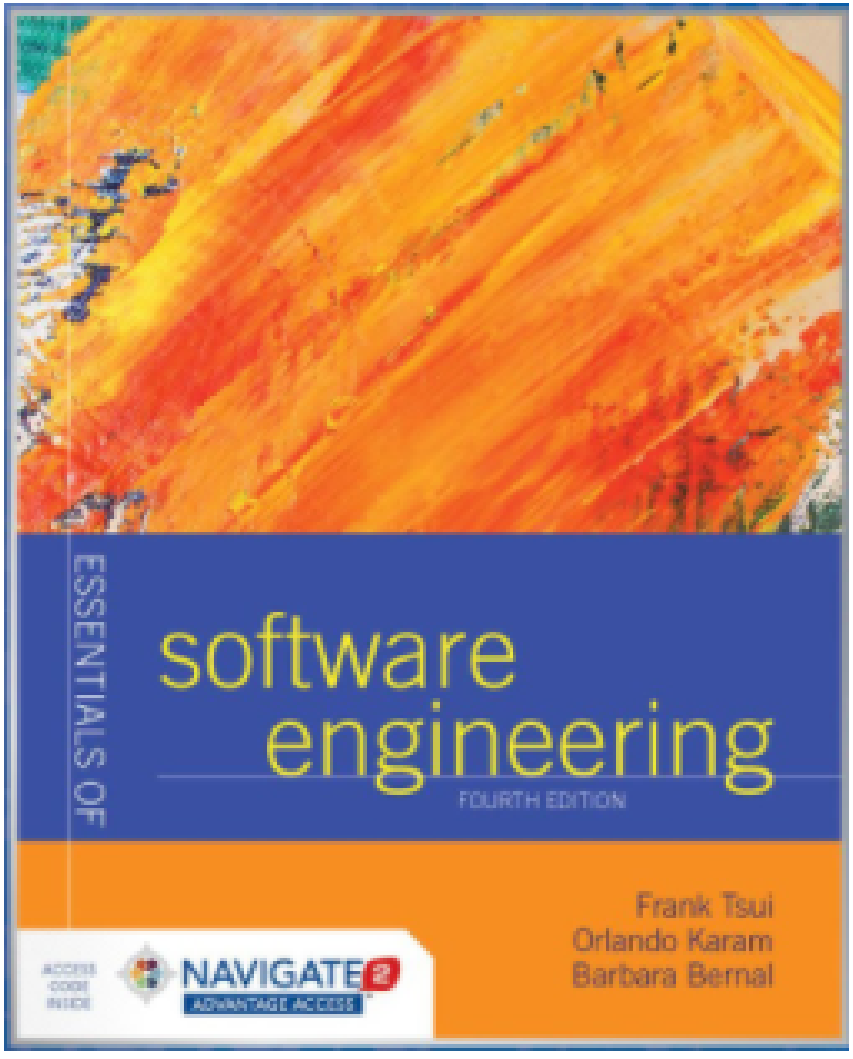
Implementation

CSC 355, Software Engineering II

Chapter 8: Design: Implementation



Implementation Topics

1. Describe:
   - characteristics of good implementations
   - best practices to achieve them
2. Understand the role of comments
3. Learn debugging techniques
4. Analyze refactoring
5. Plan for reuse

Introduction

- Implementation: transforming detailed design into a valid program
- Detailed design may be done as part of the implementation
  - Pro: faster
  - Con: less cohesive and less organized
- Includes: writing code, unit testing, debugging, configuration management

Characteristics of Good Implementations

- Readability - easy to read by others
- Maintainability - easy to maintain by others
- Performance - fast, efficient, load
- Traceability - according to RTM
- Correctness - according to RTM
- Completeness - according to RTM

Coding Guidelines

- Organization specific
- Important for consistency
- Programmers can get used to them easily
- Usually mandate:
  - Indentation style and formatting
  - Naming conventions (for files, variables, etc.)
  - Language features to use or avoid (pointers, warnings)

Style Issues

- Naming:
  - convey meaning
  - be consistent
  - rule of thumb: if you cannot think of a good name, then you might not understand the problem or the design can be improved
  - Internationalization (for example, what date is this? 4/6/2022)
- Word separation and capitalization
  - `snake_case`
  - `camelCase`

Style Issues (continued)

- Indentation and spacing
- Function size
  - when is it too big> when to break?
- File naming
- Error-prone constructs

Code Comments

- Types of comments:
  - explanation of the code -> refactor
  - marker in the code -> version control
  - summary of the code (flow of events)
  - description of intent
  - external references
- Document the “why”, not the “how”; the code itself is the “how”
- Make sure to keep the comments up-to-date.

Debugging

- Locating and fixing errors in code
- Errors noticed by testing, inspection, use
- Phases:
  - reproduction - specific test cases
  - localization - isolate source of bug
  - correction - fix without breaking more
  - verification - run test cases

Debugging (continued)

- Heuristics:
  - some routines will have many errors
  - routines with an error tend to have more
  - new code tends to have more errors
  - particular ones (for example language specific)

Debugging (continued)

- Tools
  - Code comparators
  - Extended checkers (linters)
  - Interactive debuggers
  - Debugging libraries
  - Others: profilers, test coverage, etc.

Assertions

- Pre-condition: condition a module requires in order to work
- Post-condition: condition that should be true if your module worked
- Assertion: executable statement that checks a condition and produces an error if it is not met
- Assertions are supported in many languages

Performance Optimization

- Performance trade-offs:
  - readability
  - maintainability
- Correctness is usually more important
- Profiler: runs a program an calculates how much time it spends on each part
- Cost-benefit analysis
- Measure before “optimizing”

Refactoring

- Goal: improve code quality without affecting its behavior
- Code "smells"
  - duplicated code
  - long function/method definitions
  - large classes
  - depth of control constructs
- Refactorings
  - change algorithms
  - extract functions
  - extract classes
  - use a different abstraction

Code Reuse

- Do not reinvent the wheel
- Take advantage of libraries
- Start with design patterns
- Design for reuse