Refactoring

Refactoring Techniques

Code Quality is Important!

```
public void add(Object element) {
    if(!readOnly) {
        int newSize = size + 1;
        if(newSize > elements.length) {
          Object[] newElements = new Object[elements.length+10];
            for(int i=0; i<size; i++)
                newElements[i] = elements[i];
            elements = newElements;
        }
        elements[size++] = element;
    }
}
```

```
public void add(Object element) {
    if(readOnly)
        return;
    if(atCapacity())
        grow();
    addElement(element);
}
```

Refactoring is...

"A disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior."

Characterising Refactoring...

"Each transformation does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong".

The "Two Hat" Metaphor

- Programmer always wearing one of 2 hats:
  - The developer hat
  - The refactoring hat
- If the task can be made easier if the code is structured differently, Programmer swaps hats and refactors for a while.
- Then he swaps hats again, and adds the functionality.

What is not Refactoring

- Adding new functionality is not refactoring
- Optimization is not refactoring
- Changing code that does not compile is not refactoring

Reverse Conditional

- *You have a conditional that would be easier to understand if you reversed its sense.*
- Reverse the sense of the conditional and reorder the conditional's clauses.

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

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

Rename Method

- *A method's name does not reveal its purpose*
- Change the name of the method

```
Class Customer {
//…
  public double getinvcdtlmt() {
    return creditLimit;
  }
}
```

```
Class Customer {
//…
  public double getInvoiceableCreditLimit() {
    return creditLimit;
  }
}
```

Extract Method

- *You have a code fragment that can be grouped together.*
- **Turn the fragment into a method whose name explains the purpose of the method.**

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

    //print details
    System.out.println ("name: " + name);
    System.out.println ("amount " + getOutstanding());
}
```

```
void printOwing() {
    printBanner();
    printDetails(getOutstanding());
}
void printDetails (double outstanding) {
    System.out.println ("name: " +  name);
    System.out.println ("amount " + outstanding);
}
```

Inline Method (Opposite of extract method)

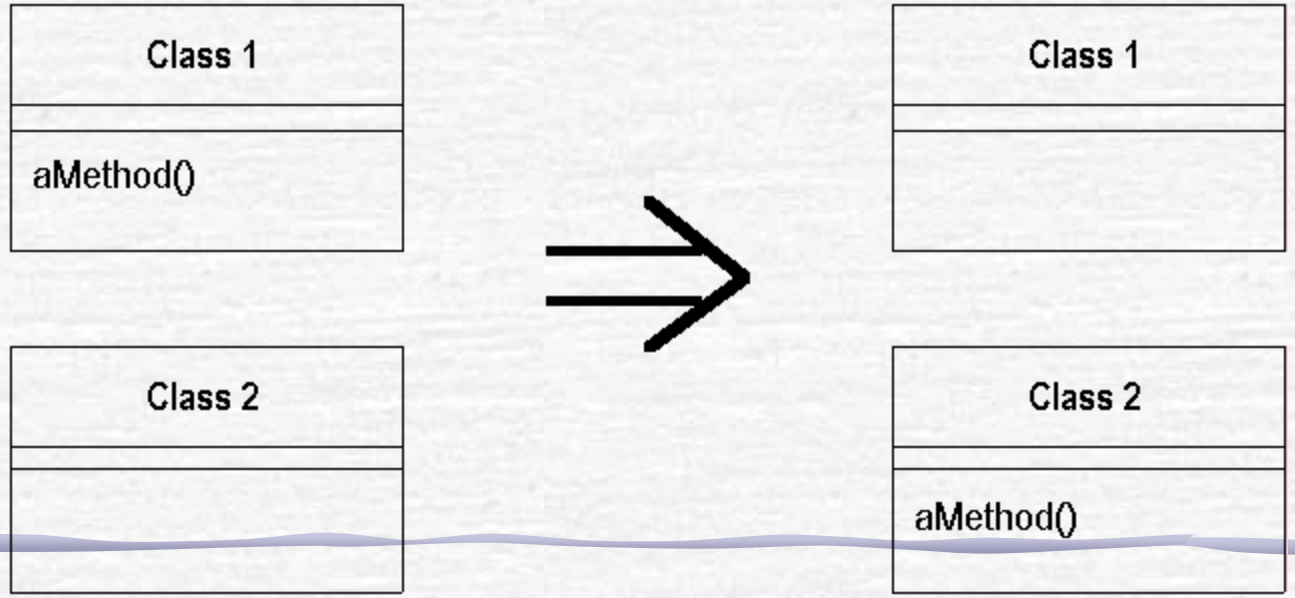
- *A method's body is just as clear as its name.*
- Put the method's body into the body of its callers and remove the method.

```
int getRating() {
  return (moreThanFiveLateDeliveries()) ? 2 : 1;
}

boolean moreThanFiveLateDeliveries() {
  return numberOfLateDeliveries > 5;
}
```

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

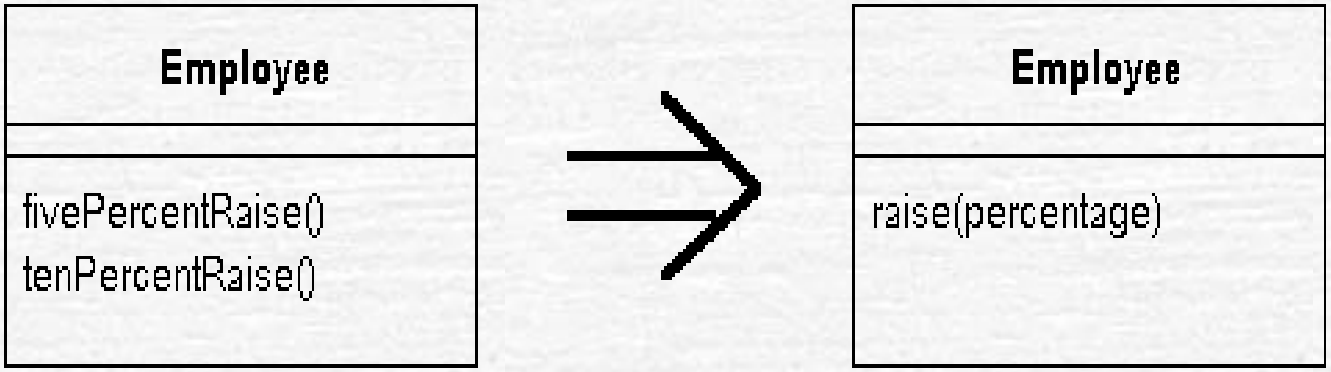
Move Method

- *A method is used by more features of another class than the class on which it is defined.*
- Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.



Parameterize Method

- *Several methods do similar things but with different values contained in the method body.*
- **Create one method that uses a parameter for the different values.**



Encapsulate Field

- *There is a public field.*
- **Make it private and provide accessors.**

```
public String name;
```

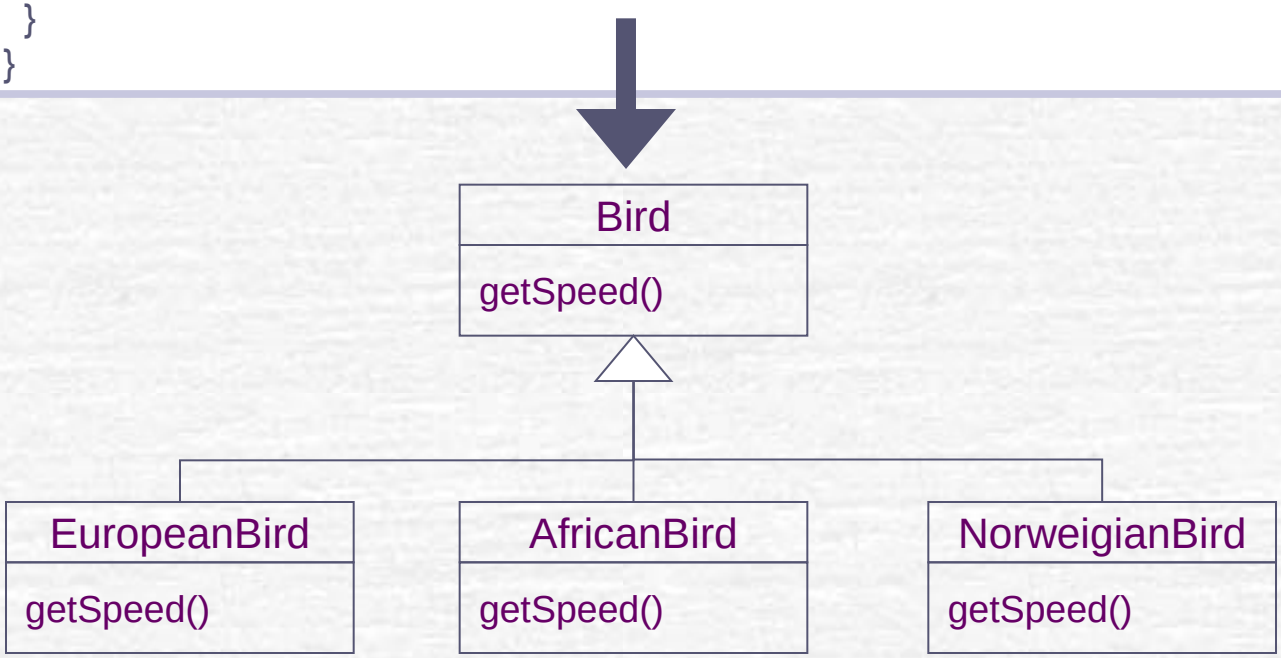
```
private String name;

public String getName() { return name; }

public void setName(String name) {
    this.name = name;
}
```

Replace Conditional With Polymorphism

- *You have a conditional that chooses different behavior depending on the type of an object.*

1. **Move each leg of the conditional to an overriding method in a subclass.**
2. **Make the parent method abstract.**

```
double getSpeed() {
  switch (_type) {
    case EUROPEAN:
      return getBaseSpeed();
    case AFRICAN:
      return getBaseSpeed() - getLoadFactor() *  _numberOfCoconuts;
    case NORWEIGIAN_BLUE:
      return (_isNailed) ? 0 : getBaseSpeed(_voltage);
  }
}
```



Replace Temp with Query

- Temporary variables can lead to poor code quality
- Create a method to compute or access the temporary 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; }
```

How Refactorings are Performed

- Either manually or automatically.
- It is always done in **small** steps!

- Larger refactorings are sequences of smaller ones

Manual Refactoring

- Manual refactoring steps should always be small, because:
  - They are safer this way, because the steps are simpler
  - It is easier to backtrack
  - Pay attention to the mechanics:
    - Mechanics should stress safety

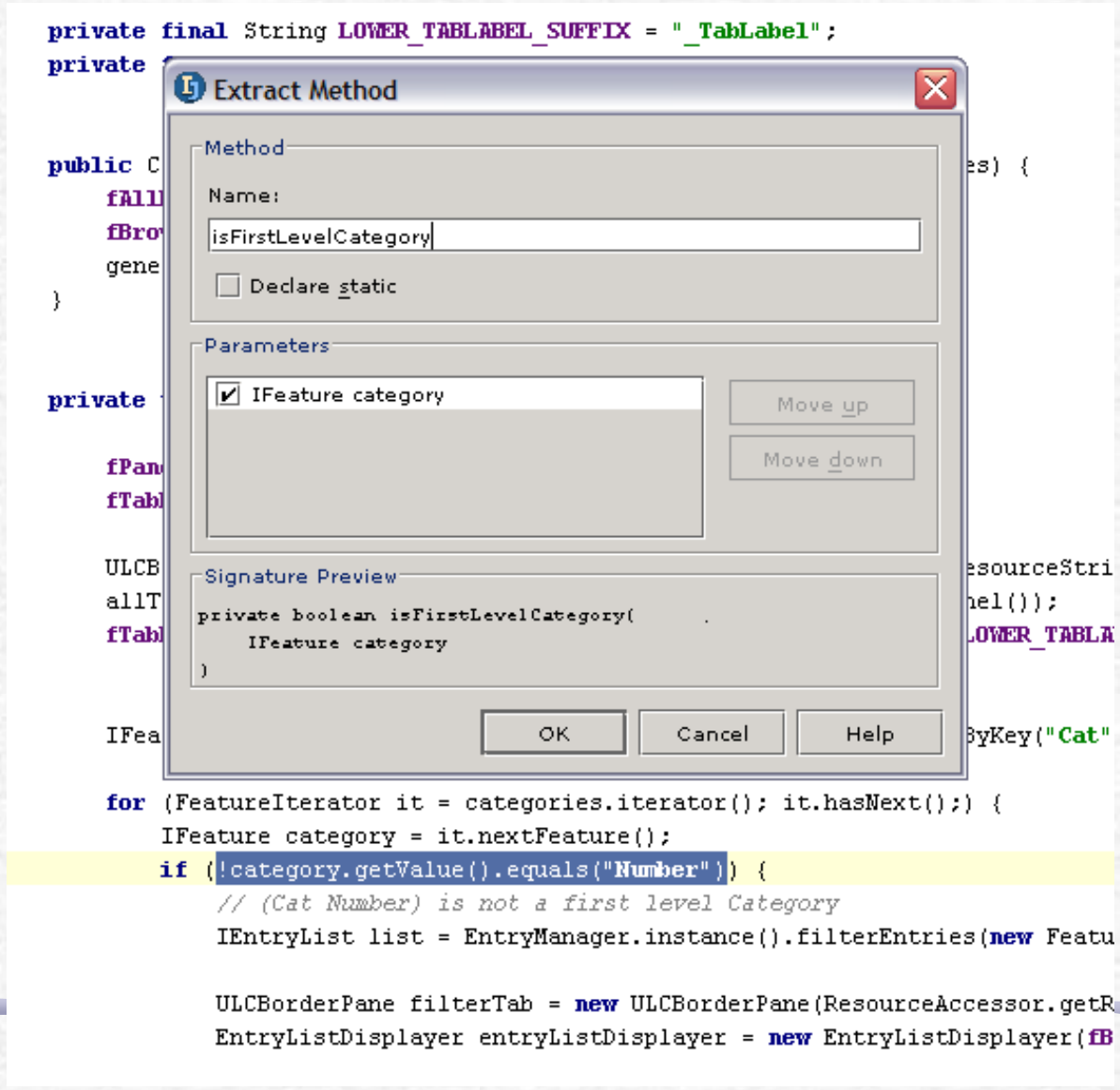
How Refactorings are Performed

- When automatic support is available, it should be preferred, but ...
- ... only if the tool is really safe.
- Example: *Rename Method*
  - Does it check for another method with the same name?
  - Does it account for overloading?
  - Does it account for overriding?

Encapsulate Field – Mechanics

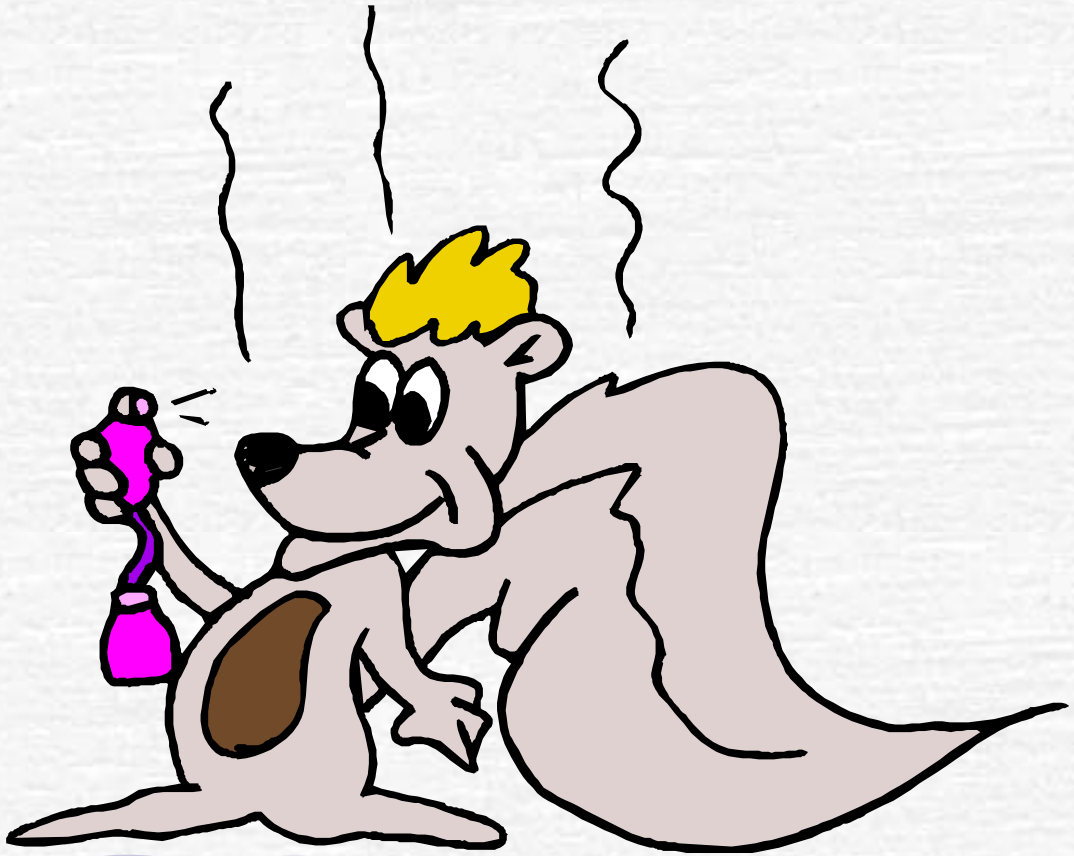
- Create getting & setting methods for the field
- Find all clients outside the class that reference the field.
- If the client uses the value, replace the reference with a call to the getting method.
  If the client changes the value, replace the reference with a call to the setting method.
- Compile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

Automatic Support



When to Refactor

We should refactor when the code stinks!



Refactoring and Code Smells

- Refactorings remove *Bad Smells in the Code* i.e., potential problems or flaws
- Some will be strong, some will be subtle
- Some smells are obvious, some aren't
- Some smells mask other problems
- Some smells go away unexpectedly when we fix something else

Replace Smelly Code

- ***Bad Smells*** include:
  - Duplicated code
  - Switch statements
  - Long methods
  - Large classes
  - Data classes (only getters and setters in the API)
  - Long parameter lists
  - Use of primitives rather than objects
  - Temporary variables and fields

When you write smelly code you are hacking...

Unit Tests

- Essential prerequisite for refactoring
- Solid tests (i.e. good unit test coverage)
  - Tests warn programmers of problems if they unknowningly break other parts of the application
  - Tests give an immediate/quick analysis of the effects of a change
- *Tests give Courage*