Software Re-Engineering: Technology Maturation Study

Marwan Abi-Antoun

17-939A: What Makes Good Research in Software Engineering?
Professor Mary Shaw
Spring 2005
The Case for Re-engineering

Once upon a time there was a Good Software Engineer whose Customers knew exactly what they wanted. The Good Software Engineer worked very hard to design the Perfect System that would solve all the Customers’ problems now and for decades. When the Perfect System was designed, implemented and finally deployed, the Customers were very happy indeed. The Maintainer of the System had very little to do to keep the Perfect System up and running, and the Customers and the Maintainer lived happily every after.

Rejuvenate Your Old Systems*

• Alternatives for legacy systems?
  – Abandon old systems when maintenance becomes too difficult?
  – Stick to *status quo* and live with low-quality?
  – Design the *Perfect System*?

Re-engineering in Perspective

• Software re-engineering is necessary
  – Correct eroding structure
  – Address changes in requirements, environments, ...

• Wide gap between research, education and practice
  – S.E. education emphasis on “forward engineering”
  – Re-engineering projects second-class citizens
  – No scalable solutions for industrial-sized software
“Reengineering is like looking at a Picasso and trying to come up with a photograph of the subject.”

(Vaughan Merlyn)
Some Definitions

- **Reverse Engineering**: examination of a subject system without the intent of changing it or creating a new system.
- **Redocumentation**: recovering design documentation about the subject system.
- **Restructuring**: transformation from one representation form to another at the same relative abstraction level.
- **Reengineering**: examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.
The Definitions Illustrated

Figure 1. Relationship between terms. Reverse engineering and related processes are transformations between or within abstraction levels, represented here in terms of lifecycle phases.

Other Terms

- Refactoring
- Round-trip engineering
- Re-code
- Re-design
- Modernization
- Renovation
- Salvaging
- Reuse
General Model for Software Re-engineering

Reverse Engineering (Abstraction) → Conceptual Requirements → Design → Implementation → Target System

(Alteration) → Conceptual Requirements → Design → Implementation

re-think → re-specify → re-design → re-build

Forward Engineering (Refinement) → Existing System

General Model for Software Re-engineering

Re-engineering Timeline Closely Fits Higher Abstraction Levels

- 1950: GOTO less approach
- 1960: Structuring Flowgraphs
- 1970: Module Clustering
- 1980: Data Re-Engineering
- 1990: Architecture Recovery
- 2000: Re-Architect

- 1950: Re-Code
- 1960: Re-Modularize
- 1970: Design Recovery
- 1980: Module Clustering
- 1990: Architecture Recovery
- 2000: Re-Engineer
Hybrid Re-engineering

Software Re-engineering. Prepared By Dr. Linda H. Rosenberg. Engineering Section head Software Assurance. Technology Center Unisys Federal Systems
http://satc.gsfc.nasa.gov/support/reengrpt.pdf
Re-engineering for Distribution

Re-engineering for Quality X

What to Re-engineer?

• Module View:
  – Affects code structure of the system
  – Improves development process
    • E.g., Build time for Sync-and-Stabilize

• Component-and-Connector View:
  – Affects runtime structure of the system
  – Improves some runtime qualities
Module View Re-engineering

- **Possible Uses:**
  - Improve compile-time dependencies (*)
- **Approaches:**
  - Automated Clustering or Partitioning
  - Metrics to quantify complexity of resulting partition
- **Difficulties:**
  - Avoid “Pot-pourri” modules
  - Stability with incremental changes

Deriving a System Description

1. Map resource exchange among modules
2. Derive a hierarchical design

Figure 3. A resource-flow diagram of modules.

Figure 4. A resource-structure diagram based on (a) maximum control; (b) minimum coupling; and (c) using our restructuring algorithm.

Source: Choi, S.C.; Scacchi, W.; Extracting and restructuring the design of large systems. In IEEE Software, Volume 7, Issue 1, Jan. 1990 Page(s):66 - 71
Re-Modularization

Evolution of “Build Architecture”

Restructuring to decrease build time

Estimated incremental recompilation time

Restructured = removed false dependencies from header files

Componentized = re-modularized

Break down the required recompilation time as a line is changed per incremental build.

Compuware OptimalAdvisor
(Package Structure Analysis Tool)

Source: http://javacentral.compuware.com/pasta/
Refactoring available in IDEs

- Refactoring actions rewrite source code
  - Within a single source file
  - Across multiple interrelated source files
  - Preserve program semantics

Source: http://www.eclipse.org/eclipse/presentation/eclipse-slides.ppt
Eclipse Refactoring

- Full preview of all ensuing code changes
  - Programmer can veto individual changes

List of changes

“before” vs. “after”
FUJABA

• Recognize instances of design patterns
• Incremental, top-down/bottom-up, iterative
• From UML to Java and Back Again
  – UML class and collaboration diagrams
  – Method behaviors as graph transformation rules
  – Java code generation
• Case Studies:
  – Abstract Window Toolkit (AWT): 110K+ LOC
  – “Our system deals comfortably with 100,000 lines of code”
**FUJABA:** Specify Pattern Rules

- An environment for roundtrip engineering with UML, Java and design patterns based on graph transformation

FUJABA: Analyze Implementation

Source: http://wwwcs.uni-paderborn.de/cs/fujaba/projects/reengineering/index.html
Architectural Refactoring

Figure 7: Substitution.

Architecture Transformation

Graph-to-Graph Transformation

Graph-to-Graph Transformation

Reflexion Models

General Problem

1. Source (Hierarchical/Nested) Graph
2. Target (Hierarchical/Nested) Graph
3. Correspondence Graph
   - Establish mapping between subgraphs of source and target graph
   - A correspondence node is connected to the nodes of these subgraphs by correspondence edges
   - Correspondence nodes may be mutually connected by edges representing dependencies between mapping decisions
4. Apply Graph Techniques
   - E.g., Triple Graph Grammars
General Problem Illustrated

Research Challenges

• Scalability
  – Toy Examples for good reason!
  – Many algorithms NP-Complete:
    • Graph Partitioning
    • Sub-graph matching (to recognize design patterns)

• Formalisms
  – How to make heavy dose of formalism (e.g., Graph Grammars) palatable to practitioners?
  – Formalisms to deal with “triple graph”
The Importance of Types

• Having type information is helpful
  – Database schemas (types) always there, from modeling to implementation
  – Data Re-engineering matured much earlier!

• Where are the types
  – UML Class diagrams
  – Good luck finding “architectural types” in your code
  – Researchers represent Architectural Styles differently
    • Graph Grammar
    • Predicates/Constraints
An Ideal Transformation

Type Graph

Instance Graph

Source: From Graph Transformation to Software Engineering and Back
Current Trends

- Aspect-Oriented Programming
  - Perceived as less invasive
  - You may need to re-engineer your concerns!
  - Conceptual Modules

- Retaining Architectural Type Information
  - ArchJava
Questions?