Teaching CS Concepts through Reverse Engineering

Adventures in Pedagogy and Binary Exploitation

G Leaden
G.Leaden1@Marist.edu

Intro Lecture

Marist College
School of Computer Science and Mathematics
Poughkeepsie, NY 12601

Advisor: Alan G. Labouseur
Alan.Labouseur@Marist.edu

What is Reverse Engineering?

Hardware

- Deciphering designs from finished products to…
    - Improve your own products
    - Analyze a competitors products

"the process of developing a set of a specifications for a complex hardware system by an orderly examination of specimens of that system"
- Michael George Rekoff

Software

- Gain a sufficient design-level understanding to…
    - **Aid** maintenance
        - Design recovery
        - Re-Documentation
    - **Strengthen** enhancement
        - Identify and patch exploits, unintended side effects, bugs
        - Security **defense** and **offense**
    - **Support** replacement
        - Restructuring
        - Reuse

Reverse Engineering Taxonomy Eliot **Chikofsky**, James **Cross II**

Requirements

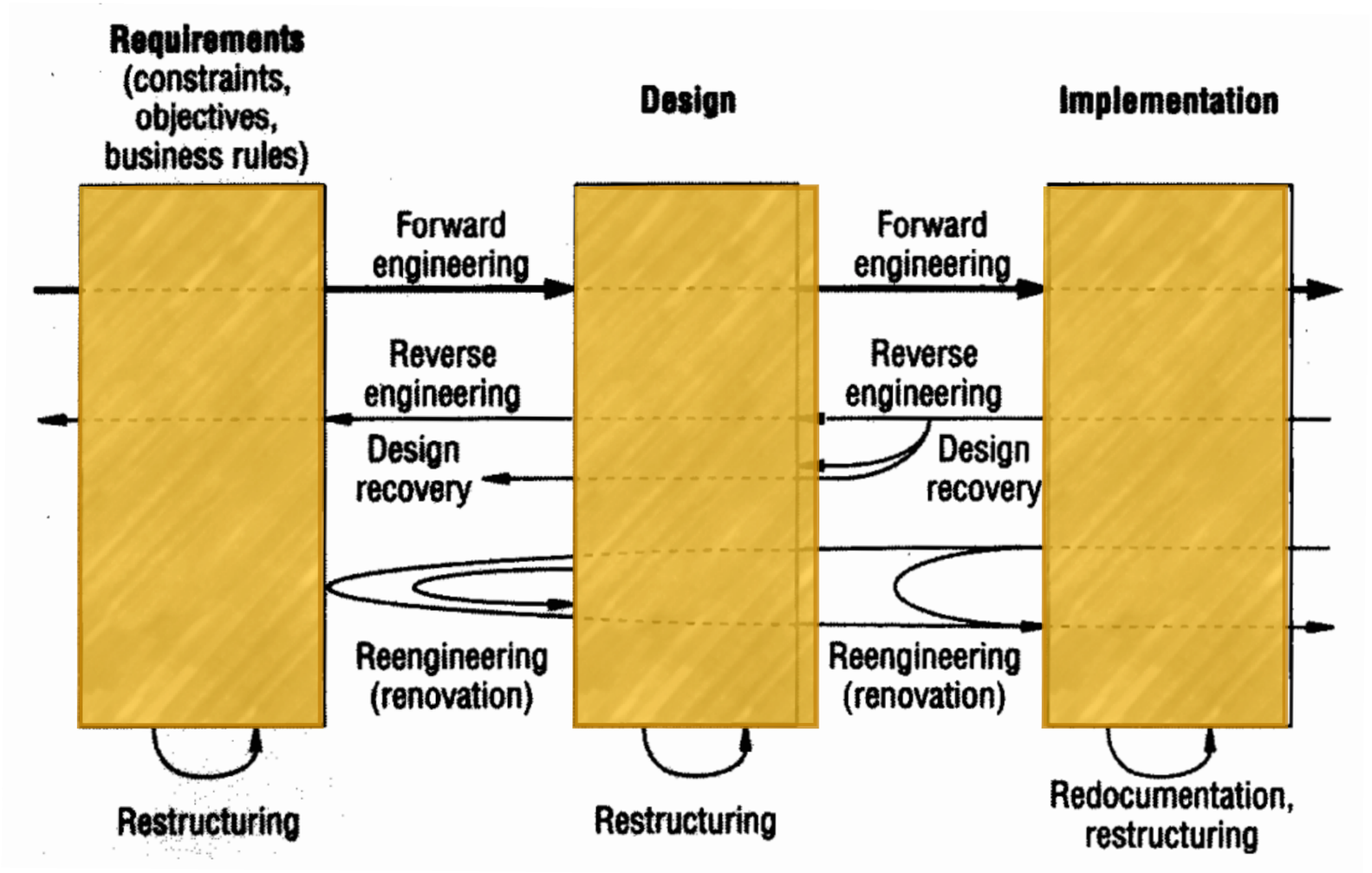
- Specification of the problem being solved, including objectives, constraints, and business rules

Design

- Specification of the solution

Implementation

- Coding, testing, and delivery of system

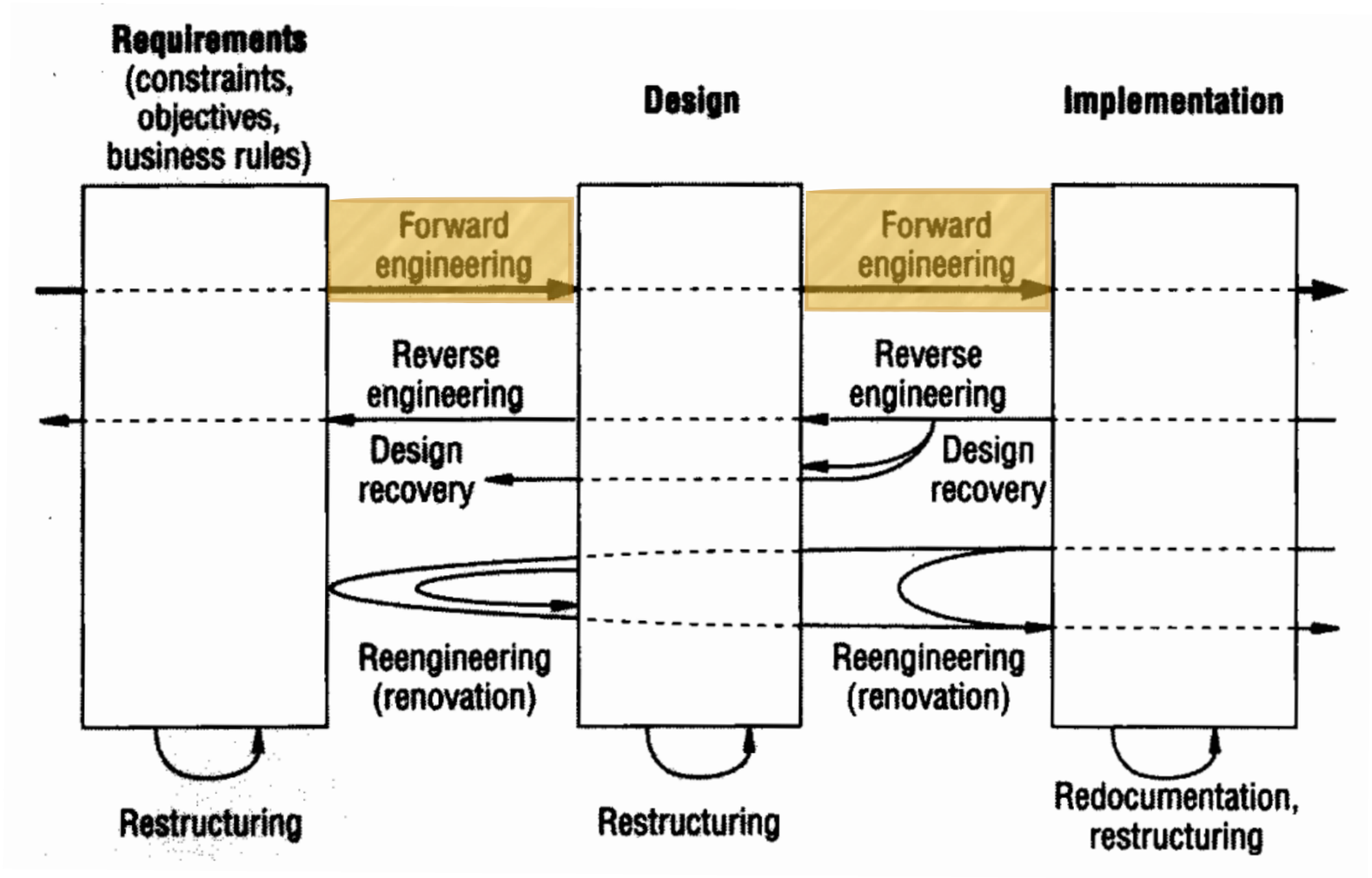


Reverse Engineering Taxonomy Eliot **Chikofsky**, James **Cross II**

Forward Engineering

- Moving **from** high level abstractions and logical implementation-independent designs **to** physical implementation

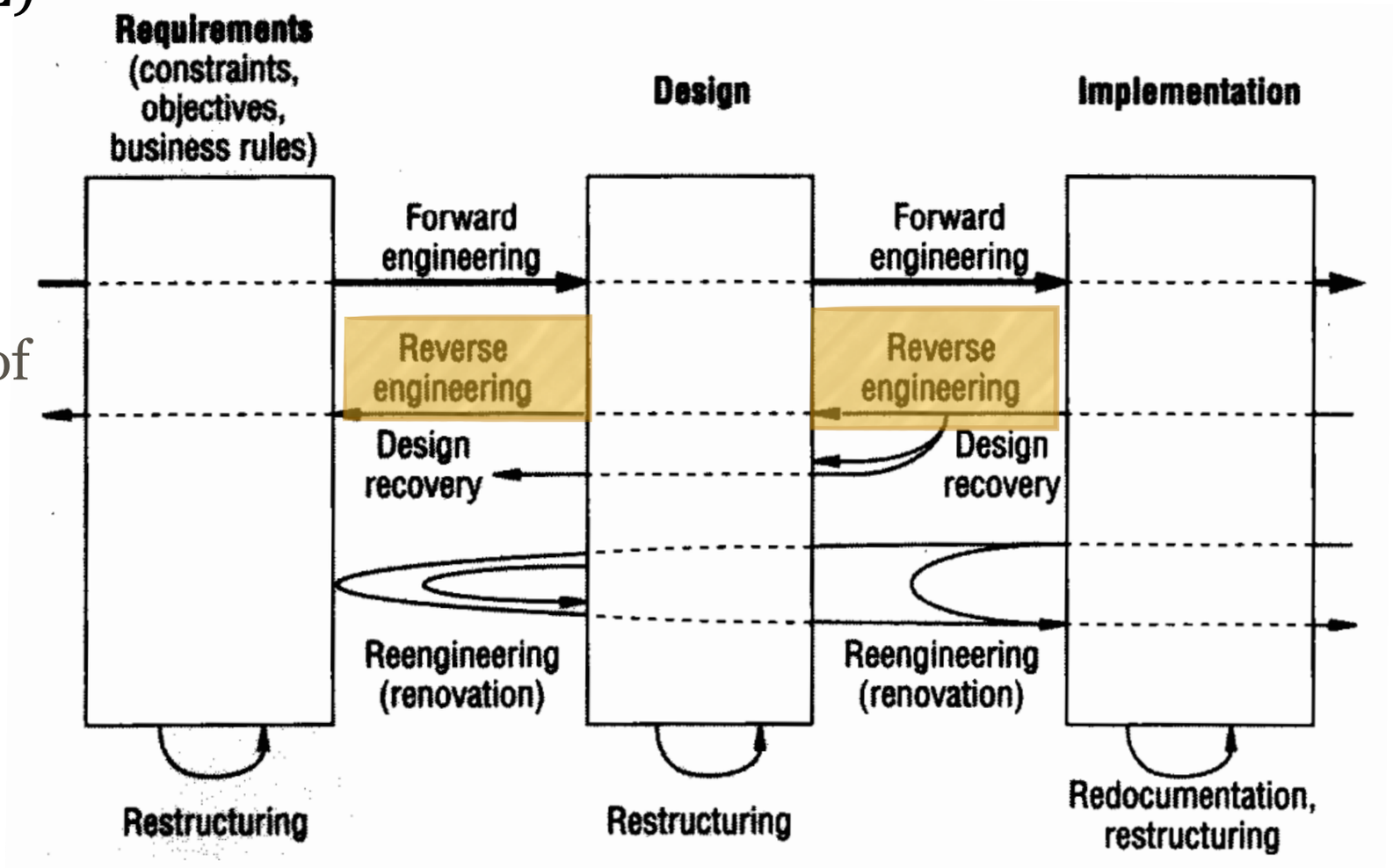
- Inverse of Reverse Engineering



Reverse Engineering Taxonomy Eliot **Chikofsky**, James **Cross II**

Reverse Engineering (RE)

- Analyzing a system to:
  - Identify the systems components and their interrelationships
  - Create representations of the system in another form or at a higher level of abstraction
- Synthesizing abstractions that are less implantation dependent
- Can be done from any level of abstraction and at any stage in the software engineering life cycle

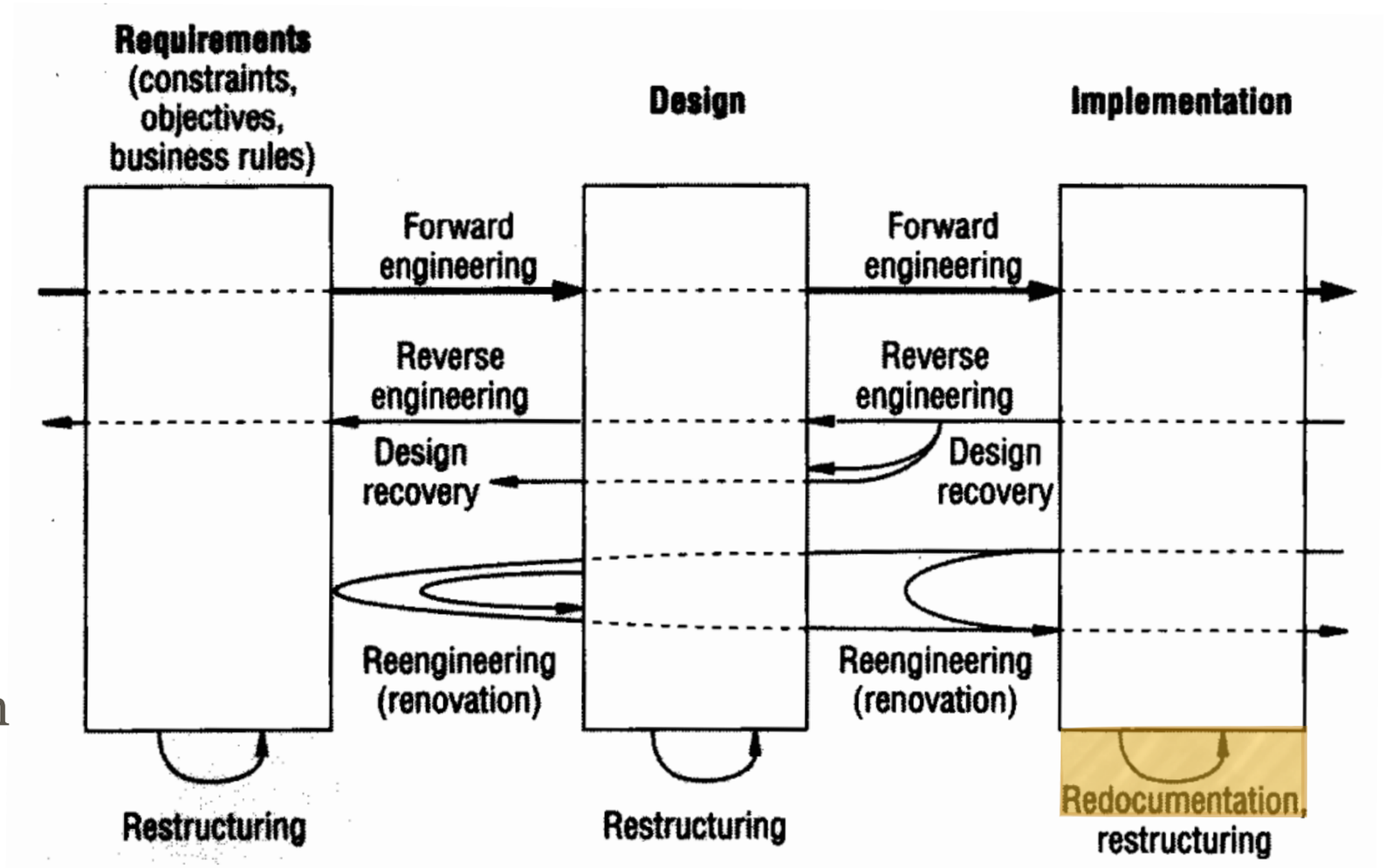


Reverse Engineering Taxonomy Eliot **Chikofsky**, James **Cross II**

Forms of RE

Redocumentation

- Simple, Old
  - Unintrusive and weak form of **restructuring**
- Creation or revision of a semantically equivalent representation with the same relative abstraction level
  - Creating documentation
    - Class Diagrams
    - Functional Diagrams
    - README
    - etc

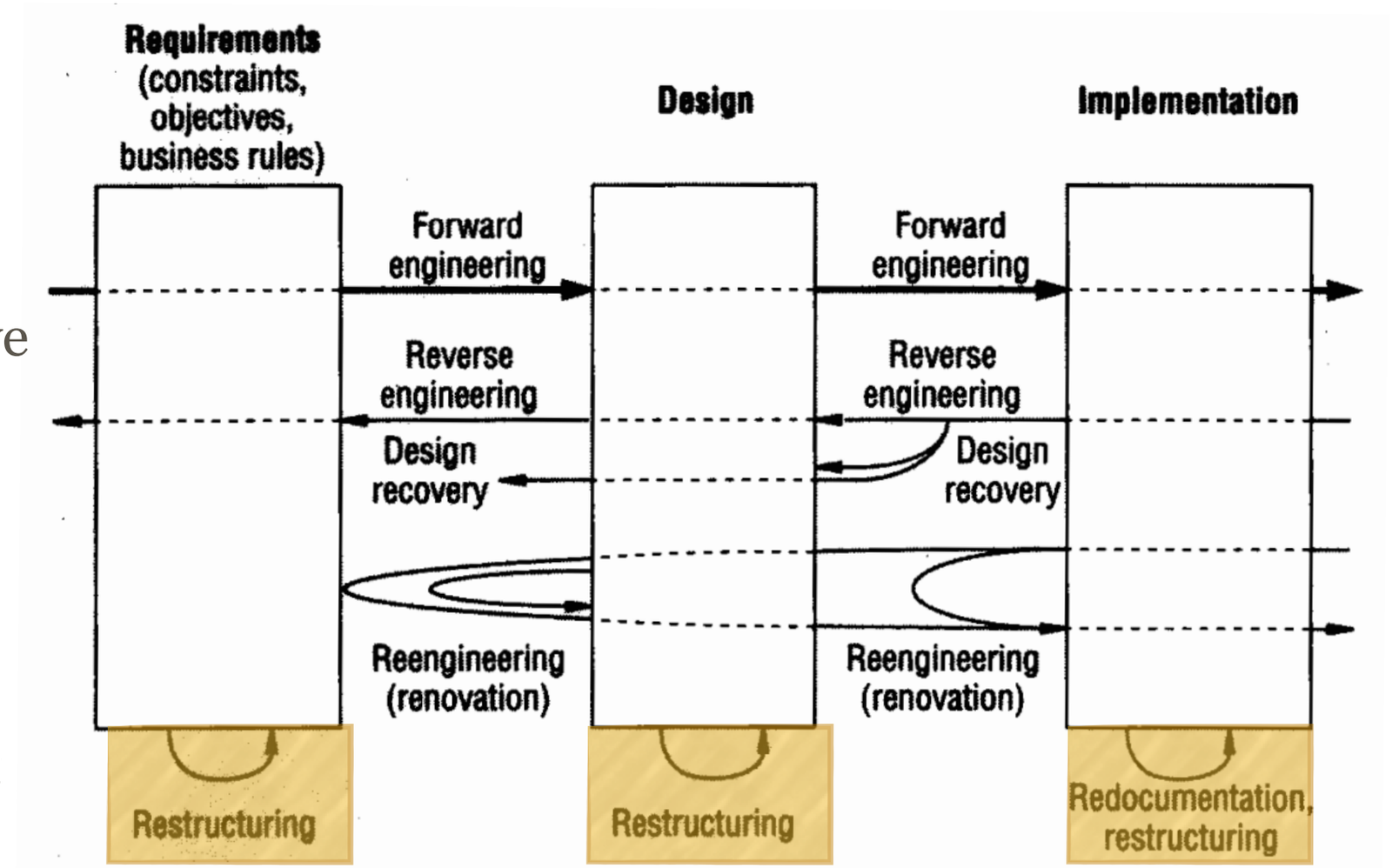


Reverse Engineering Taxonomy Eliot **Chikofsky**, James **Cross II**

Forms of RE

Restructuring

- Transformation from one representation form to another at the same relative abstraction level, *while preserving external behavior*
- Altering code to improve structure
- May lead to better observations of the subject system that could lead to the suggestion of changes that would improve aspects of the system
  - Should not change overall functionality of system itself

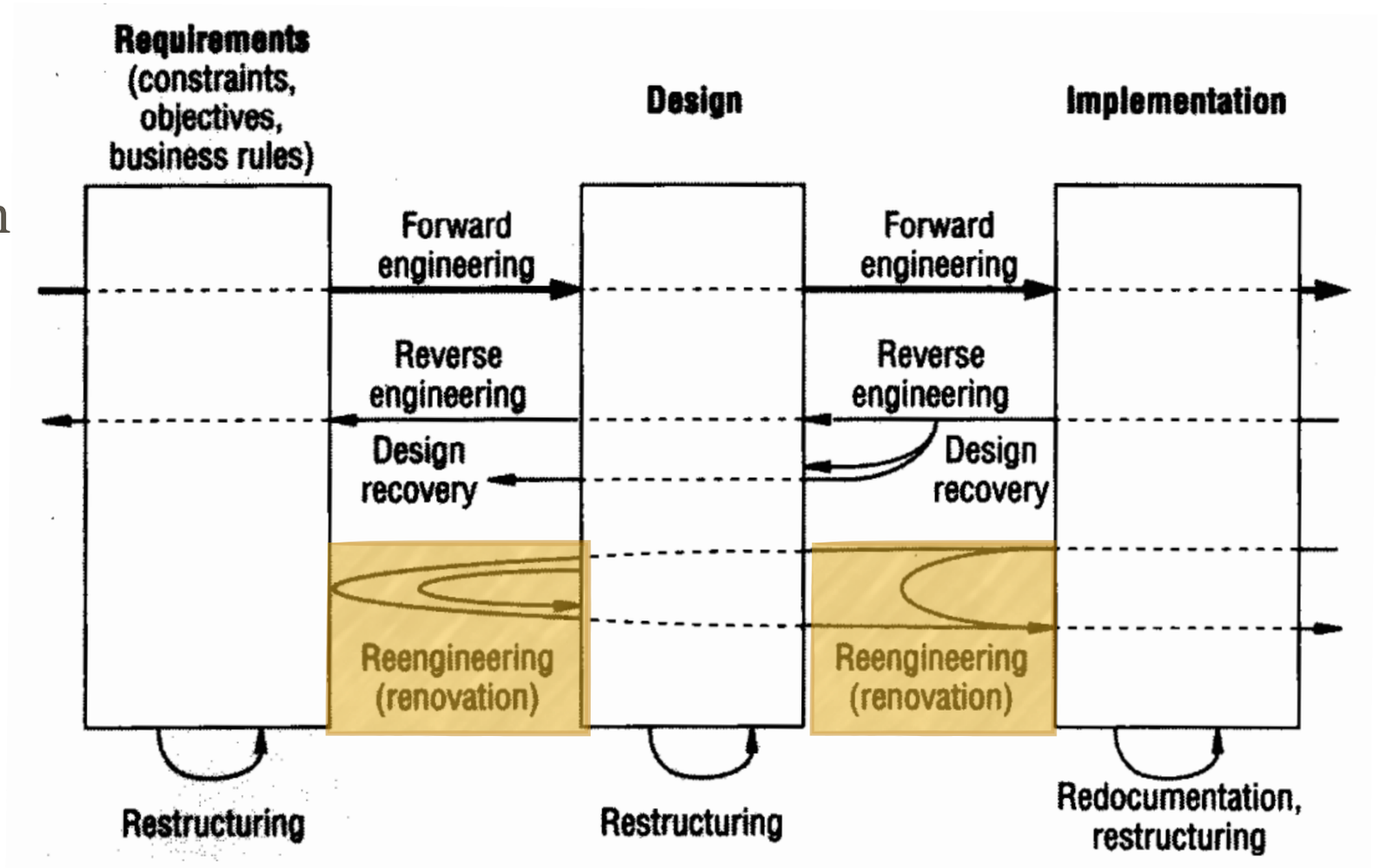


Reverse Engineering Taxonomy Eliot **Chikofsky**, James **Cross II**

Forms of RE

Reengineering

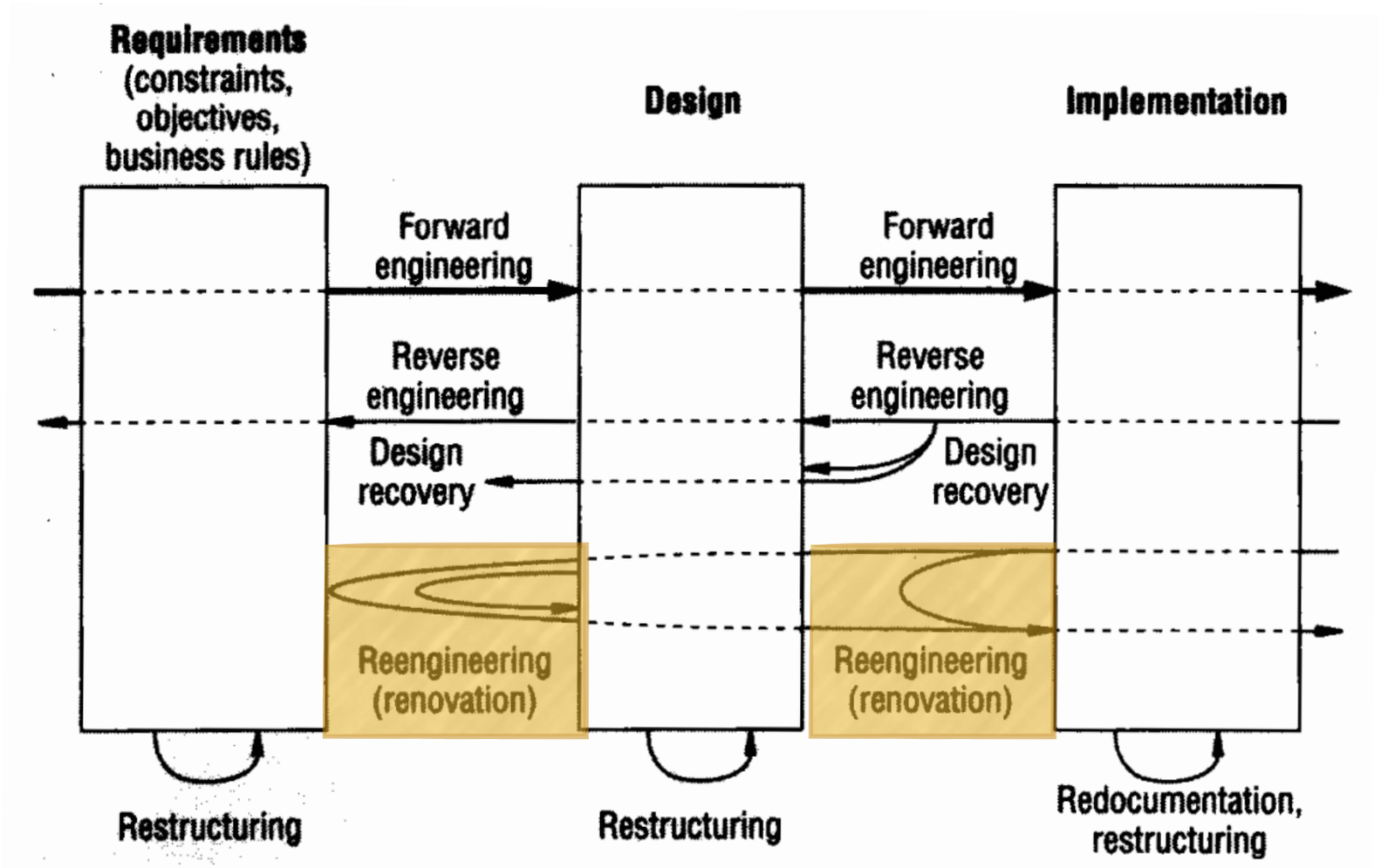
- Examination and alteration of a system to create a new system
- Implementing that new system in order to obtain a more abstract view
- Requires: Some form of **Restructuring**, **Redocumentation**, **Reverse Engineering** along with **Forward Engineering**



Forms of RE

Reengineering..

- Reengineering an information-management system
  - Organization reassess how the system implements high level business rules and makes modifications to conform to changes in the business for the future



- While involves both forward and reverse engineering, it is not a *supertype* of the two

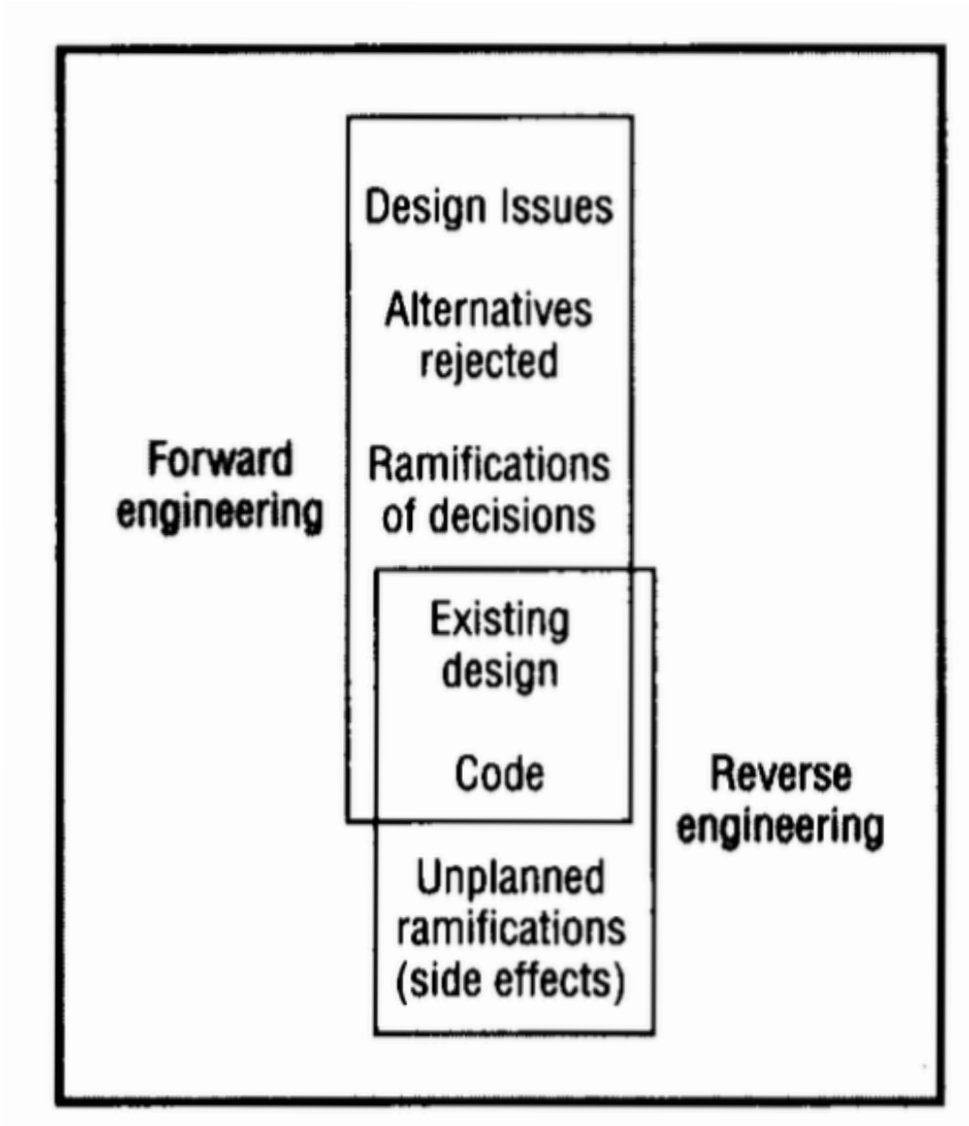
- Both RE and FE are rapidly evolving independent of their application within **reengineering**

Reverse Engineering Taxonomy Eliot **Chikofsky**, James **Cross II**

Reverse Engineering Overview

- **Purpose**
  - Increase overall comprehensibility of the system. For both **maintenance** and **development**.

- **6 Objectives**
  - **Cope** with complexity
  - **Generate** alternate views
    - **Recover** lost information
  - **Detect** side effects
  - **Synthesize** higher abstractions
  - **Facilitate** reuse



Differences in viewpoints

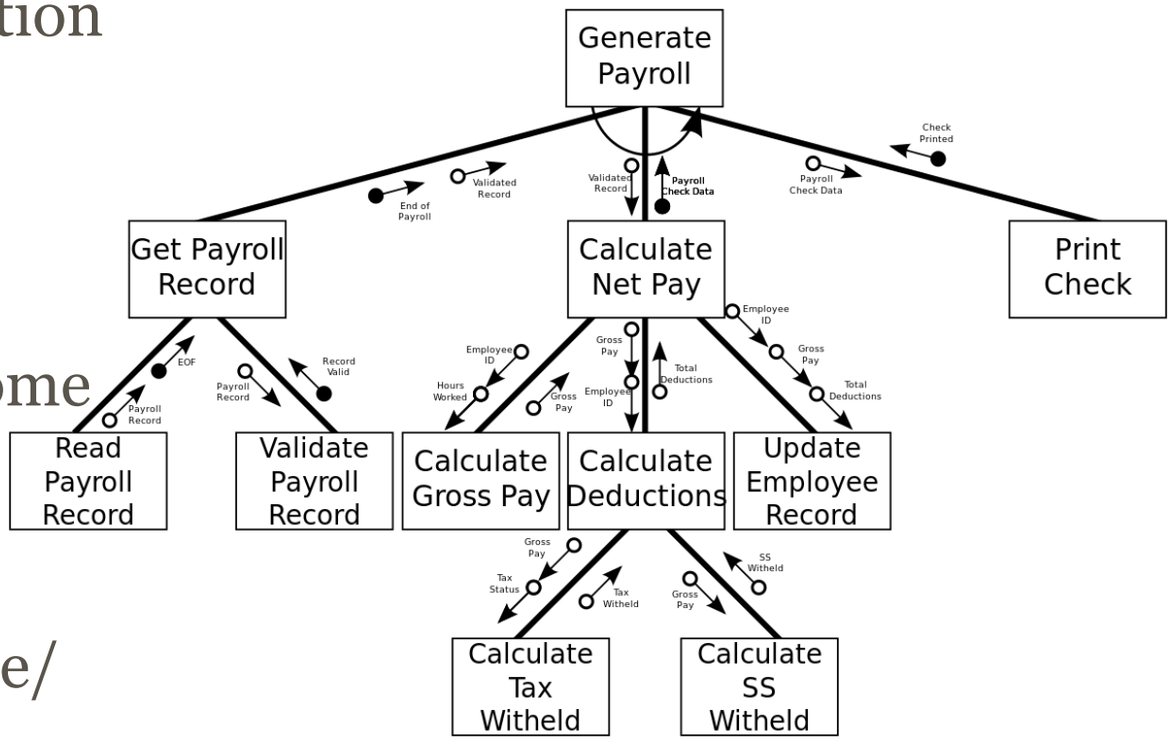
Objectives of RE

Cope with complexity

- Extracting relevant information from a system so decision makers can control both the processes and the product in systems evolution

Generate alternate views

- Graphical representations used as comprehension aids
- RE tools remove the “drag” that typically come with generating these documents by automating the process.
  - Not perfect.
  - Abstracting and generalizing from source/ compiled code
  - CASE tools



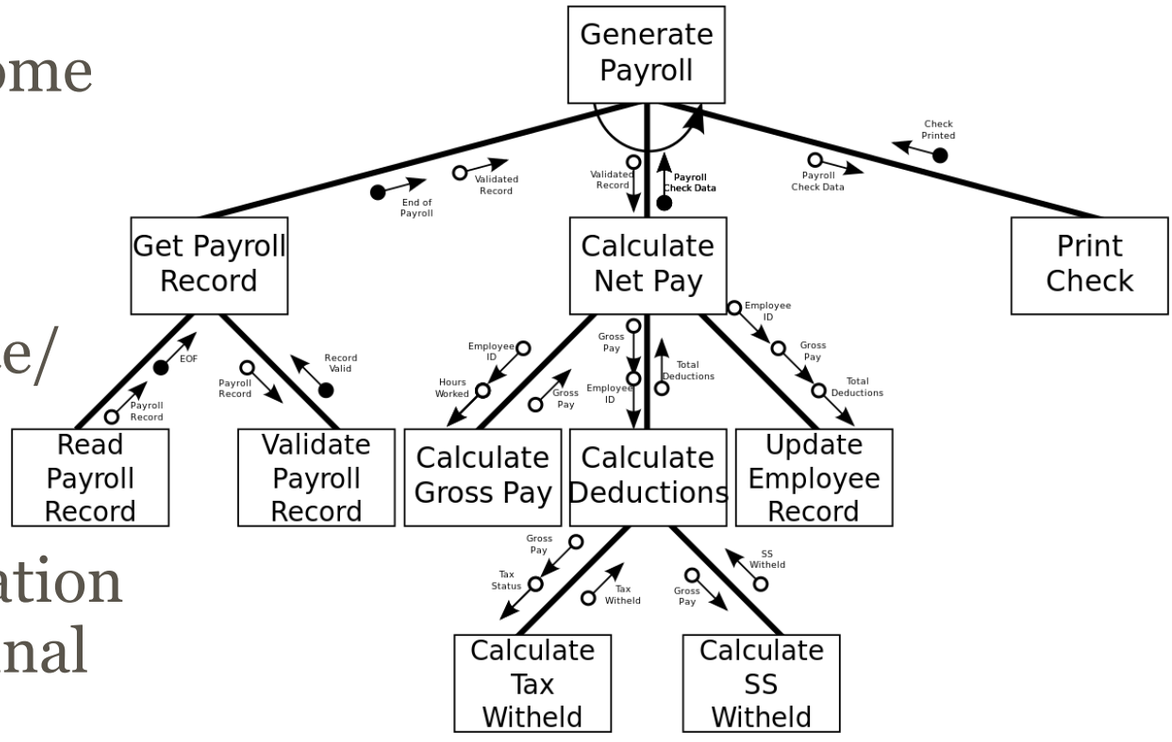
Objectives of RE

Generate alternate views

- Graphical representations used as comprehension aids
- RE tools remove the “drag” that typically come with generating these documents by automating the process.
  - Not perfect.
  - Abstracting and generalizing from source/ compiled code

Recover lost information

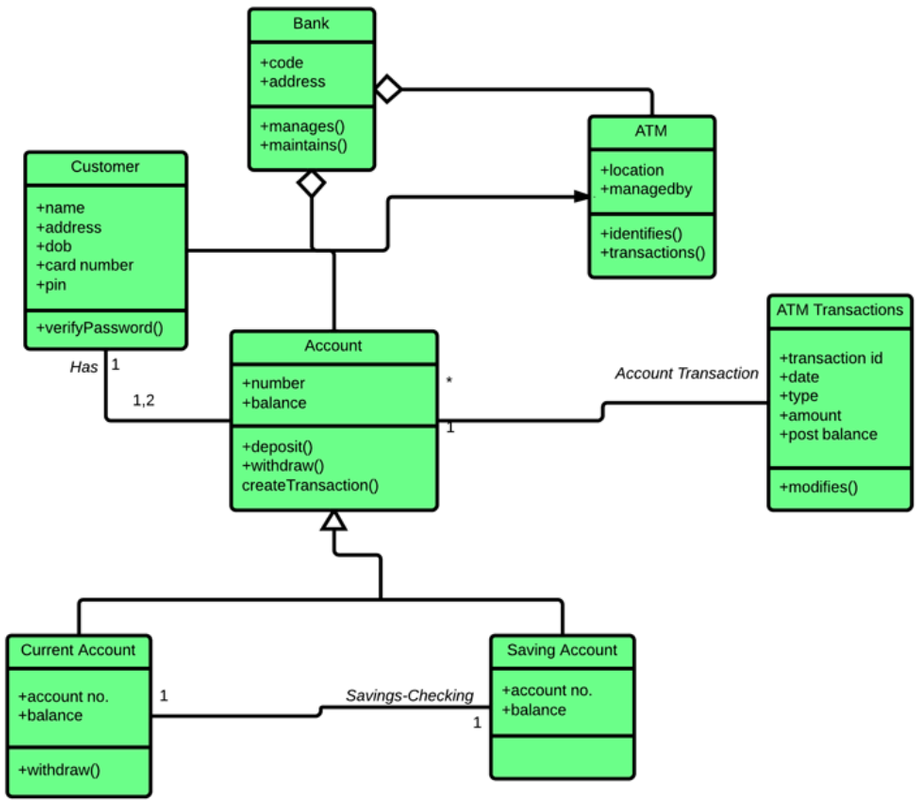
- Long lived systems tend to have information about modifications and perhaps even original designs that are lost to the passage of time



Objectives of RE

Detect side effects

- By generating designs and observing the system from its *implementation* instead of its *design*, we are able to identify unintended side effects
  - Bugs
  - Exploits
  - Errors



Synthesize higher abstractions

- Generating views such as class diagrams using CASE programs
- This is probably the least effective objective of RE.
  - There is heavy debate dating back to 30+ years ago as to how much of this process can be automated

Facilitate reuse

- RE can be used to identify components in existing systems that can be used in future reengineered versions.

Reverse Engineering Importance

Is RE Important?

- **Economics**
  - The cost of understanding software
    - **Time required** to comprehend
    - **Time lost** to misunderstanding
  - RE minimizes both, expanding to virtually all aspects of the software engineering life cycle
- **If I right perfect code, is this *important?***
  - No.
  - You don't write perfect code.
    - Maintenance is considered a cost center.
    - Documentation is often overlooked or rushed
    - Data loss cannot be 100% mitigated
  - **Is it *useful?***
    - Yes.
- **Y2K Example**

Reverse Engineering in Software Security

What is it?

- **Defensive**
  - The implementation of tools and techniques covered already, for security purposes.
    - Understanding a system
    - Identifying flaws or bugs
    - Securing against the:

- **Offensive**
  - Obtaining a desired output or state in a program with little to no knowledge on how to obtain through intended means.
    - Analyze the logic of the program in its normal behavior
    - Analyze which computations are done on test input
    - Analyze the logic of program when given unexpected input
    - Analyze behavior to see new behavior can be exploited

Why is it Important?

- To **Defend** and **Offend**

Reverse Engineering in Software Security

Offensive Example

What is Reverse Engineering?

Reverse engineering (RE) is the process of deciphering designs from an already finished product. In computer science (CS), that means gaining sufficient design-level understandings of already functional programs and systems. Reverse engineering can be used as a means to teach CS fundamentals such as computer organization, architecture, security, and the software development life cycle. Teaching through RE has the ability to give students a unique perspective into how their code looks and operates at the lowest level.

x64 Instructions, CPU Register Sizing

`mov` *D,S*	Move source to destination	`movzx` *D,S*	Move source to destination with zero-extentsion
`add` *D,S*	Add source to destination	`lea` *D,S*	Loads source address into destination register
`call` *Label*	Push return address and jump to label	`leave`	Releases stack space allocated to frame
`cmp` S_2,S_1	Set condition codes according to S_1 - S_2	`push` *S*	Pushes source onto stack
`jnz` *Label*	Jump to label if not equal to 0	`ret`	Returns to calling procedure
`jmp` *Label*	Jump to label		

RAX (64 bits)	EAX (32 bits)	AH (8 bits)	AL (8 bits)
		AX (16 bits)	

Cracking the Code (Understanding the Binary)

The assembly code on the left represents the most basic version of code that is “human-readable” before it is converted into 1’s and 0’s for the CPU to execute. This was obtained by disassembling an executable file. This file was once written in a programming language such as C/C++ and had its own source code, but was compiled then disassembled into assembly code.

This program is designed to take a password as an argument, and if the password meets the requirements, it will output “Nice Job!!”, followed by “flag{<password>}”. If the password does not meet the requirements, or the incorrect number of arguments are passed, the program will call a `usage` method that outputs:
“USAGE: ./<filename> <password>
try again!”

There are three instances where the `usage` method is called. Each occurs after a `cmp` followed by a `jnz`.

1. `2 - [rbp+var_4]`
2. `10 - strlen(rdi)`
3. `@ - al`

Once we have a password that will satisfy each of these compare instructions with a result of 0, we will have achieved our goal. Starting with nothing but an executable file, we are able to identify a pattern that will always result in a successful password. Can you crack the code?

Teaching CS Concepts through Reverse Engineering

Adventures in Pedagogy and Binary Exploitation

G Leaden

G.Leaden1@Marist.edu

Advisor: Alan G. Labouseur

Alan.Labouseur@Marist.edu