Software Evolution
CS 6/73902

Prof. Jonathan I. Maletic
Kent State University

Prerequisites

- CS 6/73901 Software Engineering Methodologies
- Strong programming background

Course Theme

- Tools and infrastructure to support the evolution and maintenance of large scale software systems
- Identification of specific software evolution tasks

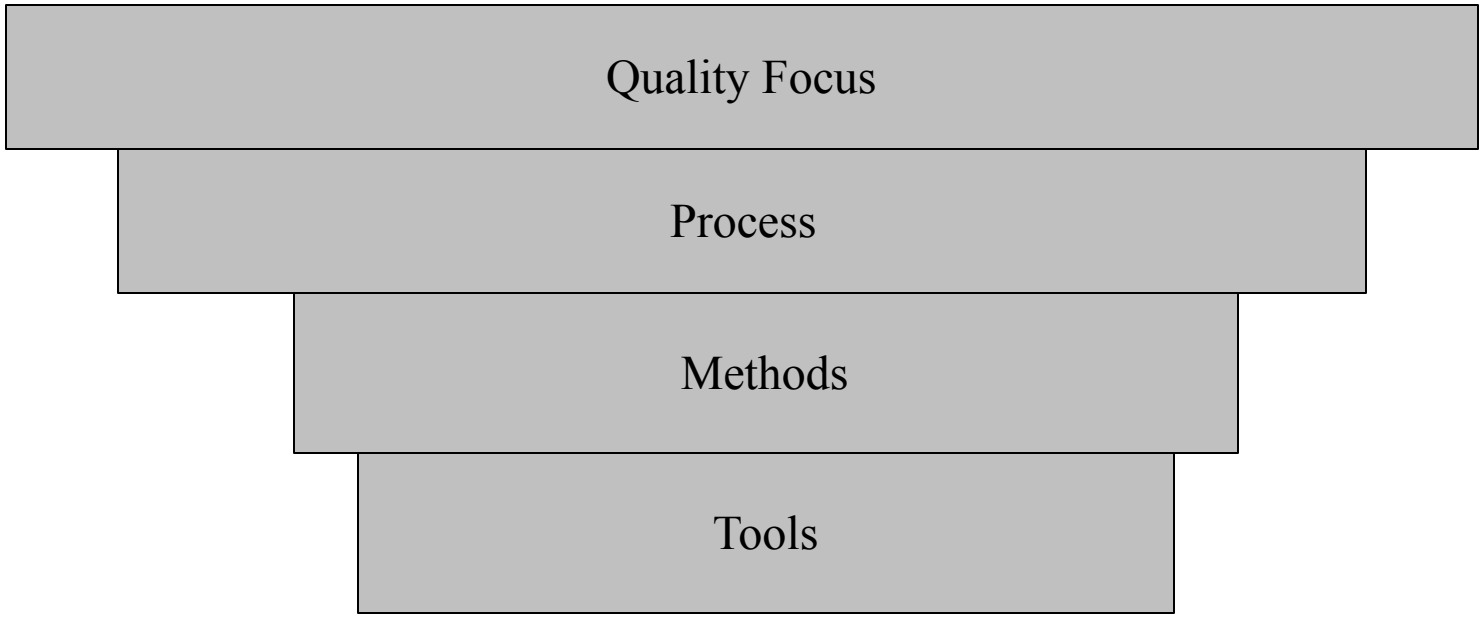
Course Topics

- Reverse Engineering, reengineering
- Impact analysis
- Design recovery
- Traceability
- Program understanding
- Refactoring and transformation

Development & Maintenance

- Development is implementation of a new software system.
- Maintenance consists of activities required to keep software operational after it is placed into production.
- Evolution is a process which changes an existing software so that it satisfies a new set of requirements.

Software Engineering



A Layered Approach

- Focus on quality (Power plant vs. Word processor)
- Process layer that enables rational and timely development of software (Agile, Spiral)
  - Key process areas must be established for effective delivery of software technology
- Methods provide support for process (OO)
- Tools provide support for methods (MSVC++)

Mini Process of Change

Change is the basic building block of both maintenance and evolution:

- Request for change
- Planning phase
- Software comprehension
- Impact analysis
- Implementation of the change
- Focus of the change
- Change propagation
- Verification
- Documentation

Definitions

- **Forward engineering** is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.
- **Reverse engineering** is the process of analyzing a subject system to (i) identify the system's components and their interrelationships and (ii) create representations of the system in another form or a higher level of abstraction.

Program Understanding

- Is a related term to reverse engineering. Program understanding implies that understanding begins with the source code while reverse engineering can start at a binary and executable form of the system or at high level descriptions of the design.
- The science of program understanding includes the cognitive science of human mental processes in program understanding. Program understanding can be achieved in an ad hoc manner and no external representation has to arise.
- While reverse engineering is the systematic approach to develop an external representation of the subject system. Program understanding is comparable with design recovery cause both of them start at source code level.

Redocumentation

- Is the creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternate views (for example, dataflow, data structures, and control flow) intended for a human audience.
- Redocumentation is the simplest and oldest form of reverse engineering, and can be considered to be an unintrusive, weak form of restructuring.

Design Recovery

- Is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself.
- Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains.

Software Comprehension

- The words "understanding" and "comprehension" are used as synonyms
- The Webster New World Dictionary:
  - com.pre.hend 1. to grasp mentally; understand 2. to include; take in; comprise.
  - un'der.stand 1. to perceive the meaning of 2. to infer 3. to interpret 4. to take as a fact 5. to learn 6. to know the nature, character, etc. of 7. to be sympathetic with.

Some Figures Available for Comprehension

- 50-90% of the maintenance time is spent in comprehension
- Fjelstad-Hamlen found that programmers:
  - studied the program 3.5 times longer than the documents
  - Spent more time in comprehension than in implementing enhancements

  - Fjeldstad, R.K., W.T. Hamlen, “Application Program Maintenance Study: Report to Our Respondents”, in G. Parikh, N. Zvegintzov, (Eds.), Tutorial on Software Maintenance, 1982, pp.13-30

Program comprehension is affected by

- Comprehensibility of the program
- Skills of the maintainer
- Information needs
- Information sources

Comprehensibility of the Program

- systems designed for comprehension, evolution, or reuse
  - easiest to comprehend
- legacy systems
- alien code

Skills of The Maintainer

- Novice
- Expert
  - Language expertise
  - Domain expertise
  - Comprehension strategies expertise
  - The specific program expertise

Information Needs

- Domain expert's view
  - Concepts of the domain and their relations
  - Scope and boundaries of the program
  - Goals of the program
- User's view
  - Operations of the program
  - Installation

Other Information Needs

- Programmer's view
  - Dependency graph
  - Program parts (classes, functions)
  - Their dependencies (call, use inheritance)
- Often not apparent from the code
  - Algorithms: How are the goals accomplished?
  - Representations: How are entities and relations of the domain reflected in the program?
  - Resource allocation: Memory size, timing

Information Sources

- Programmer's knowledge
- Code
- Documentation
- Colleagues on the project

Top-Down Strategy of Comprehension

- Domain expert's decisions:
  - Concepts, scope, etc.
- User's decisions:
  - Constraints, Operations, Installation options
- Programmer's decisions:
  - Program structure
  - Algorithms
  - Representations
  - Resource allocations

Process of Top- Down Comprehension

- Generation of hypotheses, subsidiary hypotheses
- Hypothesis is a guess of what the design decision was
  - H1
  - H2 (Subsidiary hypothesis)
  - H3
  - H4
  - H5
- The chain of hypotheses is verified; the hypotheses are either confirmed or discarded

Verifying Hypotheses

- Manually scan for beacons
  - example: Swap routine may indicate bubble sort
- Look for indicators of structures or algorithms
- While scanning code to verify current hypothesis, look for beacons for other hypothesis
  - avoid searching code later
- Help to form new subsidiary hypothesis
- Beacons may help refine hypothesis
- All code should be bound to hypotheses

Failure of Hypotheses

- Code does not support current hypothesis:
  - Discard old ones, create new ones

    H1
    H2
    H3

    H4 H6
    H5 H7
- Code supports contradictory hypothesis:
  - Incoherent code
- Code does not support any hypothesis

Bottom- Up Strategy of Comprehension: Chunking

- The components of chunking:
  - Aggregation
    - Several activities will be grouped together
  - Abstraction
    - A new abstract concept describes the group

Plans

- Sections of code with shared meaning
- Examples of plans:
  - Elementary programming techniques : swap, array traversal, etc.
  - Algorithms: bubble sort
  - Domain oriented plans: interest accrual
  - De-localization of plans:
  - Plans are scattered over a large portion of code, interleaved with each other
- De-localization makes comprehension harder

Opportunistic Comprehension

- Programmers freely switch between top-down and Bottom-up.
- Conjecture:
  - When very little information is known about a program (alien code), the programmer uses bottom-up comprehension.
  - When more is known, programmer uses top-down