Maintenance

Code Smells and Refactoring

CSC 326

© NC State Software Engineering Faculty

1

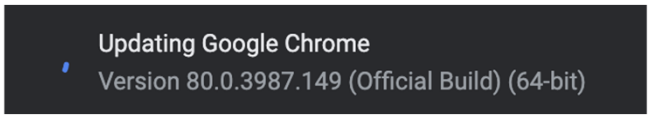
Software Change

- **Software change is inevitable**
  - New requirements emerge as software is used
  - The business/usage environment changes (e.g., transition to mobile devices)
  - Bugs must be fixed
  - New equipment must be accommodated (e.g., a new sensor for GPS)
  - The performance or reliability may need to be improved
- Key problem for organizations: implementing and managing change to their legacy systems.

© NC State Software Engineering Faculty

2

Software Versioning Numbers



Format: a.b.c(.d)

Semantic Versioning (*aka "SemVer"*)*

major.minor.patch(.build)

Breaking changes.

high risk

New features, non-breaking.

medium risk

Patches, non-breaking.

low risk

**organization dependent*

3

Software Maintenance

- *Software maintenance* is *the process of modifying a software system or component after delivery to*
  - *correct faults*
  - *improve performance or other attributes, or*
  - *adapt to a changed environment*

- Software maintenance activities are expensive
  - can consume as much as 90% of the total effort expended on a system in its lifetime

© NC State Software Engineering Faculty

4

Types of Maintenance

- Activity-based Classifications [Kitchenham, et al. 99]
  - Corrections
  - Enhancements
- Evidence-based [Chapin et al.]
- Intention-based Classifications [ISO/IEC 14764]
  - Corrective
  - Adaptive
  - Perfective
  - Preventative

© NC State Software Engineering Faculty

5

Activity-based Classifications

- **Corrections** – fix defects

- **Enhancements** – effect changes that may or may not change behavior
  - Modify existing system requirements
  - Add new system requirements
  - Modify the implementation without changing the requirements

© NC State Software Engineering Faculty

6

Evidence-based Classifications

- Maintenance on four kinds of entities
- Software Properties:
  - **Adaptive**: environment + increase COTS
  - **Performance**: impact user by improving system up time and speed
  - **Preventive**: enhance maintainability and help with transitioning technologies
  - **Groomative**: updating components/algorithms, modifying conventions, compiling source code, backups

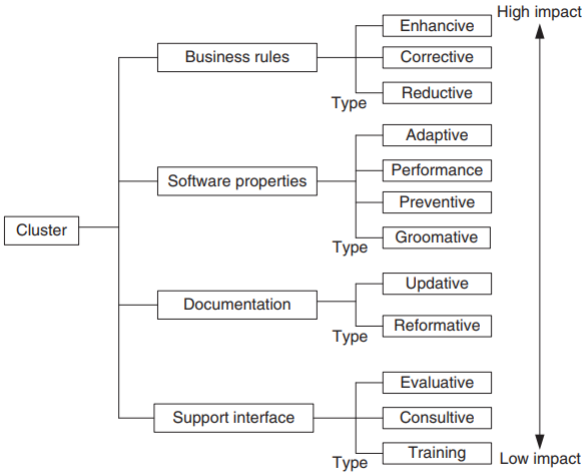


FIGURE 2.1 Groups or clusters and their types

© NC State Software Engineering Faculty

Priyadarshi Tripathy, Kshirasagar Naik, Evolution and Maintenance Models, Wiley, 2014. – Chapter 2

7

Intention-based Classifications of Maintenance

Types of maintenance

- **Corrective:** Bug fixes based on customer reports (20%)
- **Perfective:** Refinement of existing requirements (50%)
- **Adaptive:** Adjusting to external changes in environments – changes in policies, technologies, etc. (25%)
- **Preventative:** Refactoring, code restructuring, upgrading technologies (5%)

These are not mutually exclusive!

© NC State Software Engineering Faculty

8

8

Intention-based Classifications - 1

- Corrective: maintenance performed to correct faults
  - Reproduce the failure.
    - The failure cannot be reproduced. Possibly nothing at all might be wrong
  - Documentation is often obsolete or nonexistent
  - Fix without breaking anything else. Faults injected when fixing other problems are called regression faults.
    - Test to make sure the fix works and no regression faults have been introduced.

Testing →Debugging → Fixing → Testing

© NC State Software Engineering Faculty

9



10

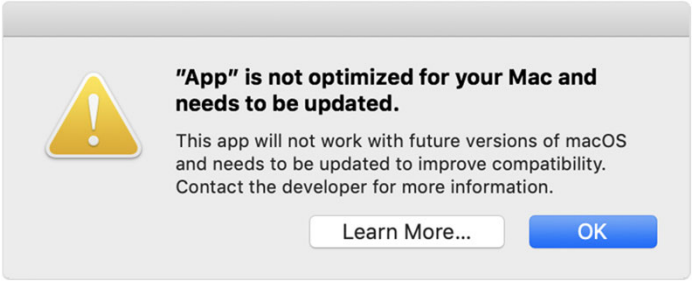
Intention-based Classifications - 2

- Adaptive: software maintenance performed to make a computer program usable in a changed environment
  - Environment: totality of all conditions and influences that act from outside the system
    - Examples: business rules, government policy, work patterns, software platforms, compilers, hardware upgrades
  - Reaction to changes in the environment to preserve existing functionality and performance.

© NC State Software Engineering Faculty

11

Apple's OSX migration to 64-bit architectures



OSX = environment
(OSX change) => (adaptive maintenance)

12

Questions about

"Bit Rot"

aka "software entropy"

Slow deterioration of a system over time. *(Software doesn't actually change.)*

Causes: Environment. Uneven maintenance.

Dormant Rot: part of the code that is unused, which stops working with everything else that changes.

Active Rot: as software evolves, it becomes harder to maintain.

13

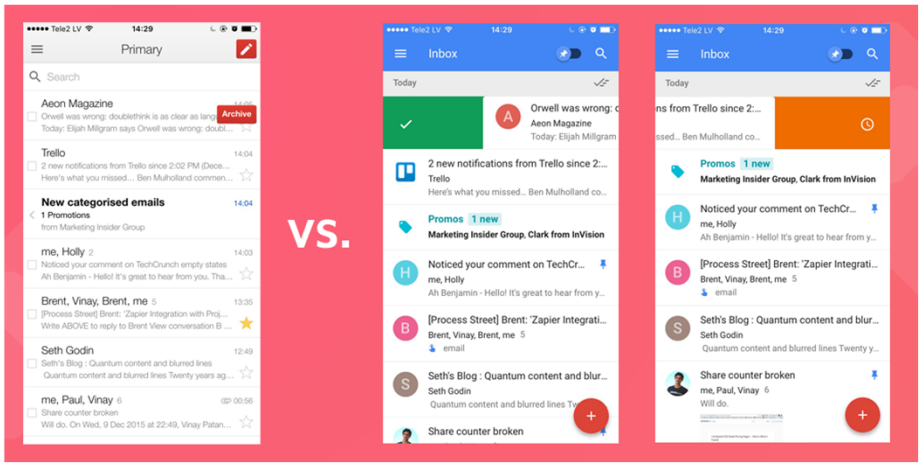
Intention-based Classifications - 3

- Perfective: software maintenance performed to improve the performance or other attributes of a computer program
  - Improvements to user experience, processing efficiency, and maintainability
  - Restructuring code
  - Create/update documentation
  - Improve performance
  - New Features

© NC State Software Engineering Faculty

14

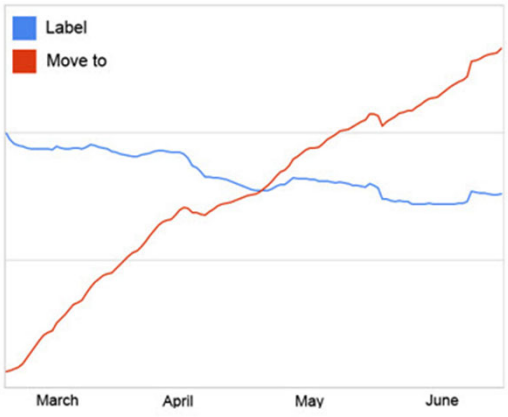
A/B Testing



15

Continuous Improvement: Gmail Labels





http://googleblog.blogspot.com/2009/07/evolution-of-gmail-labels.ht

Credit: Chris Scaffidi, OSU

© NC State

16

Intention-based Classifications - 4

- Preventative: maintenance performed for the purpose of preventing problems *before* they occur
  - Take action to prevent future risks
  - Mission and safety critical systems
  - No increase in functionality yet costs significant amounts of money
  - Refactoring
    - Process of changing a software system in such a way that it does not alter the external behavior of the code yet it improves its internal structure

© NC State Software Engineering Faculty

17

Maintenance Summary

- **Corrective**: Repair a software fault
- **Adaptive**: Change a system to a new environment (e.g., new platform, desktop to web app)
- **Perfective**: Add to or modify existing functionality, improve documentation
- **Preventative**: Increasing software maintainability and prevention of future problems (e.g., refactoring)

© NC State Software Engineering Faculty

18

Comparison of Maintenance Types

Type	Request often from customer?	Effect of change possibly visible to customers?	Need testing? (if yes, for what?)
Corrective	X	X	X (reproduce failure, confirm fix, no regressions)
Adaptive		X (may not be visible in all cases)	X (confirm adaptation, no regressions)
Perfective	X	X	X (test new features, no regressions)
Preventative			X (no regressions)

© NC State Software Engineering Faculty

19

Maintenance Costs – Organization's Perspective

- Usually greater than development costs
- Affected by both technical and non-technical factors
- Increases as software is maintained. Maintenance corrupts the software structure, so future changes are difficult.
- Aging software can have higher costs

© NC State Software Engineering Faculty

20

Maintenance Cost Factors – Organization's Perspective

Team stability

- Lots of turnover ⇒ more maintenance effort

Contractual responsibility

- Developers may have no incentive to design for future changes

Program age and structure

- As programs age, their structure is degraded and they become harder to understand and update (spaghetti)

© NC State Software Engineering Faculty

Maintenance Costs – User's Perspective

- Software Updates (patches)
  - Software designed to modify an existing program
    - Fix a problem
    - Add/modify features
  - A way to deliver the 'product' of the maintenance process to the user

© NC State Software Engineering Faculty

A brief history of software updates

- In the beginning, software patches were delivered on paper tape or punch cards to be placed into the software manually
- Later, patches came on disks
- With the Internet, patches can be downloaded

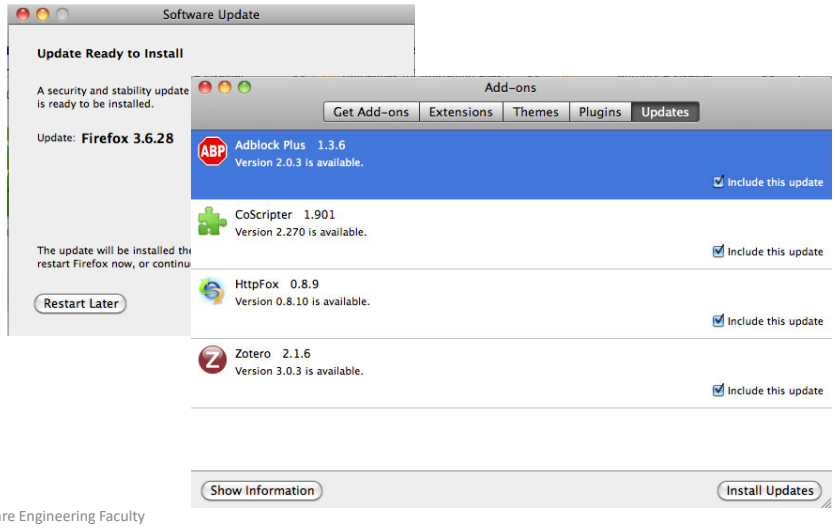
© NC State Software Engineering Faculty

Maintenance: the good and the bad

- **The Bad**
  - Require user input
  - Causes delays
- The Good
  - Continuous improvement
  - Security patches

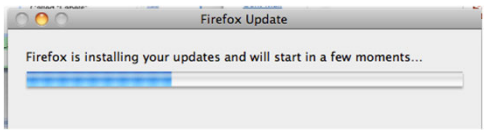
© NC State Software Engineering Faculty

Requires User Input



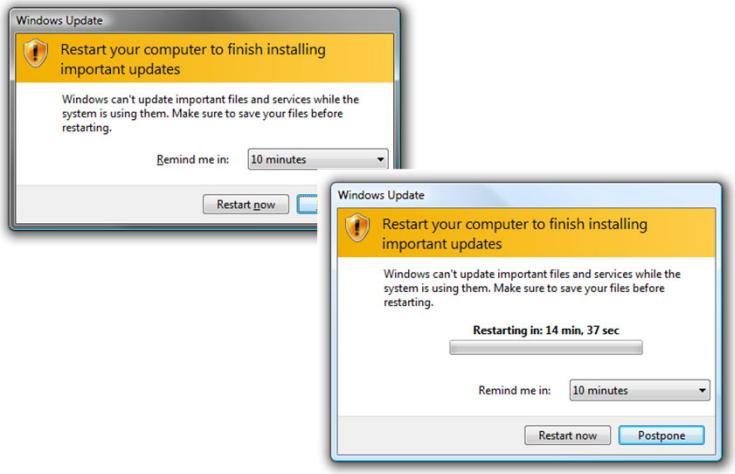
© NC State Software Engineering Faculty

Causes Delays



© NC State Software Engineering Faculty

Restarting Without Your Permission



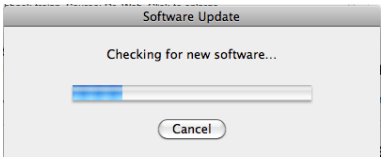
© NC State Software Engineering Faculty

Maintenance: the good and the bad

- The Bad
  - Require user input
  - Causes delays
- **The Good**
  - Continuous improvement
  - Security patches

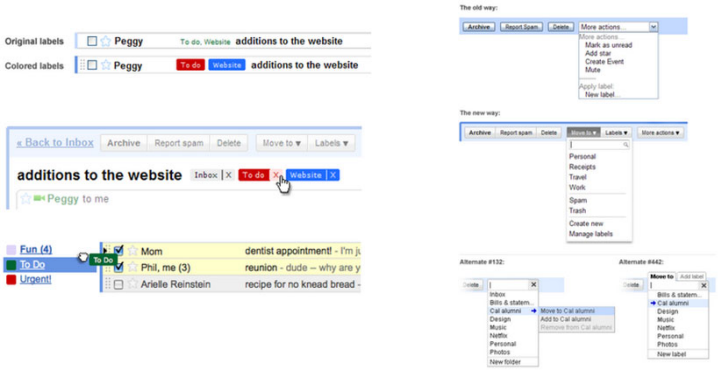
© NC State Software Engineering Faculty

User-Initiated Maintenance Checks



© NC State Software Engineering Faculty

Continuous Improvement: Gmail Labels



http://googleblog.blogspot.com/2009/07/evolution-of-gmail-labels.html

© NC State Software Engineering Faculty

Credit: Chris Scaffidi, OSU

Security Patches

Apple closes a trojan loophole after 550,000 Macs are infected

FORTUNE

157 comments

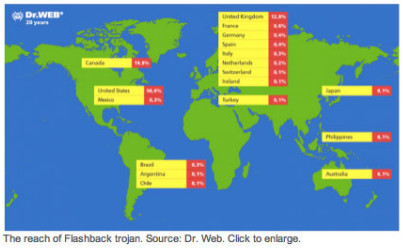
By Philip Elmer-DeWitt April 5, 2012: 8:08 AM ET

Recommend 1k Tweet 327 Share 64 +1 11 Email Print

Compare to Windows, OS X is nearly malware free. But it does run JavaScript.

Having written several times -- and taken a lot of heat from PC users -- about the relative security of Apple's (**AAPL**) operating systems (See **Why are there no Mac viruses**), I feel obliged to report that Mac OS X is under what appears to be the most serious malware attack to date.

According to a **report**



The reach of Flashback trojan. Source: Dr. Web. Click to enlarge.

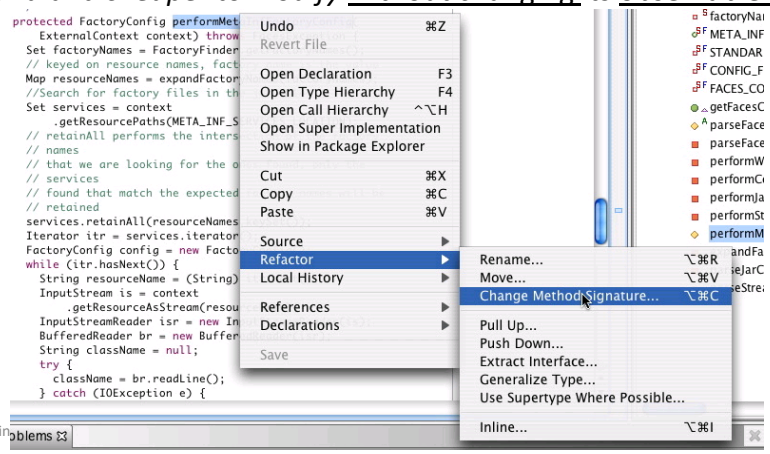
© NC State Software Engineering Faculty

Software is Maintained
It Evolves in Different Ways
Change is a Constant

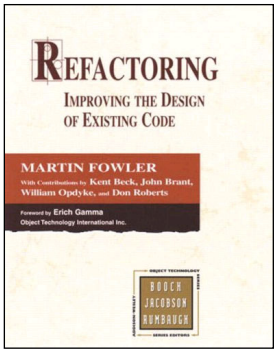
a haiku

Preventative Maintenance: Refactoring

- Definition: a change made to the internal structure of software to make it *easier to understand* and *cheaper to modify* without changing its observable behavior



Bad Smells



- Classes that are too long
- Methods that are too long
- Switch statements, instead of inheritance
- Duplicate code
- Almost (but not quite) duplicate code
- Over dependence of primitive types, instead of more domain-specific types
- Useless (or wrong!) comments
- . . . .

Does what my code looks like really matter?

The compiler doesn't care of code is clean or ugly, but a human might care when an update needs to be made

A poorly designed/written system is hard to change

- Hard to figure out where the changes are needed
- Hard to figure out what to change
- Easy to insert bugs if code is not well-understood

© NC State Software Engineering Faculty

Extract Method

- Smell: low cohesion
  - e.g., You have a code fragment that can be grouped together
  - Turn the fragment into a method whose name explains the purpose of the method

```
void print0wing(double amount) {
   printBanner();
   //print details
   System.out.println ("name" + _name);
   Sysetm.out.println ("amount" + amount);
}
```



```
void print0wing(double amount) {
   printBanner();
   printDetails(amount);
}
void printDetails(double amount); {
   System.out.println ("name" + _name);
   Sysetm.out.println ("amount" + amount);
}
```

© NC State Software Engineering Faculty

Extract Method - 2

- Increases the chances of reuse
- Allows higher-level methods to read more like a series of comments

Steps:

- Create a new method and name it after the intention of the method (what it does)
- Copy extracted source code into new target method. Replace extracted code with method call
- Scan extracted code for references to variable local to source method ➔ make parameters

© NC State Software Engineering Faculty

37

Introduce Explaining Variable -1

- Smell: You have a complicated expression.
- Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose

```
if ((platform.toUpperCase().indexOf("MAC") > -1) &&
    (browser.toUpperCase().indexOf("IE") > -1)   &&
    (wasInitialized()) && (resize > 0))   {

        // do something
}
```



```
final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResized = resize > 0;

if (isMacOS && isIEBrowser && wasInitialized() && wasResized) {
    // do something
}
```

© NC State Software Engineering Faculty

38

Introduce Explaining Variable - 2

- Temporary explaining variables can help break a complex expression down into something more manageable (and less error prone)
  - Self documenting code
- Especially good with conditional logic

Steps:

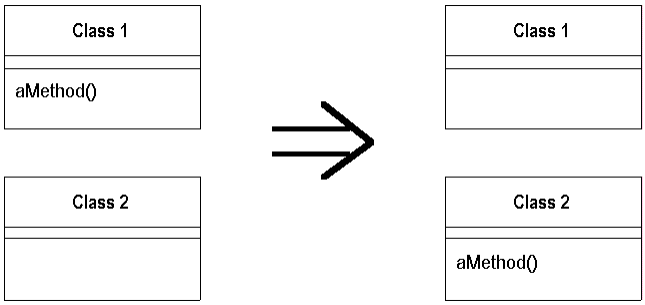
- Declare a temporary variable and set to the result of part of a complex expression
- Replace the part of the expression with the value of the temp

© NC State Software Engineering Faculty

39

Move Method - 1

- Smell: High Coupling
  - E.g., A method is, or will be, using or used by more features of another class than the class on which it is defined
  - Create a new method with a similar body in the class that uses it the most. Either turn the old method into a simple delegation, or remove it altogether.



© NC State Software Engineering Faculty

40

Move Method - 2

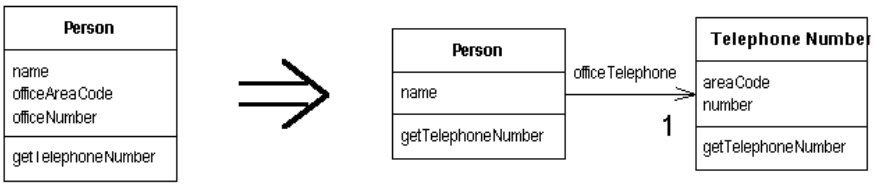
- Examine all methods used by the source method that are defined on the source class. Consider whether they should also be moved
- Check the sub- and superclass of the source class for other declarations of the method.
  - This may prevent the move.
- Declare the method in the target class.
  - Perhaps renaming, if appropriate
- Copy the code from the source method to the target. Adjust the method to make it work in its new home.

© NC State Software Engineering Faculty

41

Extract Class - 1

- Smell: Low Cohesion
  - E.g., You have one class doing work that should be done by two.
  - Create a new class and move the relevant fields and methods from the old class to the new class

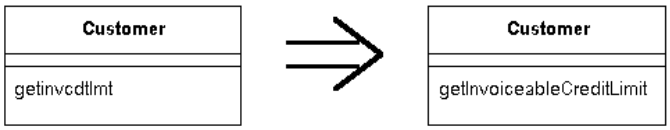


© NC State Software Engineering Faculty

42

Rename Method

- Smell: The name of a method does not reveal its purpose
  - *Change the name of the method*

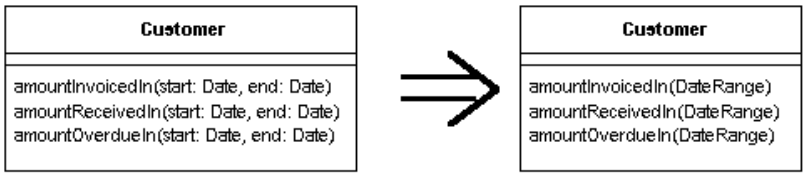


© NC State Software Engineering Faculty

43

Introduce Parameter Object - 1

- Smells: Long parameter list + Duplicate code
  - E.g., You have a group of parameters that naturally go together.
  - Replace them with an object.



© NC State Software Engineering Faculty

44

Introduce Parameter Object – 2

Steps:

- Create a new class to represent the group of parameters you are replacing.
- Modify the callers.
- Once you have made this transition, look for behavior (possible methods) that can be moved into this new class.

© NC State Software Engineering Faculty

Smell: Duplicate Code

- If you see the same code structure in more than one place, you can be sure the program will be better if they are unified
  - Often the result of copy-and-paste coding
- Possible Solution: Extract Method

© NC State Software Engineering Faculty

How to do a refactoring right

- You must have a working unit test suite
  - No refactoring is allowed until all tests pass
  - Remember - you are not changing the external behavior!
- When you smell something fishy, try out the refactoring
- Run the unit tests
- Iterate until the code smells better and all tests pass

© NC State Software Engineering Faculty

Refactor early and often

- If it takes you too long to refactor, you are not refactoring enough
  - Many tiny refactorings are easier than a single large one
  - Easier to identify errors if changes are small
  - Each refactoring makes it easier to identify opportunities for future refactorings

© NC State Software Engineering Faculty

Be alert for opportunities

- Refactor when...
  - A new feature seems too difficult to code
  - You just created a new feature and the code will be too hard for someone else to understand
  - You can't stand to look at your own code
  - You can't stand to look at your teammate's code!

© NC State Software Engineering Faculty

Problems with Refactoring

- Refactoring is coding effort that does not add features to the product
- It's hard to guarantee that the behavior of the code is the same before and after
- ... others?

© NC State Software Engineering Faculty

Refactoring Summary

- Important for maintaining proper structure of you code.
  - Even if you start with a wonderful design, it will likely degrade through time.
    - Complexity
    - Readability
- Need to do it in small baby steps.
  - Definitely one refactoring at a time.
  - Make a change, see what the compiler and your unit tests tell you about the change you just made. (Goal – no regressions!)
  - Ensure everything works fine before proceeding to the next refactoring.

© NC State Software Engineering Faculty

51

Refactoring Resources

- http://wiki.java.net/bin/view/People/SmellsToRefactorings
- http://www.refactoring.com/

© NC State Software Engineering Faculty

52

References

- [ISO/IEC 14764]: https://www.iso.org/standard/39064.html
- Priyadarshi Tripathy, Kshirasagar Naik, Evolution and Maintenance Models, Wiley, 2014.
- B. A. Kitchenham, G. H. Travassos, A. N. Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. 1999. Towards an ontologyy of software maintenance. Journal of Software Maintenance and Evolution: Research and Practice, 11, 365–389.
- N. Chapin, J. F. Hale, K. M. Khan, J. F. Ramil, and W. G. Tan. 2001. Types of software evolution and software maintenance. Journal of Software Maintenance and Evolution: Research and Practice, 13, 3–30.

© NC State Software Engineering Faculty