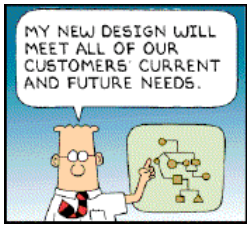
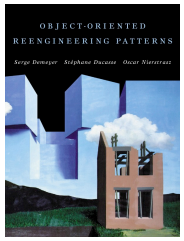
Reengineering Object-Oriented Applications

Prof. Stéphane Ducasse
ducasse@iam.unibe.ch
http://www.iam.unibe.ch/~ducasse/





1. Introduction

- Goals
- Why Reengineering ?
  - + *Lehman's Laws*
  - + *Object-Oriented Legacy*
- Typical Problems
  - + *common symptoms*
  - + *architectural problems & refactoring opportunities*
- Reverse and Reengineering
  - + *Definitions*
  - + *Techniques*
  - + *Patterns*

Goals

We will try to convince you:

- Yes, Virginia, there are *object-oriented legacy systems* too!
- Reverse engineering and reengineering are *essential activities* in the lifecycle of any successful software system. (And especially OO ones!)
- There is a large set of *lightweight tools and techniques* to help you with reengineering.
- Despite these tools and techniques, *people must do job* and they represent the most valuable resource.

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

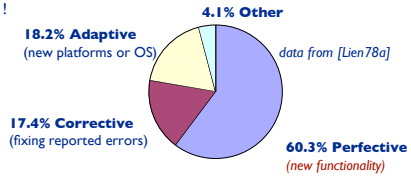
A **legacy system** is a piece of software that:
- you have *inherited*, and
- is *valuable* to you.

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- *outdated* development methods used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- *missing* or outdated documentation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*

Continuous Development

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



The bulk of the maintenance cost is due to *new functionality*
⇒ even with better requirements, 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.

Those laws are still applicable…

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*

Modern Methods & Tools ?

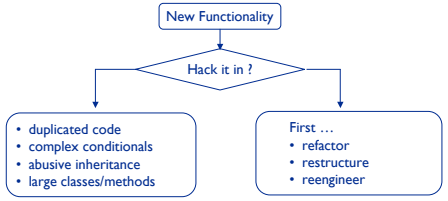
[Glas98a] quoting empirical study from Sasa Dekleva (1992)
- Modern methods(*) lead to more reliable software
- Modern methods lead to less frequent software repair
- and ...
- Modern methods lead to more total maintenance time

Contradiction ? *No!*
- modern methods make it easier to change
  ... this capacity is used to enhance functionality!

(*) process-oriented structured methods, information engineering, data-oriented methods, prototyping, CASE-tools – not OO !

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 are 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 = type-casting; C++ "friends"
- *class abuse* = classes as namespaces

Some Case Studies

Domain	LOC	Reengineering Goal
pipeline planning	55,000	*extract design*
user interface	60,000	*increase flexibility*
embedded switching	180,000	*improve modularity*
mail sorting	350,000	*portability & scalability*
network management	2,000,000	*unbundle application*
space mission	2,500,000	*identify components*

Different reengineering goals … but common themes and problems !

System evolution…

Software are living…

- Early decisions may have been good at that time
- But the context changes
- Customers change
- Technology changes
- People change

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*
  - + pretty printers
  - + diagram generators
  - + cross-reference listing generators
- *Design recovery*
  - + software metrics
  - + browsers, visualization tools
  - + static analyzers
  - + dynamic (trace) analyzers

Goals of Reengineering

- *Unbundling*
  - split a monolithic system into parts that can be separately marketed
- *Performance*
  - "first do it, then do it right, then do it fast" — experience shows this is the right sequence!
- *Port* to other Platform
  - the architecture must distinguish the platform dependent modules
- *Design extraction*
  - 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

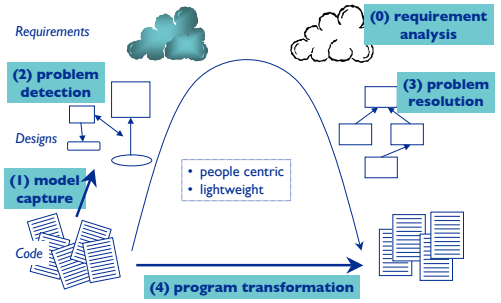
— *Chikofsky and Cross*

- *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.*

Example: **Interview During Demo**

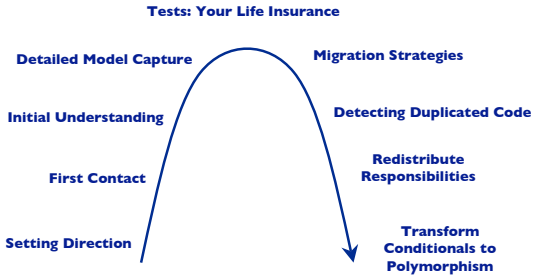
Reengineering Patterns

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*

Example: **Move Behaviour Close to Data**

A Map of Reengineering Patterns



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