Living and Working with Aging Software

Ralph Johnson

University of Illinois at Urbana-Champaign
rjohnson@illinois.edu

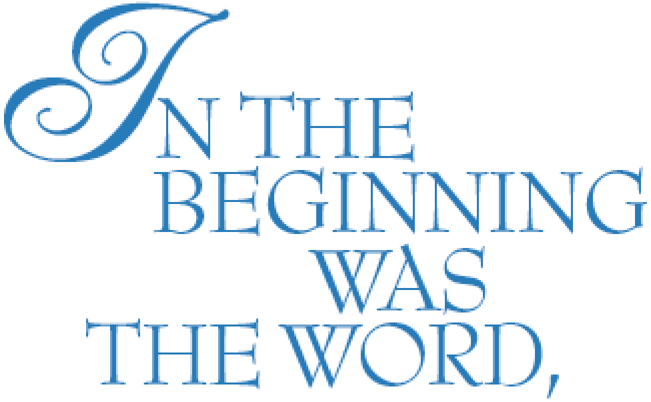
Old software gets brittle

- Hard to change
- Hard to understand

Software should be soft

History of Word

- 1983 - Word for DOS
- 1985 - Word for Mac
- 1989 - Word for Windows
- 1991 - Word 2
- 1993 - Word 6
- 1995 – Word 95
- 1997 – Word 97
- 1998 – Word 98
- 2000 – Word 2000
- 2002 – Word XP
- 2003 – Word 2003
- 2007 – Word 2007



Increase of Maintenance

- Def: Maintenance is all work on software after its first release
  - Shrink-wrap
  - Open source
  - Incremental development

The Stigma of Maintenance

- Software Evolution

- Software Revolution?

Software Capital

- As an industry matures, it becomes more capital intensive

- Is this true for software development?
- What is “capital” for software?

Software Capital

- Capital is software
- and knowing how it works.

If software is capital then …

- Expertise in the software is valuable
- Documentation is important
- Reverse-engineering is important

- Must maintain investment - keep it from depreciating

“Legacy” software

- Unfortunately, often old software
  - Has obsolete design
  - Uses technology that nobody understands
  - Uses technology that is not supported
  - Has no experts - they are all gone
  - Has no tests?

Managing 50 year old software

- Probably will last for a few more decades
  - Worthwhile to invest in the future
    - Documentation
    - Automated tests
    - Fix rare bugs
  - Worthwhile to train developers
  - Make changes slowly – mistakes are expensive
  - Programming is program transformation

Discovery and invention

- Discovery – ability to understand current system
- Invention – ability to create new system

- As system gets older, discovery becomes more important
- Current design is more important than requirements

Discovery and invention

- Discovery –
  - Reverse engineering
  - Documentation
  - Training
  - Hiring experts

Programming is program transformation

- Transform version N to version N+1
  - By adding new modules
  - By replacing modules
  - By transforming modules

Refactoring

- Behavior-preserving program transformations
- Changes to the structure of a program, but not its function
- Small, incremental design improvements
- Operations your editor should perform, but can’t

Typical refactorings

- Change name of procedure / class / variable
- Move variable / procedure from one class / module to another
- Change interface of procedure
- Extract / inline procedure

My history with refactoring

- 1985-1989 – frameworks
    - Reusable software requires iterative development
        - Software is not reusable until it has been tested
        - Test reusability by reusing it
        - Fixing reusability errors requires interface changes
    - Interface changes tend to fall into a few categories
- Bill Opdyke Ph.D. 1992
    - Developed first catalog of refactorings
    - Specified how they would work in C++

Smalltalk Refactoring Browser

- 1993 – first refactoring tool
- 1994 – start of Refactoring Browser by John Brant
- 1995 – first external users
- 1997 – port to IBM VA for Smalltalk and Envy
- 1998 – undo
- 1999 – Don Roberts PhD
- 2002 – part of Cincom’s VisualWorks 7.0

Related books

- *eXtreme Programming eXplained* by Kent Beck, 2000.
- *Refactoring: Improving the Design of Existing Code* by Martin Fowler, with Kent Beck, John Brant, Don Roberts, and William Opdyke, 1999.

Refactoring is

- The process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chance of introducing bugs. When you refactor, you are improving the design of the code after you have written it.

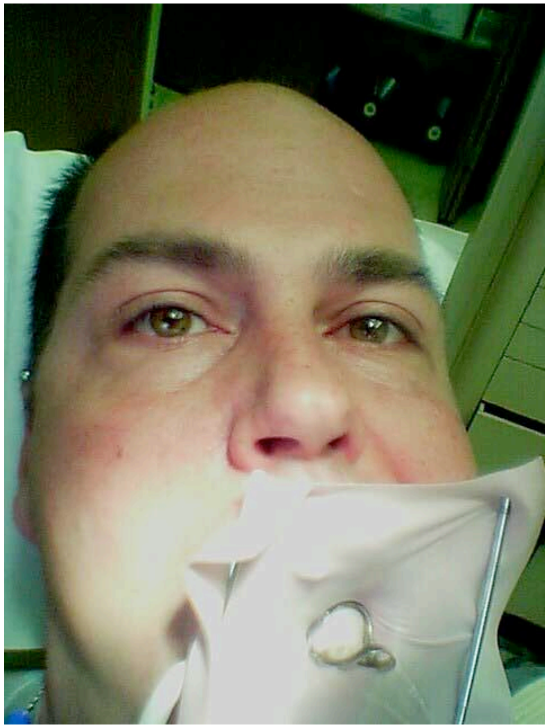
Refactoring without tools

- Start with an automated test suite
- Perform one refactoring at a time, and test after each refactoring.
  - Find mistakes quickly
  - Mistakes are easy to fix
- Be prepared to start over and redo refactoring

Lessons

- Refactoring is easier when you know how to do it
  - Tests
  - Small steps
  - Library of refactorings
- Tools can help

Flossing vs. root canal



Flossing

- Refactoring is 10% of your programming time Clean up your code after you make a change
- If a change is too hard, imagine what could have made it easier, and refactor to it
- Keep a set of goals in mind, and every time you change a file, see how you can make it better fit your goals

Root canal

- Refactoring is a project
- Make a plan, with many small steps
- Perform steps one at a time
- Keep the system running at all times
- "No battle plan survives contact with the enemy" Helmuth von Moltke
- "Plans are nothing. Planning is everything." Dwight D. Eisenhower

My recent refactoring research

- C preprocessor - Alejandra Garrido
- Library evolution - Danny Dig
- Fortran - Photran project - Jeff Overbey
- Refactoring to fix security bugs - Munawar Hafiz
- Refactoring to introduce parallelism - Stas Negara / Danny Dig

Library evolution

- Problem: libraries change with time. New version is not always compatible with old. Especially a problem with OO libraries, which are new and have complex interfaces.
- Solution:
  - Change your library by refactoring.
  - Give refactorings to users.
  - Users run the refactorings and update their applications.

Problems

- Must be able to distribute refactorings
- Refactorings might break user code
  - Need to change user code and proceed
- Framework change might not be a refactoring
  - How often?
  - Can these be carried out by hand?

- Four Java libraries
  - Eclipse 3.0
  - Struts 1.2.4
  - Log4j 1.3
  - A proprietary mortgage system
- Mature - in use more than three years
- Major releases
- Change log explaining the changes from previous version

	Eclipse 3.0	Struts 1.2.4	log4j 1.3	Mortgage
size in KLOC	1,923	97	62	52
API classes	2,579	435	349	174
Breaking changes	51	136	38	11
Change log	24	16	4	-

	Eclipse 3.0	Struts 1.2.4	log4j 1.3	Mortgage
Breaking changes	51	136	38	11
% refactorings	84	91	97	81

Danny Dig and Ralph Johnson: How do APIs evolve? A story of refactoring,

Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien Nguyen: Effective Software Merging in the Presence of Object-Oriented Refactorings,

Danny Dig, Stas Negara, Vibhu Mohindra, Ralph Johnson: ReBA: Refactoring-aware Binary Adaptation of Evolving Libraries,

https://netfiles.uiuc.edu/dig/www/research.html

Changing programming language

- Convert million lines of Delphi to C#
- Never stop adding features

- 18 months by John Brant, Don Roberts, a couple of local programmers and the local QA team

Changing architecture

- Highly integrated => highly modular
- Modular => service oriented

Software Development is Program Transformation

- Anything can be added later
  - Modularity
  - Security
  - Documentation

- Tools make transformation easer, but more important than tools are:
  - Design expertise - being able to tell good design from bad
  - Taking small steps - keep your system running
  - Have a plan
    - Flossing - direction system is evolving
    - Root canal - small steps to achieve big aim
  - Automated tests

- If software is going to last, we have to take care of it.
- Requires architectural oversight
- Make sure future change is possible
- Keep design debt small
- Refactoring is key for managing evolution
- Program transformation tools are valuable