Lecture 3
The Practice of Software Rengineering

Purpose: Look at specifics of reengineering the term project. The result of reverse engineering should be a thorough understanding of what the legacy system is supposed to do, and documented evidence of what it actually does. The artifacts of reverse engineering must document all the levels of abstraction from requirements, through design, to implementation and testing. A highly desirable goal from reverse engineering is functionally useful documentation that supports the maintenance process. The software configuration management discipline is required to establish the contents of the software configuration, naming conventions and traceability relationships. Software testing approaches are used to benchmark the legacy system, to establish what it actually does. The quality of the testing is related to how well the requirements of the legacy system are understood.

1. REQUIREMENTS DISCOVERY/RECOVERY – THE RAISON D’ETRE

- Domain Knowledge
  - Concept of data dictionary
  - User expectations
  - Contents of data dictionary
  - Phases in lifecycle where a DD is used
  - Commercial DD tools
  - Limitations of commercial tools

- Experiential Knowledge
  - User documentation
  - Hands-on usage
  - Combined with domain knowledge
  - Nuances of legacy system

- Static Examination
  - User documentation
  - Review of documentation
  - Review of design / code
  - Can you trust what you read?
2. REQUIREMENTS DISCOVERY/RECOVERY – A SYSTEMATIC APPROACH

- **SORDE Framework** – discover system from five perspectives
  - S = screens
  - O = user operations
  - R = Reports (printed, or suitable for printing)
  - D = database
  - E = exception handling
  - System Summary Matrix relates operations to screens, reports, database

- SCREENS = user visible input/output screens
  - Screens collect input from user – single or multiple inputs
  - Name screen (S1, S2, …)
  - Identify data items associated with screen
  - Classify data items as inputs/outputs

- OPERATIONS = user invocable operations
  - Name operation (O1, O2, …)
  - Menu provides navigation to user-selectable operations
  - Operation receives input(s) via screen
  - Operation transmits output(s) via screen
  - Operation may generate report(s)
  - Operation accesses/updates database
  - User operation is the **primary trigger** of data movement and data processing in system
  - Exception handling required for operations

- REPORT = printed/printable output
  - Report contains outputs
  - Name report (R1, R2, …)
  - Identify data items associated with report
  - Classify data items as inputs, retrieved, computed

- DATABASE = set of tables storing application data
  - Name each table (D1, D2, …)
  - Identify operations required for each table (some are R/W, W or R)
  - Exception handling required for operations
  - Data requirements for individual fields
    - Domain (type of value)
    - Range (min-max, length)
3. DESIGN DISCOVERY/RECOVERY – A HIERARCHICAL APPROACH

- ARCHITECTURAL DESIGN = components and their connections
  - Architecture Diagram
  - Architectural component = unit (class)

- Architectural Connections
  - Inheritance (not in DDU)
  - Declarative references (data types)
  - Function calls to a different object

- Reference Lists
  - CALLS: unit . method → unit . method
  - DATA (type or object) REFERENCES: unit . method → unit . data
  - Inheritance (not in DDU)
  - Declarative references (data types)

- Logical Design Abstractions
  - UML diagrams (object, collaboration diagrams)

- DETAILED DESIGN = component method design
  - Graphical Depictions
    - Call graphs – chain of potential calls
    - Black-box schematic – summarizes environmental impact of method execution

- Logic Design / Algorithm
  - Pseudo-code
  - PDL (program design language)
  - Decision table (truth tables)

- Metrics
  - McCabe -- complexity
  - Interface metrics (#arguments)

- Nuances = miscellaneous notes about the design that may not be clear from other documentation
  - Data or file structures (e.g., random/direct files)
  - Non-standard or advanced programming techniques (tricks) used
4. BASELINING – CONFIGURATION MANAGEMENT

- **WHY** = complete, accurate software configuration

- **Requirements**
  - Complete list of product types
  - Naming conventions -- Locate artifacts + Relate artifacts
  - Repository for storing artifacts
  - Ability to **freeze** version of artifacts

- **Traceability Across Levels of Abstraction**
  - Goal = move from artifact at one level of abstraction to related artifacts at next level of abstraction
  - Internal traceability links
  - External naming conventions
    - prefix or suffix indicating type of artifact
    - name portion identifies artifact
  - Granularity of artifact files
    - atomic vs composite
    - Each object method has its own artifact file vs only one file per object

- **Version Tracking**
  - Create new version upon making a change
  - Changes must be documented
    - Date + Reviser + Reason
  - Ability to **roll back** to earlier versions
  - UNIX provides Revision Control System (RCS)
  - Course repositories use RCS

- **Change Control**
  - Once baselined, changes follow strict process
  - Approval needed to modify
    - Change Control Board
    - Formal change proposal/request
      - Impact analysis
      - Cost estimate
    - Verification required to confirm **only** approved changes made

- **Reasons for Change**
  - Error in artifact (Primary, Direct)
  - Error in related artifact (Secondary, Indirect)
5. BENCHMARKING – PERFORMANCE vs EXPECTATIONS

- WHY = document what system does now

- Requirements define what system should do.

- Benchmarking uses requirements

- PRACTICAL APPROACH

  - Read up on data dictionaries
  - Use DDU to build DD from last semester’s class
  - Review DDU documentation to see details about system features beyond what you may use to build your DD
  - Follow the SORDE framework for requirements discovery
  - Use SORDE to organize requirements
  - Document requirements
  - Devise strategy for testing each requirement in the **nominal** and **exceptional** case.
  - Organize testing effort
    - Multiple test sessions (suites)
    - Each suite contains multiple test runs
    - Test suites may focus on one usage mode (nominal or exceptional) or system subset.
    - Test scripts/procedures developed for each test run
    - Test scripts contain test cases that include user action and expected system result
  - Perform the benchmarking test
    - Document each test run
    - Naming conventions
    - Analyze test results before concluding system deficiency

---

You can’t **reengineer or test what you don’t understand. Don’t get the cart before the horse.**