Software Reengineering & Evolution

Serge Demeyer
Stéphane Ducasse
Oscar Nierstrasz

http://www.iam.unibe.ch/~scg/OORP
Schedule

1. Introduction
   There are OO legacy systems too!

2. Reverse Engineering
   How to understand your code

3. Visualization
   Scaleable approach

4. Restructuring
   How to Refactor Your Code

5. Code Duplication
   The most typical problems

6. Software Evolution
   Learn from the past

7. Conclusion
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- *outdated* development methods used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- *missing* or outdated documentation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
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, …)?

Relative Cost of Fixing Mistakes
- Requirement: $\times 1$
- Design: $\times 5$
- Coding: $\times 10$
- Testing: $\times 20$
- Delivery: $\times 200$

Between 50% and 75% of global effort is spent on “maintenance”!
Continuous Development

17.4% Corrective
(fixing reported errors)

18.2% Adaptive
(new platforms or OS)

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

data from [Lien78a]
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!
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*
What about Components?

Components are very brittle …
After a while one inevitably resorts to glue :)

© S. Demeyer, S. Ducasse, O. Nierstrasz
Object-Oriented Reengineering
Soccer Field Metaphor

- Assume 10 lines of code = 40 tiles of 1 x 1 cm
- 12.5 million lines of code ≈ 40 soccer fields

Imagine 400 developers concurrently moving tiles around on 40 soccer fields...

A. van Deursen, De software-evolutieparadox
Intreerede TU Delft, 23 feb 2005
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

Object-Oriented Reengineering.12
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
The Reengineering Life-Cycle

Requirements

(0) requirement analysis

(1) model capture

Designs

(2) problem detection

• people centric
• lightweight

Code

(3) problem resolution

(4) program transformation
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

Object-Oriented Reengineering.15
2. Reverse Engineering

• What and Why
• First Contact
  🎨 Interview during Demo
• Initial Understanding
  🎨 Analyze the Persistent Data
• Detailed Model Capture
  🎨 Look for the Contracts
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
First Contact

- Talk with System experts
  - Chat with the Maintainers
  - Talk with developers
- Talk with end users
  - Interview during Demo
- feasibility assessment (one week time)
  - Talk about it

Software System
- Read All the Code in One Hour
- Skim the Documentation
- Do a Mock Installation
- Read it
- Read about it
- Compile it
- Verify what you hear
First Project Plan

Use *standard templates*, including:

- **project scope**
  - see "Setting Direction"

- **opportunities**
  - e.g., skilled maintainers, readable source-code, documentation

- **risks**
  - e.g., absent test-suites, missing libraries, …
  - record likelihood (unlikely, possible, likely)
  - & impact (high, moderate, low) for causing problems

- **go/no-go decision**

- **activities**
  - fish-eye view
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?
Initial Understanding

Top down

Speculate about Design

Recover design

Bottom up

Study the Exceptional Entities

Identify problems

Analyze the Persistent Data

Recover database

Understand ⇒ higher-level model

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

Object-Oriented Reengineering
Analyze the Persistent Data

Problem: Which objects represent valuable data?

Solution: Analyze the database schema

- Prepare Model
  - tables ⇒ classes; columns ⇒ attributes
  - candidate keys (naming conventions + unique indices)
  - foreign keys (column types + naming conventions + view declarations + join clauses)

- Incorporate Inheritance
  - one to one; rolled down; rolled up

- Incorporate Associations
  - association classes (e.g. many-to-many associations)
  - qualified associations

- Verification
  - Data samples + SQL statements
Example: One To One

**Person**
- id: char(5)
- name: char(40)
- address: char(60)

**Salesman**
- id: char(5)
- company: char(40)

**Patient**
- id: char(5)
- insuranceID: char(7)
- insurance: char(5)

- id: char(5)
- name: char(40)
- address: char(60)
Example: Rolled Down

- **Salesman**
  - id: char(5)
  - name: char(40)
  - addresss: char(60)
  - company: char(40)

- **Patient**
  - id: char(5)
  - name: char(40)
  - addresss: char(60)
  - insuranceID: char(7)
  - insurance: char(5)

- **Person**
  - id: char(5)
  - name: char(40)
  - addresss: char(60)

- **Salesman**
  - id: char(5)
  - company: char(40)

- **Patient**
  - id: char(5)
  - insuranceID: char(7)
  - insurance: char(5)
Example: Rolled Up

Person
id: char(5)
name: char(40)
address: char(60)
insuranceID: char(7) «optional»
insurance: char(5) «optional»
company: char(40) «optional»

Salesman
id: char(5)
company: char(40)

Patient
id: char(5)
insuranceID: char(7)
insurance: char(5)

Person
id: char(5)
name: char(40)
address: char(60)
Example: Qualified Association

Patient
id: char(5)
...

Treatment
patientID: char(5)
date: date
nr: integer
comment: varchar(255)

Patient
id: char(5)
...
addTreatment(d, n, t)
lookupTreatment(d, n)

date: Date
nr: Integer
1
1

Treatment
comment: Text
Initial Understanding (revisited)

- Top down
  - Speculate about Design
  - Study the Exceptional Entities
  - Analyze the Persistent Data

- Bottom up
  - Recover database
  - Recover design
  - Identify problems

understand ⇒ higher-level model
3. Software Visualization

- Introduction
  - The Reengineering life-cycle
- Examples
- Lightweight Approaches
  - CodeCrawler
- Dynamic Analysis
- Conclusion
The Reengineering Life-cycle

1. **Requirements**
   - (0) requirement analysis

2. **Designs**
   - (2) problem detection
     - issues
     - Tool support
     - Scalability
     - Efficiency
   - (3) problem resolution

3. **Code**
   - (1) model capture

4. **Program transformation**
Visualising Hierarchies

- **Euclidean cones**
  - **Pros:**
    - More info than 2D
  - **Cons:**
    - Lack of depth
    - Navigation

- **Hyperbolic trees**
  - **Pros:**
    - Good focus
    - Dynamic
  - **Cons:**
    - Copyright
Bottom Up Visualisation

All program entities and relations

Filter
A lightweight approach

• A combination of metrics and software visualization
  ➤ Visualize software using colored rectangles for the entities and edges for the relationships
  ➤ Render up to five metrics on one node:
    • Size (1+2)
    • Color (3)
    • Position (4+5)
System Complexity View

Nodes: Classes
Edges: Inheritance Relationships
Width: Number of attributes
Height: Number of methods
Color: Number of lines of code
Inheritance Classification View

Boxes: Classes
Edges: Inheritance
Width: Number of Methods Added
Height: Number of Methods Overridden
Color: Number of Method Extended
Data Storage Class Detection View

| Boxes:    | Classes          |
| Width:    | Number of Methods|
| Height:   | Lines of Code    |
| Color:    | Lines of Code    |
Industrial Validation

Personal experience
2-3 days to get something

Nokia  (C++ 1.2 MLOC >2300 classes)
Nokia  (C++/Java 120 kLOC >400 classes)
MGeniX  (Smalltalk 600 kLOC >2100 classes)
Bedag  (COBOL  40 kLOC)
...

Used by developers + Consultants
Program Dynamics

- Simple
- Reproducible
- Scales well

Figure 6: Inter-class call matrix
Frequency Spectrum

- Visualization of similarities in event traces
- Eliminate similarities
### Key Concept Identification

<table>
<thead>
<tr>
<th>Class</th>
<th>IC-CC' +</th>
<th>Ant docs</th>
<th>Recall (%)</th>
<th>Precision (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>✓</td>
<td>✓</td>
<td>90</td>
<td>-</td>
</tr>
<tr>
<td>UnknownElement</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Task</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Main</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>IntrospectionHelper</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ProjectHelper</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RuntimeConfigurable</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Target</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ElementHandler</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TaskContainer</td>
<td>×</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- Extract run-time coupling
- Apply datamining (“google”)
- Experiment with documented open-source cases (Ant, JMeter)
  - recall: +- 90 %
  - precision: +- 60 %
Replication

Replication is not supported, industrial cases are rare, …. In order to help the discipline mature, we think that more systematic empirical evaluation is needed. [Tonella et. Al, in Empirical Software Engineering]

Pilot Study: ATM Simulation

Assumptions

- **Feature**: invoked from the outside.
- **Map**: scenario-feature map exists
- **Recompile**: recompile or instrumentation possible
- **Isolate**: system can run in isolation (prevent noise)
- **Manual**: perform dynamic analysis without help (i.e. no operator)
- **Generic**: no limit to granularity of computational unit
Case Study: Portfolio Management

<table>
<thead>
<tr>
<th>it</th>
<th>name</th>
<th>scenarios description</th>
<th>features</th>
<th>value</th>
<th>spread</th>
<th>report</th>
<th>cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>ptf</td>
<td>inspect portfolio</td>
<td>x</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ptf-sprd</td>
<td>inspect spread of portfolio</td>
<td>x</td>
<td>x</td>
<td>n/a</td>
<td>n/a</td>
<td></td>
</tr>
<tr>
<td>2nd</td>
<td>ptf</td>
<td>inspect portfolio</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ptf2</td>
<td>inspect portfolio again</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>report</td>
<td>generate report</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>report-sprd</td>
<td>generate report with spread</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2nd iteration

3rd iteration

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

Most common situations

**Transform Conditionals to Polymorphism**
- Transform Self Type Checks
- Transform Provider Type Checks

**Redistribute Responsibilities**
- Move Behaviour Close to Data
- Eliminate Navigation Code
- Split up God Class
- Empirical Validation
Transform Conditionals to Polymorphism

- Test provider type
  - Transform Client Type Checks
    - Test null values
      - Introduce Null Object

- Test self type
  - Transform Self Type Checks
    - Factor Out Strategy
      - Test object state
        - Factor Out State

- Test external attribute
  - Transform Conditionals into Registration
    - Test object state
Example: Transform Conditional

class Message {
    private:
        int type_; void* data;

    void send (Channel* ch) {
        switch (type_) {
            case TEXT : {
                ch->nextPutAll(data);
                break;
            }
            case ACTION : {
                ch->doAction(data);
            }
        }
    }

⇒ Transform **Self** Type Checks

void makeCalls (Telephone* phoneArray []) {
    for (Telephone *p = phoneArray; p; p++) {
        switch (p-> phoneType()) {
            case TELEPHONE::POTS : {
                POTSPhone* pots = (POTSPhone*)p
                pots->tourne();
                pots->call();...
            }
            case TELEPHONE::ISDN : {
                ISDNPhone* isdn = (ISDNPhone*)p
                isdn->initLine();
                isdn->connect();...
            }
        }
    }

⇒ Transform **Client** Type Checks
Transform Self Type Check

```plaintext
switch (type_) {
  case TEXT:
    ch->nextPutAll(data);
    break;
  case ACTION:
    ch->doAction(data);
  ...
}
```
Transform Client Type Check

TelephoneBox
makeCall()

Telephone

POTSPhone
... makeCall()

ISDNPhone
... makeCall()

TelephoneBox
makeCall()

Telephone
makeCall()

POTSPhone
... makeCall()

ISDNPhone
makeCall()
Redistribute Responsibilities

- Eliminate Navigation Code
  - Move Behaviour Close to Data
  - Data containers
  - Chains of data containers

- Split Up God Class
  - Monster client of data containers
Move Behavior Close to Data (example 1/2)

Employee
+telephoneNrs
+name(): String
+address(): String

Payroll
+printEmployeeLabel()

TelephoneGuide
+printEmployeeTelephones()

System.out.println(currentEmployee.name());
System.out.println(currentEmployee.address());
for (int i=0; i < currentEmployee.telephoneNumbers.length; i++) {
    System.out.print(currentEmployee.telephoneNumbers[i]);
    System.out.print(" ");
}
System.out.println(" ");
... for ...
System.out.print(" -- ");
...
Move Behavior Close to Data (example 2/2)

```
public void printLabel (String separator) {
    System.out.println(_name);
    System.out.println(_address);
    for (int i=0; i < telephoneNumbers.length; i++) {
        System.out.print(telephoneNumbers[i]);
        System.out.print(separator);
    }
    System.out.println();
}
```

```
emp.printLabel("  ");
... emp.printLabel(" -- ");
... emp.printLabel(" " );
... emp.printLabel("  ");
... emp.printLabel(" -- ");
...
Eliminate Navigation Code

Car - engine + increaseSpeed()

Engine - carburetor + speedUp()

Carburetor + fuelValveOpen

Carburetor - fuelValveOpen + openFuelValve()

fuelValveOpen = true

engine.carburetor.fuelValveOpen = true

carburetor.openFuelValve()

carburetor.fuelValveOpen = true

... engine.speedUp()
Split Up God Class

**Problem:** Break a class which monopolizes control?

**Solution:** Incrementally eliminate navigation code

- Detection:
  - measuring size
  - class names containing Manager, System, Root, Controller
  - the class that all maintainers are avoiding

- How:
  - move behaviour close to data + eliminate navigation code
  - remove or deprecate façade

- However:
  - If God Class is stable, then don't split
    ⇒ shield client classes from the god class
Split Up God Class: 5 variants

Mail client filters incoming mail

A

Controller

Extract behavioral class

B

Controller

Filter1

Filter2

MailHeader

Extract data class

C

Controller

Filter1

Filter2

MailHeader

Extract behavioral class

D

Controller

Filter1

Filter2

MailHeader

Extract data class

E

Controller

Filter1

Filter2

MailHeader

Extract data class

NameValuePair
Empirical Validation

• **Controlled experiment** with 63 last-year master-level students (CS and ICT)

**Independent Variables**
- Institution
- God class decomposition

**Dependent Variables**
- Experimental task
- Accuracy
- Time

© S. Demeyer, S. Ducasse, O. Nierstrasz
Interpretation of Results

• “Optimal decomposition” differs w.r.t. curriculum
  ☞ Computer science: preference towards C-E
  ☞ ICT-electronics: preference towards A-C

• Advanced OO training can induce a preference towards particular styles of decomposition
  ☞ Consistent with [Arisholm et al. 2004]
5. Code Duplication

a.k.a. Software Cloning, Copy&Paste Programming

- **Code Duplication**
  - What is it?
  - Why is it harmful?
- Detecting Code Duplication
- Approaches
- A Lightweight Approach
- Visualization (dotplots)
- Duploc
The Reengineering Life-Cycle

(0) requirement analysis

(1) model capture

(2) problem detection

(3) problem resolution

Requirements

Designs

Code

(2) Problem detection

issues
- Scale
- Unknown a priori
Code is Copied

Small Example from the Mozilla Distribution (Milestone 9)
Extract from /dom/src/base/nsLocation.cpp

```c
NS_IMETHODIMP
LocationImpl::GetPathname(nsString
                        { nsAutoString href;
                          nsIURI *url;
                          nsresult result = NS_OK;
                          result = GetHref(href);
                          if (NS_OK == result) {
                            #ifndef NECKO
                              result = NS_NewURL(&url, href);
                            #else
                              result = NS_NewURI(&url, href);
                            #endif
                            if (NS_OK == result) {
                              #ifdef NECKO
                                char* file;
                                result = url->GetPath(&file);
                                if (result == NS_OK) {
                                  aPathname.SetString(file);
                                  #ifdef NECKO
                                    nsCRT::free(file);
                                  #endif
                                  NS_IF_RELEASE(url);
                                }
                              #endif
                            }
                          }
                          return result;
                        }

NS_IMETHODIMP
LocationImpl::SetPathname(const nsString
                        { nsAutoString href;
                          nsIURI *url;
                          nsresult result = NS_OK;
                          result = GetHref(href);
                          if (NS_OK == result) {
                            #ifndef NECKO
                              result = NS_NewURL(&url, href);
                            #else
                              result = NS_NewURI(&url, href);
                            #endif
                            if (NS_OK == result) {
                              char *buf = aPathname.ToNewCString();
                              #ifdef NECKO
                                url->SetPath(buf);
                              #else
                                url->SetFile(buf);
                              #endif
                              SetURL(url);
                              delete [] buf;
                              NS_RELEASE(url);
                            } return result;
                          }
                        }

NS_IMETHODIMP
LocationImpl::GetPort(nsString& aPort)
                        { nsAutoString href;
                          nsIURI *url;
                          nsresult result = NS_OK;
                          result = GetHref(href);
                          if (NS_OK == result) {
                            #ifndef NECKO
                              result = NS_NewURL(&url, href);
                            #else
                              result = NS_NewURI(&url, href);
                            #endif
                            if (NS_OK == result) {
                              aPort.SetLength(0);
                              #ifdef NECKO
                                PRInt32 port;
                                (void)url->GetPort(&port);
                              #endif
                              if (-1 != port) {
                                aPort.Append(port, 10);
                              }
                              NS_RELEASE(url);
                            } return result;
                          }
```
How Much Code is Duplicated?

Usual estimates: 8 to 12% in normal industrial code
15 to 25 % is already a lot!

<table>
<thead>
<tr>
<th>Case Study</th>
<th>LOC</th>
<th>Duplication without comments</th>
<th>with comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>460’000</td>
<td>8.7%</td>
<td>5.6%</td>
</tr>
<tr>
<td>Database Server</td>
<td>245’000</td>
<td>36.4%</td>
<td>23.3%</td>
</tr>
<tr>
<td>Payroll</td>
<td>40’000</td>
<td>59.3%</td>
<td>25.4%</td>
</tr>
<tr>
<td>Message Board</td>
<td>6’500</td>
<td>29.4%</td>
<td>17.4%</td>
</tr>
</tbody>
</table>
Copied Code Problems

- General negative effect:
  - Code bloat
- Negative effects on *Software Maintenance*
  - Copied Defects
  - Changes take double, triple, quadruple, ... Work
  - Dead code
  - Add to the cognitive load of future maintainers
- Copying as additional source of defects
  - Errors in the systematic renaming produce unintended aliasing
- Metaphorically speaking:
  - Software Aging, “hardening of the arteries”,
  - “Software Entropy” increases even small design changes become very difficult to effect
Code Duplication Detection

Nontrivial problem:
- No a priori knowledge about which code has been copied
- How to find all clone pairs among all possible pairs of segments?

Lexical Equivalence
Syntactical Equivalence
Semantic Equivalence
General Schema of Detection Process

<table>
<thead>
<tr>
<th>Author</th>
<th>Level</th>
<th>Transformed Code</th>
<th>Comparison Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>[John94a]</td>
<td>Lexical</td>
<td>Substrings</td>
<td>String-Matching</td>
</tr>
<tr>
<td>[Duca99a]</td>
<td>Lexical</td>
<td>Normalized Strings</td>
<td>String-Matching</td>
</tr>
<tr>
<td>[Bake95a]</td>
<td>Syntactical</td>
<td>Parameterized Strings</td>
<td>String-Matching</td>
</tr>
<tr>
<td>[Mayr96a]</td>
<td>Syntactical</td>
<td>Metric Tuples</td>
<td>Discrete comparison</td>
</tr>
<tr>
<td>[Kont97a]</td>
<td>Syntactical</td>
<td>Metric Tuples</td>
<td>Euclidean distance</td>
</tr>
<tr>
<td>[Baxt98a]</td>
<td>Syntactical</td>
<td>AST</td>
<td>Tree-Matching</td>
</tr>
</tbody>
</table>
Simple Detection Approach (i)

- **Assumption:**
  - Code segments are just copied and changed at a few places

- **Code Transformation Step**
  - remove white space, comments
  - remove lines that contain uninteresting code elements (e.g., just ‘else’ or ‘}’)

```c
//assign same fastid as container
fastid = NULL;
const char* fidptr = get_fastid();
if(fidptr != NULL) {
    int l = strlen(fidptr);
    fastid = newchar[ l + 1 ];
}
```

```c
fastid=NULL;
const char*fidptr=get_fastid();
if(fidptr!=NULL)
intl=strlen(fidptr)
fastid = newchar[1+]
```
Simple Detection Approach (ii)

- **Code Comparison Step**
  - Line based comparison (Assumption: Layout did not change during copying)
  - Compare each line with each other line.
  - Reduce search space by hashing:
    1. Preprocessing: Compute the hash value for each line
    2. Actual Comparison: Compare all lines in the same hash bucket

- **Evaluation of the Approach**
  - Advantages: Simple, language independent
  - Disadvantages: Difficult interpretation
A Perl script for C++ (i)

```perl
while (<>) {
    chomp;
    $totalLines++;

    # remove comments of type /* */
    my $codeOnly = "";
    while(($inComment && m|/*|) || (!$inComment && m|/\*|)) {
        unless($inComment) {
            $codeOnly .= $`
        }
        $codeOnly = !$inComment;
        $inComment = !$inComment;
        $_ = $'
    }

    # remove comments of type //
    $codeOnly .= $_ unless $inComment;
    $_ = $codeOnly;

    # remove white space
    s//\s+/g;

    # remove keywords
    s/$keywordsRegExp//og if $removeKeywords;
}
```

$equivalenceClassMinimalSize = 1;
$slidingWindowSize = 5;
$removeKeywords = 0;
@keywords = qw(if then else);

@keywordsRegEx = join '|', @keywords;

@unwantedLines = qw( else return return
                      { }
                      ;
                    );
push @unwantedLines, @keywords;
```

© S. Demeyer, S. Ducasse, O. Nierstrasz Object-Oriented Reengineering.67
A Perl script for C++ (ii)

```
$codeLines++; push @currentLines, $_; push @currentLineNos, $.;
if($slidingWindowSize < @currentLines) {
    shift @currentLines;
    shift @currentLineNos;}
#print STDERR "Line $totalLines >$_<\n";
my $lineToBeCompared = join ', @currentLines;
my $lineNumbersCompared = "<$ARGV">'; # append the name of the file
$lineNumbersCompared .= join '/', @currentLineNos;
#print STDERR "$lineNumbersCompared\n";
if($bucketRef = $eqLines{$lineToBeCompared}) {
    push @$bucketRef, $lineNumbersCompared;
} else {$eqLines{$lineToBeCompared} = [
    $lineNumbersCompared ];}
if(eof) { close ARGV } # Reset linenumber-count for next file
```

- Handles multiple files
- Removes comments and white spaces
- Controls noise (if, {,}
- Granularity (number of lines)
- Possible to remove keywords
Output Sample

Lines:
create_property(pd,pnImplObjects,stReference,false,*iImplObjects);
create_property(pd,pnElttype,stReference,true,*iEltType);
create_property(pd,pnMinelt,stInteger,true,*iMinelt);
create_property(pd,pnMaxelt,stInteger,true,*iMaxelt);
create_property(pd,pnOwnership,stBool,true,*iOwnership);
Locations: </face/typesystem/SCTypesystem.C>6178/6179/6180/6181/6182
</face/typesystem/SCTypesystem.C>6198/6199/6200/6201/6202

Lines:
create_property(pd,pnSupertype,stReference,true,*iSupertype);
create_property(pd,pnImplObjects,stReference,false,*iImplObjects);
create_property(pd,pnElttype,stReference,true,*iEltType);
create_property(pd,pMinelt,stInteger,true,*iMinelt);
create_property(pd,pMaxelt,stInteger,true,*iMaxelt);
Locations: </face/typesystem/SCTypesystem.C>6177/6178
</face/typesystem/SCTypesystem.C>6229/6230

Lines = duplicated lines
Locations = file names and line number
Visualization of Duplicated Code

• Visualization provides insights into the duplication situation
• A simple version can be implemented in three days
• Scalability issue

• Dotplots — Technique from DNA Analysis
  • Code is put on vertical as well as horizontal axis
  • A match between two elements is a dot in the matrix

© S. Demeyer, S. Ducasse, O. Nierstrasz
Visualization of Copied Code Sequences

**Detected Problem**
File A contains two copies of a piece of code

File B contains another copy of this code

**Possible Solution**
Extract Method

All examples are made using Duploc from an industrial case study (1 Mio LOC C++ System)
Visualization of Repetitive Structures

**Detected Problem**
4 Object factory clones: a switch statement over a type variable is used to call individual construction code

**Possible Solution**
Strategy Method
**Visualization of Cloned Classes**

**Detected Problem:**
Class A is an edited copy of class B. Editing & Insertion

**Possible Solution**
Subclassing …
Visualization of Clone Families

20 Classes implementing lists for different data types
6. Software Evolution

• Exploiting the Version Control System
  ☞ Visualizing CVS changes
• The Evolution Matrix
• Yesterday's weather

It is not *age* that turns a piece of software into a legacy system, but the *rate* at which it has been developed and adapted without being reengineered.

[Demeyer, Ducasse and Nierstrasz: Object-Oriented Reengineering Patterns]
The Reengineering Life-Cycle

(0) Requirement analysis

(1) Model capture

(2) Problem detection
  - Issues
    - Scale

(3) Problem resolution

Requirements

Designs

Code
Analyse CVS changes

1) Vertical lines = Frequent Changers
2) Horizontal line = Shotgun Surgery
3) Triangle = Core Reduces
4) Block Shift = Design Change
Pulsar & Supernova

**Pulsar:** Repeated Modifications make it grow and shrink. System Hotspot: Every System Version requires changes.

**Supernova:** Sudden increase in size. Possible Reasons:
- Massive shift of functionality towards a class.
- Data holder class for which it is easy to grow.
- *Sleeper:* Developers knew exactly what to fill in.
Example: MooseFinder (38 Versions)
Yesterday’s Weather: Stability of Changes
Test history

integration tests

d… affect unit tests

test+production

phased testing

single test
7. Conclusion

1. **Introduction**
   There are OO legacy systems too!

2. **Reverse Engineering**
   How to understand your code

3. **Visualization**
   Scaleable approach

4. **Restructuring**
   How to Refactor Your Code

4. **Code Duplication**
   The most typical problems

5. **Software Evolution**
   Learn from the past

6. **Conclusion**
   Did we convince you?
Goals

We will try to convince you:

• Yes, Virginia, there are *object-oriented legacy systems* too!
  ☞ … actually, that's a sign of health

• Reverse engineering and reengineering are *essential activities* in the lifecycle of any successful software system. (And especially OO ones!)
  ☞ … consequently, do not consider it second class work

• There is a large set of *lightweight tools and techniques* to help you with reengineering.
  ☞ … check our book, but remember the list is growing

• Despite these tools and techniques, *people must do job* and represent the most valuable resource.
  ☞ … pick them carefully and reward them properly

⇒ Did we convince you?