You Want to Achieve

- Quality control (objective vs. subjective, quantitative)
- Project planning & reporting
- Accountability (e.g. in case of an audit)
- Process improvement
- Measuring is not a goal in itself

Questions: What Do You Want to Know

- What is the quality of the system functionality?
- How much extra (time, staff) will it take to complete the system?
- What is the branch coverage of the tests?
- How complex is the system?
- How much time was spent on defect removal?

Metrics: What/How Do You Want to Measure

- What actually to measure?
- What measurement to use? Alternatives, trade-offs
- How to measure? (organize the measurement process)
- What to do with measurement results?

Measurement Process

1. Select metric(s) appropriate for goals and questions.

   Also consider: validation, calibration, tuning, staff training
2. Collect and store measurement data.
3. Consolidate and report: graphs, trend charts, . . .
4. Interpret results w.r.t. goals and questions.
5. React: close the feedback loop.

What to measure

- Product-related
- Process-related

Fundamental Metrics

- Size of product (LOC, SLOC, FP, ABC, . . .)
- Cost of project (Euro) [N.B. Not: product pricing]
- Duration of project (calendar months)
- Effort for project (person-months)
- Quality of product (number of remaining defects)
- Relationships, models
- Economy or diseconomy of scale: Effort = Size^e, $e < 1$ or $e > 1$

Size versus Complexity

- LOC, SLOC: (Source) Lines Of Code (SLOC = nonempty without comments)
- % of lines with comments
- # classes; # (public) methods or inst. var. per class
- # parameters or LOC per method
- depth of inheritance hierarchy
- # overridden methods
- % of duplicated code

Cyclomatic Complexity

- McCabe 1976
- Measure for testability, understandability (maintainability)
- # linearly independent paths in flow graph
- # edges − # vertices + 2 ∗ # components
- # binary decisions + 1
- Typical reasonable upper bound to impose per module: 10

Code Metrics Example

Python code:

```
1 p, q = 0, N   # given A[0..N)
2
3 while p <> q :
4 if A[p] : p = q
5 else : p = p + 1
6
7 if p == N : print "Not found"
8 else : print "Found at", p
```

LOC = 8; SLOC = 6
A, B, C = 4, 2, 3
Cyclomatic complexity = 4

Measurement Tools

- Configuration management tools
- Defect/issue trackers
- Test tools (also measure coverage)
- Static code analyzers (e.g. JDepend)

Benchmark

- Point of reference, specifically for program/processor performance
- SPEC: Standard Performance Evaluation Corporation `www.spec.org`
- Dhrystone → SPEC CINT2000 (without floating point)
- Whetstone → SPEC CFP2000 (with floating point)

Future Developments

- Increasing software/system complexity
- Further integration with hardware: sensors, robotics, 3D output
- Further standardization; certification of product, process, staff
- Higher-level formal models, e.g. for architecture, to generate code
- Dynamically evolving, self-adjusting, "organic" code
- Other kinds of processors: neural networks

Lasting Principles

- Rigor & formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incrementality

Looking Back

- Software (and) Engineering
- Product, Process, Project, Documentation
- Requirements Engineering
- Configuration Management
- Dealing with Errors in Engineering
- Verification by Review and by Testing
- Architecture
- Quarto by **N***ew* **M***ind* **A***pplications*

What Was the (Real) Goal?

Look beyond Planting the Flag

- Planting the flag is highly visible, but not the ultimate goal.

  Though the media would often like you to believe it is.
- Planting the flag brings you halfway, at best.
- The cost after planting the flag exceeds that of before.
- You must prepare for the second half before you depart.
- In software, planting the flag is the yell "it works".

  After that comes maintenance (and usually much more).

References

- "Applying the ABC Metric to C, C++, and Java" by Jerry Fitzpatrick. *C++ Report*, June 1997.