Refactoring

Joseph W. Yoder
The Refactory, Inc.

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

The Refactory Principals

John Brant Don Roberts
Brian Foote Joe Yoder
Ralph Johnson

Refactory Affiliates

Joseph Bergin Fred Grossman
Bill Opdyke Rebecca Wirfs-Brock

Short Instructor Bio

Joseph Yoder (Founder and Senior Architect, The Refactory; Hillside Board President; ACM Member)

pattern enthusiast, an author of Big Ball of Mud;
programs adaptive software,
runs a development company,
consults top companies on software needs,
amateur photographer,
motorcycle enthusiast,
enjoys dancing samba!!!

Slide - 3

Evolved from The UIUC SAG

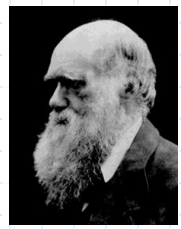
Throughout the 90's we were studying objects, frameworks, components, reusability, patterns, "good" architecture.

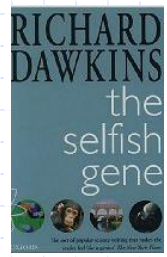
However, in our SAG group we often noticed that although we talk a good game, many successful systems do not have a good internal structure at all.

Slide - 4

Selfish Class

Brian and I had published a paper called Selfish Class which takes a *code's-eye view of software reuse and evolution.*





In contrast, our BBoM paper noted that in reality, a lot of code was hard to (re)-use.

Escape from the Spaghetti Code Jungle (Big Balls of Mud)

Brian Foote

&

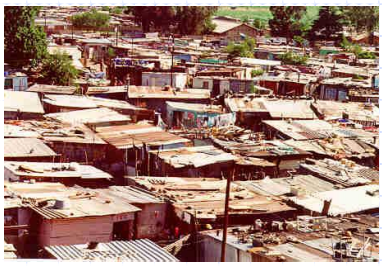
Joseph Yoder

PLoP D4 Book
Addison Wesley

Big Ball of Mud

Alias: Shantytown, Spaghetti Code

A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle.



The de-facto standard software architecture. Why is the gap between what we **preach** and what we **practice** so large?

We preach we want to build high quality systems but why are BBoMs so prevalent?

Slide - 7

Where does Mud Come From

Software Tectonics

Reconstruction

- Major Upheaval
- Throw it away

Incremental Change

- Evolution
- Piecemeal Growth

Throwaway Code

Legacy Mush

Urban Sprawl

Slash and Burn Tactics

Merciless Deadlines

Sheer Neglect

Slide - 8

Throwaway Code

- Sometimes this is the *right* approach
- There is the danger that such code will take on a life of its own

Slide - 9

Piecemeal Growth

- Mir was designed to accommodate maintenance and growth

- **Core** *1986*
- **Kvant 1** *1987*
- **Kvant 2** *1989*
- **Kristall** *1990*
- **Spekter** *1995*
- **Docking** *1995*
- **Priroda** *1996*

Slide - 10

Sweep It Under the Rug

- You may not know how to get rid of a problem, but at least you can cordon it off... (maybe then...)
  - A kind of Refactoring!!!

Agile Principles & Refactoring

- Scrum, TDD, Refactoring, Regular Feedback, Testing, More Eyes, ...
- Good People! Face-To-Face conversation.
- Continuous attention to technical excellence!
- Motivated individuals with the environment and support they need. Retrospectives!
- Allow Requirements to Change! Encourage Software Evolution as needed!

Agile Encourages Changes

- Very Little Upfront Design!
- Piecemeal Growth! Small Iterations!
- Late changes to the requirements of the system!
- Continuously Evolving the Architecture!

Adapting to Changes requires the code to change and Refactoring supports changes to the code in a "safe" way.

Agile & Refactoring

- A key part of Agile is to allow things to change and adapt as dictated by business needs!
- To support these changes, Refactoring is encouraged by most Agile practitioners.
- Agile has helped Refactoring be accepted into the mainstream development process (even sometimes encouraged).

Agile Refactoring & Testing

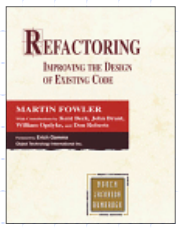
- Many Agile practices highly encourage Testing as one of their core practices.
- Processes like XP support developers writing many unit tests (XUnit).
- Test Driven Development (TDD) is usually considered a key principle of Agile.
- **Testing** was a key principle of **Refactoring** before there was Agile.

Ground Breaking Work

- Before there was Agile…
- Started at the University Of Illinois
- Bill Opdyke & Don Roberts PHD Thesis
- Refactoring Browser by John Brant and Don Roberts (first commercial tool)

Refactoring Definition

Refactoring is the process of **changing** a **software system** in such a way that it does not alter the external behavior of the code, yet **improves** its **internal structure**…Martin Fowler, **Refactoring** *Improving the Design of Existing Code; Addison Wesley , 1999*



Slide - 17

Refactoring

Refactoring is the engine of Consolidation

Refactorings are **program transformations** that **preserve** program semantics, while **improving** structure, lot's of small steps

Refactoring was originally done by hand, but standard **tools** have **emerged**

Languages differ significantly in the degree to which they support refactoring

Slide - 18

Refactorings

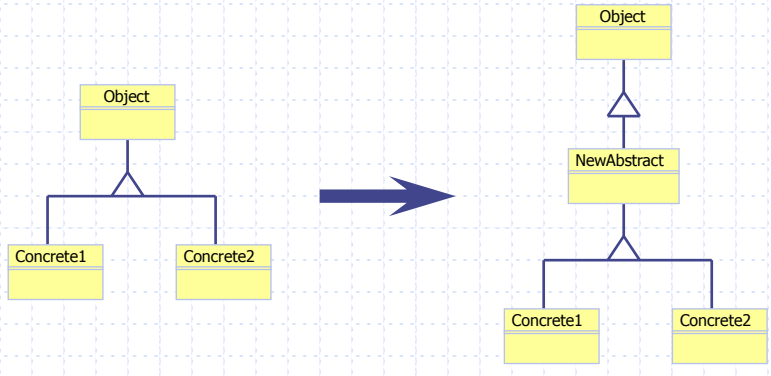
Behavior Preserving Program Transformations

- Rename Instance Variable
- Promote Method to Superclass
- Move Method to Component

Always done for a reason!!!

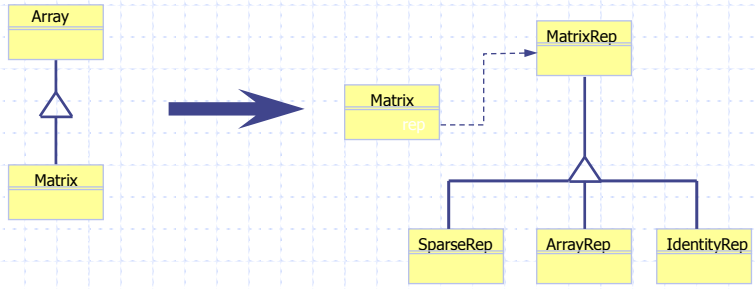
A Simple Refactoring

Create Empty Class



Borrowed from Don Roberts, The Refactory, Inc.

A Complex Refactoring



Borrowed from Don Roberts, The Refactory, Inc.

Why Refactor?

Need to fix a bug
Add new enhancements

Code is brittle and hard to maintain
There is much code entanglement

Do not refactor to just refactor!!!

Refactoring Code Smells

A *code smell* is a **hint** that something has **gone wrong** somewhere in your code. Use the smell to **track** down the **problem** KentBeck ...

Bad Smells in Code was an essay by KentBeck and MartinFowler, published as Chapter 3 of Refactoring Improving The Design Of ExistingCode.

----Wards Wiki

Code Smells (1)

Code Smell	Description
Duplicate Code	Name tells all….you have duplicate code that you want to remove.
Long Method	If it is too long, break it up into smaller pieces.
Large Class	Big bloated class doing many different thing….consider breaking it up.
Long Parameter List	Many parameters being passed around…Use Replace Param with Method.
Divergent Change	Each change requires changes to many methods…might need a new object.
Shotgun Surgery	A change requires many changes to many classes…Move Method or Field.
Feature Envy	Method in one class is always interested in another class…Move Method.
Data Clumps	Sets of data is always used together….put them together in same object.
Primitive Obsession	Too many primitive data types….use more real objects like Address
Switch Statements	Many case statements…replace with objects or move methods.

Code Smells (2)

Code Smell	Description
Parallel Inheritance	Two class hierarchies are tightly coupled….remove delegation.
Lazy Class	Class is not really doing much work…move methods and fields.
Speculative Generality	Trying to generalize where you are not ready to generalize.
Temporary Field	Have a field that is only sometimes set…use a subclass for that case.
Message Chains	Chaining many messages for a result…use hide delegation to remove.
Middle Man	A middle class does little work between classes…..remove the class.
Inappropriate Intimacy	Two classes are tightly coupled and working too closely…separate changes.
Incomplete Lib Classes	Not enough function in library…use add foreign method to add to classes.
Data Class	Dumb data holder classes so either add methods working on data or move fields to where the data is being used.
Refused Bequest	Inheriting too much behavior you don't need…push down where it belongs.
Comments	Methods with many comments…similar to large methods….extract method.

Duplicate Code

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

Fixing Code Duplication

- Move identical methods up to superclass
- Move methods into common components
- Break up Large Methods

Feature Envy

- Method is accessing values and doing work for other classes
- Method might be in wrong place
- Move the method to the class where it is usually doing the work for

Switch Statements

- Many switch statements or nested conditionals throughout methods
- Rather than switching use method names to do the cases (double dispatch)
- Use polymorphism or overriding of hook methods (new cases do not change existing code)

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

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.
- Need both Unit Testing and Integrated Tests for making sure nothing breaks.

Deciding Whether A Refactoring is Safe

- Refactoring should **not 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 needs to perform the **same** after a refactoring as before."
- Satisfying timing constraints.

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, Nunit, …)
- Take small steps, testing between each
- Java and C# tools are pretty good

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!!!!**

Catalogue of Refactorings

- Composing Method
- Moving Features
- Organize Data
- Simplifying Conditionals
- Simpler Method Calls
- Generalization

From Fowlers Book

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);
    ……a lot more details….

}
```


Extract Method (2)

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

void printDetails (double amount) {
   System.out.println("name:" + _name);
   System.out.println("amount:" + amount);
 ……a lot more details….
}
```

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.

Compile, Build and Test!!!

Replace the extracted code in the source method.

Inline Method

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

```
int getRating() {
    return (_numberOfLateDeliveries > 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.

Slide - 43

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;}
```


Slide - 44

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

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

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

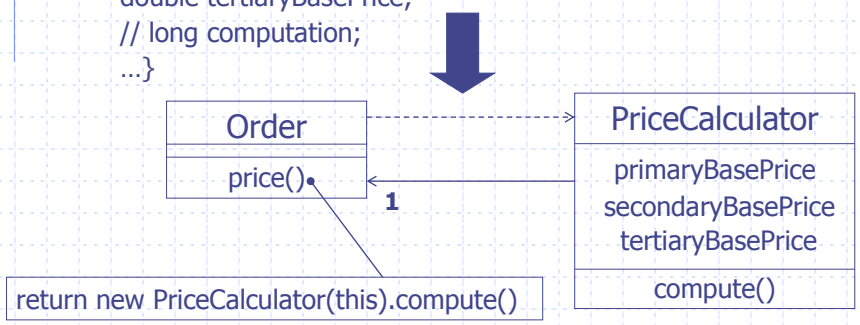
Could evolve to apply Replace Temp with Query Refactoring

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.

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.

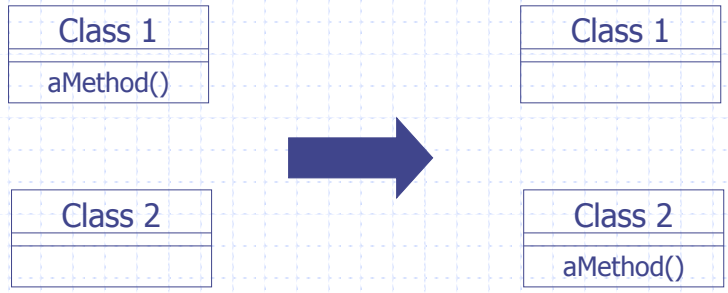
Slide - 49

Moving Features Catalogue

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

Slide - 50

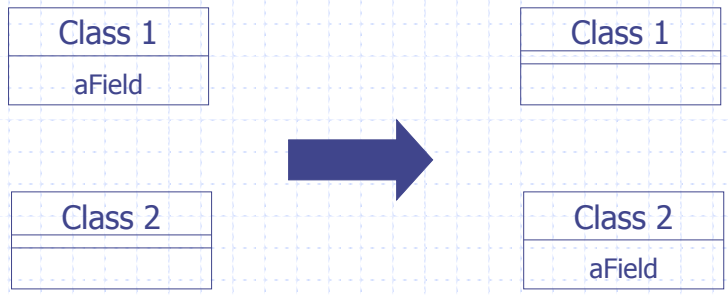
Move Method



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.

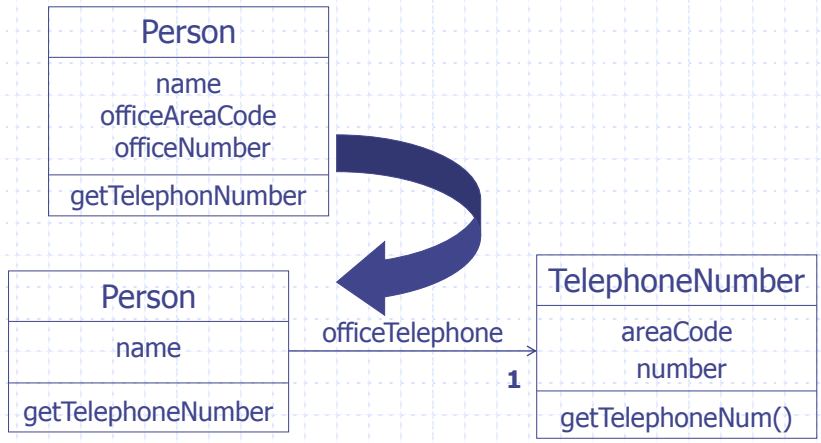
Move Field



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



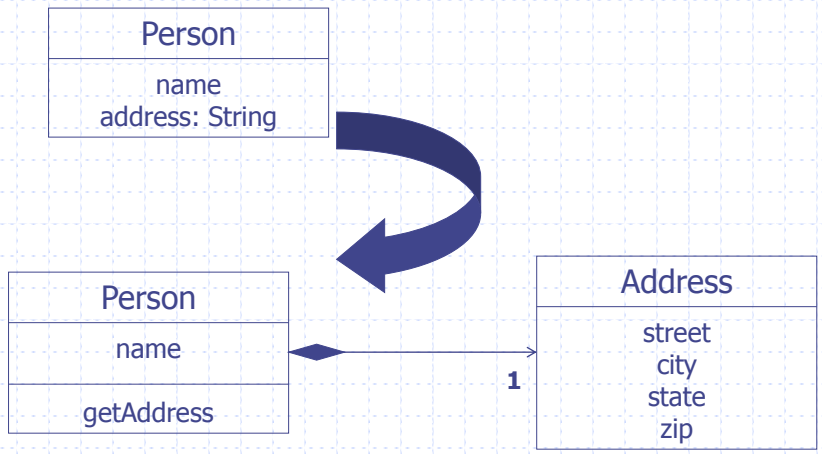
Extract Class Mechanics

- *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.

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, ……..

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.

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.

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;}}
    return result;}
```

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

Replace Nested Mechanics

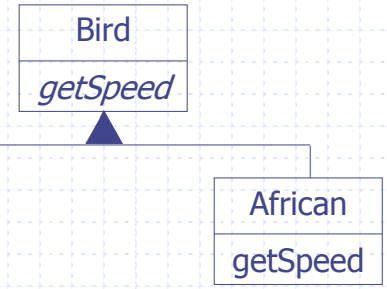
- For each check put in the guard clause.
- Compile and Test after each check is replace with a guard clause.
- Might consider consolidate conditional if the guards use the same result.

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.

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.

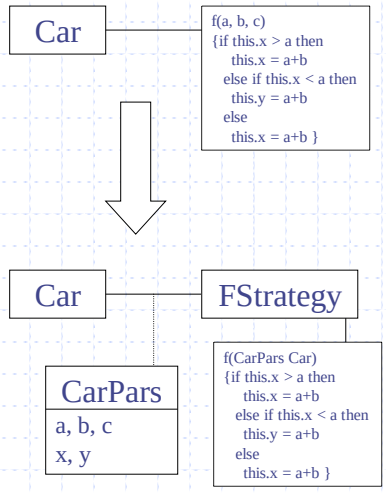
Moving function f from class X to class B

```
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;
    } ...
```


Slide - 75

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.



Slide - 76

Introduce Null Object

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

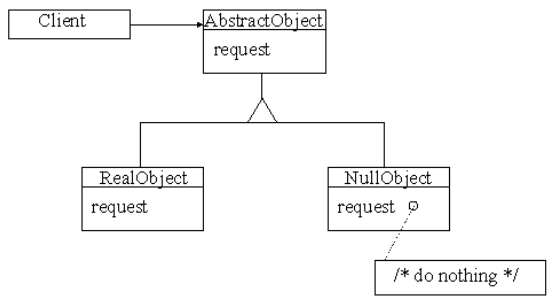
```
if (address == null) Console::WriteLine("")
else Console::WriteLine(this->address);
if (phone == null) Console::WriteLine("")
else Console::WriteLine(this->phone);
```


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:

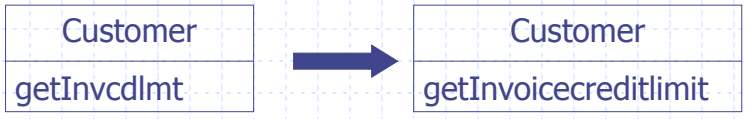


Simpler Method Calls Catalogue

- 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

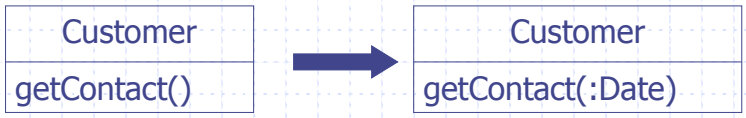
Rename Method

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



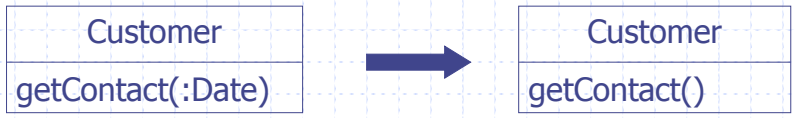
Add Parameter

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



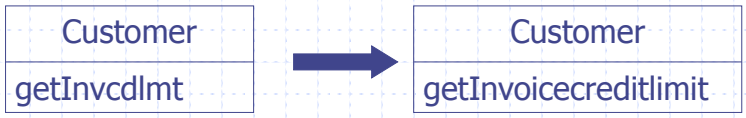
Remove Parameter

- 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!*

Refactoring Summary

- We have seen the mechanics for some of the Refactorings.
- When you find smelly code, you often apply Refactorings to clean your code.
- Refactorings do often apply Design Patterns.
- Testing is a Key Principle of Good Refactoring!

Patterns and Refactoring

Understanding Design Patterns and good design is a prerequisite to good OO refactorings!

- **Refactoring is the Redesign or re-architecture of a system!**
- **Apply well-known design principles such as "*design patterns*" to separate what changes from what doesn't!**

Refactoring and Design Patterns (1)

What varies	Design Pattern
Algorithms	Strategy, Visitor
Actions	Command
Implementations	Bridge
Response to change	Observer

Refactoring and Design Patterns (2)

What varies	Design Pattern
Interactions between objects	Mediator
Object being created	Factory Method, Abstract Factory, Prototype
Structure being created	Builder
Traversal Algorithm	Iterator
Object interfaces	Adapter
Object behavior	Decorator, State

C++ Refactoring

- "C++ syntax parsing can hard" pointers, headers, memory management, casting, ...
- "C++ has this nasty preprocessing" which can make refactoring harder
- "C++ has templates" So, declaration constructs can be a more complex
- There are some tools that can help

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.

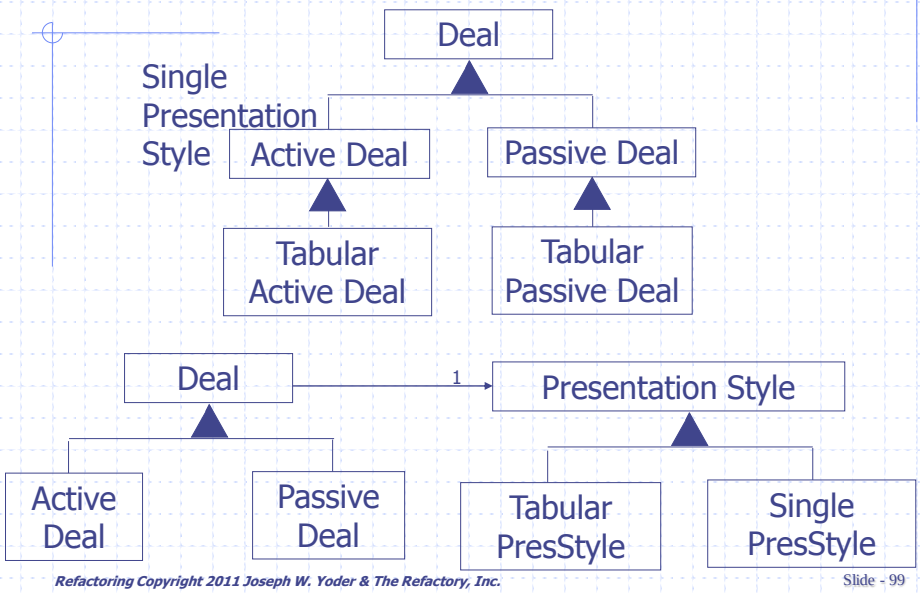
Four Big Refactorings

- 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



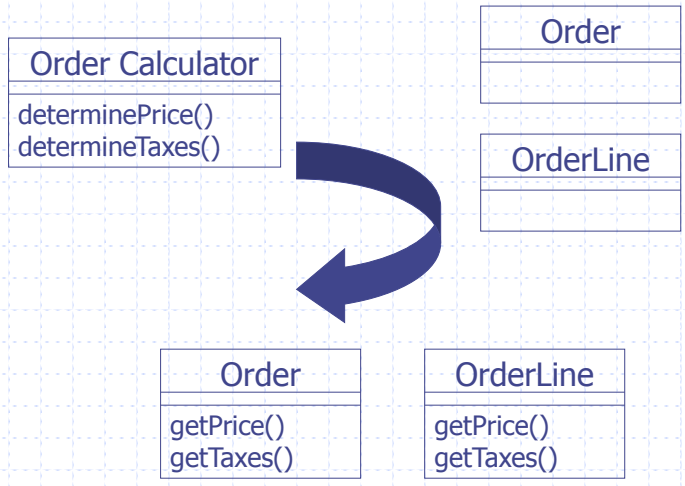
Tease Apart Mechanics

- 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 a lot of 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.

Separate Domain from Presentation

- You have GUI classes that contain domain logic.
- *Separate the domain logic into separate domain classes.*

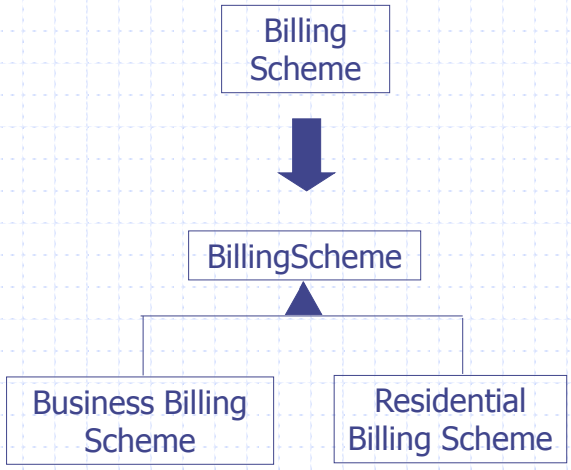
Separate Domain Mechanics

- Create a domain class for each window.
- If you have a grid, create a class to represent rows on the grid.
- Examine the data on the window. If it is used for the domain logic, use *Move Method* to move it into the domain object.
- Use Extract Method to separate presentation from domain logic.
- Apply more refactorings to domain object once the separation is complete.

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.

Despite The 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.

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….this is part of Agile!!!

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):
  - Extend – refactor
  - Refactor – extend
  - Debug – refactor
  - Refactor – debug
  - Refactoring to understand.

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 (isn't this core to Agile???)
  - 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.

Papers and Web Sites

- Bill Opdyke's Thesis
- Don Robert's Thesis
  - http://st-www.cs.illinois.edu/users/droberts/thesis.pdf
- Refactoring Browser
  - http://st-www.cs.illinois.edu/users/brant/Refactory
- Evolving Frameworks
  - http://st-www.cs.uiuc.edu/users/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
- The Refactory, Inc.
  - http://www.refactory.com
- Martin Fowler's Refactoring Pages
  - http://www.refactoring.com/
- Adaptive Object Models
  - http://www.adaptiveobjectmodel.com/

That's All