Software Maintenance

- ***Software maintenance*** **is defined as *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 can consume as much as 90% of the total effort expended on a system in its lifetime.**
- **Failures continue to be discovered in software for years.**
- **Often first job upon graduating is maintenance.**

Why so much maintenance?

- **Software is a model of reality. Reality changes.**
- **If software is found to be useful, satisfied users want to extend the functionality of the system.**
- **Software is much cheaper to change than hardware. As a result, changes are made in software wherever possible.**
- **Successful software survives well beyond the lifetime of the hardware for which it was written. Software need to be modified to run on new hardware and operating system.**

Types of Maintenance - 1

- **Corrective:** maintenance performed to correct faults in hardware or software → need to have strong debugging and interpersonal skills
  - Reproduce the failure.
  - The failure cannot be reproduced. Possibly nothing at all might be wrong
  - Might be due to changes in hardware, OS, another application. These are very difficult to reproduce and are fixed via adaptive maintenance.
  - Might be in the code
    - Documentation is often obsolete or nonexistent
    - Fix without breaking anything else. Faults injected when fixing other problems are called regression faults. Update the documentation.
    - Test to make sure the fix works and no regression faults have been introduced.

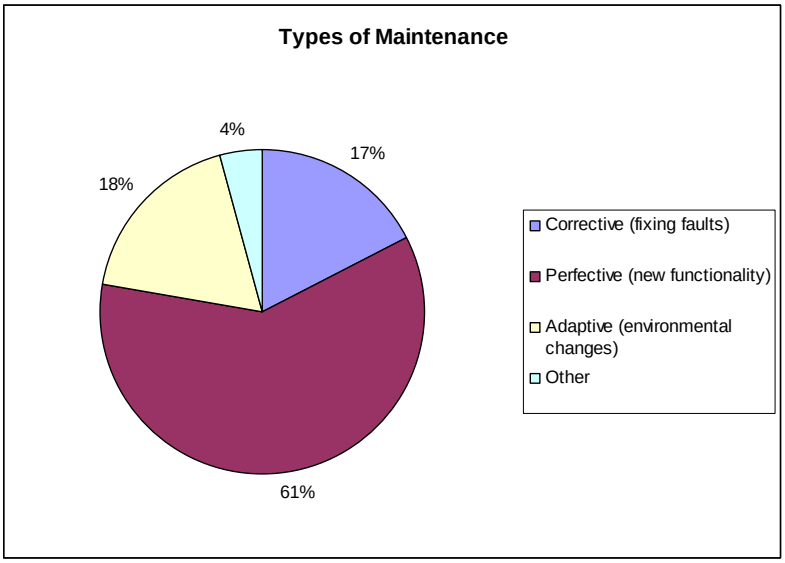
Types of maintenance - 2

- **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 (business rules, government policy, work patterns, software platforms, compilers, hardware upgrades (e.g. European countries switching to the Euro)
  - Reaction to changes in the environment to preserve existing functionality and performance.
- **Perfective:** software maintenance performed to improve the performance, maintainability, or other attributes of a computer program.
  - Extend the software beyond its original functional or non-functional requirements.
  - Happy users want more.

Types of Maintenance - 3

- **Preventative:** maintenance performed for the purpose of preventing problems before they occur
  - Prevent aging; make more easily corrected, adapted, and enhanced
  - No increase in functionality yet costs significant amounts of money
  - Also called software reengineering
    - Data restructuring
    - Code restructuring
  - Mini restructurings → 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

Types of Maintenance



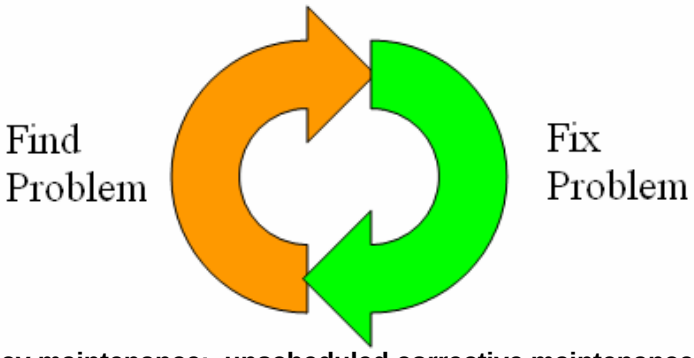
Lehman's Laws of Software Evolution

No.	Law	Description
I	Continuing change	Any system must be continually adapted or else it becomes progressively less satisfactory.
II	Increasing complexity	As an system evolves, its complexity increases unless work is done to maintain or reduce it.
III	Self regulation	Program evolution is a self-regulating control process, often accomplished via feedback mechanisms.
IV	Conservation of organizational stability	Over a system's lifetime, its activity rate (elements handled per release) is approximately constant.
V	Conservation of familiarity	Over the lifetime of a system, the average incremental growth tends to decline due to factors such as decreasing interest in the product.
VI	Continuing growth	A system must be continually grown to satisfactorily support new situations and circumstances.
VII	Declining quality	Unless rigorously adapted to take into account changes in the operational environment, the quality of a system will appear to decline as it is evolved.
VIII	Feedback System	Evolution processes are multi-level, multi-loop, multi-agent feedback systems.

Maintenance Process

- Change request form; bug report; change control board
  - Evaluate for resources and impact on system

Quick Fix



Emergency maintenance: unscheduled corrective maintenance performed to keep a system operational.

Requirements, design, and code become inconsistent; software aging is accelerated; unforeseen ripple effects impact system quality [small changes 40x more likely to cause problems]

Iterative Enhancement

- **Implementation of changes to a system through its lifetime is an iterative process**
- **Assumes documentation is available**

- **Analysis**
- **Characterization of proposed modifications**
- **Redesign and implementation**

Percentage of use

- Quick fix, no doc changes – 57%
- Quick fix, doc changes – 27%
- Iterative enhancement + reuse based – 16%

Testing

- Recreate the problem
- Write a test case that will fail because of the problem
- Fix
- Make sure test case now passes
- Regression testing of fixes very important

Why so expensive

- **Team stability**
- **Contractual responsibility – developers not responsible for maintenance so don't care**
- **Staff skills – newbie, unskilled**
- **Program age and structure**

Maintainability/maintenance metrics

- **Maintainability: ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to changed environment**
- **Mean time to repair (MTTR) – expected or observed time required to repair a system or component and return it to normal operation**
- **Number of requests for corrective maintenance – maybe injecting more than removing**
- **Average time for impact analysis – growing more and more components are being affected**
- **Complexity – metrics (coupling, cohesion, etc.)**

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

- **"If it stinks, change it."**

  -- Grandma Beck, discussing child-rearing philosophy

- **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)**
- **Too much string addition**
- **Useless (or wrong!) comments**
- **. . . .**

Extract Method

- 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);
}
```


Extract Method - 2

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

- Create a new method and name it after the intention of the method (what it <u>does</u>)
- 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
- Local scope modified by extracted code?
  - One? Return as result
  - More? Can't extract as it stands

Introduce Explaining Variable -1

- 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().index0f("MAC") > -1) &&
    (browser.toUpperCase().index0f("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
}
```


Introduce Explaining Variable - 2

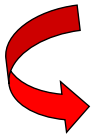
- Temporary explaining variables can help break a complex expression down into something more manageable (and less error prone)
- Especially good with conditional logic

- Declare a final 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

Inline Temp

- You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings
- Replace all references to that temp with the expression

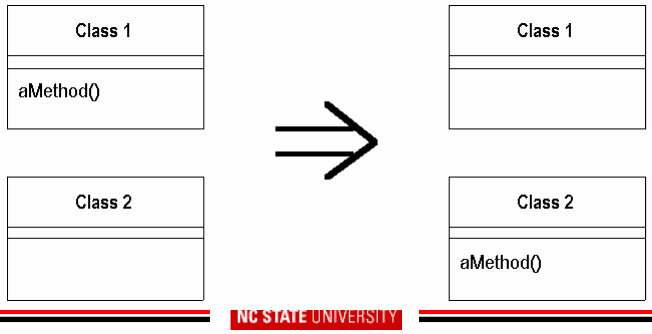
```
double basePrice = anOrder.basePrice();
return (basePrice > 1000);
```



```
return (anOrder.basePrice() > 1000)
```


Move Method - 1

- 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 it uses most. Either turn the old method into a simple delegation, or remove it altogether.

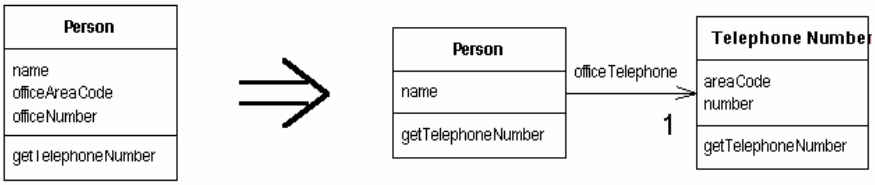


Move Method - 2

- Examine all methods used by the source method that are defined on the source class. Consider whether they should also be moved
  - If a method is used only by the method you are about to move, move it too. If the method is used by other methods, considering moving them as well. (Sometimes it is easier to move in bunches.)
- 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.

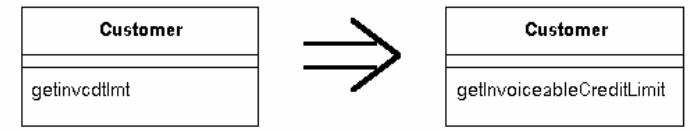
Extract Class - 1

- 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



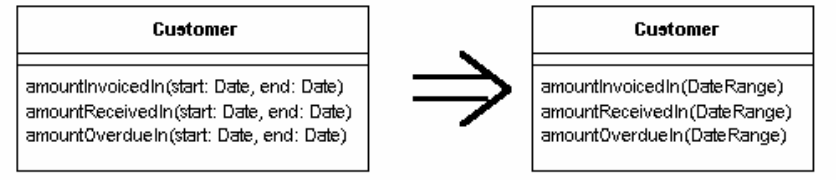
Rename Method

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



Introduce Parameter Object - 1

- You have a group of parameters that naturally go together.
- Replace them with an object.



Introduce Parameter Object – 2

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

Resources

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

Refactoring Summary

- **Important for maintaining proper structure of you code.**
  - Even if you start with a wonderful design, it will 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 test tell you about the change you just made.
  - Ensure everything works fine before proceeding to the next.