Software Engineering

Prof. Dr. Bertrand Meyer
Dr. Manuel Oriol
Dr. Bernd Schoeller

Lectures 22: Legacy Software
Today

- Legacy Software
  - Definition and facts
- Reengineering
  - Refactoring
  - Reengineering
- Reverse Engineering
- Software Migration
  - Wrapping
  - Recoding
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

Source: 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 AND REENGINEERING
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.

- Risks: sometimes backward incompatible changes
Prof. Stéphane Ducasse

ducasse@iam.unibe.ch
http://www.iam.unibe.ch/~ducasse/

Slides from Stephane Ducasse
REVERSE ENGINEERING
3. Reverse Engineering

- What and Why
- Setting Direction
  - Most Valuable First
- First Contact
  - Chat with the Maintainers
  - Interview during Demo
- Initial Understanding
  - Analyze the Persistent Data
  - Study Exceptional Entities
- Detailed Model Capture
  - Tie Code and Questions
  - Step through the Execution
  - Look for the Contracts
- Conclusion

© S. Demeyer, S. Ducasse, O. Nierstrasz
SOFTWARE MIGRATION
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

```eiffel
feature {NONE} -- wrapped features
  genrand_int32: INTEGER is
    -- Encapsulation of a C routine with no parameter
    -- returning a the next integer in the mersene twister
    -- pseudo random suite of number
    external
      "C (): int | "%mt19937ar.h""
    alias
      "genrand_int32"
  end

  init_genrand(seed_32 : INTEGER_32) is
    -- Encapsulation of a C routine with one parameter
    -- the routine sets the seed of the mersene twister
    external
      "C (unsigned long) | "%mt19937ar.h"
    alias
      "init_genrand"
  end
```

From: http://merseneturisterrandom.origo.ethz.ch/ (tiny project)
Example: Apache and CGI Scripts

From Apache Tutorial:
http://www.ftt.co.uk/tutorials/Apache_tutorial4.html
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
  ☞ 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”!

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

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

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

© S. Demeyer, S. Ducasse, O. Nierstrasz
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

<table>
<thead>
<tr>
<th>Domain</th>
<th>LOC</th>
<th>Reengineering Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>pipeline planning</td>
<td>55,000</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
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

- **(0) requirement analysis**
- **(1) model capture**
  - people centric
  - lightweight
- **(2) problem detection**
- **(3) problem resolution**
- **(4) program transformation**

© S. Demeyer, S. Ducasse, O. Nierstrasz
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

Tests: Your Life Insurance

Detailed Model Capture

Initial Understanding

First Contact

Setting Direction

Migration Strategies

Detecting Duplicated Code

Redistribute Responsibilities

Transform Conditionals to Polymorphism

© S. Demeyer, S. Ducasse, O. Nierstrasz

Intro.24
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
YOU MUST LEARN THAT CHANCE IS GOOD.
CHANGE IS

ANY QUESTIONS?

WHO WANTS THIS ONE?
I GOT IT.

QUESTION: WHY DON'T YOU TRIPLE OUR PAY? THAT WOULD BE A CHANGE.

THAT WOULD NOT BE IN THE BEST INTEREST OF SHAREHOLDERS.

OKAY, WHY DON'T YOU WORK FOR FREE? THAT'S A CHANGE THAT'S GOOD FOR SHAREHOLDERS.

OR WOULD IT BE BETTER TO ADMIT THAT CHANGE CAN BE VERY BAD?

MY FAVORITE PART WAS WHEN HE YELLED, "STOP RUINING MY SLOGANS WITH YOUR LOGIC!"

© UFS, Inc.
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 the system’s 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
(cfr. 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

(0) req. analysis
(1) model capture
issues
- scale
- speed
- accuracy
- politics

(2) problem detection

(3) problem resolution

(4) program transformation

Requirements

Designs

Code

© S. Demeyer, S. Ducasse, O. Nierstrasz
Reverse Engineering.3
Forces — 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?
Setting Direction

Agree on Maxims

Set direction

Maintain direction

Coordinate direction

Most Valuable First

Where to start

Fix Problems, Not Symptoms

What to do

If It Ain't Broke Don't Fix It

What not to do

Keep it Simple

How to do it

Fix Problems, Not Symptoms

Keep it Simple

Principles & Guidelines for Software project management especially relevant for reengineering projects

© S. Demeyer, S. Ducasse, O. Nierstrasz

Reverse Engineering 5
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 — First Contact

• 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

Talk with developers

Chat with the Maintainers

Interview during Demo

Talk with end users

feasibility assessment (one week time)

Talk about it

Verify what you hear

Read All the Code in One Hour

Read it

Skim the Documentation

Read about it

Compile it

Do a Mock Installation

© S. Demeyer, S. Ducasse, O. Nierstrasz
Interview during Demo

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”
Initial Understanding

- Top down
  - Recover design
  - Speculate about Design
- Bottom up
  - Identify problems
  - Study the Exceptional Entities
  - Analyze the Persistent Data
  - Recover database

understand $\Rightarrow$ higher-level model
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
Detailed Model Capture

Tie Code and Questions

Keep track of your understanding

Expose design

Exposé the design & make sure it stays exposed

Refactor to Understand

Expose collaborations

Step through the Execution

Expose contracts

Look for the Contracts

Look for:
- Use Your Tools
- Look for Key Methods
- Look for Constructor Calls
- Look for Template/Hook Methods
- Look for Super Calls

Write Tests to Understand

Expose evolution

Learn from the Past

Reverse Engineering.15
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