Software Re-Engineering

COSC 6431

Vassilios Tzerpos
bil@cse.yorku.ca
CSEB 3024

The Legacy Dilemma

Legacy Systems

- Older software systems that remain vital to an organization
- Software systems that are developed specially for an organization have a long lifetime
- Many software systems that are still in use were developed many years ago using technologies that are now obsolete

Legacy System Replacement

- There is a business risk in scrapping a legacy system and replacing it with a modern system:
  - Legacy systems rarely have a complete specification.
  - Business processes rely on the legacy system.
  - The system may embed business rules that are not formally documented elsewhere.
  - New software development is risky and may not be successful.

Laws of Software Evolution

Also known as Lehman's Laws

Law of Increasing Complexity

As a program is evolved its complexity increases unless work is done to maintain or reduce it

Law of Continuing Growth

Functional content of a program must be continually increased to maintain user satisfaction over its lifetime

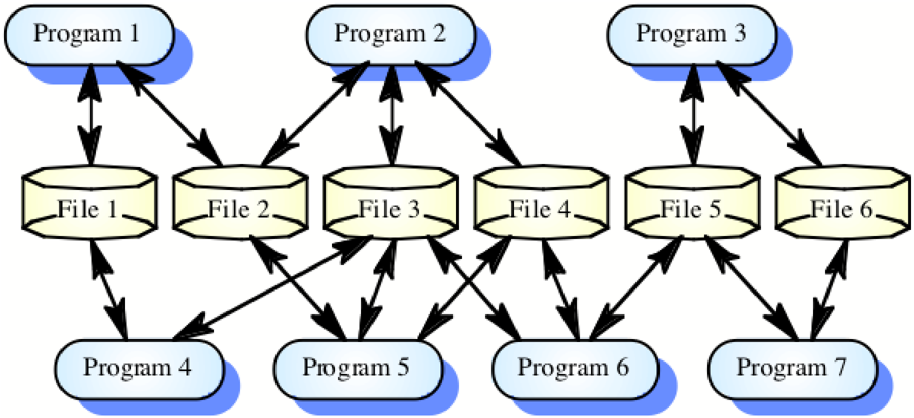
Legacy System Change is Expensive

- Different parts of the system are implemented by different teams.
- The system may use an obsolete programming language.
- The system documentation is often out-of-date.
- The system structure may be corrupted by many years of maintenance.
- Techniques to save space or increase speed at the expense of understandability may have been used

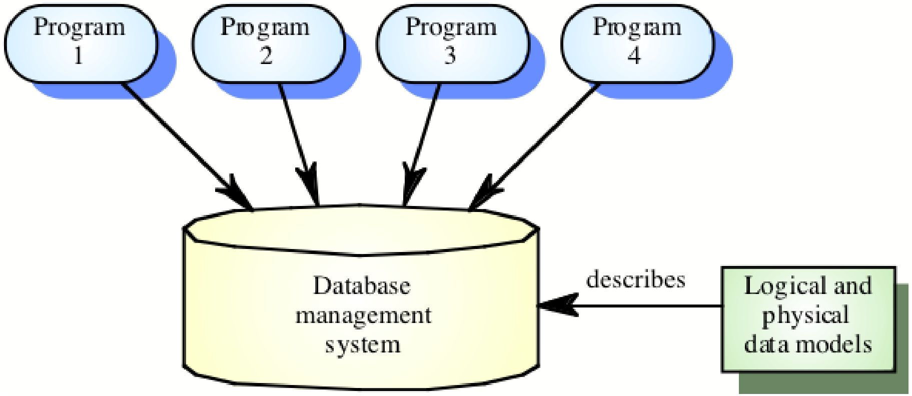
The Legacy Dilemma

- It is expensive and risky to replace the legacy system.
- It is expensive to maintain the legacy system.
- Businesses may choose to extend the system lifetime using techniques such as reverse engineering.

Example of a Legacy Application System



After Re-engineering: Database-Centred System



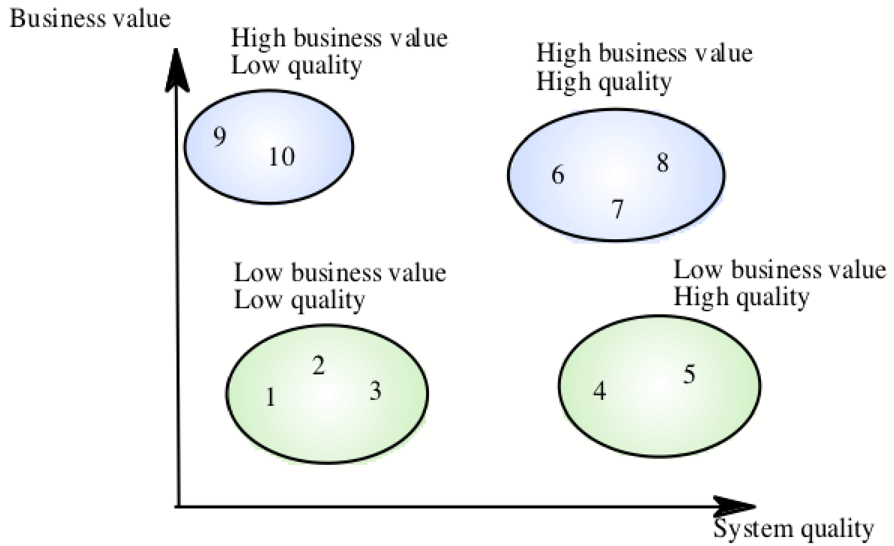
Legacy System Design

- Most legacy systems were designed before object-oriented development was used.
- Rather than being organized as a set of interacting objects, these systems have been designed using a function-oriented design strategy.

Legacy System Assessment

- Organizations that rely on legacy systems must choose a strategy for evolving these systems:
  - Replace the old system with a new one.
  - Continue maintaining the system.
  - Transform the system by re-engineering to improve its maintainability.
- The strategy chosen should depend on the system quality and its business value.

System Quality and Business Value



Legacy System Categories

- Low quality, low business value
  - These systems should be scrapped
- Low-quality, high-business value
  - Should be re-engineered or replaced
- High-quality, low-business value
  - Replace, scrap, or maintain
- High-quality, high business value
  - Continue in operation using normal system maintenance

Software Maintenance

Managing the processes of system change

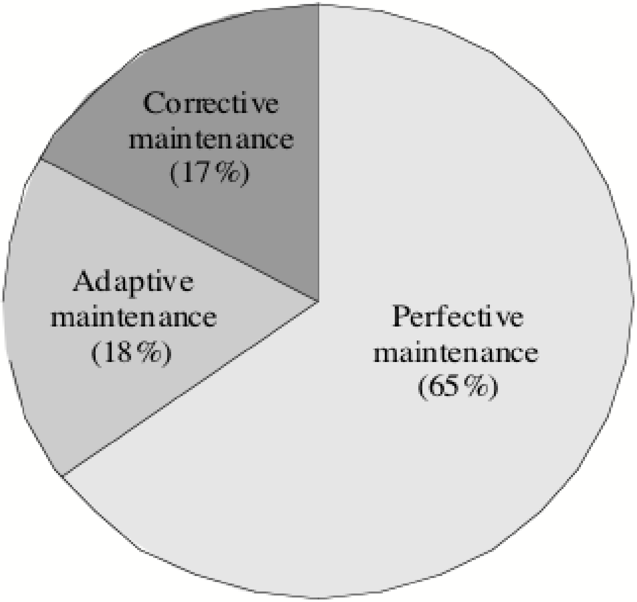
Maintenance is Inevitable

- The system requirements are likely to change while the system is being developed because the environment is changing.
- When a system is installed in an environment it changes that environment and therefore changes the system requirements.

Types of Maintenance

- Perfective maintenance
  - Adding or modifying the system's functionality to meet new requirements.
- Adaptive maintenance
  - Changing a system to adapt it to new hardware or operating system.
- Corrective maintenance
  - Changing a system to fix coding, design, or requirements errors.

Which type of maintenance is the most common one?



Evolving Systems

- It is usually more expensive to add functionality after a system has been developed rather than design it into the system:
  - Maintenance staff are often inexperienced and unfamiliar with the application domain.
  - Programs may be poorly structured and hard to understand.
  - Changes may introduce new faults as the complexity of the system makes impact assessment difficult.
  - The structure may be degraded due to continual change.
  - There may be no documentation available to describe the program.

The Maintenance Process

- Maintenance is triggered by change requests from customers or marketing requirements.
- Changes are normally batched and implemented in a new release of the system.
- Programs sometimes need to be repaired without a complete process iteration but this is dangerous as it leads to documentation and programs getting out of step.

Maintenance Costs

- Usually greater than development costs (2 to 100* depending on the application).
- Affected by both technical and non-technical factors.
- Maintenance corrupts the software structure so makes further maintenance more difficult.
- Aging software can have high support costs, e.g. old languages, compilers etc.

Maintenance Cost Factors

- Module independence
  - It should be possible to change one module without affecting others.
- Programming language
  - High-level language programs are easier to maintain.
- Programming style
  - Well-structured programs are easier to maintain.
- Program validation and testing
  - Well-validated programs tend to require fewer changes due to corrective maintenance.

Maintenance Cost Factors

- Documentation
  - Good documentation makes programs easier to understand.
- Configuration management
  - Good CM means that links between programs and their documentation are maintained.
- Application domain
  - Maintenance is easier in mature and well-understood application domains.
- Staff stability
  - Maintenance costs are reduced if the same staff are involved with them for some time.

Maintenance Cost Factors

- Program age
  - The older the program, the more expensive it is to maintain (usually).
- External environment
  - If a program is dependent on its external environment, it may have to be changed to reflect environmental changes.
- Hardware stability
  - Programs designed for stable hardware will not require to change as the hardware changes.

How to measure maintainability?

- Control complexity
  - Can be measured by examining the conditional statements in the program.
- Data complexity
  - Complexity of data structures and component interfaces.
- Length of identifier names
  - Longer names imply readability.
- Program comments
  - Perhaps more comments mean easier maintenance.

How to measure maintainability?

- Coupling
  - How much use is made of other components or data structures.
- Degree of user interaction
  - The more user I/O, the more likely the component is to require change.
- Speed and space requirements
  - Require tricky programming, harder to maintain.

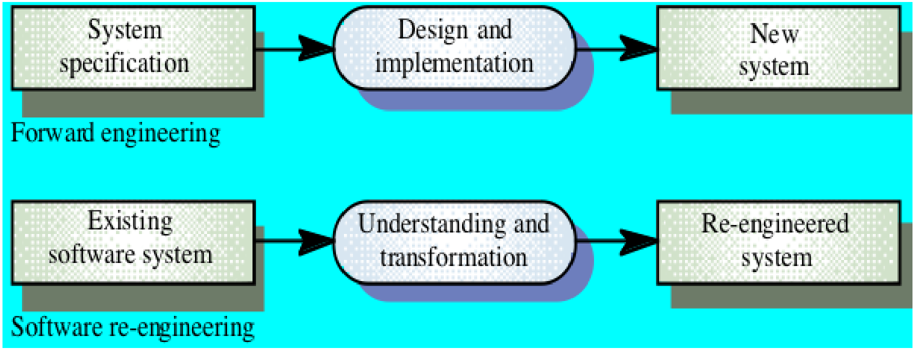
Process Measurements

- Number of requests for corrective maintenance.
- Average time taken to implement a change request.
- Number of outstanding change requests.
- If any or all of these is increasing, this may indicate a decline in maintainability.

Software Re-Engineering

Reorganizing and modifying existing software systems to make them more maintainable

Forward Engineering and Re-Engineering



When to Re-Engineer

- When system changes are mostly confined to part of the system, then re-engineer that part.
- When hardware or software support becomes obsolete.
- When tools to support re-structuring are available.

Re-Engineering Advantages

- Reduced risk
  - There is a high risk in new software development.
- Reduced cost
  - The cost of re-engineering is often significantly less than the costs of developing new software.

Re-engineering Cost Factors

- The quality of the software to be re-engineered.
- The tool support available for re-engineering.
- The extent of the data conversion which is required.
- The availability of expert staff for re-engineering.

Reverse Engineering

- Reverse Engineering is the process of determining how a system works by analyzing its internal constituents and/or its external behaviour.
- In the software world one would say that reverse engineering is trying to figure out how a system works by:
    - Inspecting the source code and documentation (if it exists)
    - Exercising the executable programs and observing their behavior.

Why is Reverse Engineering Important/Necessary?

- Most software that is developed is not “from scratch”.
- Understanding someone else’s source code, specifications, designs, is difficult.
  - Why is this so?
  - What makes software more difficult to understand than a toaster or a car?

Software Maintenance Problem

- A company hires a bright software developer to maintain a system.
- The project manager points the developer to a source code directory and says “become an expert in the system as soon as possible”.
- The IBM TOBEY back-end compiler project allowed for a 1 year learning curve (but this is quite rare).

Reverse Engineering Research

- The focus has been primarily on the development of tools to help software developers understand software quicker and with less effort.
- Not much work has been done on reverse engineering methods, however.

Sherlock Holmes Analogy
by Spiros Mancoridis

We have developed good detective tools (e.g., magnifying glasses, fingerprint matchers, etc) but we have little insight on how to train someone to be a good detective (e.g., guidelines, processes, etc)

Progress Has Been Made In ...

- Source code analysis
- Program tracing and profiling
- Automatic modularization (software clustering)
- But still a research area in its infancy ...

Lecture schedule

- Sep 6 & 11: Introduction, administrivia
- Sep 18 & 20: Static and dynamic analysis
- Sep 25 & 27: Software clustering
- Oct 2 & 4: Evaluation of clustering techniques
- Oct 9 & 11: Introduction to Design Patterns
- Oct 16 & 18: Design Pattern Detection
- Oct 23 & 25: Refactoring
- Oct 30 & Nov 1: No classes (WCRE conference)
- Nov 6 & 8: Program Transformation
- Nov 13 & 15: Re-Engineering Patterns
- Nov 20 & 22: Research paper presentations
- Nov 27 & 29: Research paper presentations

Grading

- 10% - Participation (paper discussion)
- 20% - Assignment
- 30% - Research paper presentation
- 40% - Project report

Workload

- September: 2 papers a week
- October: Assignment + 2 papers a week
- November: 1 paper to present + project