Introduction to Software Engineering

14. Software Evolution

Based on a lecture by Oscar Nierstarz and the SDE Course at the University of Bern.
Roadmap

- Laws of Software Evolution
- Mining Software Evolution
- Reengineering Software
- Tools for Evolution
MY NEW DESIGN WILL MEET ALL OF OUR CUSTOMERS’ CURRENT AND FUTURE NEEDS
Lehman & Belady’s Classification of Systems

- P-type
- S-type
- E-Type

IBM System 360
Two of Lehman & Belady’s Laws of Software Evolution

**Continuing change**
— A program that is used in a real-world environment *must change*, or become progressively less useful in that environment.

**Increasing entropy/complexity**
— As a program evolves, it becomes *more complex*, and extra resources are needed to preserve and simplify its structure.

Continuous Development

The bulk of the maintenance/evolution cost is due to new functionality ⇒ even with better requirements, it is hard to predict new functions

data from [Lien78a]
So what to do?

Before:
Build a family of solutions
So what to do?

After: Reengineer
Never try to rewrite the system. There is too much knowledge encoded in the running system.
Laws about people

*Characterizing people as non-linear, first-order components in software development*, Alistair Cockburn

— Communication degradation
— Inconsistency of people
— Good citizenship
— Diversity of people
It’s all about people: Peopleware

- Project management
- Work environment
- The concept of “flow”
“You know that what you need to do is possible to do, even though difficult, and sense of time disappears. You forget yourself. You feel part of something larger.”

Mihaly Csikszentmihalyi on experiencing ‘flow’
Roadmap

- Laws of Software Evolution
- **Mining Software Evolution**
- Reengineering Software
- Tools for Evolution
20 years of **VCS** before people start doing research in analyzing software repositories.

20 more years until software evolution research results are integrated in the IDE.
Learn about how people build software
SC Seminar Study by Julian S. & Roger K.

Are people migrating to the new concurrency libraries of Java?

Source: GitHub
Projects: 880
Size: 16,5 GB
Can we learn from developer behavior?
Logical Coupling

> Learning from VCS history
  — by Gall et al.
  — things that changed together might change again together
  — advantages over static analysis
    - *can detect IO*
    - *works beyond source code*
API Specification Mining

Data mining reveals frequent patterns
- Matching Method Pairs
- State Machines

Principles
1. Mines from history
2. API specific errors
3. Co-addition is a pattern
4. Small commits are fixes
Piatra Craiului
Mountains, Romania

“You can always make another step”
“You can always make another step”

“You can always take another step”

( wins in googlefight against “make” )
Clone analysis for querying

14% - 17% of methods in SqueakSource are clones

On how often is code cloned across repositories

Schwarz et al.
Build better development tools
Replaying past changes in multi-developer projects

Lile Hattori et al.
Evaluation
+ Correctness (10%)
+ Completion time (6%)

But the coolest thing (imho)
+ It proved useful during its own development!
Recording IDE Interactions

> Kersten & Murphy ‘05
  — Mylin
  — Task-Focused Interface
  — Degree of Interest ranking

> Robbes ’08
  — Spyware
  — Applications in Reverse Engineering
  — Code completion

How to filter the large amount of information available in the IDE?
Roadmap

- Laws of Software Evolution
- Mining Software Evolution
- Reengineering Software
- Tools for Evolution
Common Symptoms of Software Aging (2)

Process Failuers

> too long to turn things over to production
> constant bug fixes
> difficulties separating products

⇒ simple changes take too long

Code symptoms

• duplicated code
• technical debt
• code smells (big build times?)
Common Symptoms of Software Aging

Lack of Knowledge

> *obsolete* or no documentation

> *departure* of the original developers or users

> *limited understanding* of entire system (*missing tests?*)
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* = 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
The Reengineering Life-Cycle

(0) requirement analysis

(2) problem detection

(1) model capture

(3) problem resolution

• people centric
• lightweight

(4) program transformation

Requirements

Designs

Code
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 Reengineering

> **Untangling**
> — split a monolithic system into parts that can be separately marketed
> — increase the understandability of the code

> **Performance**
> — “first do it, then do it right, then do it fast” — experience shows this is the right sequence!

> **Porting**
> — 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.
Goals of Reverse Engineering

> Facilitate *reuse*
  — detect candidate reusable artifacts and components

> Generate *alternative views*
  — automatically generate different ways to view systems

> Synthesize *higher abstractions*
  — identify latent abstractions in software

> Cope with *complexity*
  — need techniques to understand large, complex systems

> Recover *lost information*
  — extract what changes have been made and why

> Detect *side effects*
  — help understand ramifications of changes

— 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
Reengineering Patterns

Reengineering 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.

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
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
Setting Direction

Agree on Maxims

Set direction

Maintain direction

Appoint a Navigator

Speak to the Round Table

Coordinate direction

Fix Problems, Not Symptoms

Most Valuable First

Where to start

What to do

If It Ain't Broke Don't Fix It

What not to do

How to do it

Keep it Simple

Principles & Guidelines for Software project management especially relevant for reengineering projects
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*
First Contact

- Talk with System experts
  - Chat with the Maintainers
  - Interview during Demo
  - Talk with end users

Software System
- Read it
- Read about it
- Skim the Documentation
- Compile it
- Do a Mock Installation

Feasibility assessment (one week time)

- Verify what you hear
Initial Understanding

Top down

Recover design

Speculate about Design

Bottom up

understand $\Rightarrow$

Obtain a higher-level model

Analyze the Persistent Data

Read it

Compile it

Study the Exceptional Entities
Pattern: Study the Exceptional Entities

**Problem**
— How can you quickly gain insight into complex software?

**Solution**
— *Measure* software entities and *study the anomalous ones*

**Steps**
— Use simple metrics
— Visualize metrics to get an overview
— Browse the code to get insight into the anomalies
System Complexity View

Nodes = Classes
Edges = Inheritance Relationships

Width = Number of Attributes
Height = Number of Methods
Color = Number of Lines of Code
Detailed Model Capture

Tie Code and Questions
- Keep track of your understanding

Exposure design

Refactor to Understand

Exposure collaborations

Step through the Execution
- Expose contracts

Look for the Contracts

Write Tests to Understand
- Expose the design & make sure it remains exposed

Learn from the Past

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

Write Tests to Enable Evolution

Managing tests

Grow Your Test Base Incrementally

Write Tests to Understand

• Test Fuzzy features
• Test Old Bugs
• Retest Persistent Problems

Regression Test after Every Change

Use a Testing Framework

Designing tests

Test the Interface, Not the Implementation

Record Business Rules as Tests

Migration Strategies
Migration

Migrate Systems Incrementally

- Involve the Users
- Why
- Prototype the Target Solution
- Where to
- Always Have a Running Version
- Regression Test after Every Change
- Tests, your Life-Insurance

- Build Confidence
- Why
- Use Profiler before Optimizing
- How
- Make a Bridge to the New Town
- Present the Right Interface
- Deprecate Obsolete Interfaces
- Distinguish Public from Published Interfaces

Conserve Familiarity

Why
Detecting Duplicated Code

- Detect
- Compare Code Mechanically
- Visualize Code as Dotplots
- Redistribute Responsibilities
- Transform Conditionals to Polymorphism
- Understand
Pattern: Visualize Code as Dotplots

Problem
— How can you effectively identify significant duplication in a complex software system?

Solution
— Visualize the code as a dotplot, where dots represent duplication.

Steps
— Normalize the source files
— Compare files line-by-line
— Visualize and interpret the dotplots
Clone detection by string-matching

**Solid diagonals** indicate significant duplication between or within source files.
Dotplot Visualization

Sample Dot Configurations:

- **Exact Copies**: 
  - a b c d e f a b c d e f

- **Copies with Variations**: 
  - a b c d e f a b x y e f

- **Inserts/Deletes**: 
  - a b c d e a b x y c d e

- **Repetitive Code Elements** (Helfman, 1995)
  - a x b c x d e x f g x h
Redistribute Responsibilities

Chains of data containers

Eliminate Navigation Code

Data containers

Move Behaviour Close to Data

Monster client of data containers

Split Up God Class
High-level refactorings make use of many low-level refactorings.
Transform Conditionals to Polymorphism

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

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

Transform Conditionals into Registration
- Test external attribute
Roadmap

- Laws of Software Evolution
- Mining Software Evolution
- Reengineering Software
- Tools for Evolution
Moose — an extensible platform for software and data analysis

FAMIX meta-models family

Import/export format for models

Data parsing support

…more
Explicit metamodels enable change

The Story of Moose: an Agile Reengineering Environment. Nierstrasz, Ducasse, Gîrba. ESEC/FSE 2005
Programming visualizations with CodeCrawler
Moose visualizations
Moose — a platform for collaborative research

Moose is a platform for software and data analysis.

It is an open source project since 1997. The community has grown and spread to several research groups worldwide.

The research around Moose has led to over 150 scientific publications.

Download
Softwarenaut - Interactive Architecture Recovery

Softwarenaut on Apache Lucene

http://scg.unibe.ch/softwarenaut
Information flows... across continents on travel.stackexchange.org

Reputation Distribution and Flow of Information (link)
Dennis Schenk (Masterarbeit in progress...)
MY NEW DESIGN WILL MEET ALL OF OUR CUSTOMERS' CURRENT AND FUTURE NEEDS.
What you should know!

> Lehman & Belady’s Law of Continuing Change
> Why do software systems become more complex over time
> Why is duplicated code considered to be a bad code smell?
> When should one keep an older version of a system rather than rewrite?
> What is meant by “reverse engineering”?
> Does studying exceptional entities help you understand a software system?
Can you answer the following questions?

- How would you ensure that documentation stays in sync with implementation?
- When should you start a reengineering project?
- When should you rewrite a project?
- How can you use the history of a system to improve development tools?
- What are the dangers of trying to fix the buggiest code first?
References

http://scg.unibe.ch/download/oorp/

http://www.joelonsoftware.com/
Attribution-ShareAlike 3.0

You are free:
▪ to copy, distribute, display, and perform the work
▪ to make derivative works
▪ to make commercial use of the work

Under the following conditions:

**BY:** Attribution. You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

▪ For any reuse or distribution, you must make clear to others the license terms of this work.
▪ Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

[link to license] http://creativecommons.org/licenses/by-sa/3.0/