SOFTWARE ENGINEERING

SOFTWARE EVOLUTION

Saulius Ragaišis
saulius.ragaisis@mif.vu.lt
CSC2008 SE “Software Evolution”

Learning Objectives:

• Identify the principal issues associated with software evolution and explain their impact on the software life cycle.
• Discuss the challenges of maintaining legacy systems and the need for reverse engineering.
• Outline the process of regression testing and its role in release management.
• Estimate the impact of a change request to an existing product of medium size.
• Develop a plan for re-engineering a medium-sized product in response to a change request.
• Discuss the advantages and disadvantages of software reuse.
• Exploit opportunities for software reuse in a given context.
• Identify weaknesses in a given simple design, and highlight how they can be removed through refactoring.
SOFTWARE MAINTENANCE
Concepts

• Software Maintenance
• Software Support
• Software Maintenance and Support
• Software Operation
Software maintenance

• SM is the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment [IEEE].

• The purpose of the SM process is to provide cost-effective support to a delivered software product. Pre-delivery SM activities include planning for post-delivery operations, supportability, and logistics determination. Post-delivery activities include software modification and operational support, such as training or operating a help desk [ISO/IEC 12207].
Software operation

• The purpose of the SO process is to operate the software product in its intended environment and to provide support to the customers of the software product.
Challenging problems of SM

• Program comprehension
• Impact analysis
• Regression testing
Maintenance costs

• Software maintenance consumes 60% to 80% of the total life cycle costs.
• Maintenance costs are largely due to enhancements (often 75–80%), rather than corrections.
• Program comprehension ranges from 50% up to 90% of maintenance time.
Maintainability

- The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. [IEEE Std. 610.12]

- Maintainability is one of the six primary characteristics of software quality and it depends on 5 sub-characteristics: analyzability, changeability, stability, testability, and compliance. [ISO/IEC 9126]
The second law of Lehman

As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving the semantics and simplifying the structure.
Categories of software maintenance

E.B. Swanson, R.S. Pressman

- Corrective maintenance
- Adaptive maintenance
- Perfective maintenance (or enhancement)
- Preventive maintenance (or reengineering)
ISO/IEC 9000-3 categories

• Problem resolution, which involves the detection, analysis, and correction of software nonconformities causing operational problems;

• Interface modifications, required when additions or changes are made to the hardware system controlled by the software;

• Functional expansion or performance improvement, which may be required by the purchaser in the maintenance stage.
IEEE categories

• **Corrective maintenance**: reactive modification of a software product performed after delivery to correct discovered faults.

• **Adaptive maintenance**: modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment.

• **Perfective maintenance**: modification of a software product performed after delivery to improve performance or maintainability.

• **Emergency maintenance**: unscheduled corrective maintenance performed to keep a system operational.
## Classification of IEEE categories

<table>
<thead>
<tr>
<th></th>
<th>Unscheduled</th>
<th>Scheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reactive</strong></td>
<td>Emergency</td>
<td>Corrective Adaptive</td>
</tr>
<tr>
<td><strong>Proactive</strong></td>
<td></td>
<td>Perfective</td>
</tr>
</tbody>
</table>
Quick-fix model
Full-reuse model

<table>
<thead>
<tr>
<th>Old System</th>
<th>Repository</th>
<th>New System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>{R_i}</td>
<td>Requirements</td>
</tr>
<tr>
<td>Design</td>
<td>{D_i}</td>
<td>Design</td>
</tr>
<tr>
<td>Code</td>
<td>{C_i}</td>
<td>Code</td>
</tr>
<tr>
<td>Test</td>
<td>{T_i}</td>
<td>Test</td>
</tr>
</tbody>
</table>
IEEE-1219 maintenance processes

• Problem/modification identification, classification, and prioritization
• Analysis
• Design
• Implementation
• Regression/system testing
• Acceptance testing
• Delivery
IEEE-1219 maintenance process
Maintenance management

• Planning
• Organizing
• Staffing
• Leading
• Controlling
LEGACY SYSTEMS
Legacy systems

The systems developed over the past 20/30 years (or even more). They have typically been conceived in a mainframe environment using non-standard development techniques and obsolete programming languages. The structure has often been degraded by a long history of changes and adaptations and neither consistent documentation nor adequate test suites are available. Nevertheless, these are crucial systems to the business they support (most legacy systems hold terabytes of live data) and encapsulate a great deal of knowledge and expertise of the application domain. Sometimes the legacy code is the only place where domain knowledge and business rules are recorded.
Definitions

• Large software systems that we don’t know how to cope with but that are vital to our organization. [K. H. Bennett]

• An information system that significantly resists modifications and evolution to meet new and constantly changing business requirements. [M. L. Brodie, M. Stonebraker]
Typical solutions

• discarding the legacy system and building a replacement system;
• freezing the system and using it as a component of a new larger system;
• carrying on maintaining the system for another period;
• modifying the system to give it another lease of life.
Value factors of a legacy system

- obsolescence,
- deterioration,
- decomposability, and
- business value.
A life cycle model for legacy systems
REENGINEERING
Business process reengineering

• Business process reengineering (BPR): “the search for, and the implementation of, radical change in business process to achieve breakthrough results”. [T.A. Stewart]

• BPR extends far beyond the scope of information technologies and software engineering.
BPR model

1. Process identification
2. Process specification and design
3. Prototyping
4. Refinement and instantiation
5. Business definition

Software reengineering
Economics of reengineering

Costs:
- $C_{cm}$ – current maintenance per year
- $C_{co}$ – current operation per year
- $C_{pm}$ – predicted maintenance per year
- $C_{po}$ – predicted operation per year
- $C_{re}$ – estimated reengineering

Business value of application:
- $BV_c$ – current
- $BV_p$ – predicted

Reengineering:
- $T_{re}$ – estimated calendar time
- $RF_{re}$ – risk factor (1.0 is nominal)
- $L$ – expected life of the system
Economics of reengineering (2)

Costs_{maint} = 
\[ BV_c - (C_{cm} + C_{co}) \] * L

Costs_{reeng} = 
\[ BV_p - (C_{pm} + C_{po}) \] * (L - T_{re}) - (C_{re} * RF_{re})

Cost benefit = 
Costs_{maint} - Costs_{reeng}
What is Refactoring?

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring. [Martin Fowler]
Why Should You Refactor?

- Refactoring Improves the Design of Software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster
When Should You Refactor?

- The Rule of Three
- Refactor When You Add Function
- Refactor When You Need to Fix a Bug
- Refactor As You Do a Code Review
Why Refactoring Works?

• Programs that are hard to read are hard to modify.
• Programs that have duplicated logic are hard to modify.
• Programs that require additional behavior that requires you to change running code are hard to modify.
• Programs with complex conditional logic are hard to modify.
Refactoring. Composing Methods

Long methods are troublesome because they often contain lots of information, which gets buried by the complex logic that usually gets dragged in. [http://sourcemaking.com/refactoring]

• Extract Method
• Inline Method
• Inline Temp
• Introduce Explaining Variable
• Remove Assignments to Parameters
• Replace Method with Method Object
• Replace Temp with Query
• Split Temporary Variable
• Substitute Algorithm
Refactoring. Moving Features Between Objects

One of the most fundamental decision in object design is deciding where to put responsibilities.

- Extract Class
- Hide Delegate
- Inline Class
- Introduce Foreign Method
- Introduce Local Extension
- Move Field
- Move Method
- Remove Middle Man
Refactoring. Organizing Data

- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Duplicate Observed Data
- Encapsulate Collection
- Encapsulate Field
- Replace Array with Object
- Replace Data Value with Object
- Replace Magic Number with Symbolic Constant
- Replace Record with Data Class
- Replace Subclass with Fields
- Replace Type Code with Class
- Replace Type Code with State/Strategy
- Replace Type Code with Subclasses
- Self Encapsulate Field
Refactoring. Simplifying Conditional Expressions

- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Decompose Conditional
- Introduce Assertion
- Introduce Null Object
- Remove Control Flag
- Replace Conditional with Polymorphism
- Replace Nested Conditional with Guard Clauses
Refactoring. Making Method Calls Simpler

The refactorings that make interfaces more straightforward.

- Add Parameter
- Encapsulate Downcast
- Hide Method
- Introduce Parameter Object
- Parameterize Method
- Preserve Whole Object
- Remove Parameter
- Remove Setting Method
- Rename Method
- Replace Constructor with Factory Method
- Replace Error Code with Exception
- Replace Exception with Test
- Replace Parameter with Explicit Methods
- Replace Parameter with Method
- Separate Query from Modifier
Refactoring. Dealing with Generalization

These refactorings mostly deal with moving methods around a hierarchy of inheritance.

- Collapse Hierarchy
- Extract Interface
- Extract Subclass
- Extract Superclass
- Form Template Method
- Pull Up Constructor Body
- Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Replace Delegation with Inheritance
- Replace Inheritance with Delegation
Refactoring. Big Refactorings

You are refactoring to some purpose, not just to avoid making progress (at least usually you are refactoring to some purpose). What does the whole game look like?

- Convert Procedural Design to Objects
- Extract Hierarchy
- Separate Domain from Presentation
- Tease Apart Inheritance
- The Nature of the Game
SOFTWARE REUSE
Definition

Software reuse is the process of creating software systems from predefined software components.

• The systematic development of reusable components.
• The systematic reuse of these components as building blocks to create new systems.

A reusable component may be code, but the bigger benefits of reuse come from a broader and higher-level view of what can be reused. Software specifications, designs, tests cases, data, prototypes, plans, documentation, frameworks, and templates are all candidates for reuse.
The advantage of software reuse

- Increase software productivity.
- Shorten software development time.
- Improve software system interoperability.
- Develop software with fewer people.
- Move personnel more easily from project to project.
- Reduce software development and maintenance costs.
- Produce more standardized software.
- Produce better quality software and provide a powerful competitive advantage.
Factors of reuse

• Technical
  – Programming language support
  – Repositories
  – Tools for searching reusable artifacts

• Non-technical
  – Organization
  – Process
  – Business drivers
  – Human involvement
Some conclusions from reuse survey

- Type of software production has predictive importance. Although type of software and application domain has no significance.
- OO ≠ Reuse.
- All successful projects had top management commitment. Projects with top management commitment failed as well.
- Human factors must be addressed.
- Introducing key reuse roles or setting up a repository are not sufficient for successful reuse.
- A reward policy to promote reuse never succeeded.
- Qualification of reuse assets and configuration managements are important.
Key factors in adopting a reuse strategy

• Reuse potential
  – Product family has more potential

• Reuse capability
  – Obtain management commitment for resources

• Reuse implementation
  – Add reuse specific processes
  – Modify non-reuse processes
  – Address human factors
  – Setup a repository
  – Qualify assets
Reuse techniques

• High Level Languages
• Source code components
• Software architectures
• Design and code scavenging
• Application generators
• Transformational systems
• Very High Level Languages (VHLL)
• Software schemas
Designing for reuse

- defining the interface/specification
- defining the implementation
- object oriented programming/inheritance
- generics/templates
- module decomposition/parameter passing
- user defined types/strong typing
- portability features
- generality
Examples of reuse humor

Look

http://alpha.fdu.edu/~levine/reuse_course/humor/index.html
What we have learned?

• Software maintenance
• Legacy systems
• Reengineering
• Refactoring
• Software reuse
QUESTIONS?