### Legacy Systems

**Definition of Legacy Software**

- Definition: Legacy software is old software that is still useful.
- In particular:
  - It cannot be simply discarded because it contains experienced and validated knowledge.
  - It was developed to answer precise needs that are not covered by existing software.
  - The system might be running on old hardware.
  - The system might be currently used by other people.

### Facts

- COBOL Programs (1959) ran 80% of the world’s business in 1997 (200 billions lines of code in mainframes alone).
- Most programs to make transactions with stock exchange run in COBOL.

![Software Maintenance Cost Chart](http://users.jyu.fi/~koskinen/smcosts.htm)
Why change an old system?

- *"If it ain't broke, don't fix it!"
- If it fulfills its tasks why recode and change it?
  - Y2K?
  - Improving readability
  - Debugging
  - Changing requirements
  - ...

Refactoring

- Refactoring the code means making small changes to the code that do not change the results
  - E.g. Renaming a class or a parameter
- Goal: improve readability

Refactoring Tools

- All IDE have a refactoring tool now.

  ![Refactoring Tools Image]

  - Risks: sometimes backward incompatible changes

---

Reengineering Object-Oriented Applications

---

Slides from Stephane Ducasse

---

REVERSE ENGINEERING
Software Migration

- Problems:
  1. You have this old piece of code that you use on a daily basis... and you are scared to switch to the X operating system/platform
  2. You want to use this fancy new platform and say you do (you are up-to-date...)

- What to do?

Two main solutions

- Wrap
- Port

Wrapping the code

- The idea is to use directly the interface from another platform and
- That's the easiest solution if the tool has a clear textual interface or API
- It does not really solve problem 1 though!

Example: Wrapping C code in Eiffel

(from: [http://mersenewtwisterrandom.origo.ethz.ch/](http://mersenewtwisterrandom.origo.ethz.ch/) (tiny project))
Example: Apache and CGI Scripts

Porting the Code

- Porting the code means reimplementing it in another language
- One would think it is easier... see part on reverse engineering!
- Many issues may happen:
  - Users want a similar interface
  - The original code might not work that well!!!
  - The new version usually require additional features

Today

- Legacy Software
  - Definition
- Reengineering
  - Refactoring
  - Reengineering
- Reverse Engineering
- Software Migration
  - Wrapping
  - Porting

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
**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?**

*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

---

**Continuous Development**

<table>
<thead>
<tr>
<th>Relative Maintenance Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between 50% and 75% of global effort is spent on “maintenance”!</td>
</tr>
</tbody>
</table>

- 18.2% Adaptive (new platforms or OS)
- 17.4% Corrective (fixing reported errors)
- 60.3% Perfective (new functionality)
- 4.1% Other

Data from [Law78a]

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[1] 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!

How to deal with Legacy?

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

New Functionality

Hack it in!

• duplicated code
• complex conditionals
• abusive inheritance
• large classes/methods

First...

• refactor
• restructure
• reengineer

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

Code symptoms
• duplicated code
• code smells
• large build times

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

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
• revision of inheritance
• code reuse vs polymorphism
• misplaced operations
• operations outside classes
• violation of encapsulation
• type-casting; C++ “friends”
• class abuse
• classes as namespaces

Some Case Studies

<table>
<thead>
<tr>
<th>Domain</th>
<th>LOC</th>
<th>Reengineering Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipeline planning</td>
<td>51,000</td>
<td>extract design</td>
</tr>
<tr>
<td>user interface</td>
<td>60,000</td>
<td>increase flexibility</td>
</tr>
<tr>
<td>embedded switching</td>
<td>180,000</td>
<td>improve modularity</td>
</tr>
<tr>
<td>mail sorting</td>
<td>350,000</td>
<td>portability &amp; scalability</td>
</tr>
<tr>
<td>network management</td>
<td>2,000,000</td>
<td>unbundle application</td>
</tr>
<tr>
<td>space mission</td>
<td>2,500,000</td>
<td>identify components</td>
</tr>
</tbody>
</table>

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

— Sommerville, ch 32

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 controlling multiple databases
  - unifying multiple, inconsistent representations
  - upgrading data models

• Refactoring
  - renaming/moving methods/classes etc.

— Sommerville, ch 32

The Reengineering Life-Cycle

(0) requirement analysis
(1) problem identification
(2) problem detection
(3) problem resolution
(4) program transformation
(5) model capture
(6) requirements refiguration
(7) people centered
(8) lightweight

— Chikofsky and Cross [in Arnold, 1993]
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

2. Reverse Engineering

- What and Why
- Setting Direction
  - Most Valuable First
- First Contact
  - Interview during Demo
- Initial Understanding
  - Study Exceptional Entities
- Detailed Model Capture
  - Tie Code and Questions
- Conclusion
What and Why?

**Definition**
Reverse Engineering is the process of analysing a subject system to identify its components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. — Chikofsky & Cross, '90

**Motivation**
Understanding other people’s code (cf. newcomers in the team, code reviewing, original developers left,...)

Generating UML diagrams is NOT reverse engineering ... but it is a valuable support tool.

The Reengineering Life-Cycle

<table>
<thead>
<tr>
<th>(0) requirement analysis</th>
<th>(1) model capture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issues</td>
<td></td>
</tr>
<tr>
<td>• scale</td>
<td></td>
</tr>
<tr>
<td>• speed</td>
<td></td>
</tr>
<tr>
<td>• accuracy</td>
<td></td>
</tr>
<tr>
<td>• politics</td>
<td></td>
</tr>
</tbody>
</table>

Setting Direction

- **Conflicting interests** (technical, ergonomic, economic, political)
- Presence/absence original developers
- Legacy architecture
- Which problems to tackle?
  - Interesting vs important problems?
  - Wrap, refactor or rewrite?

Most Valuable First

- **Problem**: Which problems should you focus on first?
- **Solution**: Work on aspects that are most valuable to your customer
  - Maximize commitment, early results; build confidence
  - Difficulties and hints:
    - Which stakeholder do you listen to?
    - What measurable goal to aim for?
    - Consult change logs for high activity
    - Play the Planning Game
    - Wrap, refactor or rewrite! — Fix Problems, not Symptoms

Forces — Setting Direction

- Legacy systems are large and complex
  - Split the system into manageable pieces
- Time is scarce
  - Apply lightweight techniques to assess feasibility and risks
- First impressions are dangerous
  - Always double-check your sources
- People have different agendas
  - Build confidence; be wary of skeptics
First Contact

System experts
Chat with the Maintainers
Interview during Demo
Talk with end users
Talk with developers
Feasibility assessment (one week time)

Discuss the Solution:
- Interview during Demo
  - Select several users
  - Demo puts a user in a positive mindset
  - Demo steers the interview

Test Drive:
- Read All the Code in One Hour
- Do a Mock Installation
- Skim the Documentation
- Compile it
- Read it

Study the Exceptional Entities

Problem: How can you quickly identify design problems?
Solution: Measure software entities and study the anomalous ones

- Use simple metrics
- Visualize metrics to get an overview
- Browse the code to get insight into the anomalies

Initial Understanding

 Forces — Initial Understanding

- Data is deceptive ⇒ Always double-check your sources
- Understanding entails iteration ⇒ Plan iteration and feedback loops
- Knowledge must be shared ⇒ “Put the map on the wall”
- Teams need to communicate ⇒ “Use their language”

Study the Exceptional Entities

Visualizing Metrics

Problem: What are the typical usage scenarios?
Solution: Ask the user!

- Solution: interview during demo
  - select several users
  - demo puts a user in a positive mindset
  - demo steers the interview

... however
- Which user?
- Users complain
- What should you ask?

forces — initial understanding

• data is deceptive ⇒ always double-check your sources
• understanding entails iteration ⇒ plan iteration and feedback loops
• knowledge must be shared ⇒ “put the map on the wall”
• teams need to communicate ⇒ “use their language”

study the exceptional entities

problem: how can you quickly identify design problems?
solution: measure software entities and study the anomalous ones

• use simple metrics
• visualize metrics to get an overview
• browse the code to get insight into the anomalies

visualizing metrics

use simple metrics and layout algorithms

visualise up to 5 metrics per node
Forces — Detailed Model Capture

- Details matter
  ⇒ Pay attention to the details!
- Design remains implicit
  ⇒ Record design rationale when you discover it!
- Design evolves
  ⇒ Important issues are reflected in changes to the code!
- Code only exposes static structure
  ⇒ Study dynamic behaviour to extract detailed design

Tie Code and Questions

Problem: How do you keep track of your understanding?

Solution: Annotate the code

- List questions, hypotheses, tasks and observations.
- Identify yourself!
- Use conventions to locate/extract annotations.
- Annotate as comments, or as methods

Conclusion

- Setting Direction + First Contact
  ⇒ First Project Plan

- Initial Understanding + Detailed Model Capture
  ⇒ Plan the work... and Work the plan
  ⇒ Frequent and Short Iterations

- Issues
  ⇒ scale
  ⇒ speed vs. accuracy
  ⇒ politics