Quality Assurance and Software Evolution

Course outline

1. Software Evolution and Reengineering
2. Model Capture and Design Extraction
3. Object-Oriented Harmony ... and Its Disharmonies
4. Detecting Disharmonies
5. Refactoring and Restructuring
6. Defect and Change Prediction

Textbooks

1. Introduction

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

Goals of this course

We will try to convince you:

- 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 the quality assessment and the evolution of your software.
- Despite these tools and techniques, people must do the 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.

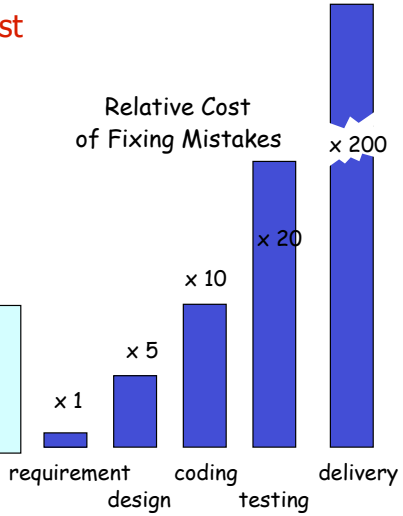
Typical **problems** with legacy systems are:
- original developers no longer available
- outdated development methods used
- extensive patches and modifications
- missing or outdated documentation

⇒ so, further evolution and development may be prohibitively expensive

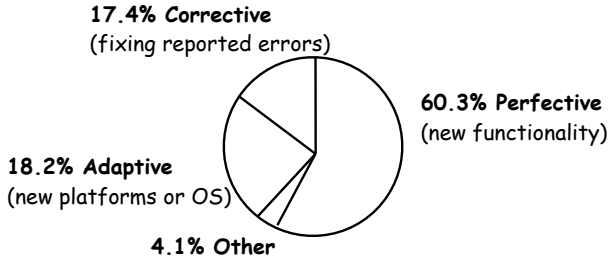
Software Maintenance - Cost

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, ...)



Requirements Engineering ?



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.

Continuous 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.

These 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

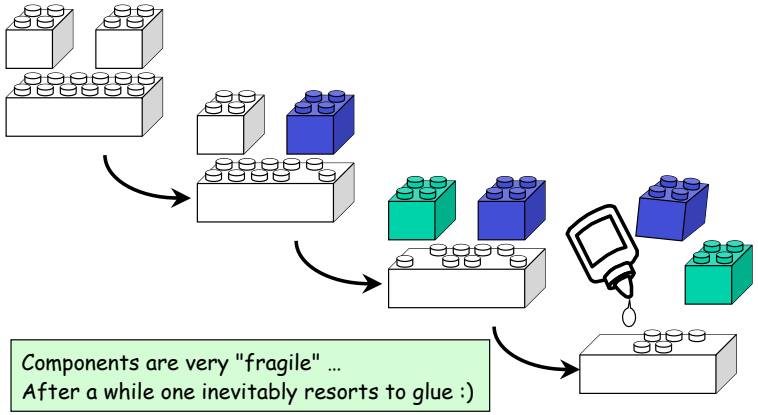
- The symptoms and the source of the problems are the same
  - ravioli code instead of spaghetti code ;)
- The technical details and solutions may differ

OO techniques promise better

- flexibility,
- reusability,
- maintainability
- …

⇒ they do not come for free

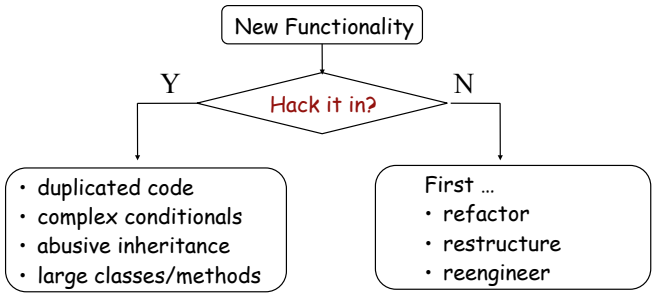
What about Components ?



Components are very "fragile" …
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

When, due to constraints,
I design *quickly and dirty*,
my project is loaded with

technical debt.

Cunningham, 1992

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
  - simple changes take too long
- need for constant bug fixes
- maintenance dependencies
- difficulties separating products

Code symptoms

- big build times
- duplicated code
- code smells

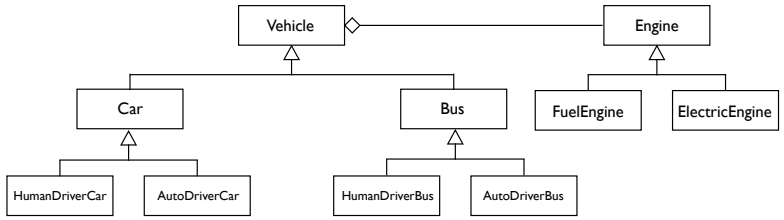
Common Problems

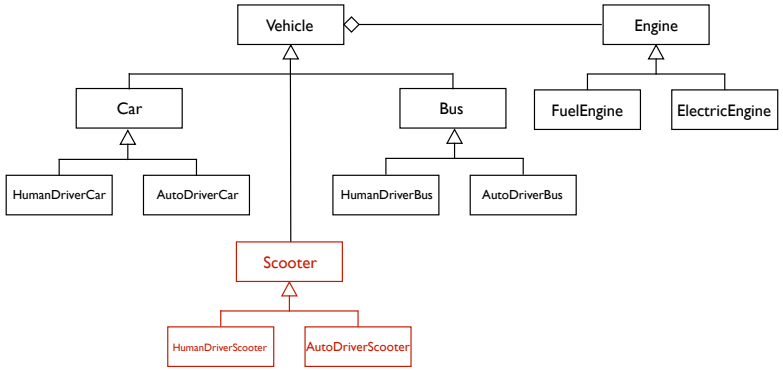
Architectural Problems

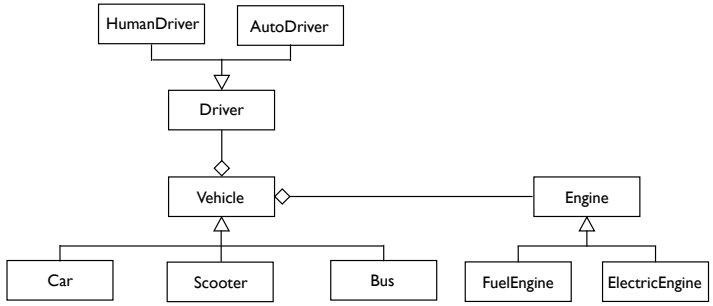
- insufficient documentation
  - non-existent or out-of-date
- improper layering
  - too few are too many layers
- lack of modularity
  - strong coupling
- duplicated functionality
  - similar functionality by separate teams

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
- duplicated code
  - copy, paste & edit code







Favor object composition over class inheritance!

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]

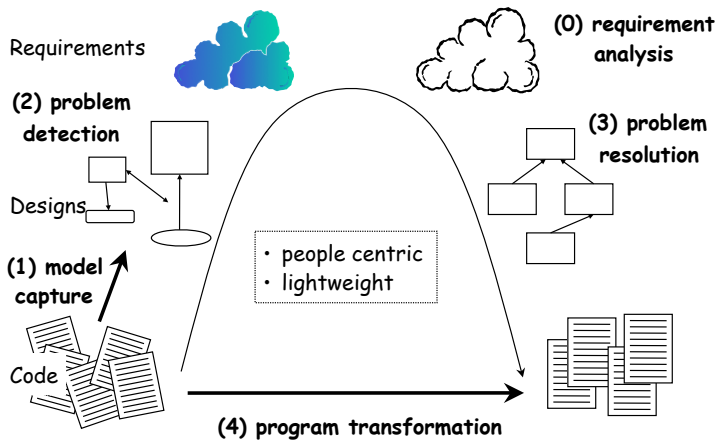
Goals of Software Evolution (reengineering)

- 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!

- Design refinement
  - to improve maintainability, portability, etc.
- Port to other Platform
  - the architecture must distinguish the platform dependent modules
- Exploitation of New Technology
  - i.e., new language features, standards, libraries, etc.

The Reengineering (Evolution) Life-Cycle



Sample Projects

FAMOOS 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

Different goals … but common themes and problems !

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