Refactoring

What to refactor

Refactor to what

How to conduct the refactoring

This website is also very informative
https://refactoring.com/catalog/
Definitions

• Changing/improving the code structure w/o changing the program semantics
Key principles in refactoring

• Where to refactor
  • Code smell

• Refactor to what
  • Is it worthwhile to refactor?

• How to refactor?
  • What to change? (don’t miss anything!)
  • What are the steps? (keep each step as small as possible!)
  • Testing after every step of change

Use automated refactoring tool whenever you can
Example 0

• What if the name of a method is not clear?

• Why should we make this change?

• What should we change?
  • Document
  • Function declaration
  • Reference to the function
  • Test code
  • Sub, super classes
What to change?

• Method declaration
• Caller
• **Super classes, sub classes**
• Test cases
• Documentation
How to change?
How to change?

• Check if the method is inherited from super class
  • ...
• Create a new method, declare it, copy the code
• Let the old method calls the new method
  • If the old method is used in many places
• Replace the old method every place it is called
• Remove the old method
Example 1 how to add a parameter?
Steps to perform

• Check superclasses and subclasses
• Make copy of old method, add parameter
• Change body of old method so that it calls new one
• Find all references to the old method and change them to refer to the new
• Test should run after each change
• Remove old method
Example 2

- What if the parameter list is too long?
When is this change worthwhile?
When is this change worthwhile?

• Many methods have same parameters
• The parameter list is very long
What needs to be done?
Refactoring steps
Introduce Parameter Object (1)

• Make a new class for the group of parameters
• Use Add Parameter for the new class
  • Use a new object for the parameter in all the callers
• For each of the original parameters: ...
Introduction Parameter Object (2)

• For each of the original parameters:
  • Modify caller to store parameter in the new object and omit parameter from call
  • Modify method body to omit original parameter and to use the value stored in the new parameter
  • If method body calls another method with parameter object, use existing parameter object instead of making a new one
class Account ...

double getFlowBetween(Date start, Date end) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
        Entry each = (Entry) e.nextElement();
        Date date = each.getDate();
        if (date.equals(start) || date.equals(end) ||
            (date.after(start) && date.before(end))) {
            result += each.getValue();
        }
    }
    return result;
}
class DateRange {
    DateRange (Date start, Date end) {
        _start = start;
        _end = end;
    }
    Date getStart() {
        return _start;
    }
    Date getEnd() {
        return _end;
    }
    private final Date _start;
    private final Date _end;
}
class Account ...

double getFlowBetween(Date start, Date end, DateRange range) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
        Entry each = (Entry) e.nextElement();
        Date date = each.getDate();
        if (date.equals(start) || date.equals(end) ||
            (date.after(start) && date.before(end))) {
            result += each.getValue();
        }
    }
    return result;
}
Changing callers (1)

double flow = anAccount.getFlowBetween(startDate, endDate);

double flow = anAccount.getFlowBetween(startDate, endDate, new DateRange(null, null))
Changing callers (2)

double flow = anAccount.getFlowBetween(startDate, endDate, new DateRange(null, null))

double flow = anAccount.getFlowBetween(endDate, new DateRange(startDate, null))
class Account ...

double getFlowBetween(Date end, DateRange range) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
        Entry each = (Entry) e.nextElement();
        Date date = each.getDate();
        if (date.equals(range.getStart()) || date.equals(end) ||
            (date.after(range.getStart()) && date.before(end))) {
            result += each.getValue();
        }
    }
    return result;
}
class Account ...

double getFlowBetween(DateRange range) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
        Entry each = (Entry) e.nextElement();
        Date date = each.getDate();
        if (date.equals(range.getStart()) ||
            date.equals(range.getEnd()) ||
            (date.after(range.getStart()) &&
             date.before(range.getEnd()))) {
            result += each.getValue();
        }
    }
    return result;
}
Changing callers (3)

double flow = anAccount.getFlowBetween(endDate, new DateRange(startDate, null))

double flow = anAccount.getFlowBetween(new DateRange(startDate, endDate))
Introduce Parameter Object

After introducing a parameter object, look to see if code should be moved to its methods

??
Introduce Parameter Object

After introducing a parameter object, look to see if code should be moved to its methods

class DateRange ...

    boolean includes (Date arg) {
        return (arg.equals(_start) || arg.equals(_end) || (arg.after(_start) && arg.before(_end)));
    }


class Account ...

double getFlowBetween(DateRange range) {
    double result = 0;
    Enumeration e = _entries.elements();
    while (e.hasMoreElements()) {
        Entry each = (Entry) e.nextElement();
        if (range.includes(each.getDate())) {
            result += each.getValue();
        }
    }
    return result;
}
Lessons

• Refactorings should be small
  • Test cases
  • Version control

• Check after each step to make sure you didn’t make a mistake

• One refactoring leads to another

• Major change requires many refactorings
More OO refactoring
Example 4 pull up method

• What if there is code duplication across two classes?

• Why is it worthwhile?
• What to do?
• What are the steps?
• The example on the next page requires a series of code refactoring that include pull up methods and will help remove code redundancy
Class Person{
    private:
    string First;
    string Last;
    string Address;
}

Class Female: public Person{
    public:
    void printName() {
        cout << "Ms. " << First << " " << Last;
    }
    void printAddress() {
        cout << "Ms. " << First << " " 
                << Last << endl << Address;
    }
}

Class Male: public Person{
    public:
    void printName() {
        cout << "Mr. " << First << " " << Last;
    }
    void printAddress() {
        cout << "Mr. " << First << " " 
                << Last << endl << Address;
    }
}
Class Person{
  private:
    string First;
    string Last;
    string Address;
  
  Public:
    void printName();
    void printAddress();
}

Class Person::PrintAddress()
{
  printName();
  cout << endl << Address;
}

Class Female: public Person{
  
  public:
    void printName() {
      cout << "Ms. " << First << " " << Last;
    }
}

Class Male: public Person{
  
  public:
    void printName() {
      cout << "Mr. " << First << " " << Last;
    }
}
Example 5 push down methods

• When does that happen?
  • When the super class’ default implementation does not work for most of the sub-classes

• What to do?
  • Remove the default implementation, turn that into a virtual method
  • Make sure that every sub-class has its implementation of that method

• This refactoring common comes together with “extract sub-class”
Example 6: extract sub-class

• Extract sub-class
  • When to use what?
    • We have a class A
    • Some of its properties are used under context 1, some other are used under context 2
    • Its method implementation contains if/else, switch/case depending on context 1 or 2

• What to do?
  • Create sub-classes for class A that represent different contexts
  • Some properties that are only used for one context can be pushed down to sub-classes
  • Some methods that are implemented using if/else can be pushed down to sub-classes with polymorphism there
• The example on the next slide requires extract sub-class refactoring

• The JobItem class has two usage contexts:
  • 1. The job item is an item, the cost is about material cost
  • 2. the job item is about labor, the cost is about labor fee
  • The “_employee” property of the JobItem has no meaning when it is a non-labor JobItem
  • The “getUnitPrice” method contains if/else depending on the context

• Refactoring for this example
  • Create a LaborJobItem sub-class
  • Move _employee property down to that sub-class
  • Replace if/else in getUnitPrice with polymorphism of getUnitPrice
class JobItem {
    public JobItem (int unitPrice, int quantity, boolean isLabor, Employee employee) {
        _unitPrice = unitPrice;
        _quantity = quantity;
        _isLabor = isLabor;
        _employee = employee;
    }

    public int getTotalPrice() {
        return getUnitPrice() * _quantity;
    }

    public int getUnitPrice() {
        return (_isLabor) ?
            _employee.getRate():
            _unitPrice;
    }
}

class Employee {
    public Employee (int rate) {
        _rate = rate;
    }
}

public int getQuantity() {
    return _quantity;
}

public Employee getEmployee() {
    return _employee;
}
private int _unitPrice;
private int _quantity;
private Employee _employee;
private boolean _isLabor;

private int _rate;

Example 7: extract super-class

• When to do?
• What to do?
Example 7: extract super-class

• When to do?
  • Two classes share many properties and operations

• What to do?
  • Create a super class
  • Move common properties and operations up
    • Leave unique properties and operations in each sub-class
  • Turn some if/else, switch/case into simple method call (polymorphism) ...
• The code on the next two pages smell
• Desired refactoring:
  • create a super class for Employee and Department
class Employee...
public Employee (String name, String id, int annualCost) {
    _name = name;
    _id = id;
    _annualCost = annualCost;
}
public int getAnnualCost() {
    return _annualCost;
}
public String getId() {
    return _id;
}
public String getName() {
    return _name;
}
private String _name;
private int _annualCost;
private String _id;
public class Department...
public Department (String name) {
    _name = name;
}

public int getTotalAnnualCost() {
    Enumeration e = getStaff();
    int result = 0;
    while (e.hasMoreElements()) {
        Employee each = (Employee) e.nextElement();
        result += each.getAnnualCost();
    }
    return result;
}

public int getHeadCount() {
    return _staff.size();
}

public Enumeration getStaff() {
    return _staff.elements();
}

public void addStaff(Employee arg) {
    _staff.addElement(arg);
}

public String getName() {
    return _name;
}

private String _name;
private Vector _staff = new Vector();
Be careful ...

• Separate changing behavior from refactoring
  • Changing behavior requires new tests
  • Refactoring must pass all tests

• Only refactor when you need to
  • Before you change behavior
  • After you change behavior
  • To understand
Some other refactorings

• Composing methods
• Extract method
• Inline method
• Inline temporary variable
• Introduce explaining variable
• Split temporary variable
• Replace method with method object
• ...

We didn’t talk about these in lecture. These won’t be in exams/quizzes.
Automated refactoring support

• Deciding where to refactor
  • Tools for measuring cohesion, size, etc.
  • Tools for measuring code duplication/cloning

• Performing the change
  • Refactoring Browser for Smalltalk, first
  • Over a dozen of tools for Java
  • Eclipse