SOFTWARE MAINTENANCE & EVOLUTION

LINGI2252 – PROF. KIM MENS
LINGI2252 – PROF. KIM MENS

CODE REFACTORING

Refactoring: Improving the Design of Existing Code

One of the best references on software refactoring, with illustrative examples in Java:

*Refactoring: Improving the Design of Existing Code.*

See also [www.refactoring.com](http://www.refactoring.com)

Overview of this presentation

A. Refactoring basics

B. Categories of refactoring

C. Words of warning
A. REFACTORING BASICS

What is refactoring?

A **refactoring** is a software transformation that

**preserves the external behaviour** of the software;

**improves the internal structure** of the software.

It is a disciplined way to clean up code that minimises the chances of introducing bugs.
Definition of Refactoring [Fowler 2000]

[noun] “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour”

[verb] “to restructure software by applying a series of refactorings without changing its observable behaviour”

typically with the purpose of making the software easier to understand and modify
Why should you refactor?

**THE LIFE OF A SOFTWARE ENGINEER.**

Clean slate. Solid foundations. This time I will build things the right way.

**MUCH LATER...**

Oh my. I’ve done it again, haven’t I?
Why should you refactor?

To improve the design of software

To counter code decay (software ageing)
refactoring helps code to remain in shape

To increase software comprehensibility

To find bugs and write more robust code

To increase productivity (program faster)

on a long term basis, not on a short term basis
Why should you refactor?

To reduce costs of software maintenance

To reduce testing
  automatic refactorings are guaranteed to be behaviour preserving

To prepare for / facilitate future customisations

To turn an OO application into a framework

To introduce design patterns in a behaviourally preserving way
When should you refactor?

Whenever you see the need for it

Do it all the time in little bursts

Not on a pre-set periodical basis

Apply the rule of three

1\text{st} time : implement from scratch

2\text{nd} time : implement something similar by code duplication

3\text{rd} time : do not implement similar things again, but refactor
When should you refactor?

Refactor when adding new features or functions

   Especially if feature is difficult to integrate with the existing code

Refactor during bug fixing

   If a bug is very hard to trace, refactor first to make the code more understandable, so that you can understand better where the bug is located

Refactor during code reviews
When should you refactor?

Refactoring also fits naturally in the *agile methods* philosophy.

Is needed to address the principle “Maintain simplicity”

Wherever possible, actively work to eliminate complexity from the system.

By refactoring the code.
What do you tell the manager?

When (s)he’s technically aware (s)he’ll understand why refactoring is important.

When (s)he’s interested in quality, (s)he’ll understand that refactoring will improve software quality.

When (s)he’s only interested in the schedule, don’t tell that you’re doing refactoring, just do it anyway.

In the end refactoring will make you more productive.
When shouldn’t you refactor?

When the existing code is such a mess that although you could refactor it, it would be easier to rewrite everything from scratch instead.

When you are too close to a deadline.

The productivity gain would appear after the deadline and thus be too late.

However, when you are not close to a deadline you should never put off refactoring because you don’t have the time.

Not having enough time usually is a sign that refactoring is needed.
B. CATEGORIES OF REFACTORINGS

### Categories of refactorings

<table>
<thead>
<tr>
<th>Small refactorings</th>
<th>Big refactorings</th>
</tr>
</thead>
<tbody>
<tr>
<td>(de)composing methods</td>
<td>Tease apart inheritance</td>
</tr>
<tr>
<td>moving features between objects</td>
<td>Extract hierarchy</td>
</tr>
<tr>
<td>organising data</td>
<td>Convert procedural design to objects</td>
</tr>
<tr>
<td>simplifying conditional expressions</td>
<td>Separate domain from presentation</td>
</tr>
<tr>
<td>dealing with generalisation</td>
<td></td>
</tr>
<tr>
<td>simplifying method calls</td>
<td></td>
</tr>
</tbody>
</table>
Small refactorings

(de)composing methods [9 refactorings]

moving features between objects [8 refactorings]

organising data [16 refactorings]

simplifying conditional expressions [8 refactorings]

dealing with generalisation [12 refactorings]

simplifying method calls [15 refactorings]
Small Refactorings: (de)composing methods

1. Extract Method
2. Inline Method
3. Inline Temp
4. Replace Temp With Query
5. Introduce Explaining Variable
6. Split Temporary Variable
7. Remove Assignments to Parameter
8. Replace Method With Method Object
9. Substitute Algorithm

Legend:
= we will zoom in on these
= home reading
(De)composing methods: 1. Extract Method

What? When you have a fragment of code that can be grouped together, turn it into a method with a name that explains the purpose of the method

Why? improves clarity, removes redundancy

Example:

```java
public void accept(Packet p) {
    if ((p.getAddressee() == this) &&
        (this.isASCII(p.getContents())))
        this.print(p);
    else
        super.accept(p); }
```

```java
public void accept(Packet p) {
    if this.isDestFor(p) this.print(p);
    else super.accept(p); }
```

```java
public boolean isDestFor(Packet p) {
    return ((p.getAddressee() == this) &&
            (this.isASCII(p.getContents()))); }
```

Beware of local variables!
(De)composing methods: 2. Inline Method

(OPPOSITE OF EXTRACT METHOD)

**What?** When a method’s body is just as clear as its name, put the method’s body into the body of its caller and remove the method.

**Why?** To remove too much indirection and delegation.

**Example:**

```java
int getRating() {
    return moreThanFiveLate Deliveries();
}

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

```java
int getRating() {
    return (_numberOfLateDeliveries > 5);
}
```
What? When you have a temp that is assigned once with a simple expression, and the temp is getting in the way of refactorings, replace all references to that temp with the expression.

Why? (Part of **Replace Temp with Query** refactoring)

Example:

double basePrice = anOrder.basePrice();
return (basePrice > 100)

return (anOrder. basePrice() > 100)
(De)composing methods: 4. Replace Temp with Query

What? When you use a temporary variable to hold the result of an expression, extract the expression into a method and replace all references to the temp with a method call.

Why? Cleaner code

Example:

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

```java
double basePrice()
{
    return _quantity * _itemPrice;
}
if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;
...```
(De)composing methods:
5. Introduce Explaining Variable

**What?** When you have a complex expression, put the result of the (parts of the) expression in a temporary variable with a name that explains the purpose.

**Why?** Breaking down complex expressions for clarity.

**Example:**

```java
if ((platform.toUpperCase().indexOf("MAC") > -1) &&
    (browser.toUpperCase().indexOf("IE") > -1) &&
    wasInitialized() && resize > 0 )
{
    //ACTION
}
```

```java
final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResized = resize > 0;
```

```java
if (isMacOs && isIEBrowser && wasInitialized() && wasResized){
    //ACTION
}
```
**What?** When you assign a temporary variable more than once, but it is not a loop variable nor a collecting temporary variable, make a separate temporary variable for each assignment.

**Why?** Using temps more than once is confusing.

**Example:**

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

```java
final double perimeter =
2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (area);
```
(De)composing methods:
7. Remove Assignments To Parameters

What? When the code assigns to a parameter, use a temporary variable instead.

Why? Lack of clarity and confusion between “pass by value” and “pass by reference”

Example:

```java
int discount (int inputVal, int quantity, int yearToDate){
    if (inputVal > 50) inputVal -= 2;
    ... MORE CODE HERE ...
}
```

```java
int discount (int inputVal, int quantity, int yearToDate){
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    ... MORE CODE HERE ...
}
```
(De)composing methods:
8. Replace Method with Method Object

**What?** When you have local variables but cannot use **extract method**, turn the method into its own object, with the local variables as its fields.

**Why?** Extracting pieces out of large methods makes things more comprehensible.

**Example:**

```java
Order
price() double primaryBasePrice;
    double secondaryBasePrice;
    // long computation

PriceCalculator
primaryBasePrice secondaryBasePrice
compute()

return new PriceCalculator(this).compute()
```

```java
Order
price() 1

PriceCalculator
primaryBasePrice secondaryBasePrice
compute()

return new PriceCalculator(this).compute()
```
(De)composing methods: 9. Substitute Algorithm

**What?** When you want to replace an algorithm with a clearer alternative, replace the body of the method with the new algorithm.

**Why?** To replace complicated algorithms with clearer ones.

**Example:**

```java
String foundPerson(String[] people) {
    List candidates = Arrays.asList(new String[] {"John", "Jack"})
    for (int i = 0; i < people.length; i++)
        if (candidates[i]. contains (people[i]))
            return people[i];
}
```

```java
String foundPerson(String[] people) {
    for (int i = 0; i < people.length; i++)
        if (people[i]. equals ("John")) {
            return "John";
        }
        if (people[i]. equals ("Jack")) {
            return "Jack";
        }
    }
}
```
Small refactorings

(de)composing methods [9 refactorings]

moving features between objects [8 refactorings]

organising data [16 refactorings]

simplifying conditional expressions [8 refactorings]

dealing with generalisation [12 refactorings]

simplifying method calls [15 refactorings]
Small Refactorings: moving features between objects

1. Move Method
2. Move Field
3. Extract Class
4. Inline Class
5. Hide Delegate
6. Remove Middle Man
7. Introduce Foreign Method
8. Introduce Local Extension

Legend:
- = we will zoom in on these
- = home reading

29
Moving features between objects:

1,2. Move Method / Field

**What?** When a method (resp. field) is used by or uses more features of another class than its own, create a similar method (resp. field) in the other class; remove or delegate original method (resp. field) and redirect all references to it.

**Why?** Essence of refactoring

**Example:**

```
Class 1
  aMethod()

Class 2

Class 1
  aMethod()

Class 2
```
What? When you have a class doing work that should be done by two, create a new class and move the relevant fields and methods to the new class.

Why? Large classes are hard to understand.

Example:

<table>
<thead>
<tr>
<th>Person</th>
<th>PhoneNumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>areaCode</td>
</tr>
<tr>
<td>officeAreaCode</td>
<td>number</td>
</tr>
<tr>
<td>officeNumber</td>
<td></td>
</tr>
<tr>
<td>homeAreaCode</td>
<td></td>
</tr>
<tr>
<td>homeNumber</td>
<td></td>
</tr>
<tr>
<td>getOfficePhone</td>
<td></td>
</tr>
<tr>
<td>getHomePhone</td>
<td></td>
</tr>
<tr>
<td>getOfficePhone</td>
<td></td>
</tr>
<tr>
<td>getHomePhone</td>
<td></td>
</tr>
</tbody>
</table>
Moving features between objects:
4. Inline Class

**What?** When you have a class that does not do very much, move all its features into another class and delete it.

**Why?** To remove useless classes (as a result of other refactorings).

**Example:**

<table>
<thead>
<tr>
<th>Person</th>
<th>PhoneNumber</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>areaCode</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>number</td>
<td>officeAreaCode</td>
</tr>
<tr>
<td>getPhoneNumber()</td>
<td>getPhoneNumber()</td>
<td>getPhoneNumber()</td>
</tr>
</tbody>
</table>

In the example, the Person class with its getPhoneNumber() method is moved to the PhoneNumber class, which then acquires the name, officeAreaCode, and officeNumber features.
Moving features between objects:
5. Hide Delegate

**What?** When you have a client calling a delegate class of an object, create methods on the server to hide the delegate.

**Why?** Increase encapsulation.

**Example:**

```
Client Class

Person
- getDepartment()

Department
- getManager()

Client Class

Person
- getManager()

Department
- getManager()
```
Moving features between objects:

6. Remove Middle Man

**What?** When a class is doing too much simple delegation, get the client to call the delegate directly.

**Why?** To remove too much indirection (as a result of other refactorings).

**Example:**

Client Class

Person

Department

Client Class

Person

Department

Client Class
Moving features between objects: 7. Introduce Foreign Method

**What?** When a server class needs an additional method, but you cannot modify the class, create a method in the client class with an instance of the server class as its first argument.

**Why?** To introduce one additional service.

**Example:**

```java
Date newStart = new Date (previousEnd.getYear(),
   previousEnd.getMonth(), previousEnd.getDate() + 1);
```

```java
Date newStart = nextDay(previousEnd);
```

```java
private static Date nextDay(Date arg) {
   return new Date (arg.getYear(),
      arg.getMonth(), arg.getDate() + 1);
}
```
What? When a server class needs several additional methods but you cannot modify the class, create a new class containing the extra methods; make the extension class a subclass or wrapper.

Why? To introduce several additional services.

Example:

<table>
<thead>
<tr>
<th>Client Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextDayDate(Date): Date</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>MfDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextDay(): Date</td>
</tr>
</tbody>
</table>
Small refactorings

(de)composing methods [9 refactorings]

moving features between objects [8 refactorings]

organising data [16 refactorings]

simplifying conditional expressions [8 refactorings]

dealing with generalisation [12 refactorings]

simplifying method calls [15 refactorings]
Small Refactorings: organising data

1. Encapsulate field
2. Replace data value with object
3. Change value to reference
4. Change reference to value
5. Replace array with object
6. Duplicate observed data
7. Change unidirectional association to bidirectional
8. Change bidirectional association to unidirectional
9. Replace magic number with symbolic constant
10. Encapsulate collection
11. Replace record with data class
12. Replace subclass with fields
13-16. Replace type code with class / subclass / state / strategy
Organising Data:
1. Encapsulate Field

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

**Why?** Encapsulating state increases modularity, and facilitates code reuse and maintenance.
When the state of an object is represented as a collection of private variables, the internal representation can be changed without modifying the external interface.

**Example:**

```java
private String name;
public String getName() {
    return this.name;
}
public void setName(String s) {
    this.name = s;
}
```

public String name;
private String contents;
public String getContents() {
    return this.contents;
}
public void setContents(String s) {
    this.contents = s;
}

private Document doc;
public String getContents() {
    return this.doc.getContents();
}
public void setContents(String s) {
    this.doc.setContents(s);
}

public class Document {
    private String contents;

    public String getContents() {
        return this.contents;
    }
    public void setContents(String s) {
        this.contents = s;
    }
}
Organising Data:  
13. Replace Type Code with Subclass

**PROBLEM**
YOU HAVE A CODED TYPE FIELD OF WHICH THE VALUES DIRECTLY AFFECT TRIGGER DIFFERENT BEHAVIOUR IN CONDITIONALS.

**What?** An immutable type code affects the behaviour of a class

**Example:**

<table>
<thead>
<tr>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>const Engineer=0</td>
</tr>
<tr>
<td>const Salesman=1</td>
</tr>
<tr>
<td>const Manager=2</td>
</tr>
<tr>
<td>type:Int</td>
</tr>
</tbody>
</table>

**SOLUTION**
CREATE SUBCLASSES FOR EACH VALUE OF THE CODED TYPE. EXTRACT RELEVANT BEHAVIORS FROM THE ORIGINAL CLASS TO THESE SUBCLASSES. REPLACE THE CONTROL FLOW CODE WITH POLYMORPHISM.
When? If subclassing cannot be used, e.g. because of dynamic type changes during object lifetime (e.g. promotion of employees)

Example:

<table>
<thead>
<tr>
<th>Employee</th>
<th>EmployeeType</th>
</tr>
</thead>
<tbody>
<tr>
<td>const Engineer=0</td>
<td>Engineer</td>
</tr>
<tr>
<td>const Salesman=1</td>
<td>Salesman</td>
</tr>
<tr>
<td>const Manager=2</td>
<td>Manager</td>
</tr>
</tbody>
</table>

Organising Data: 15,16. Replace Type Code with State/Strategy

Makes use of state pattern or strategy design pattern
Organising Data:
12. Replace Subclass with Fields

**What?** Subclasses vary only in methods that return constant data

**Solution:** Change methods to superclass fields and eliminate subclasses

**Example:**

```
Person
  Male
  Female

Person
  sex: [M, F]
```

Similar to **replace inheritance with aggregation**
Small refactorings

(de)composing methods [9 refactorings]
moving features between objects [8 refactorings]
organising data [16 refactorings]
simplifying conditional expressions [8 refactorings]
dealing with generalisation [12 refactorings]
simplifying method calls [15 refactorings]
Small Refactorings: simplifying conditional expressions

1. Decompose conditional
2. Consolidate conditional expression
3. Consolidate duplicate conditional fragments
4. Remove control flag
5. Replace nested conditional with guard clauses
6. Replace conditional with polymorphism
7. Introduce null objects
8. Introduce assertion
Small refactorings

(de)composing methods [9 refactorings]

moving features between objects [8 refactorings]

organising data [16 refactorings]

simplifying conditional expressions [8 refactorings]

dealing with generalisation [12 refactorings]

simplifying method calls [15 refactorings]
Small Refactorings: dealing with generalisation

1. Push down method / field
2. Pull up method / field / constructor body
3. Extract subclass / superclass / interface
4. Collapse hierarchy
5. Form template method
6. Replace inheritance with delegation (and vice versa)
Dealing with Generalisation:  
1. Push Down Method

When behaviour on a superclass is relevant only for some of its subclasses, move it to those subclasses.
Dealing with Generalisation:
2. Pull Up Method

Simple variant: look for methods with same name in subclasses that do not appear in superclass

More complex variant: do not look at the name but at the behaviour of the method

If the method that is being pulled up already exists in the superclass as an abstract method, make it concrete with the common behaviour
Dealing with Generalisation: 3. Extract Superclass

When you have 2 classes with similar features
Small refactorings

(de)composing methods [9 refactorings]

moving features between objects [8 refactorings]

organising data [16 refactorings]

simplifying conditional expressions [8 refactorings]

dealing with generalisation [12 refactorings]

simplifying method calls [15 refactorings]
Small Refactorings: simplifying method calls

1. Rename method
2. Add parameter
3. Remove parameter
4. Separate query from modifier
5. Parameterise method
6. Replace parameter with method
7. Replace parameter with explicit methods
8. Preserve whole object
9. Introduce parameter object
10. Remove setting method
11.Hide method
12. Replace constructor with factory method
13. Encapsulate downcast
14. Replace error code with exception
15. Replace exception with test
### Simplifying method calls:

**9. Introduce Parameter Object**

**Problem**

Your methods contain a repeating group of parameters.

<table>
<thead>
<tr>
<th>Customer</th>
<th>Customer</th>
<th>DataRange</th>
</tr>
</thead>
<tbody>
<tr>
<td>amountInvoicedIn(from:Date,to:Date)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>amountReceivedIn(from:Date,to:Date)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>amountOverdueIn(from:Date,to:Date)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>amountInvoicedIn(r:DateRange)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>amountReceivedIn(r:DateRange)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>amountOverdueIn(r:DateRange)</td>
<td></td>
</tr>
</tbody>
</table>

**DataRange**

<table>
<thead>
<tr>
<th>from : Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>to : Date</td>
</tr>
</tbody>
</table>

**Solution**

Replace these parameters with an object and use that object as parameter instead.
Simplifying method calls:
14. Replace Error Code with Exception

What? When a method returns a special code to indicate an error, throw an exception instead.

Why? Clearly separate normal processing from error processing.

Example:

```java
int withdraw(int amount) {
    if (amount > balance)
        return -1
    else
        {balance -= amount;
        return 0}
}
```

```java
void withdraw(int amount) throws BalanceException {
    if (amount > balance) throw new BalanceException();
    balance -= amount;
}
```
Categories of refactorings (according to [Fowler2000])

Small refactorings

(de)composing methods [9]

moving features between objects [8]

organising data [16]

simplifying conditional expressions [8]

dealing with generalisation [12]

simplifying method calls [15]

Big refactorings

Tease apart inheritance

Extract hierarchy

Convert procedural design to objects

Separate domain from presentation
Big refactorings

Require a large amount of time (> 1 month)

Require a degree of agreement among the development team

No instant satisfaction, no visible progress
Big Refactorings

1. Tease apart inheritance
2. Extract hierarchy
3. Convert procedural design to objects
4. Separate domain from presentation
Big refactorings:
1. Tease apart inheritance

**Problem**

A tangled inheritance hierarchy that is doing 2 jobs at once

**Solution**

Create 2 separate hierarchies and use delegation to invoke one from the other
Big refactorings:
1. Tease apart inheritance

**Approach**

Identify the different jobs done by the hierarchy.

Extract least important job into a separate hierarchy.

Use *extract class* to create common parent of new hierarchy.

Create appropriate subclasses.

Use *move method* to move part of the behaviour from the old hierarchy to the new one.
Big refactoring:
1. Tease apart inheritance
Big refactorings:
1. Tease apart inheritance

Related design patterns

Bridge

decouples an abstraction from its implementation so that the two can vary independently

Strategy / Visitor / Iterator / State
Big refactorings:
2. Extract hierarchy

Problem

An overly-complex class that is doing too much work, at least in part through many conditional statements.

Solution

Turn class into a hierarchy where each subclass represents a special case.
Big refactorings:
2. Extract hierarchy

Approach

Create a subclass for each special case.

Use one of the following refactorings to return the appropriate subclass for each variation:

- replace constructor with factory method
- replace type code with subclasses
- replace type code with state/strategy

Take methods with conditional logic and apply:

- replace conditional with polymorphism
Calculating electricity bills.

Lots of conditional logic needed to cover many different cases:

- different charges for summer/winter
- different tax rates
- different billing plans for personal / business / government / …
- reduced rates for persons with disabilities or social security
Big refactorings:
3. Convert procedural design into objects

Problem

You have code written in a procedural style.

Solution

Turn the data records into objects, break up the behaviour, and move the behaviour to the objects.

Smaller refactorings used

extract method, move method, …
4. Separate domain from presentation

Goal

Change a two-tier design (user interface/database) into a three-tier one (UI/business logic/database).

Solution

Separate domain logic into separate domain classes.

Smaller refactorings used

extract method, move method/field, duplicate observed data, …
C. REFACTORYING TOOLS
AUTOMATED CODE REFACTORING TOOLS

Available for all major programming languages
(and OO programming languages in particular)

Java : IntelliJ IDEA, Eclipse, NetBeans, JDeveloper, ...

JavaScript : WebStorm, Eclipse, ...

C++ : VisualStudio, Eclipse, ...

ObjectiveC and SWIFT : XCode

.NET : VisualStudio

Smalltalk, PHP, Ruby, Python, C#, Delphi, ...
LIMITATIONS OF MOST REFACTORIZATION TOOLS

Only support for primitive refactorings

class refactorings

  add (sub)class to hierarchy, rename class, remove class

method refactorings

  add to class, rename, remove, push down, pull up, add parameter, move to component, extract code

variable refactorings

  add to class, rename, remove, push down, pull up, create accessors, abstract variable

Often no support for higher-level refactorings
REFACTORING IN ECLIPSE

The refactoring tool in Eclipse supports a number of transformations described in Martin Fowler's book Refactoring.

Refactoring can be accessed via the Refactor menu.

Refactoring commands are also available from the context menus in many views or appear as quick assists.
SUPPORTED REFACTORING ACTIONS IN ECLIPSE (2016)

- Rename, Move, Change Method Signature
- Extract Method, Extract Local Variable, Extract Constant
- Inline, Move Type to New File, Use Supertype Where Possible
- Convert Anonymous Class to Nested, Convert Local Variable to Field
- Extract Superclass, Extract Interface, Extract Class
- Push Down, Pull Up, Encapsulate Field
- Introduce Parameter Object, Introduce Indirection
- Introduce Factory, Introduce Parameter
- Generalize Declared Type, Infer Generic Type Arguments
  (and more)
CODE REFACTORING – REFACTORING TOOLS

Changes to be performed

- Node.java – LANwithTests
- Workstation.java – LANwithTests
- NodeTest.java – LANwithTests
- PacketTest.java – LANwithTests

Java Source Compare

Original Source

```java
public class Node {
    public String name;

    public Node nextNode;
    public Node(String s) {
        name = s;
    }
    public Node(String s, Node n) {
        this(s); // calls the constructor Node()
        nextNode = n;
    }
}
```

Refactored Source

```java
public class Node {
    private String name;

    public Node nextNode;
    public Node(String s) {
        setName(s);
    }
    public Node(String s, Node n) {
        this(s); // calls the constructor Node()
        nextNode = n;
    }
}
```

[Image of code refactoring interface]
D. WORDS OF WARNING
A WORD OF WARNING (1)

Know what you are doing

If not applied well, refactoring may decrease quality rather than improve it
"Bad smells" are symptoms that something is wrong

Refactoring are supposed to remove “bad smells”
A WORD OF WARNING (1)

Refactoring should not introduce new smells

```plaintext
Person
  name
  gender

getOfficePhone
getHomePhone

HumanBeing
  gender

SMELLS LIKE A TOO ABSTRACT CLASS

EXTRACT SUPERCLASS

Person
  name

getOfficePhone
getHomePhone
```
Bad code smells

indicate that your code is ripe for refactoring

Refactoring is about

*how* to change code

Bad smells are about

*when* to modify it
A WORD OF WARNING (2)

Independently applied refactorings can introduce subtle merge conflicts.

REFACTORING CONFLICT:

In the new version, Safe should not be handled by Bank, but by Agency.
Learning objectives:
- Definition and difference between maintenance, evolution, reuse
- Different types of maintenance
- Causes of maintenance and changes
- Technical differences of evolution
- Reuse
25. Give a definition of **refactoring** in your own words and illustrate it with a concrete example of a refactoring.

26. Explain **why** it is important to refactor.

27. Explain **when** (= at what moment) refactoring should (or should not) be performed.

28. Like refactoring, **performance optimisation** does not usually change the behaviour of code (other than its speed); it only alters the internal structure. So how does it differ from refactoring?

29. Explain and illustrate one of the following refactorings in detail:

   - Extract Method, Move Method, Extract Class, Replace Type Code with Subclass, Replace Subclass with Fields, Pull Up Method, Introduce Parameter Object

30. Give a concrete example of how a refactoring could accidentally **reduce quality**.

31. Give a concrete example of how to independently applied refactorings could accidentally introduce a subtle **merge conflict**.
CLASS . . . IS . . . DISMISSED.