Software Re-Engineering

Lecture: 03

Dr. Syed Muazzam Ali Shah

Department of Software Engineering

National University of Computer & Emerging Sciences

muazzam.ali@nu.edu.pk

Sequence [Todays Agenda]

Content of Lecture

- Why Reengineer?
  - Lehman's Laws
  - Object-Oriented Legacy
- Typical Problems
  - Common symptoms
  - Architectural problems & refactoring opportunities
- Reverse and Reengineering
  - Definitions
  - Techniques
  - Patterns (intro.)

What is a Legacy System ?

"legacy"

A sum of money, or a specified article, given to another by will; anything handed down by an ancestor or predecessor. —
Oxford English Dictionary

"legacy system – in computing"

A legacy system is an old method, technology, , computer system, or application program of relating to or being a previous or outdated system, still in use.

A **legacy system** is a piece of software that:

- you have *inherited*, and
- is *valuable* to you.

Typical **problems** with legacy systems:

- original developers *not available*
- *outdated* development methods used
- extensive patches and *modifications* have been made
- *missing* or outdated documentation

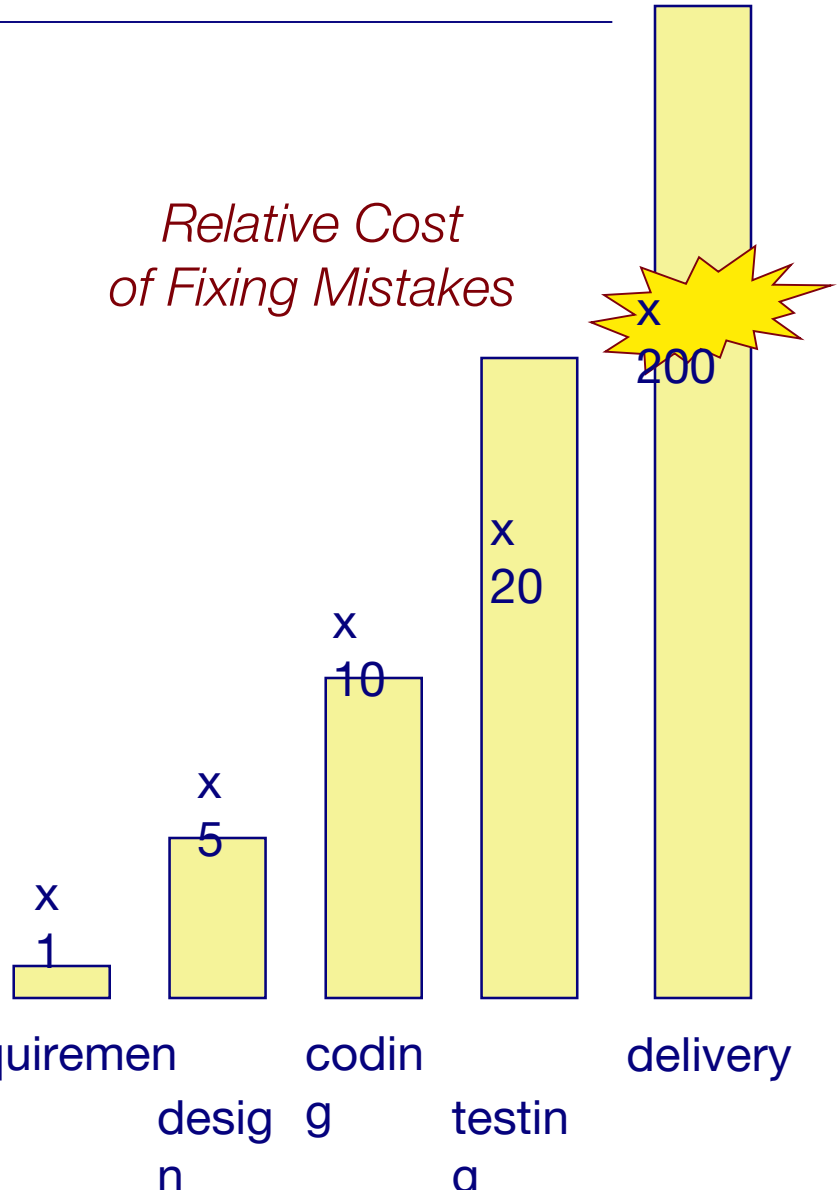
⇒ *so, further evolution and development may be prohibitively expensive*

Software Maintenance - Cost

Relative Maintenance Effort
Between 50% and 75% of global effort is spent on "maintenance" !

Solution ?
- Better requirements engineering?
- Better software methods & tools (database schemas, CASE-tools, objects, components, …)?

Relative Cost of Fixing Mistakes

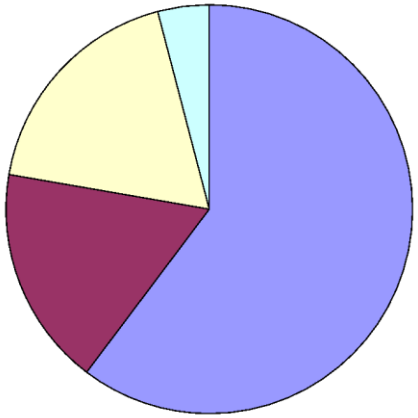


Continuous Development

4.1% Other

data from [Lien78a]

18.2% Adaptive
(new platforms or OS)



17.4% Corrective
(fixing reported errors)

60.3% Perfective
(new functionality)

The bulk of the maintenance cost is due to *new functionality*
⇒ even with better requirements engineering, it is hard to predict new functions

Lehman's Laws

A classic study by Lehman and Belady [Lehm85a] identified several "laws" of system change.

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.

What about Objects ?

Object-oriented legacy systems

- = successful OO systems whose architecture and design no longer responds to changing requirements

Compared to traditional legacy systems

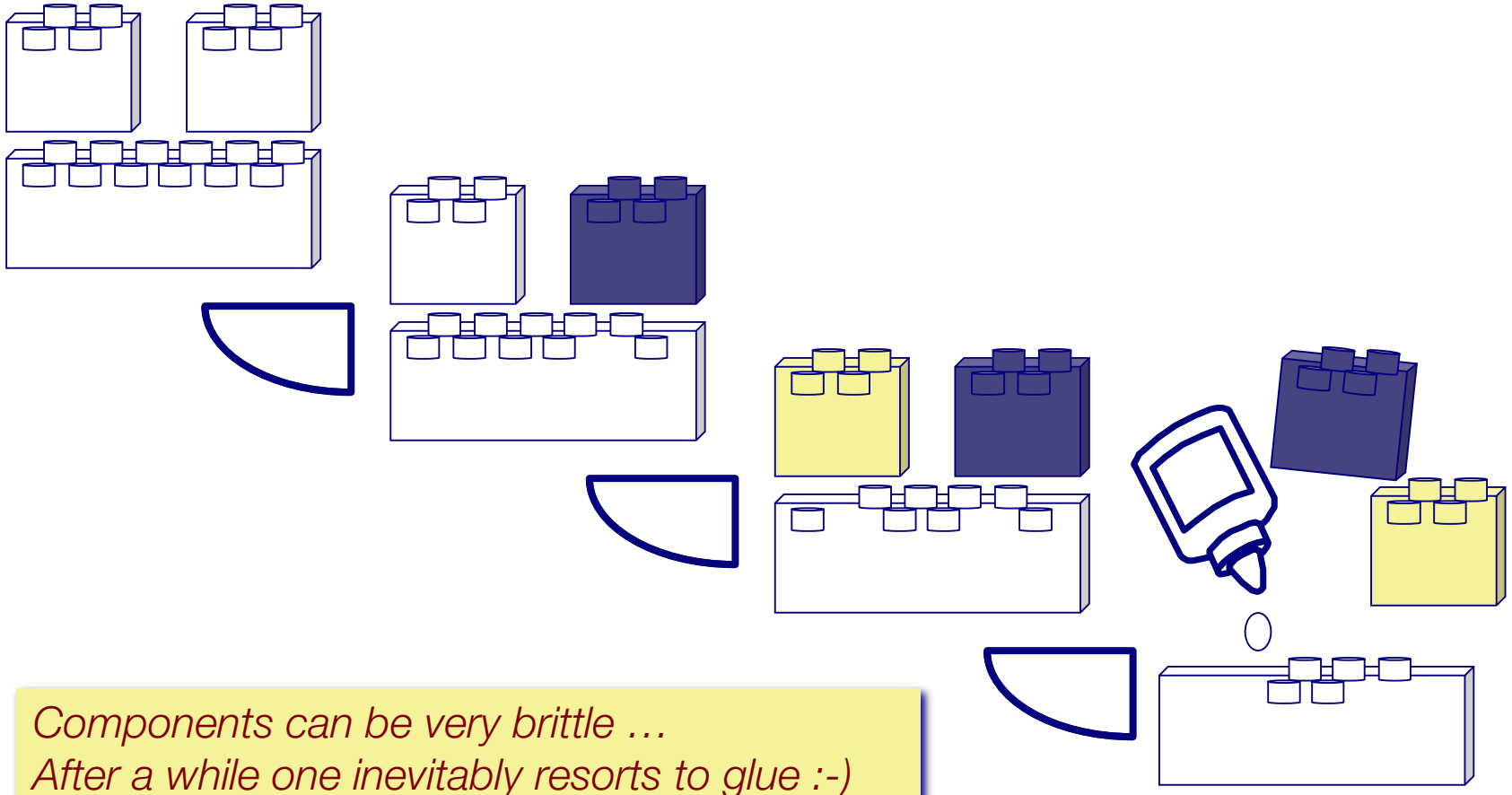
- The *symptoms* and the source of the problems are the *same*
- The *technical details* and solutions may *differ*

OO techniques promise better

- flexibility,
- reusability,
- maintainability
- …

⇒ *they do not come for free*

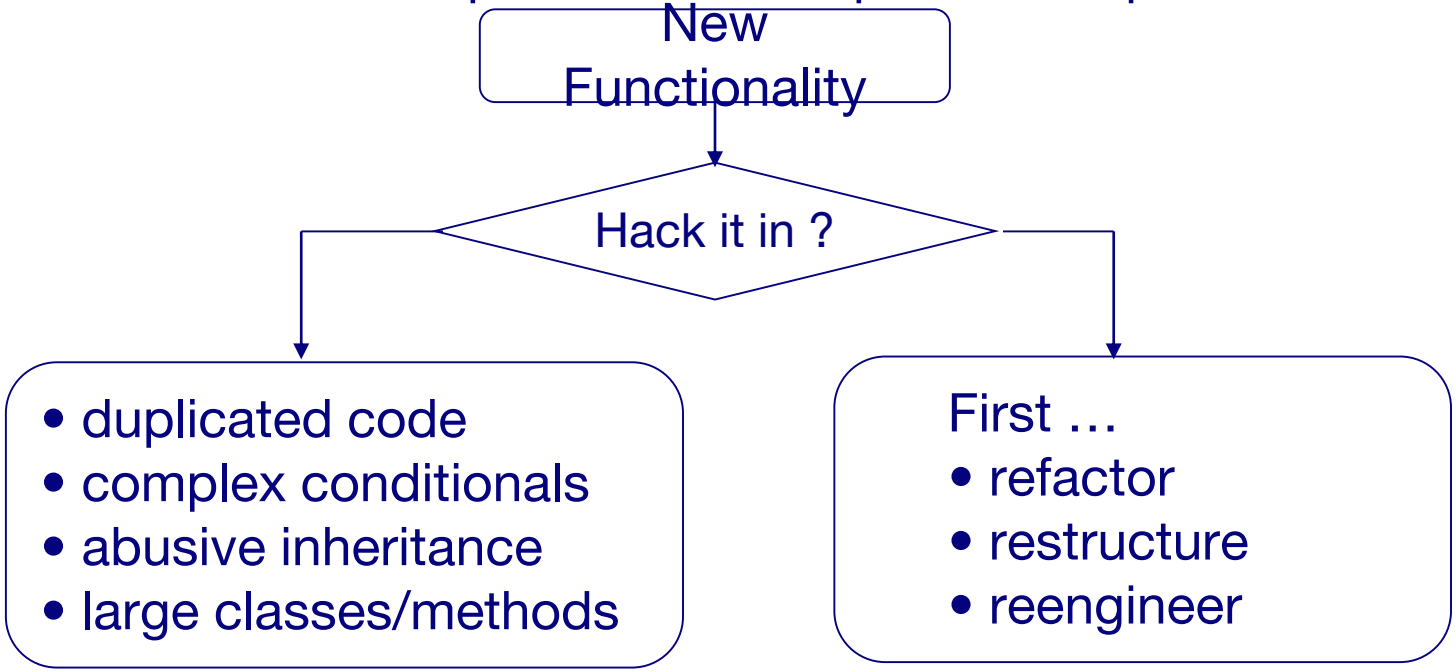
What about Components ?



Components can be very brittle …
After a while one inevitably resorts to glue :-)

How to deal with Legacy ?

New or changing requirements will gradually degrade original design
… unless extra development effort is spent to adapt the structure



Take a *loan* on your software
⇒ pay back via reengineering

Investment for the future
⇒ paid back during maintenance

Common Symptoms

Lack of Knowledge

- *obsolete* or no documentation
- *departure* of the original developers or users
- *disappearance of inside knowledge* about the system
- *limited understanding* of entire system

  ⇒ *missing tests*

Process symptoms

- *too long* to turn things over to production
- need for *constant bug fixes*
- *maintenance dependencies*
- *difficulties separating* products

  ⇒ *simple changes take too long*

Code symptoms

- *duplicated* code
- *code smells*

  ⇒ *big build times*

Common Problems

Architectural Problems

- insufficient *documentation*
  = non-existent or out-of-date
- improper *layering*
  = too few or too many layers
- lack of *modularity*
  = strong coupling
- *duplicated code*
  = copy, paste & edit code
- duplicated *functionality*
  = similar functionality
  by separate teams

Refactoring opportunities

- *misuse* of inheritance
  = code reuse vs polymorphism
- *missing* inheritance
  = duplication, case-statements
- *misplaced* operations
  = operations outside classes
- *violation* of encapsulation

FAMOOS Case Studies

Domain	***LOC***	***Reengineering Goal***
Pipeline Planning	55,000	*extract design*
User Interface	60,000	*increase flexibility*
Embedded Realtime System for controlling hardware	180,000	*improve modularity*
Mail Sorting	350,000	*portability*
Cellular Network Management	2,000,000	*unbundle application*
Space Mission Management	2,500,000	*identify components (to improve reliability)*

Different reengineering goals … but common themes and problems !

Some Terminology

"*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]*

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*
  - identify latent abstractions in software
- Facilitate *reuse*
  - detect candidate reusable artifacts and components

— Chikofsky and Cross [in Arnold, 1993]

Reverse Engineering Techniques

- *Redocumentation*
  - Redocumentation is the simplest and oldest form of reverse engineering, and many consider it to be a weak form of restructuring. The “re-” prefix implies that the intent is to recover documentation about the subject system that existed or should have existed.

- *Design recovery*
  - Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domain.

Goals of Reengineering

- *Unbundling*
  - Unbundle the software system into subsystems that can be tested, delivered and marketed separately.
- *Performance*
  - “first do it, then do it right, then do it fast” — experience shows this is the right sequence!
  - Improving performance is sometimes a goal and sometimes considered as a potential problem once the system is reengineered.
- *Port to other Platform*
  - the architecture must distinguish the platform dependent modules
- *Design extraction:*
  - Always a necessary step in understanding the system; sometimes even an explicit reengineering goal.
  - to improve maintainability, portability, etc.
- *Exploitation of New Technology*
  - i.e., new language features, standards, libraries, etc.

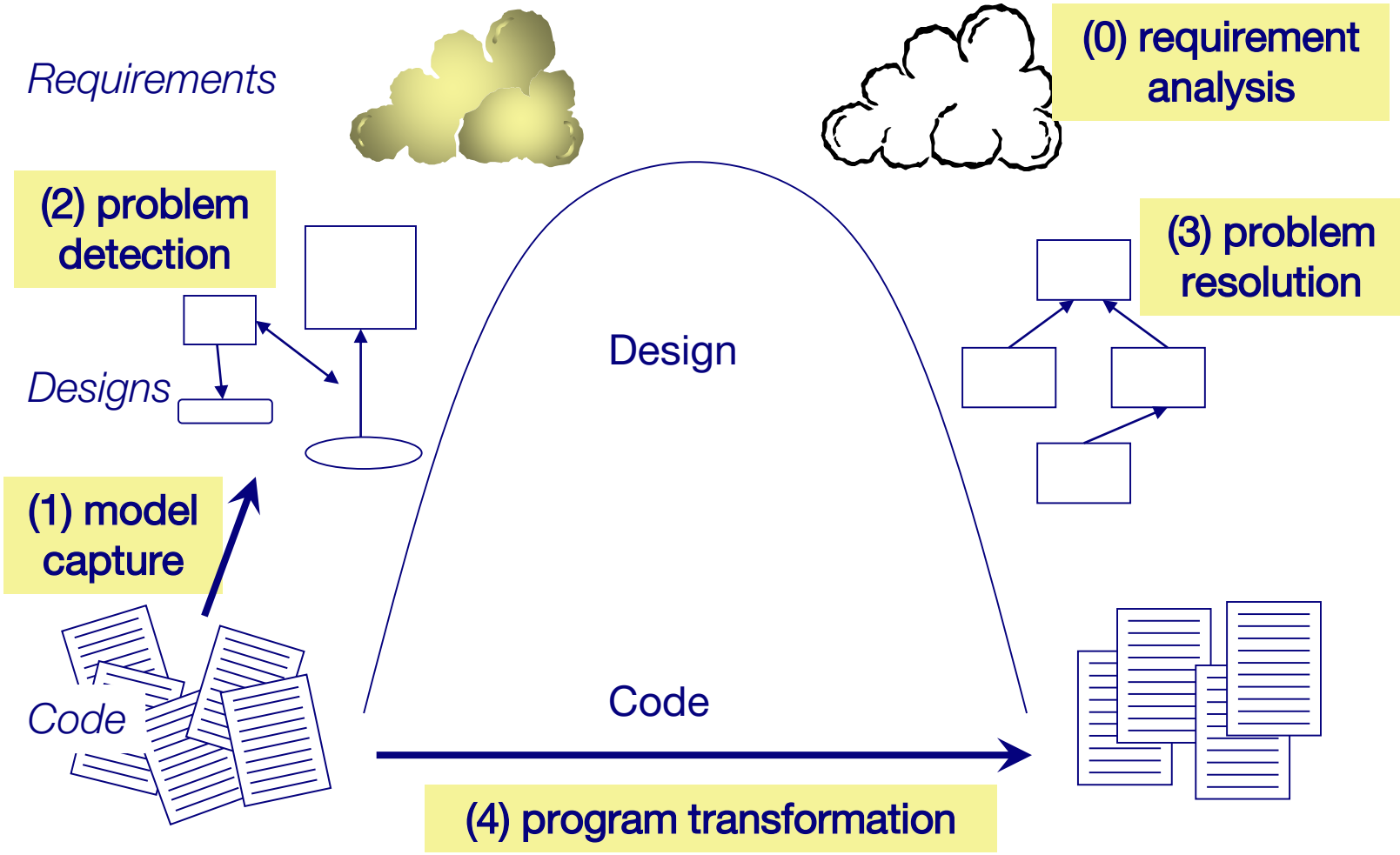
Reengineering Techniques

- *Restructuring*
  - automatic conversion from unstructured to structured code
  - source code translation

  — *Chikofsky and Cross*
- *Data reengineering*
  - integrating and centralizing multiple databases
  - unifying multiple, inconsistent representations
  - upgrading data models

  — *Sommerville, ch 32*
- *Refactoring*
  - renaming/moving methods/classes etc.

The Reengineering Life-Cycle



Reverse engineering Patterns

Reverse engineering patterns *encode expertise and trade-offs* in *extracting design* from source code, running systems and people.

- *Even if design documents exist, they are typically out of sync with reality.*

Reengineering Patterns

Reengineering patterns encode expertise and trade-offs in *transforming legacy code* to resolve problems that have emerged.

- *These problems are typically not apparent in original design but are due to architectural drift as requirements evolve*

Summary

- Software “maintenance” is really *continuous development*
- *Object-oriented* software also suffers from *legacy* symptoms
- Reengineering *goals* differ; *symptoms* don’t
- Common, *lightweight* techniques can be applied to keep software healthy


Thank You!