ELEC 876: Software Reengineering
(Terminology)

Dr. Ying Zou

Department of Electrical & Computer Engineering
Queen’s University
Software Life-Cycle – General Taxonomy

• Software production is composed of:
  – Software development
    • Requirements
    • Specification
    • Design
    • Implementation
    • Testing
  – Software maintenance

• Some researchers and practitioners use *software evolution* as a preferable substitute for maintenance
Software Maintenance

- **Software Maintenance** is the “modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment” [ANSI/IEEE Std. 729-1983]
- Software maintenance is classified as:
  - Adaptive
  - Corrective
  - Perfective
Software Maintenance – Classification

- **Adaptive maintenance**: Changes are made in response to changes in the environment the product operates.
- **Corrective maintenance**: Changes are made for the removal of faults, leaving the specifications unchanged.
- **Perfective maintenance**: Changes are made for the improvement of product effectiveness (e.g., additional functionality, decreased response time).
Software Maintenance – Classification (Con’t)

Corrective maintenance (17%)
fixing reported errors in the software

Adaptive maintenance (18%)  
adapting the software to a new environment (e.g., platform or O/S)

Perfective maintenance (65%)  
implementing new functional or non-functional requirements
Software Maintenance – Approximate Relative Costs

- Requirements: 2%
- Specification: 5%
- Design: 6%
- Implementation: 12%
- Integration: 8%
- Maintenance: 67%
Why is Software Maintenance Expensive?

• Costs can be high because:
  – Maintenance staff are often inexperienced and unfamiliar with the application domain
  – Programs being maintained may have been developed without modern techniques; they may be unstructured, or optimized for efficiency, not maintainability
  – Changes may introduce new faults, which trigger further changes
  – As a system is changed, its structure tends to degrade, which makes it harder to change
  – With time, documentation may no longer reflect the implementation
Factors Affecting Maintenance

- Module independence
- Programming language
- Programming style
- Program validation and testing
- Quality of documentation
- Configuration management techniques
- Application domain
- Staff stability
- Age of program
- Dependence on external environment
- Hardware stability
Lehman’s Laws of Evolution

• A classic study by Lehman and Belady (1985) identified several “laws” of system changes.

• Continuing change
  – A program that is used in a real-world environment must change, or become progressively less useful in that environment

• Increasing complexity
  – As a program evolves, it becomes more complex, and extra resources are needed to preserve and simplify its structure
History of Eclipse

• 1997 – IBM VisualAge for Java (implemented in small talk)

• 1999 – IBM VisualAge for Java micro-edition (Eclipse code based from here)

• 2001 – Eclipse (change name for marketing issue)

• 2003 – Eclipse.org foundation

• 2005 – Eclipse V3.1

• 2006 – Eclipse V3.2
History of Microsoft Word

• 1983 – MS Word for DOS
• 1985 – MS Word for Mac
• 1990 – MS Word for Windows
• 1991- MS Word 2
• 1993 – MS Word 6
• 1995 – MS Word 95
• 1997 – MS Word 97
• 1998 – MS Word 98
• 2000 – MS Word 2000
• 2002 – MS Word XP
• 2003 – MS Word 2003
Staged Model of Software Lifecycle

• Initial development
  – Deliver the first version of software
  – May lack some features
  – Possesses the architecture and knowledge

• Evolution
  – Take place when the first version is successful
  – Adapt the application to user requirements and operating environments
  – Keep architecture integrity while introducing changes
Staged Model of Software Lifecycle (con’t)

• Servicing (stage of code decay)
  – loss of architecture coherent
  – Loss of knowledge triggered by loss of key personnel

• Phase-out
  – No more servicing is undertaken
  – Software may be in production

• Close-down
  – Software use is disconnected
  – Users are directed towards a replacement
The Versioned Staged Model

Initial development

first running version

evolution changes

Evolution Version 1

Servicing Version 1

servicing patches

evolution of new version

evolution changes

Phase-out Version 1

Evolution Version 2

Servicing Version 2

servicing patches

evolution of new version

Phase-out Version 2

Evolution Version . . .

Close-down Version 1

Close-down Version 2
What is a Legacy System?

- A legacy system is a large software system
- They are old, often more than 10 years old
- They are written in legacy languages (e.g., COBOL), and built around legacy database services
- Legacy systems are autonomous, and mission critical
- They are inflexible and brittle
- They are responsible for the consumption of at least 80% of the IT budget
Problems of Legacy Systems

• Availability of original developers
• Lack of documentation
• Size and complexity of the software system
• Accumulated past maintenance activities
• Volatile user environment
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
  – identify the system’s components and their interrelationships and
  – create representations of the system in another form or at a higher level of abstraction.”

• “**Reengineering** ... is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.”

  — Chikofsky and Cross [in Arnold, 1993]
Reverse and Reengineering

Requirements

Reengineering

New requirements

Designs (models)

System (software)
Reverse Engineering Techniques

- **Re-documentation** is the creation or revision of a semantically equivalent representation within the same relative abstraction level.
  - pretty printers
  - diagram generators
  - cross-reference listing generators

- **Design recovery** recreates design abstractions from a combination of code, existing documentation (if available), personal experience, and general knowledge about problem and application domains.
  - software metrics
  - browsers, visualization tools
  - static analyzers
  - dynamic (trace) analyzers
Example Design Recovery tool: BPS
Example Redocumentation tool: Rigi
Example: Design Recovery

OrderAccessBean abOrder =
   new OrderAccessBean();
Vector vOrderItems =
   abOrderJ.findStaleOrderItems();

// Turn the Vector into an Enumeration for
// performance considerations
Enumeration enumOrderItems =
   vOrderItems.elements();
getCommandContext().getTransactionCache().flush();

try {
   TransactionManager.commit();
}
catch (Exception ex) {
   throw new
   ECSysytemException(ECMessage._STA_COMM
   IT_DB_FAILURE, iClassName, ..., ex);
}

DeallocateInventoryCmd
deallocateInventoryCmd =
   (DeallocateInventoryCmd)
CommandFactory.createCommand(...);
Reverse Engineering Techniques (con’t)

- **Restructuring** is the transformation from one representation form to another at the same relative abstraction level, while preserving the system’s external behaviour
  - automatic conversion from unstructured code to structured code
  - source code translation
- **Data reengineering** is the process of analyzing and reorganizing the data structures (and sometimes the data values) in a system to make it more understandable
  - integrating and centralizing multiple databases
  - unifying multiple, inconsistent representations
  - upgrading data models
Reverse Engineering Techniques (con’t)

• **Refactoring** is restructuring within an object-oriented context
  – Misuse of inheritance
    • change inheritance to delegation if the subclass doesn’t use attributes
  – Missing inheritance
    • duplicated code, and case statements to select behaviour
  – Misplaced operations
    • unexploited cohesion — operations outside instead of inside classes
  – Violation of encapsulation
    • explicit type-casting, C++ “friends” ...
  – Class misuse
    • lack of cohesion — classes as namespaces
Tool Architectures

• Most tools for reverse engineering, restructure reengineering use the same basic architecture
Goals of Reverse Engineering

• Cope with complexity
  – need techniques to understand large, complex systems

• Generate alternative views
  – automatically generate different ways to view systems

• Recover lost information
  – extract what changes have been made and why

• Detect side effects
  – help understand ramifications of changes

• Synthesize higher abstractions
  – Create alternative views that transcend to higher abstracts of software

• Facilitate reuse
  – detect candidate reusable artifacts and components
    — Chikofsky and Cross [in Arnold, 1993]
Software Life-Cycle Schematic (Reengineering View)

Software Reengineering Terms and Relationships
Reengineering Objectives

• The main objective of reengineering is:
  – Improve the understanding of a software system through analysis
  – Capture and preserve the existing knowledge for a software system through analysis and modeling
  – Improve the software for increased maintainability, reusability or evolvability, through alteration at the code, architecture, or design level
Reengineering Terms

• Synonyms for reengineering
  – Improvement
  – Renewal
  – Renovation
  – Refurbishing
  – Redevelopment engineering
  – Modernization
  – Reclamation
  – Reuse engineering
Why Reengineering?

• Pros:
  – build on existing software than embarking on a high risk re-implementation
  – reduce maintenance costs
  – evaluate software assets
  – integrate strategic legacy applications with new software packages

• Cons:
  – Reengineering projects if not carefully planned and technically justified can be very high risk projects
  – Technical and business success criteria often are not in alignment
  – There are not generic success criteria for a reengineering project
Reengineering – Stepwise Approach

• Step 1: Reverse engineering:
  – Recover existing application’s process logic and design to a more abstract specification model that accurately reflects the system’s current capability

• Step 2: Revising the specification model:
  – Remove obsolete functionality
  – Upgrade the model to accommodate interfaces with new technologies
  – Validate and document any changes
Reengineering – Stepwise Approach (con’t)

• Step 3: Forward engineering:
  – Reuse the stable and robust components of the existing software
  – Design and implement the new components in a way that is provable to reduce maintenance costs
  – Perform regression testing to validate the overall reengineering effort
Software Reengineering – Scenarios

- Changes in operating systems or hardware platforms
- Data type mismatch in expressions (i.e., data type mismatch problems not handled by the compiler)
- Appropriateness of data structures used (e.g., all fields of a structure are used)
- Detection of inefficient and error prone code (high complexity, high complex module interaction)
- Memory allocation / deallocation (memory leaks)
Software Reengineering – Scenarios (con’t)

• Code flow analysis and non initialized data (control/data flow, value ranges, constant propagation)
• Reengineering high complexity and memory consuming algorithms
• Information hiding, variable naming, documentation
• Excessive fan-out (i.e., excessive number of function calls per block or module)
• High coupling (i.e., high interface complexity between modules)
Software Reengineering Process Models

• The goal of software reengineering is
  – to take an existing system and generate from it to form a new system

• To ensure a higher success rate, the software reengineering should follow a well-planned process to accomplish a reengineering task

• A process model is a purely descriptive representation of process, in terms of key activities and their relationships
Software Reengineering Process Models (con’t)

• Three well known approaches to model reengineering process model
  – The horseshoe process model
  – Goal driven process model
  – Incremental Process model
The Horseshoe Process Model

http://www.sei.cmu.edu/reengineering/horseshoe_model.html
The Horseshoe Process Model (con’t)

• Distinguishes different levels of reengineering analysis
• Provides a foundation for transformations at each level, especially for transformations to the architecture level
• There are three basic reengineering processes
  – Analysis of an existing system
  – Logical transformation
  – Development of a new system
The Horseshoe Process Model (con’t)

• The horseshoe process model can be divided into four levels of abstractions that represent the logical structure of the subject system
  – *Source level*, which is source code in textual representation.
  – *Code-Structure Representation*, which includes source code and artifacts such as abstract syntax trees (ASTs) and flow graphs obtained through parsing and analytical operations.
  – *Function-Level Representation*, which describes the interrelations between the program artifacts, such as functions, data structures, function calls, and data references.
  – *Concept*, which represents the system in terms of subsystems, and their interactions. The subsystem is a cluster of the functions and data structure and fulfills one specific task for the system.
Goal Driven Process Model

- Non-functional requirements define system properties, constraints and software qualities of the system being developed
  - such as reusability, maintainability, performance, portability and security
- However, for existing legacy systems, system properties, and qualities are constantly deteriorating and deviating from their original specifications due to prolonged maintenance and technology updates.
- Software re-engineering activities should not occur in a vacuum, and it is important to incorporate non-functional requirements in the re-engineering process.
- The reengineered system may conform to specific target objectives
  - such as better performance and higher maintainability.
Goal Driven Process Model (con’t)

1. AST Annotation
2. System Modeling / Analysis
3. Transformation Rule Application
4. Quality Goal Modeling and Representation
5. Quality Evaluation
6. New System
7. Target System

- Parse
- Original System
Incremental Process Model

• It can be monumental task to reengineer an entire code base in one sweep
• Incremental process model is provided to eliminate the risk and complexity involved in reengineering a large system
  – The process model is divided into phases
  – A software system is decomposed into a set of manageable components
  – Each component is reengineered at a phase
• At each phase, the reengineering process can adopt the horseshoe process model or goal driven process model
Incremental Process Model (con’t)
Possible Project Topics

- Case studies on the clones and their evolution in mobile apps
- Case studies on the design patterns/antipatterns in mobile apps
- Case study on the evolution of web systems
- Licensing issues in the code reuse
- Techniques for detecting semantic clones
- Architecture recovery and their evolution
- Code migration
- Change impact analysis and estimation