Refactoring Principles

Joseph W. Yoder

The Refactory, Inc.
University of Illinois

yoder@refactory.com
http://www.refactory.com

Presenter

- Joseph Yoder
  - e-mail: yoder@refactory.com
  - www: http://www.joeyoder.com

  - www.refactory.com

UIUC Patterns Group Software Architecture Group Ralph Johnson's Group

- Objects
- Reuse
- Frameworks
- Adaptive Architecture
- Components
- Refactoring
- Evolution
- Patterns

The Refactory, Inc.

- The Refactory, Inc. was founded in 1998 as a consortium of object-oriented experts dedicated to helping organizations succeed with objects.
- Founders and Affiliates have a total of over 120 years of combined software development experience with over 80 years dedicated to Object-Oriented development.

The Refactory Principals

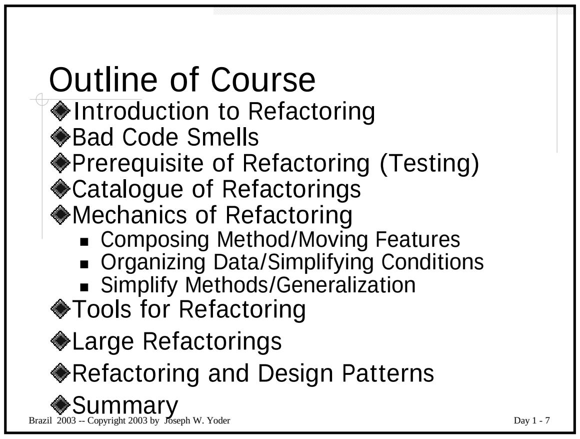
John Brant
Brian Foote
Ralph Johnson
Don Roberts
Joe Yoder

Refactory Affiliates

Dragos Manolescu
Brian Marick
Bill Opdyke

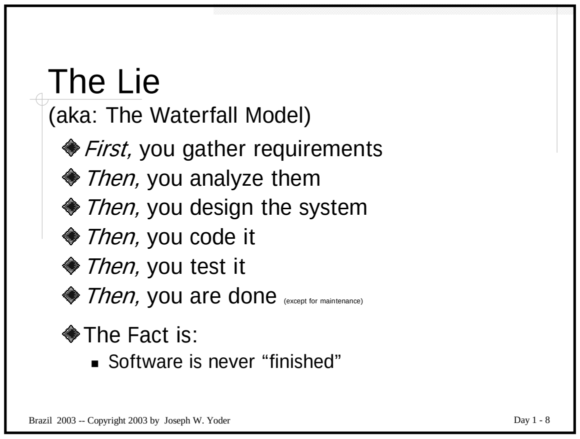
The Refactory, Inc.

The Refactory principles are experienced in software development, especially in object-oriented technology. We've been studying and developing software since 1973. Our current focus has been object-oriented technology, software architecture, and patterns. We have developed frameworks using Smalltalk, C++, and Java, have helped design several applications, and mentored many new Smalltalk, Java, and C++, C# developers. Highly experienced with Frameworks, Software Evolution, Refactoring, Objects, Flexible and Adaptable Systems (Adaptive Object-Models), Testing, Workflow Systems, and Agile Software Development including methods like eXtreme Programming (XP).



Outline of Course

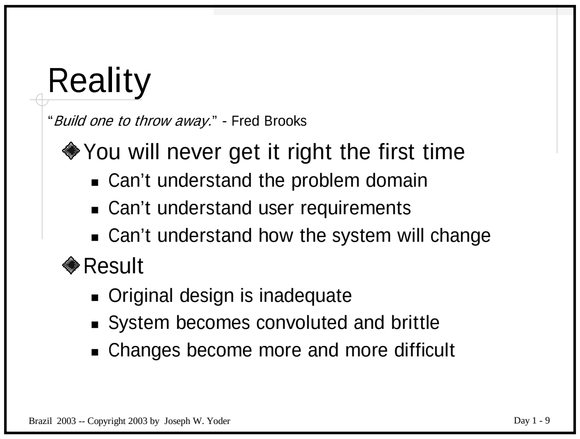
- Introduction to Refactoring
- Bad Code Smells
- Prerequisite of Refactoring (Testing)
- Catalogue of Refactorings
- Mechanics of Refactoring
  - Composing Method/Moving Features
  - Organizing Data/Simplifying Conditions
  - Simplify Methods/Generalization
- Tools for Refactoring
- Large Refactorings
- Refactoring and Design Patterns
- Summary



The Lie

(aka: The Waterfall Model)

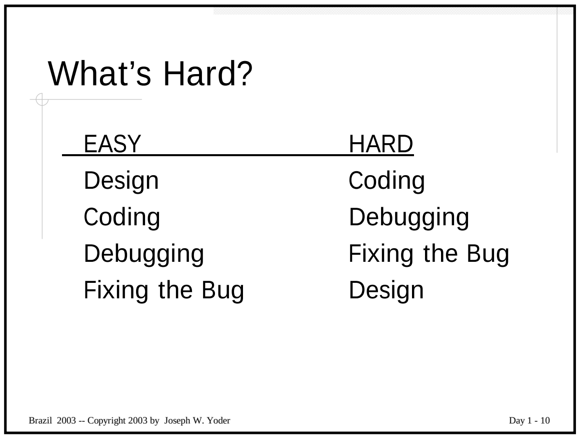
- *First*, you gather requirements
- *Then*, you analyze them
- *Then*, you design the system
- *Then*, you code it
- *Then*, you test it
- *Then*, you are done (except for maintenance)
- The Fact is:
  - Software is never "finished"



Reality

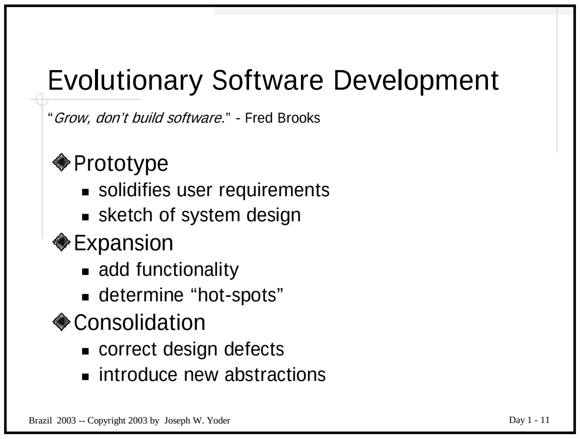
"*Build one to throw away.*" - Fred Brooks

- You will never get it right the first time
  - Can't understand the problem domain
  - Can't understand user requirements
  - Can't understand how the system will change
- Result
  - Original design is inadequate
  - System becomes convoluted and brittle
  - Changes become more and more difficult



What’s Hard?

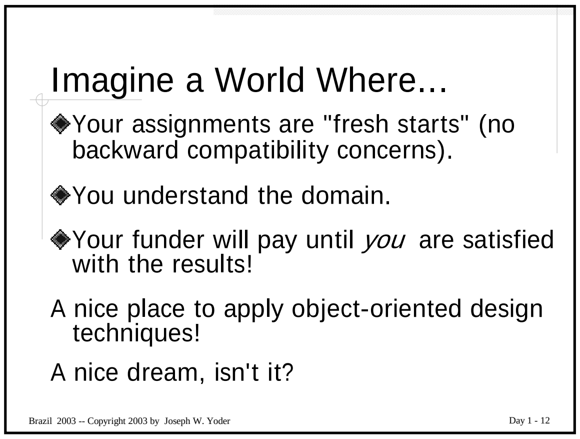
EASY	HARD
Design	Coding
Coding	Debugging
Debugging	Fixing the Bug
Fixing the Bug	Design



Evolutionary Software Development

“*Grow, don’t build software.*” - Fred Brooks

- Prototype
  - solidifies user requirements
  - sketch of system design
- Expansion
  - add functionality
  - determine “hot-spots”
- Consolidation
  - correct design defects
  - introduce new abstractions

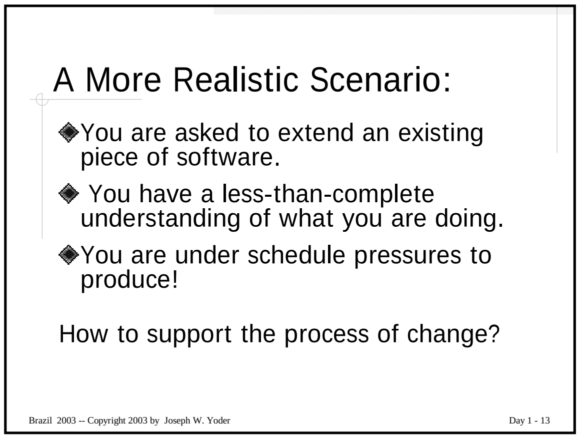


Imagine a World Where...

- Your assignments are "fresh starts" (no backward compatibility concerns).
- You understand the domain.
- Your funder will pay until *you* are satisfied with the results!

A nice place to apply object-oriented design techniques!

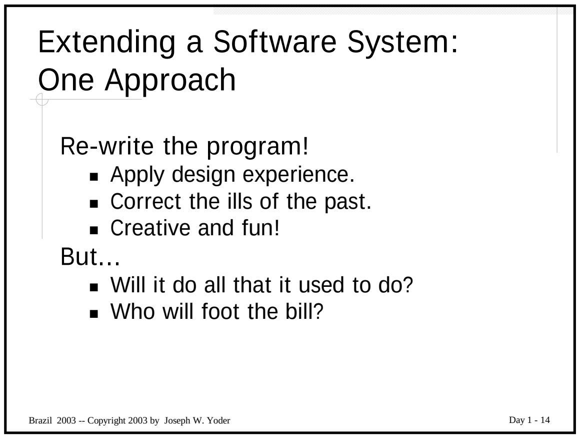
A nice dream, isn't it?



A More Realistic Scenario:

- You are asked to extend an existing piece of software.
- You have a less-than-complete understanding of what you are doing.
- You are under schedule pressures to produce!

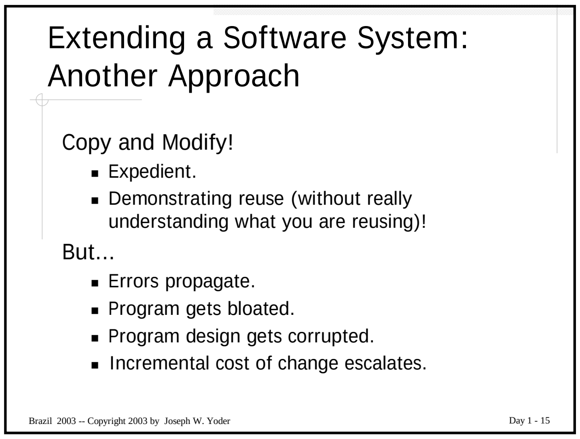
How to support the process of change?



Extending a Software System: One Approach

Re-write the program!
- Apply design experience.
- Correct the ills of the past.
- Creative and fun!

But…
- Will it do all that it used to do?
- Who will foot the bill?



Extending a Software System: Another Approach

Copy and Modify!
- Expedient.
- Demonstrating reuse (without really understanding what you are reusing)!

But…
- Errors propagate.
- Program gets bloated.
- Program design gets corrupted.
- Incremental cost of change escalates.

Extending a Software System: A Middle Ground

Restructure (*refactor*) the existing software:

- Start with the existing software base.
- Apply design insights; extract reusable abstractions and components.
- Clarify the software architecture.
- Prepare program to make the additions easier.

Then, add your new features!

Some Advantages:

- leverage past investment
- reduce duplication
- streamline program.

Definitions

- Refectory - n. - A dining hall, especially in a monastery.
- Refractory - n. - 1. Referring to a period of unresponsiveness to a nervous or sexual stimulus after such a stimulus. 2. The period of time during which our Chief of State conducts business.

Definition of Refactoring

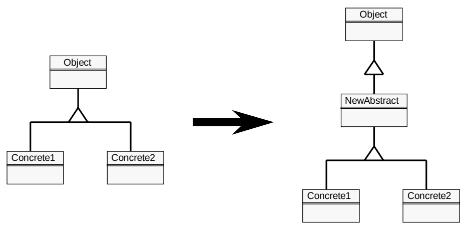
Interface design and functional factoring constitute the key intellectual content of software and are far more difficult to create or re-create than code. - Peter Deutsch

- *vt.* - The process of redesigning the abstractions in a program
- *n.* - A behavior-preserving source-to-source program transformation

Substantial changes to software can be characterized as refactorings plus additions.

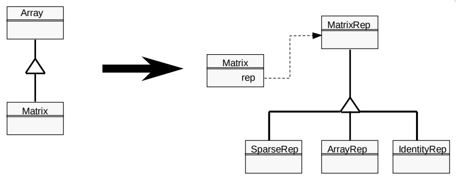
A Simple Refactoring

Create Empty Class



Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 19

A Complex Refactoring



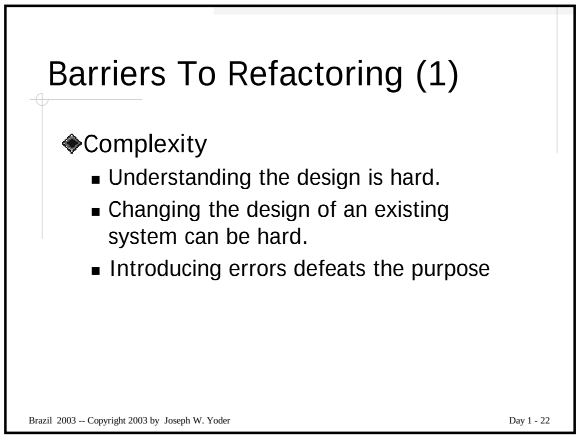
Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 20

Where Refactorings Come From

- Application Maintenance
- Application Extension
- Application Development
- Framework Development

"Refactoring often applies Design Patterns"

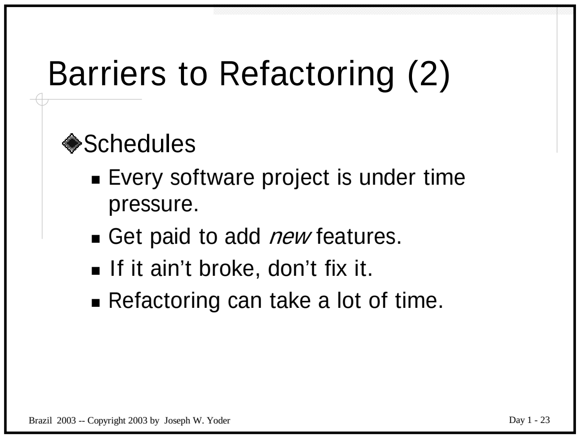
Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 21



Barriers To Refactoring (1)

- Complexity
  - Understanding the design is hard.
  - Changing the design of an existing system can be hard.
  - Introducing errors defeats the purpose

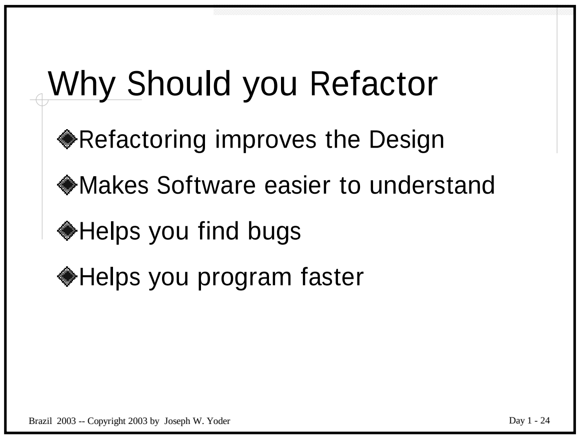
Day 1 - 22



Barriers to Refactoring (2)

- Schedules
  - Every software project is under time pressure.
  - Get paid to add *new* features.
  - If it ain’t broke, don’t fix it.
  - Refactoring can take a lot of time.

Day 1 - 23



Why Should you Refactor

- Refactoring improves the Design
- Makes Software easier to understand
- Helps you find bugs
- Helps you program faster

Day 1 - 24

When Should you Refactor

- The Rule of Three
- Refactor when you add function
- Refactor when you need to fix a bug
- Refactor during code reviews

There are also forces for when you should not refactor but there is a debt to pay

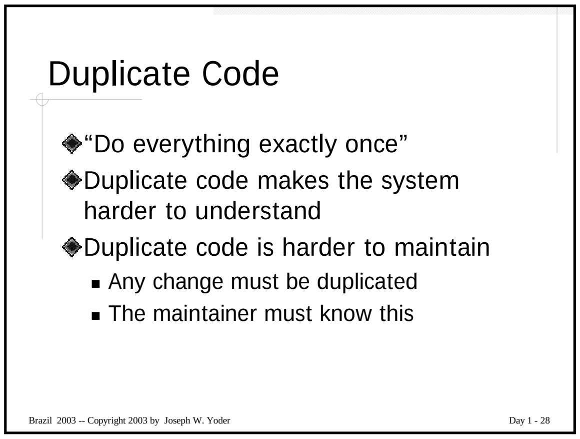
Consequences of not Refactoring

- Changes are made in the most expedient way
- Design becomes more corrupt
- Code becomes more brittle
- Changes become more expensive and more frequent
- Big Balls of Mud – Foote & Yoder

Code Smells

If it stinks, change it.......Grandma Beck

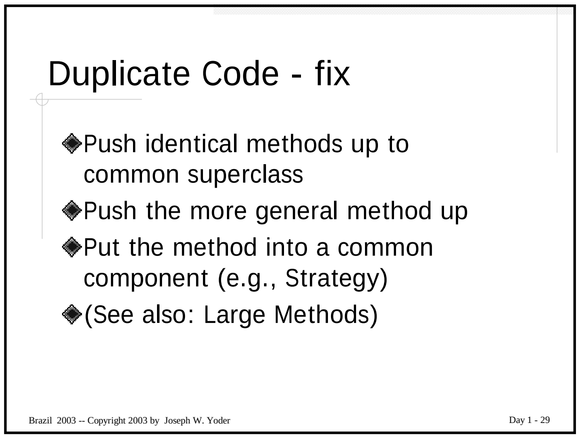
There are certain things that lead to code smells and there are certain refactorings that help us deal with these smells



Duplicate Code

- "Do everything exactly once"
- Duplicate code makes the system harder to understand
- Duplicate code is harder to maintain
  - Any change must be duplicated
  - The maintainer must know this

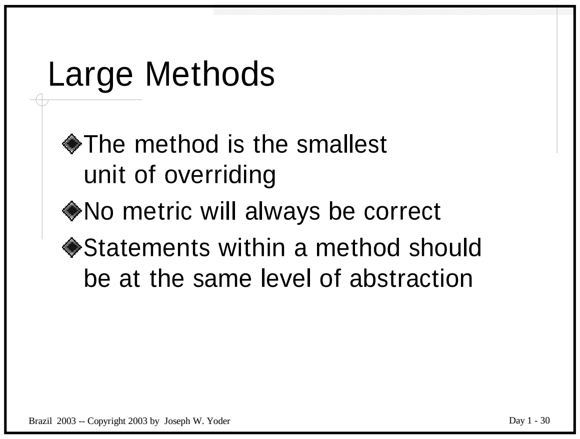
Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 28



Duplicate Code - fix

- Push identical methods up to common superclass
- Push the more general method up
- Put the method into a common component (e.g., Strategy)
- (See also: Large Methods)

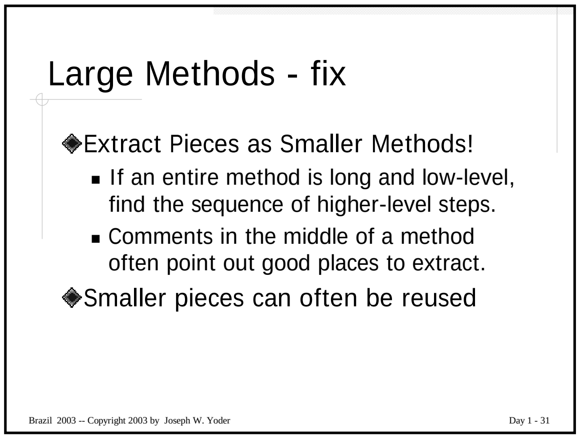
Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 29



Large Methods

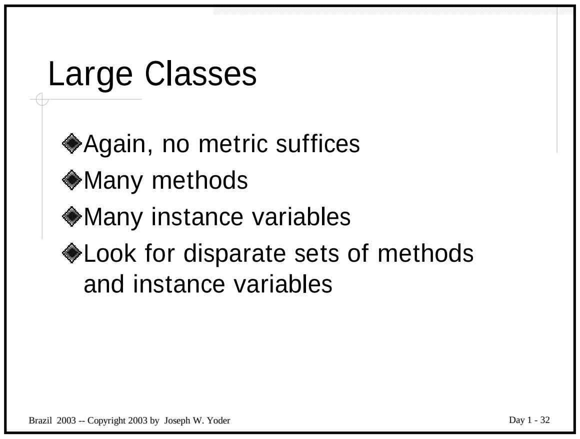
- The method is the smallest unit of overriding
- No metric will always be correct
- Statements within a method should be at the same level of abstraction

Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 30



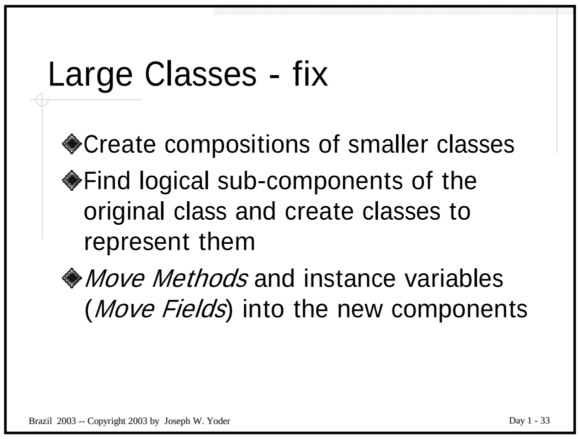
Large Methods - fix

- Extract Pieces as Smaller Methods!
  - If an entire method is long and low-level, find the sequence of higher-level steps.
  - Comments in the middle of a method often point out good places to extract.
- Smaller pieces can often be reused



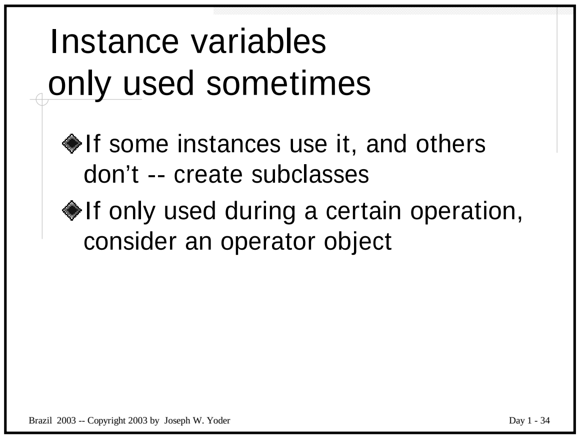
Large Classes

- Again, no metric suffices
- Many methods
- Many instance variables
- Look for disparate sets of methods and instance variables



Large Classes - fix

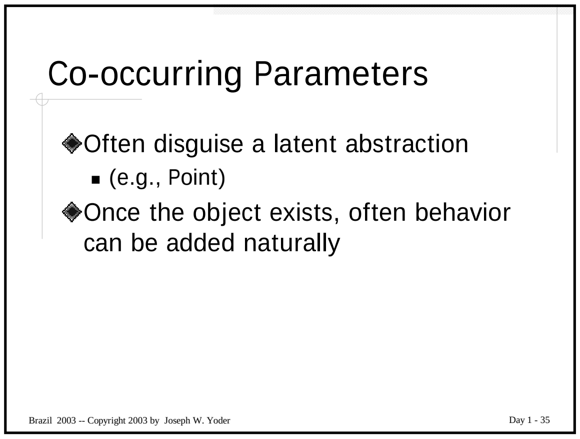
- Create compositions of smaller classes
- Find logical sub-components of the original class and create classes to represent them
- *Move Methods* and instance variables (*Move Fields*) into the new components



Instance variables only used sometimes

- If some instances use it, and others don’t -- create subclasses
- If only used during a certain operation, consider an operator object

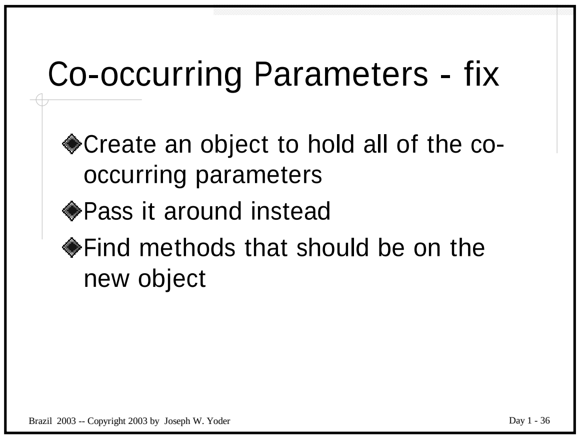
Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 34



Co-occurring Parameters

- Often disguise a latent abstraction
  - (e.g., Point)
- Once the object exists, often behavior can be added naturally

Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 35



Co-occurring Parameters - fix

- Create an object to hold all of the co-occurring parameters
- Pass it around instead
- Find methods that should be on the new object

Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 36

Feature Envy

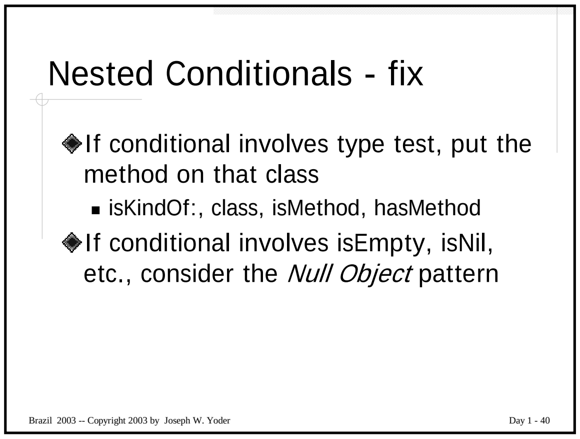
- Symptom of methods in the wrong place
- A method is always accessing values from another class
- Seems to be more interested in another class rather than the class it lives

Feature Envy - fix

- Use *Move Method* to put the method into the class it is usually working on
- If it is interested in multiple classes, put the method into the class where it accesses most of the values

Nested Conditionals

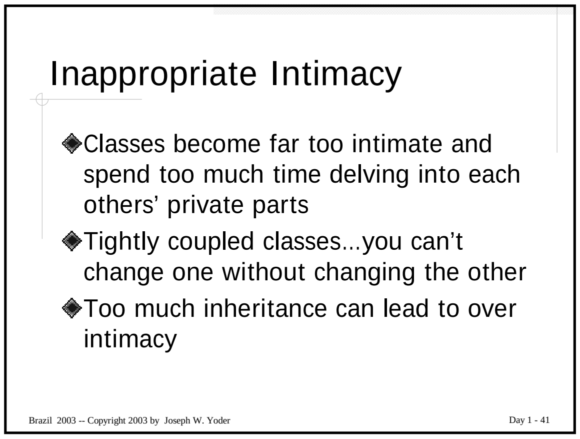
- Symptom of methods in the wrong place
- Rather than switching allow method lookup to do the switching
- New cases do not require changing existing code (The Ultimate Goal)



Nested Conditionals - fix

- If conditional involves type test, put the method on that class
  - isKindOf:, class, isMethod, hasMethod
- If conditional involves isEmpty, isNil, etc., consider the *Null Object* pattern

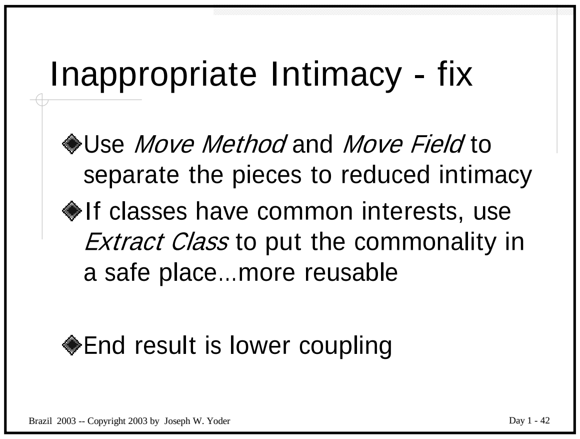
Brazil 2003 -- Copyright 2003 by Joseph W. Yoder   Day 1 - 40



Inappropriate Intimacy

- Classes become far too intimate and spend too much time delving into each others' private parts
- Tightly coupled classes...you can't change one without changing the other
- Too much inheritance can lead to over intimacy

Brazil 2003 -- Copyright 2003 by Joseph W. Yoder   Day 1 - 41



Inappropriate Intimacy - fix

- Use *Move Method* and *Move Field* to separate the pieces to reduced intimacy
- If classes have common interests, use *Extract Class* to put the commonality in a safe place...more reusable

- End result is lower coupling

Brazil 2003 -- Copyright 2003 by Joseph W. Yoder   Day 1 - 42

Parallel Hierarchies

- Every time you make a subclass of one class, you have to make a subclass of another class from another hierarchy.
- If you used good naming techniques, you can recognize this since the prefix of your class names will be the same for both hierarchies
- Move Method and Move Field can help the referring class disappear

Comments

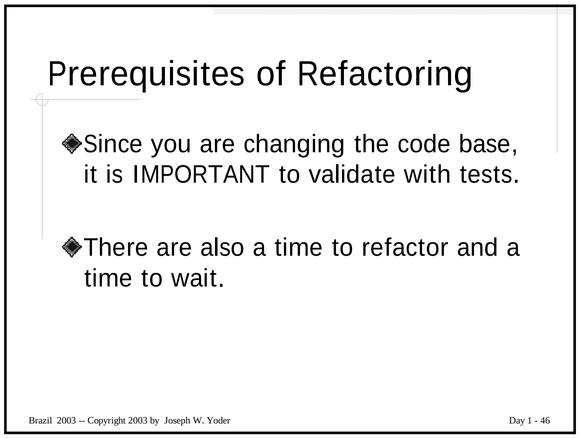
- We are not against comments but…
- If you see large methods that have places where the code is commented, use *Extract Method* to pull that out to a comment

Comments Example

```
void printOwing (double amount) {
    printBanner();
    //print details
    System.out.println (name: “ + _name);
    System.out.println (amount: “ + amount);
    …}
```

↓

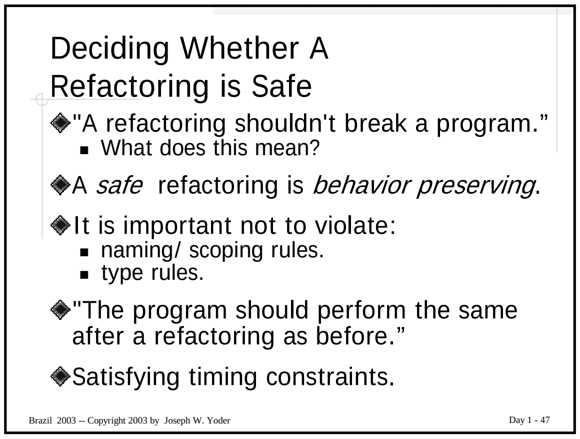
```
void printOwing (double amount) {
    printBanner();
    printDetails();}
void printDetails (double amount) {
    System.out.println (name: “ + _name);
    System.out.println (amount: “ + amount);
    …}
```



Prerequisites of Refactoring

- Since you are changing the code base, it is IMPORTANT to validate with tests.
- There are also a time to refactor and a time to wait.

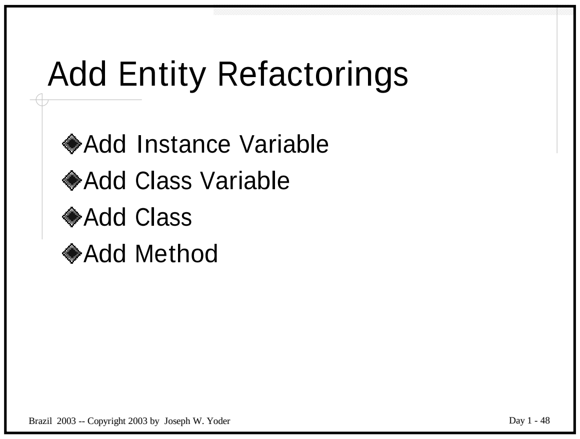
Day 1 - 46



Deciding Whether A Refactoring is Safe

- "A refactoring shouldn't break a program."
  - What does this mean?
- A *safe* refactoring is *behavior preserving*.
- It is important not to violate:
  - naming/ scoping rules.
  - type rules.
- "The program should perform the same after a refactoring as before."
- Satisfying timing constraints.

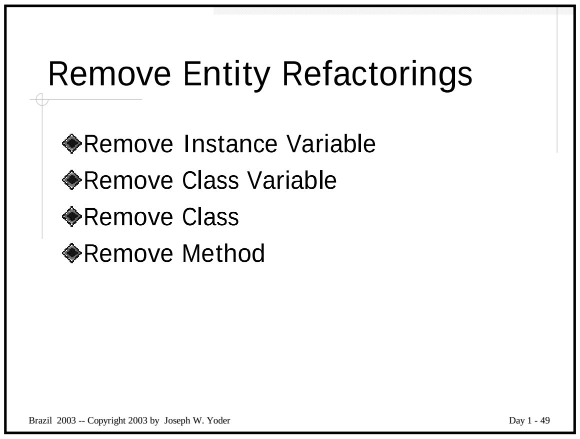
Day 1 - 47



Add Entity Refactorings

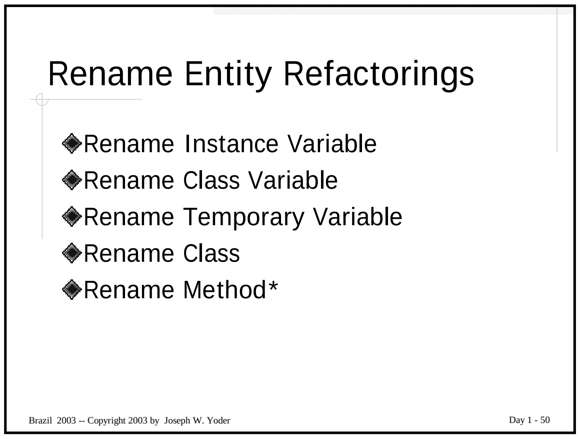
- Add Instance Variable
- Add Class Variable
- Add Class
- Add Method

Day 1 - 48



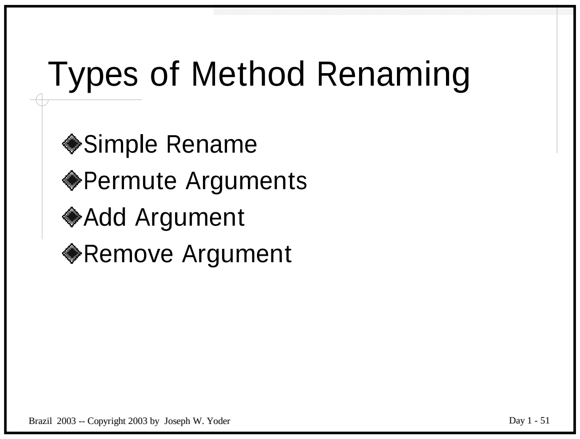
Remove Entity Refactorings

- Remove Instance Variable
- Remove Class Variable
- Remove Class
- Remove Method



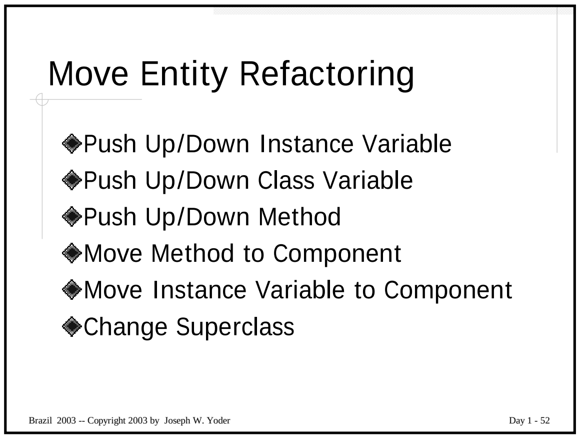
Rename Entity Refactorings

- Rename Instance Variable
- Rename Class Variable
- Rename Temporary Variable
- Rename Class
- Rename Method*



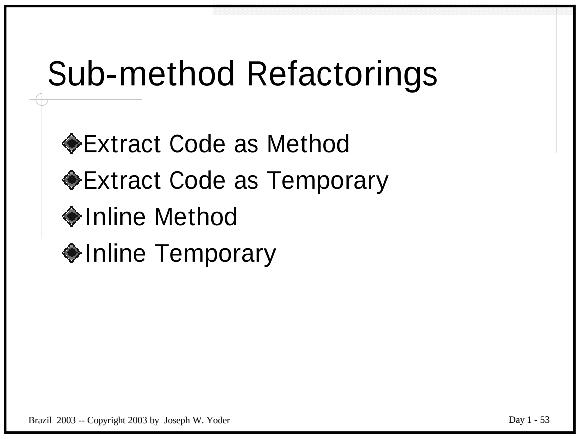
Types of Method Renaming

- Simple Rename
- Permute Arguments
- Add Argument
- Remove Argument



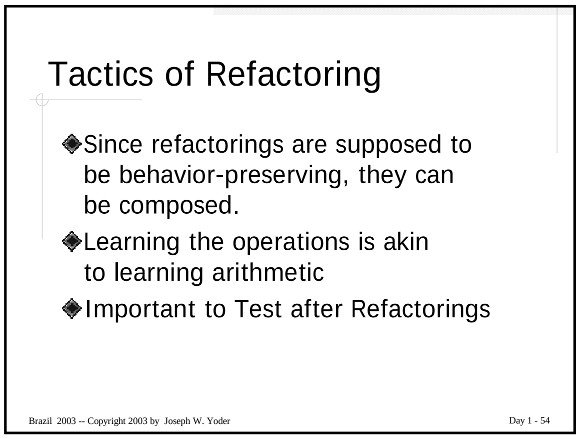
Move Entity Refactoring

- ◈Push Up/Down Instance Variable
- ◈Push Up/Down Class Variable
- ◈Push Up/Down Method
- ◈Move Method to Component
- ◈Move Instance Variable to Component
- ◈Change Superclass



Sub-method Refactorings

- ◈Extract Code as Method
- ◈Extract Code as Temporary
- ◈Inline Method
- ◈Inline Temporary



Tactics of Refactoring

- ◈Since refactorings are supposed to be behavior-preserving, they can be composed.
- ◈Learning the operations is akin to learning arithmetic
- ◈Important to Test after Refactorings

Using Standard Tools

- To be safe, must have tests
- Should pass the tests before and after refactoring
  - Commercial Testing Tools
  - Kent Beck's Testing Framework (SUnit, JUnit)
- Take small steps, testing between each
- Java is getting better tools

Refactoring Scripts

- Experts develop internal scripts
- Rename method
  - 1. Browse all implementers
  - 2. Browse all senders
  - 3. Edit and rename all implementers
  - 4. Edit and rename all senders
  - 5. Remove all implementers
  - 6. TEST!!!!

Composing Method Catalogue

- Extract Method, Inline Method,
- Inline Temp, Replace Temp with Query,
- Introduce Explaining Variable,
- Split Temporary Variable,
- Remove Assignments to Parameters,
- Replace Method with Method Object,
- Substitute Algorithm

Extract Method

```
void printOwing(double amount) {
   printBanner();

   //print details
   System.out.println("name:" + _name);
   System.out.println("amount:" + _amount);

}
```

Extract Method (2)

```
void printOwing (double amount) {
   printBanner();
   printDetails(amount);
}

void printDetails (double amount) {
   System.out.println("name:" + _name);
   System.out.println("amount:" + _amount);
}
```

Extract Method Mechanics

Create a new method and name it after the intention of the code to extract.

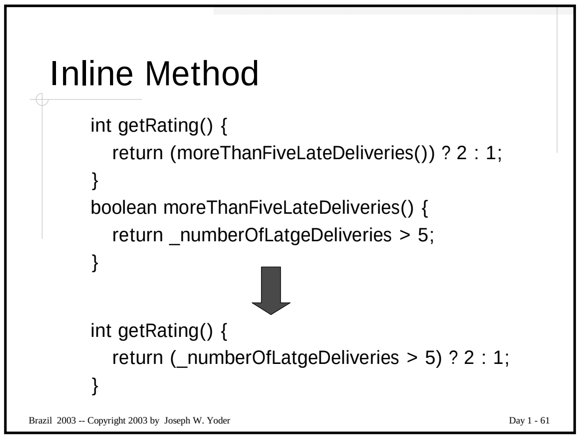
Copy the extracted code from the source to the new method.

Scan the extracted code for references to any variables that are local to the source method.

See whether any temps are used only within extracted code.

Pass into target method as parameters local-scope variables that are read from the extracted code.

Replace the extracted code in the source method.

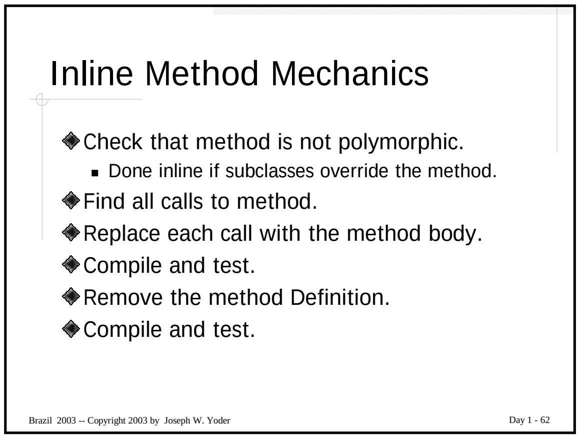


Inline Method

```
int getRating() {
   return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
   return _numberOfLatgeDeliveries > 5;
}
```

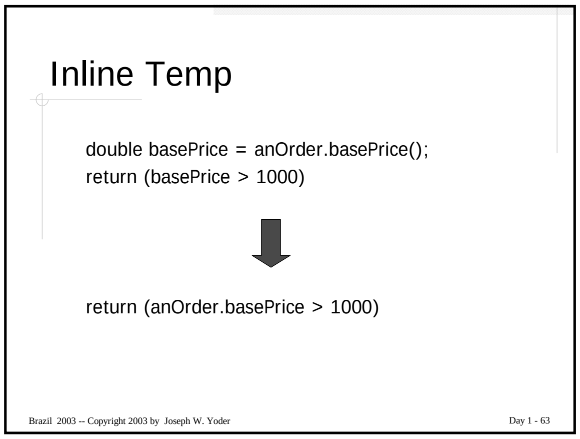
⬇

```
int getRating() {
   return (_numberOfLatgeDeliveries > 5) ? 2 : 1;
}
```



Inline Method Mechanics

- Check that method is not polymorphic.
  - Done inline if subclasses override the method.
- Find all calls to method.
- Replace each call with the method body.
- Compile and test.
- Remove the method Definition.
- Compile and test.



Inline Temp

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

⬇

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

Inline Temp Mechanics

- Declare the temp as final.
  - Insures the temp is really assigned only once.
- Find all references to the temp and replace with the right-hand side of the assignment.
- Compile and test after each change.
- Remove the declaration and the assignment.
- Compile and test.

Replace Temp with Query

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98
```

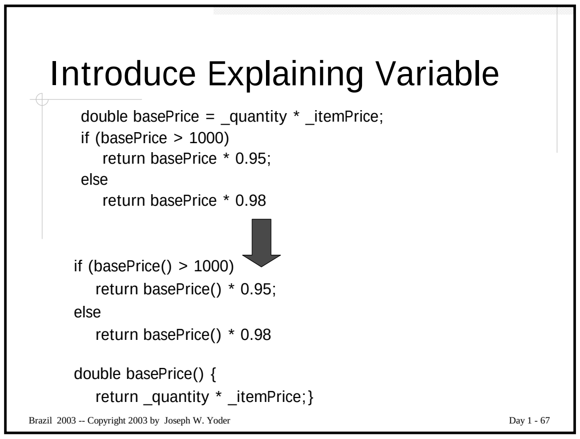
↓

```
if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98

double basePrice() {
    return _quantity * _itemPrice;}
```

Replace Temp Mechanics

- Look for temp assigned to once.
  - If more than once, consider split temporary.
- Declare the temp as Final.
- Compile.
- Extract the right hand side into a method.
- Compile and test.

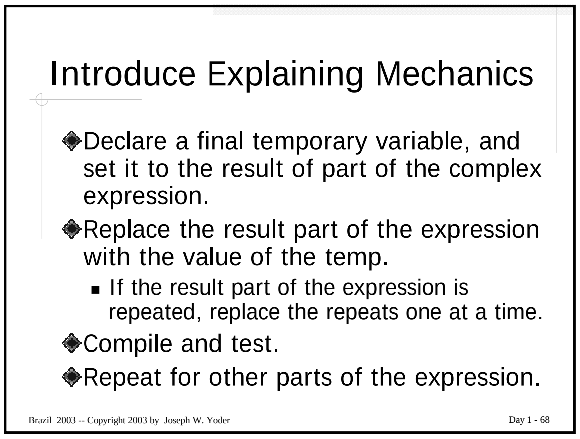


Introduce Explaining Variable

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98
```

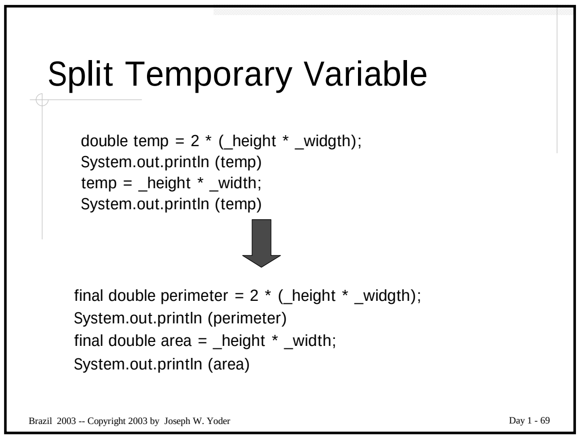
```
if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98

double basePrice() {
    return _quantity * _itemPrice;}
```



Introduce Explaining Mechanics

- Declare a final temporary variable, and set it to the result of part of the complex expression.
- Replace the result part of the expression with the value of the temp.
  - If the result part of the expression is repeated, replace the repeats one at a time.
- Compile and test.
- Repeat for other parts of the expression.



Split Temporary Variable

```
double temp = 2 * (_height * _widgth);
System.out.println (temp)
temp = _height * _width;
System.out.println (temp)
```

```
final double perimeter = 2 * (_height * _widgth);
System.out.println (perimeter)
final double area = _height * _width;
System.out.println (area)
```

Split Temporary Mechanics

- ◈Change the name of the temp at is declaration and its final assignment.
- ◈Declare new temp as final.
- ◈Change all the references of the temp up to its second assignment.
- ◈Declare the temp as its second assignment.
- ◈Compile and test.
- ◈Repeat in stages.

Remove Assignments to Params

```
int discount (int inputVal, int quantity, int yearToDate) {
   if (inputVal  > 50) inputVal -= 2;
```

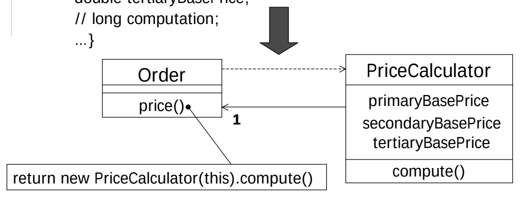
```
int discount (int inputVal, int quantity, int yearToDate) {
   int result = inputVal;
   if (inputVal  > 50) result -= 2;
```

Remove Assignments Mechanics

- ◈Create a temp variable for the parameter.
- ◈Replace all references to the parameter, made after the assignment, to the temporary variable.
- ◈Change the assignment to assign to the temporary variable.
- ◈Compile and test.

Remove Method with Method Object

```
Class Order ...
   double price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // long computation;
        ...}
```



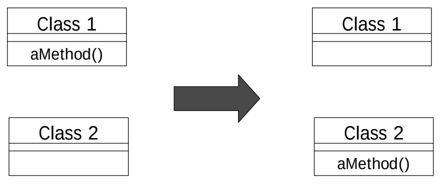
Remove Method to Method Object Mechanics

- Create a new class named after method.
- Give the new class a final field for the object that hosted the original method.
- Give the new class a constructor for the original object and each parameter.
- Give the new class a compute method.
- Copy the body of original method to compute.
- Compile.
- Replace the old method with the one that creates the new object and calls compute.

Moving Features Catalogue

- Move Method, Move Field,
- Extract Class, Inline Class,
- Hide Delegate
- Remove Middle Man
- Introduce Foreign Method
- Introduce Local Extension

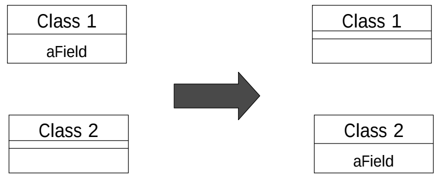
Move Method



Move Method Mechanics

- Examine all features used by the source method that are defined in the source class. Consider whether they also should be moved.
- Check the sub and superclasses of the source class for other definitions.
- Declare the method in the target class.
- Copy the code from the source method to the target.
- Compile the target class.
- Determine how to reference the correct target object.
- Turn the source method into a delegating method.
- Compile and test.
- Decide whether to remove the source method or retain it as delegating method.

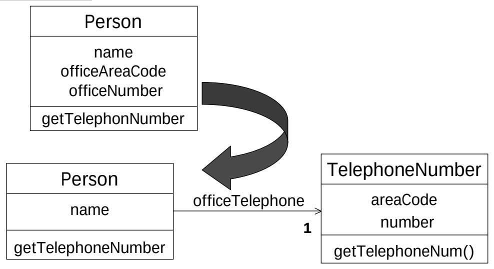
Move Field



Move Field Mechanics

- If the field is public, use encapsulate field.
- Create a field in the target class with getters and setters.
- Compile the target class.
- Determine how to reference target object from the source.
- Remove the field on the source class.
- Replace all references to the source field with references to the appropriate method on the target.

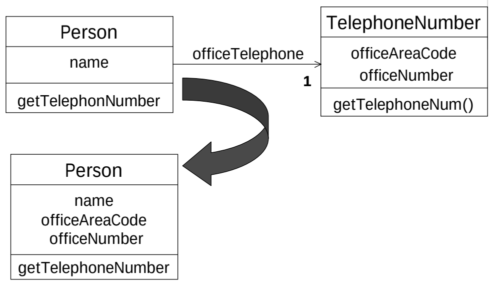
Extract Class



Extract Class Mechanics

- Decide how to split the responsibilities of the class.
- Create a new class to split responsibilities.
- Make a link from the old to the new class.
- Use Move Field on each field to move.
- Compile and Test.
- Use Move Method on each desired method.
- Compile and Test.
- Review and reduce the interfaces of the class.

Inline Class



Organize Data Catalogue

- Self Encapsulate Field, Replace Data Value with Object, Change Value to Reference, Change Reference to Value, Replace Array with Object, Duplicate Observed Data, Change (Uni|Bi) directional Association to (Bi|Uni) directional, Replace Magic Number, Encapsulate (Field|Collection), Replace Record with Data Class, ……..

Self Encapsulate Field

```
private int _low, _high;
boolean includes (int arg) {
   return arg >= _low && arg <= _high; }
```

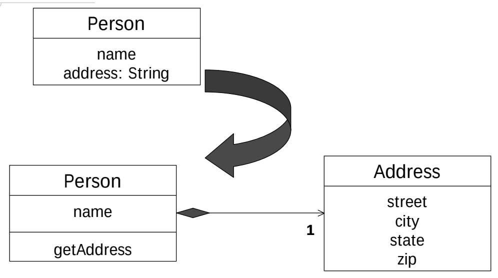
⬇

```
private int _low, _high;
boolean includes (int arg) {
   return arg >= getLow() && arg <= getHigh(); }
int getLow() {return _low;);
int getHigh() {return _high;);
```

Self Encapsulate Mechanics

- Create a getting and setting method for the field.
- Find all references to the field and replace them with a getting or setting method.
- Make the field private.
- Compile and Test.

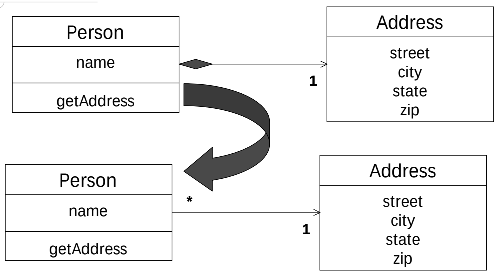
Replace Data Value with Object



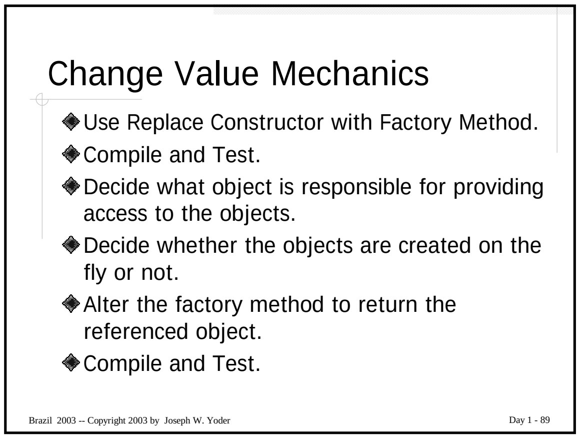
Replace Data Value Mechanics

- Create the class for the value.
- Compile.
- Change the type of the field in the source class to the new class.
- Change the getter in the source class to call the getter in the new class.
- If the field is mentioned in the source class constructor, assign the field.
- Change the getting message to create a new instance of the new class.
- Compile and Test.

Change Value to Reference



Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 88

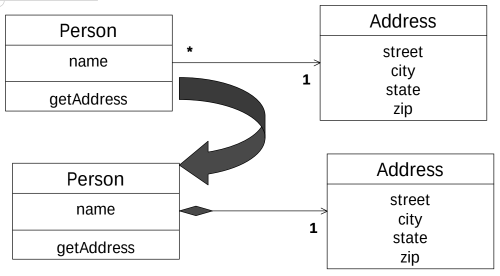


Change Value Mechanics

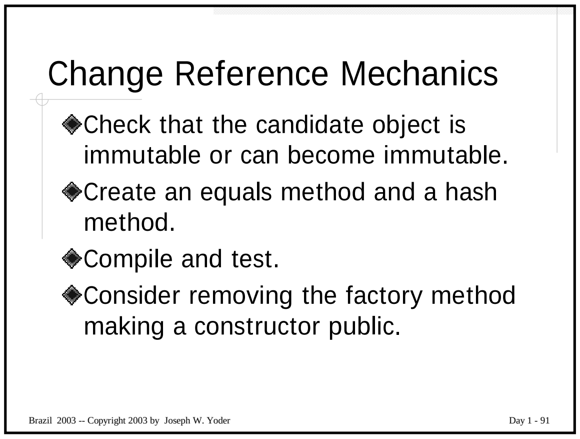
- Use Replace Constructor with Factory Method.
- Compile and Test.
- Decide what object is responsible for providing access to the objects.
- Decide whether the objects are created on the fly or not.
- Alter the factory method to return the referenced object.
- Compile and Test.

Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 89

Change Reference to Value

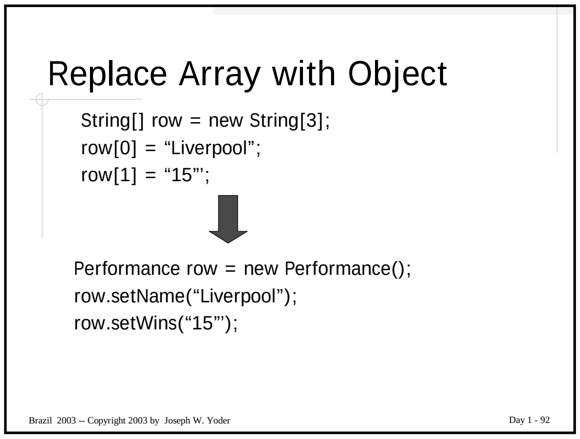


Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 90



Change Reference Mechanics

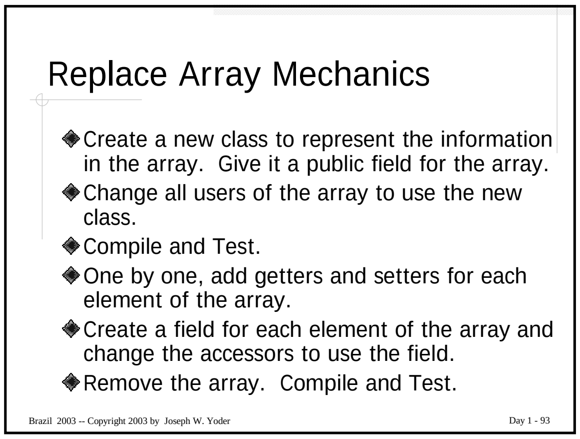
- Check that the candidate object is immutable or can become immutable.
- Create an equals method and a hash method.
- Compile and test.
- Consider removing the factory method making a constructor public.



Replace Array with Object

```
String[] row = new String[3];
row[0] = “Liverpool”;
row[1] = “15”';
```

```
Performance row = new Performance();
row.setName(“Liverpool”);
row.setWins(“15”');
```

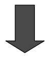


Replace Array Mechanics

- Create a new class to represent the information in the array. Give it a public field for the array.
- Change all users of the array to use the new class.
- Compile and Test.
- One by one, add getters and setters for each element of the array.
- Create a field for each element of the array and change the accessors to use the field.
- Remove the array. Compile and Test.

Encapsulate Field

```
public String _name;
```



```
private String _name;
public String getName() {return _name;};
public void setName(String arg) {_name = arg;);
```

Encapsulate Field Mechanics

- Create getting and setting methods for the field.
- Find out clients outside the class that reference the field. If the client uses the value, replace the reference with a call to the getting method.
- Compile and Test after each change.
- Once all clients are changed, declare the field as private.
- Compile and Test.

Simplifying Conditionals Catalogue

- Decompose Conditional, Consolidate Conditional Expression, Consolidate Duplicate Conditional Fragments, Remove Control Flag, Replace Nested Conditionals with (Guard Clauses | Polymorphism), Introduce Null Object, Introduce Assertion

Decompose Conditional

```
if (date.before (SUMMER_START) || date.after (SUMMER_END)
    charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

⬇

```
if (notSummer(date))
    charge = winterCharge(quantity)
else charge = summerCharge(quantity);
```

Decompose Mechanics

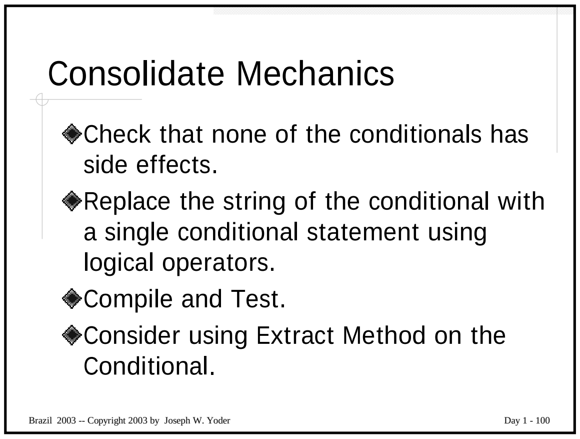
- Extract the conditional into its own method.
- Extract the then part and the else part into their own methods.

Consolidate Conditional

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    //compute the disability amount
```

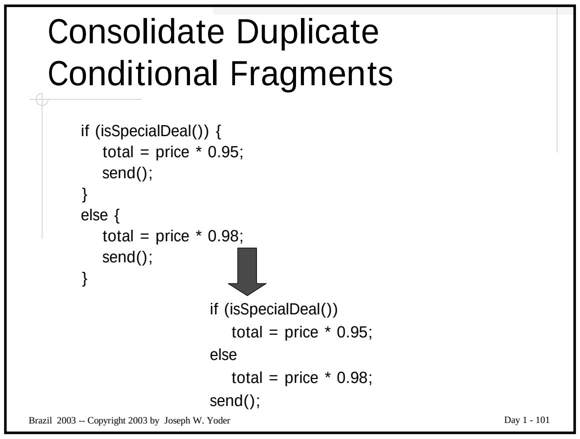
⬇

```
double disabilityAmount() {
    if (isNotEligableForDisability()) return 0;
    //compute the disability amount
```



Consolidate Mechanics

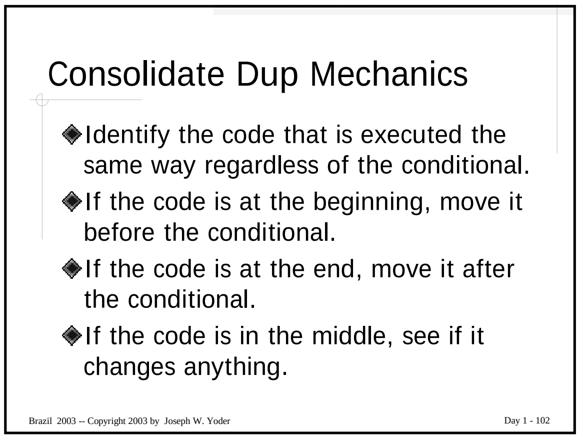
- ◈Check that none of the conditionals has side effects.
- ◈Replace the string of the conditional with a single conditional statement using logical operators.
- ◈Compile and Test.
- ◈Consider using Extract Method on the Conditional.



Consolidate Duplicate Conditional Fragments

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;
send();
```



Consolidate Dup Mechanics

- ◈Identify the code that is executed the same way regardless of the conditional.
- ◈If the code is at the beginning, move it before the conditional.
- ◈If the code is at the end, move it after the conditional.
- ◈If the code is in the middle, see if it changes anything.

Replace Nested Conditional with Guard Clauses

- A method has conditional behavior that does not make clear the normal path of execution.
- *Use guard clauses for the special cases!*

Replace Nested Conditional with Guard Clauses

```
double getPayAmount() {
   double result;
   if (_isDead) result = deadAmount();
   else {
        if (_isSeparated) result = separatedAmount();
        else {
                if (_isRetired) result = retiredAmount();
                else result = normalAmount;}}}
```

⇓

```
double getPayAmount() {
   if (_isDead) return deadAmount();
   if (_isSeparated) return separatedAmount();
   if (_isRetired) return retiredAmount();
   return normalAmount;}}}
```

Replace Nested Mechanics

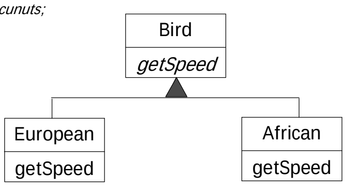
- For each check put in the guard clause.
- Compile and Test after each check is replace with a guard clause.
- Might consider consolidate conditional if the guards use the same result.

Replace Conditional with Polymorphism

- ◈You have a conditional that chooses different behavior depending on the type of an object
- ◈*Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract!*

Replace Conditional with Polymorphism

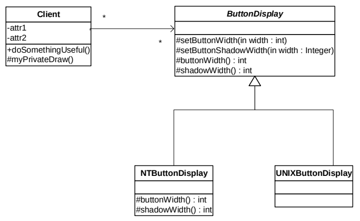
```
double getSpeed() {
   switch (_type) {
         case EUROPEAN:
                 return getBaseSpeed();
         case AFRICAN:
                 return getBaseSpeed() – getLoadFactor() * _number
   ofCocunuts;
   ...}
```



Replace Polymorphism Mechanics

- ◈If the conditional is part of a larger statement, take apart the conditional and use Extract Method.
- ◈If necessary, use Move Method to place the conditional at the top of the inheritance hierarchy.
- ◈Create classes and copy the body of the leg of the conditional into the subclass.
- ◈Compile and Test…and continue on.

Strategy
(an example)



Moving Code "Refactoring"

To move a function to a different class, add an argument to refer to the original class of which it was a member and change all references to member variables to use the new argument.

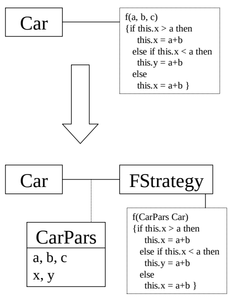
If you are moving it to the class of one of the arguments, you can make the argument be the receiver.

Moving function f from class X to class B

```
class X {
    int f(A anA, B aB){
          return (anA.size + size) / aB.size;
    } ...
class B {
    int f(A anA, X anX){
          return (anA.size + anX.size) / size;
    } ...
```

Moving Code "Refactoring"

- You can also pass in a parameter object which gives the algorithm all of the values that it will need.
- Inner Classes can help by providing access to values that the algorithm may need.



```
f(a, b, c)
{if this.x > a then
    this.x = a+b
  else if this.x < a then
    this.y = a+b
  else
    this.x = a+b }
```

```
f(CarPars Car)
{if this.x > a then
    this.x = a+b
  else if this.x < a then
    this.y = a+b
  else
    this.x = a+b }
```

Introduce Null Object

- You have repeated checks for a null value.
- *Repalce the null value with a null object!*

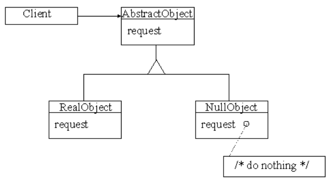
if (address = null) System.out.println(“”)
else System.out.println(address);

NullObject

Author: Bobby Woolf, PLoPD 3

Intent: Provide surrogate for another object that shares same interface usually does nothing but can provide default behavior encapsulate implementation decisions of how to do nothing.

Structure:

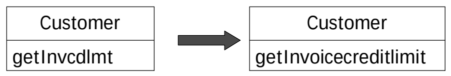


Simpler Method Calls Catalogue

- Rename Method, (Remove|Add) Parameter, Separate Query with Modifier, Parameterize Method, Replace Parameter with Explicit Methods, Preserve Whole Object, Replace Parameter with Method, Introduce Parameter Object, Remove Setting Method, Hide Method, Replace Constructor with Factory Method, Encapsulate Downcast, Replace Error Code with Exception, Replace Exception with Test

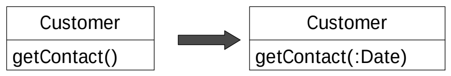
Rename Method

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



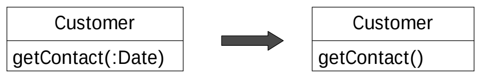
Add Parameter

- A method needs more information from the caller.
- *Add a parameter for an object that can pass on this information!*



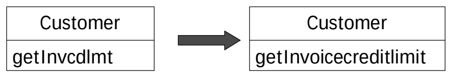
Remove Parameter

- A parameter is no longer used by the method body.
- *Remove the parameter!*



Rename Method

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

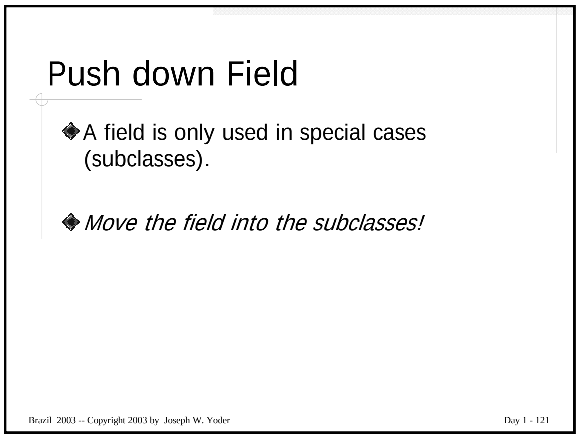


Generalization Catalogue

- Pull Up (Field|Method|Constructor Body)
- Push Down (Method|Field)
- Extract (Subclass|Superclass|Interface)
- Collapse Hierarchy
- Form Template Method
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance

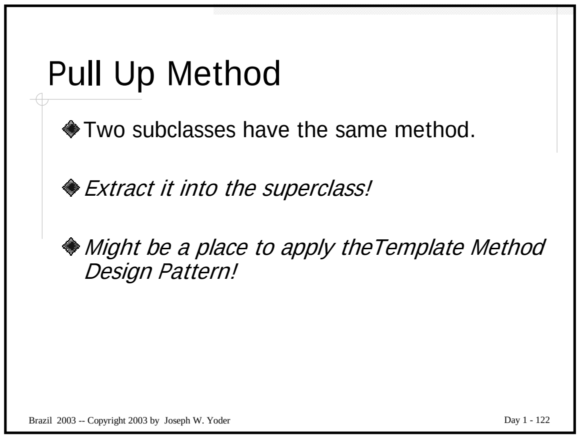
Pull Up Field

- Two subclasses have the same field.
- *Extract it into the superclass!*



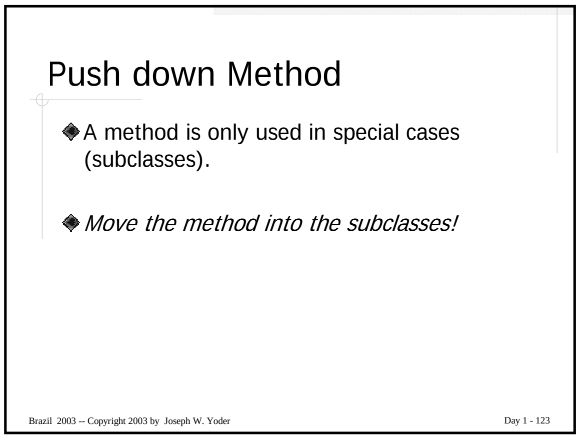
Push down Field

- A field is only used in special cases (subclasses).
- *Move the field into the subclasses!*



Pull Up Method

- Two subclasses have the same method.
- *Extract it into the superclass!*
- *Might be a place to apply theTemplate Method Design Pattern!*



Push down Method

- A method is only used in special cases (subclasses).
- *Move the method into the subclasses!*

Extract Superclass

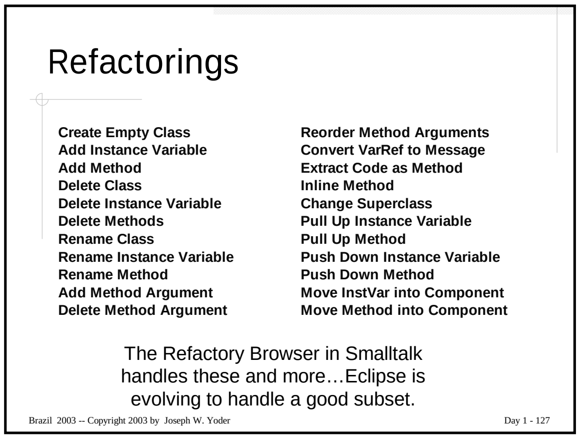
- You have two classes with similar features.
- *Create a superclass and move the common features to the superclass!*

Extract Interface

- Several Clients use the same subset of a class's interface, or two classes have part of their interface in common.
- *Extract the subset into an interface!*

Summary

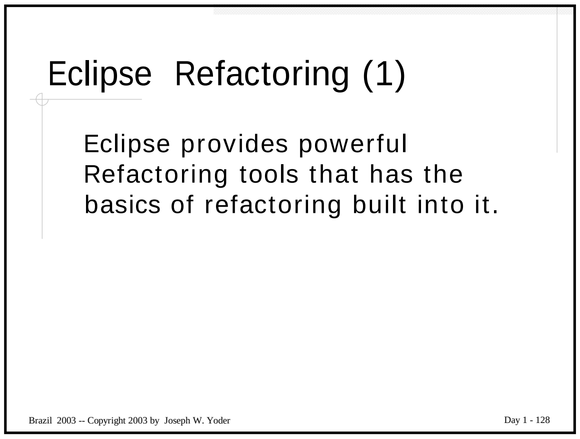
- We have seen the mechanics for many of the Refactorings.
- When you find smelly code, you often apply Refactorings to clean your code.
- Refactorings do often apply Design Patterns.



Refactorings

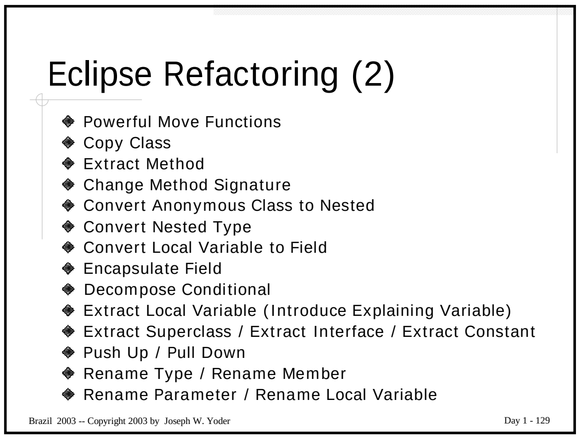
- Create Empty Class
- Add Instance Variable
- Add Method
- Delete Class
- Delete Instance Variable
- Delete Methods
- Rename Class
- Rename Instance Variable
- Rename Method
- Add Method Argument
- Delete Method Argument
- Reorder Method Arguments
- Convert VarRef to Message
- Extract Code as Method
- Inline Method
- Change Superclass
- Pull Up Instance Variable
- Pull Up Method
- Push Down Instance Variable
- Push Down Method
- Move InstVar into Component
- Move Method into Component

The Refactory Browser in Smalltalk handles these and more…Eclipse is evolving to handle a good subset.



Eclipse Refactoring (1)

Eclipse provides powerful Refactoring tools that has the basics of refactoring built into it.



Eclipse Refactoring (2)

- Powerful Move Functions
- Copy Class
- Extract Method
- Change Method Signature
- Convert Anonymous Class to Nested
- Convert Nested Type
- Convert Local Variable to Field
- Encapsulate Field
- Decompose Conditional
- Extract Local Variable (Introduce Explaining Variable)
- Extract Superclass / Extract Interface / Extract Constant
- Push Up / Pull Down
- Rename Type / Rename Member
- Rename Parameter / Rename Local Variable

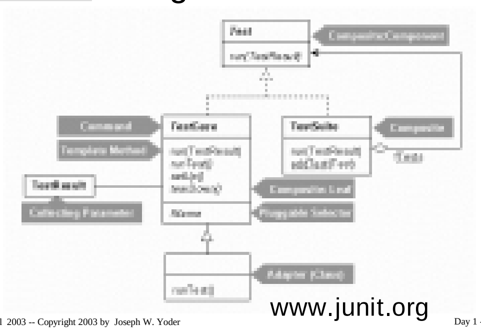
C# Refactoring (1)

C# Refactory is a refactoring, metrics and productivity add-in for Microsoft Visual Studio.NET and is a must-have if you do test driven development with a unit testing tool like csUnit.

C# Refactoring (2)

- Extract Method
- Decompose Conditional
- Extract Variable (Introduce Explaining Variable)
- Extract Superclass
- Extract Interface
- Copy Class
- Push Up Members
- Rename Type
- Rename Member
- Rename Parameter
- Rename Local Variable

Unit Testing JUnit & SUnit



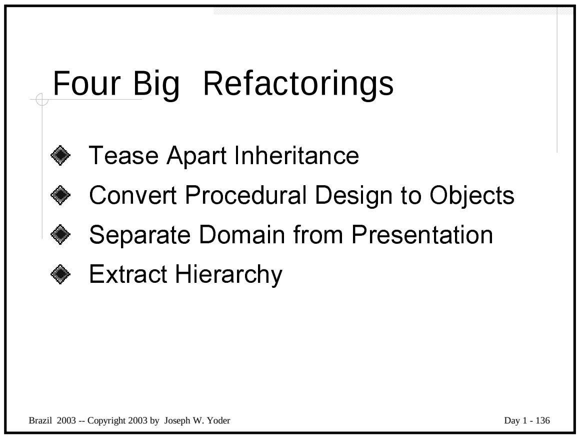
www.junit.org

Design Patterns (1)

Design Patterns (2)

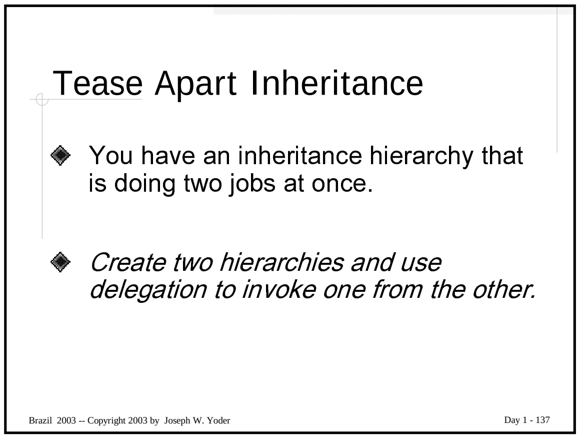
Large Refactorings

- Accumulations of many problems overtime can lead to a muddy design.
- You no longer understand the system.
- Accumulation of half-understood design decisions chokes a program.



Four Big Refactorings

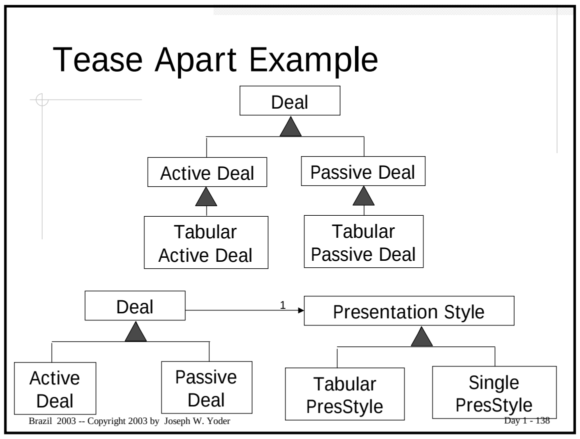
- Tease Apart Inheritance
- Convert Procedural Design to Objects
- Separate Domain from Presentation
- Extract Hierarchy



Tease Apart Inheritance

- You have an inheritance hierarchy that is doing two jobs at once.
- *Create two hierarchies and use delegation to invoke one from the other.*

Tease Apart Example



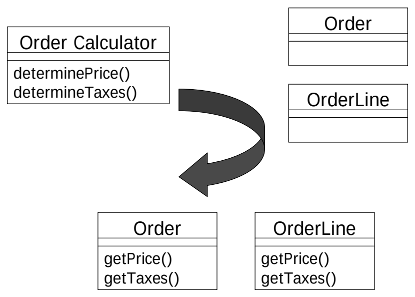
Tease Apart Mechanics

- Identify the different jobs being done by the hierarchy.
- Decide which job is more important.
- Use Extract Class at the common superclass to create an object for the additional job and add an instance variable for this object.
- Create subclasses of the extracted object for each subclass.
- Use Move Method to move the behavior in each subclass.

Convert Procedural To Object

- You have code written in Procedural Style.
- *Turn the data records into objects, break up the behavior and move the behavior into the objects.*

Convert Procedural Example



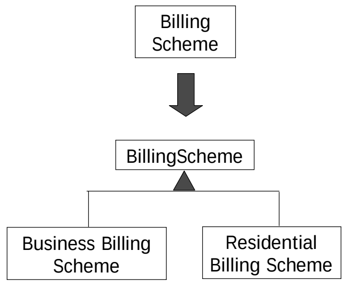
Convert Procedural Mechanics

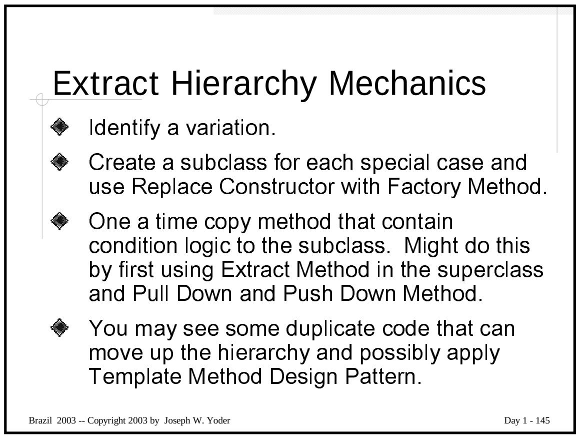
- Take each record type and turn it into a dumb data object with accessors.
- Take all procedural code and put it into a single class.
- Take each long procedure and apply Extract Method and the related refactorings to break it down. As you break it down, use Move Method to move to the appropriate class.
- Continue until you've moved all of the behavior away from the original class.

Extract Hierarchy

- You have class that is doing too much work, at least in part through many conditionals.
- *Create two hierarchy of classes in which each subclass represents a special case.*

Extract Hierarchy

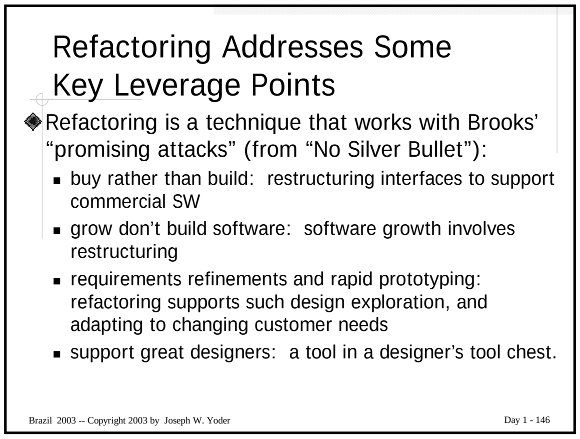




Extract Hierarchy Mechanics

- Identify a variation.
- Create a subclass for each special case and use Replace Constructor with Factory Method.
- One a time copy method that contain condition logic to the subclass. Might do this by first using Extract Method in the superclass and Pull Down and Push Down Method.
- You may see some duplicate code that can move up the hierarchy and possibly apply Template Method Design Pattern.

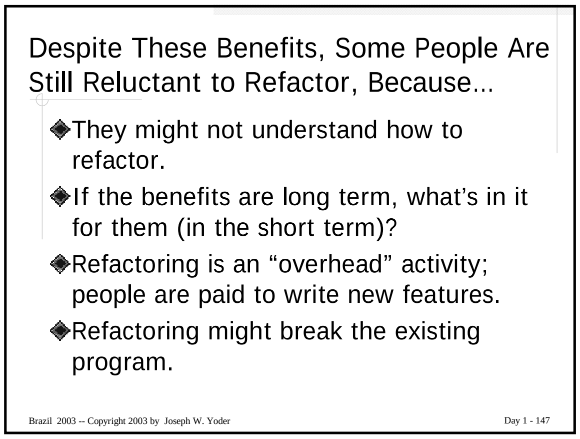
Day 1 - 145



Refactoring Addresses Some Key Leverage Points

- Refactoring is a technique that works with Brooks' "promising attacks" (from "No Silver Bullet"):
  - buy rather than build: restructuring interfaces to support commercial SW
  - grow don't build software: software growth involves restructuring
  - requirements refinements and rapid prototyping: refactoring supports such design exploration, and adapting to changing customer needs
  - support great designers: a tool in a designer's tool chest.

Day 1 - 146



Despite These Benefits, Some People Are Still Reluctant to Refactor, Because...

- They might not understand how to refactor.
- If the benefits are long term, what's in it for them (in the short term)?
- Refactoring is an "overhead" activity; people are paid to write new features.
- Refactoring might break the existing program.

Day 1 - 147

Addressing Concerns About Refactoring

- Understanding how to refactor:
  - Opdyke and Roberts doctoral thesis, and related publications.
  - *Refactoring: Improving the Design of Existing Code* (Fowler, Beck, Brant, Opdyke, and Roberts; Addison-Wesley, 1999).
- Achieving near-term benefits:
  - Interleave refactoring and incremental additions.

Addressing Concerns About Refactoring

- Reducing the overhead of refactoring:
  - Use browsers, text editors, and tools to reduce manual effort.
  - Try it! Refactoring saves in overall development time near term.
- Refactoring safely:
  - Need to have unit-level test suites that test the functionality of each module.
  - Apply precondition checking and test suites described in refactoring references.

Refactoring: Two Hard Problems

- Is it safe to apply a refactoring?
  - (Discussed earlier.)
- Which refactorings should you apply?

Deciding What Refactorings To Apply

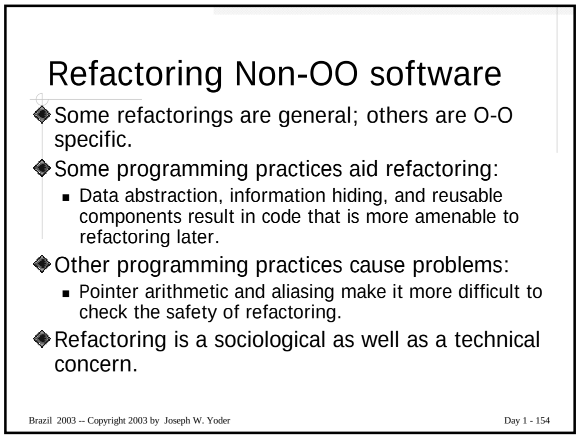
- It is the role of the designer to understand the goals of their application.
- Reasoning based upon program structure:
  - more powerful than upon simple textual scans.
- Heuristics can be applied to automatically detect some structural abnormalities.

Deciding What Refactorings To Apply

- Commonality analysis:
  - for example: common names/ types may suggest a common abstraction.
- Complexity analysis:
  - For example: very large, complex classes (or large functions, or functions with many arguments) are candidates for simplification/ splitting.
- Useful references:
  - Johnson/ Foote "Designing Reusable Classes"
    - http://www.laputan.org/drc/drc.html
  - Beck/ Fowler "Bad Smells in Code"
    - Refactoring text/ chapter 3.

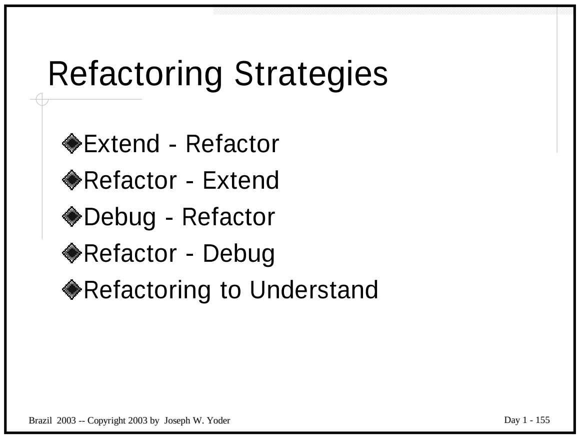
Refactoring Strategies

- Foote/ Opdyke "Lifecycle and Refactoring Patterns That Support Evolution & Reuse" (PLOP '94).
  - Prototype/ Initial Design; Expand; Consolidate.
  - http://www.laputan.org/lifecycle/Lifecycle.html
- Various Strategies
  (Compiled by Roberts & Yoder):
  - Extend – refactor
  - Refactor – extend
  - Debug – refactor
  - Refactor – debug
  - Refactoring to understand.



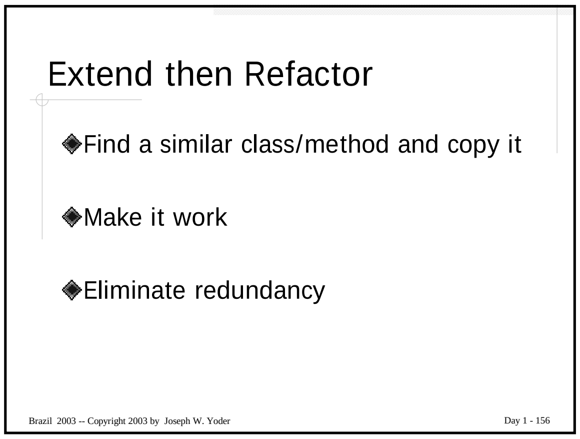
Refactoring Non-OO software

- Some refactorings are general; others are O-O specific.
- Some programming practices aid refactoring:
  - Data abstraction, information hiding, and reusable components result in code that is more amenable to refactoring later.
- Other programming practices cause problems:
  - Pointer arithmetic and aliasing make it more difficult to check the safety of refactoring.
- Refactoring is a sociological as well as a technical concern.



Refactoring Strategies

- Extend - Refactor
- Refactor - Extend
- Debug - Refactor
- Refactor - Debug
- Refactoring to Understand



Extend then Refactor

- Find a similar class/method and copy it
- Make it work
- Eliminate redundancy

Refactor then Extend

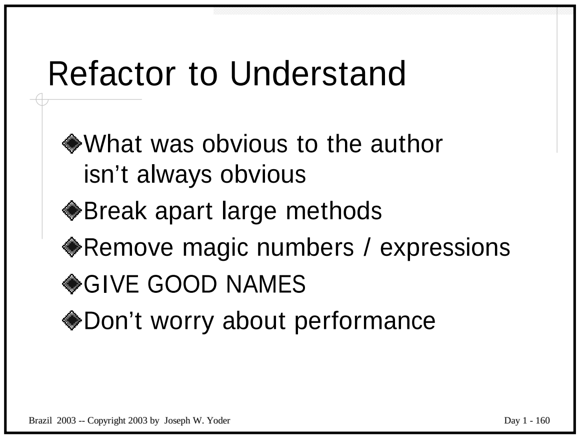
- Something seems too awkward to implement
- Refactor the design to make the change easy
- Make the change

Debug then Refactor

- Fix the bug
- Refactor the code to make the bug obvious
  - Extract method
  - Assign good names
  - Get rid of magic numbers and expressions

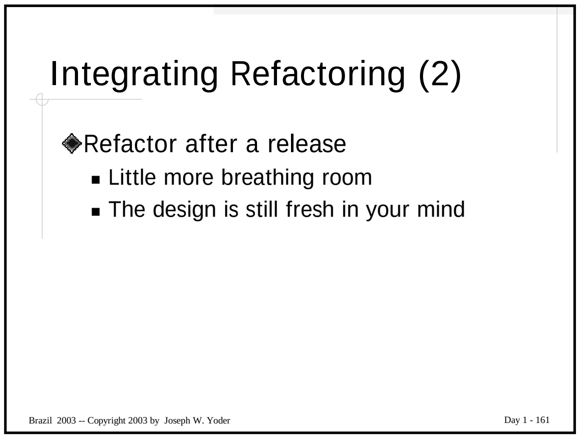
Refactor then Debug

- Since refactoring is behavior-preserving, it preserves bad behavior.
- Refactor to simplify complicated methods
- Debug it



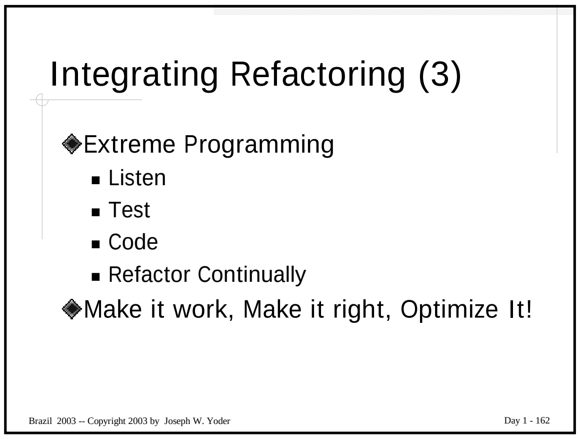
Refactor to Understand

- What was obvious to the author isn't always obvious
- Break apart large methods
- Remove magic numbers / expressions
- GIVE GOOD NAMES
- Don't worry about performance



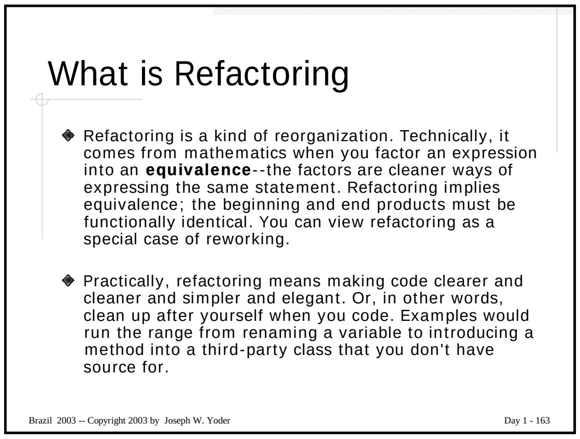
Integrating Refactoring (2)

- Refactor after a release
  - Little more breathing room
  - The design is still fresh in your mind



Integrating Refactoring (3)

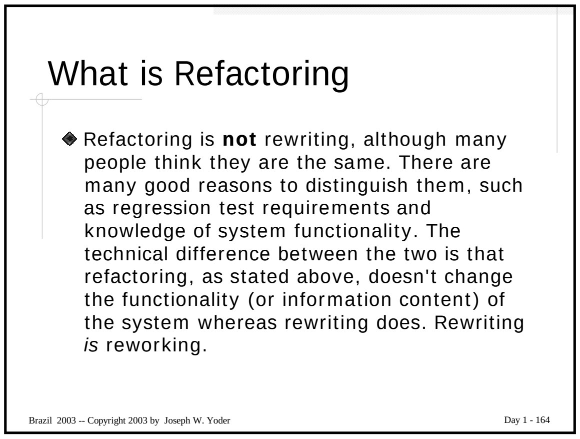
- Extreme Programming
  - Listen
  - Test
  - Code
  - Refactor Continually
- Make it work, Make it right, Optimize It!



What is Refactoring

- Refactoring is a kind of reorganization. Technically, it comes from mathematics when you factor an expression into an **equivalence**--the factors are cleaner ways of expressing the same statement. Refactoring implies equivalence; the beginning and end products must be functionally identical. You can view refactoring as a special case of reworking.
- Practically, refactoring means making code clearer and cleaner and simpler and elegant. Or, in other words, clean up after yourself when you code. Examples would run the range from renaming a variable to introducing a method into a third-party class that you don't have source for.

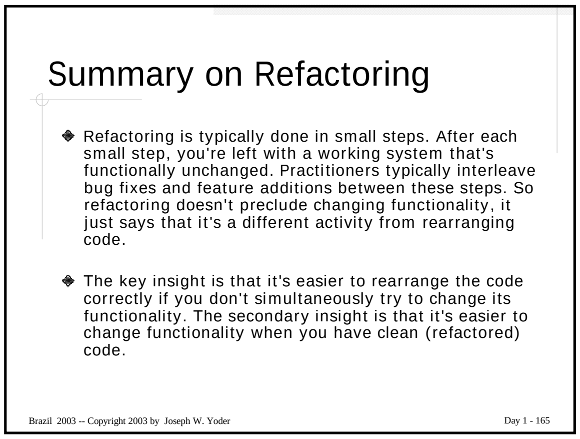
Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 163



What is Refactoring

- Refactoring is **not** rewriting, although many people think they are the same. There are many good reasons to distinguish them, such as regression test requirements and knowledge of system functionality. The technical difference between the two is that refactoring, as stated above, doesn't change the functionality (or information content) of the system whereas rewriting does. Rewriting *is* reworking.

Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 164



Summary on Refactoring

- Refactoring is typically done in small steps. After each small step, you're left with a working system that's functionally unchanged. Practitioners typically interleave bug fixes and feature additions between these steps. So refactoring doesn't preclude changing functionality, it just says that it's a different activity from rearranging code.
- The key insight is that it's easier to rearrange the code correctly if you don't simultaneously try to change its functionality. The secondary insight is that it's easier to change functionality when you have clean (refactored) code.

Brazil 2003 -- Copyright 2003 by Joseph W. Yoder Day 1 - 165

Papers and Web Sites

- Bill Opdyke's Thesis
- Don's Thesis
  - http://st-www.cs.uiuc.edu/~droberts/thesis.pdf
- Refactoring Browser
  - **http://st-www.cs.uiuc.edu/~brant/Refactory/RefactoringBrowser.html**
- Evolving Frameworks
  - http://st-www.cs.uiuc.edu/~droberts/evolve.html
- Extreme Programming
  - http://www.c2.com/cgi-bin/wiki?ExtremeProgramming

More Web Sites

- Wiki wiki web
  - http://c2.com/cgi/wiki?WikiPagesAboutRefactoring
- Refactoring Swiki
  - http://brain.cs.uiuc.edu:8080/RefactoringBrowser
- The Refactory, Inc.
  - http://www.refactory.com
- Martin Fowler's Refactoring Pages
  - http://www.refactoring.com/

That's All for Day 1