An introduction to
Software Refactoring

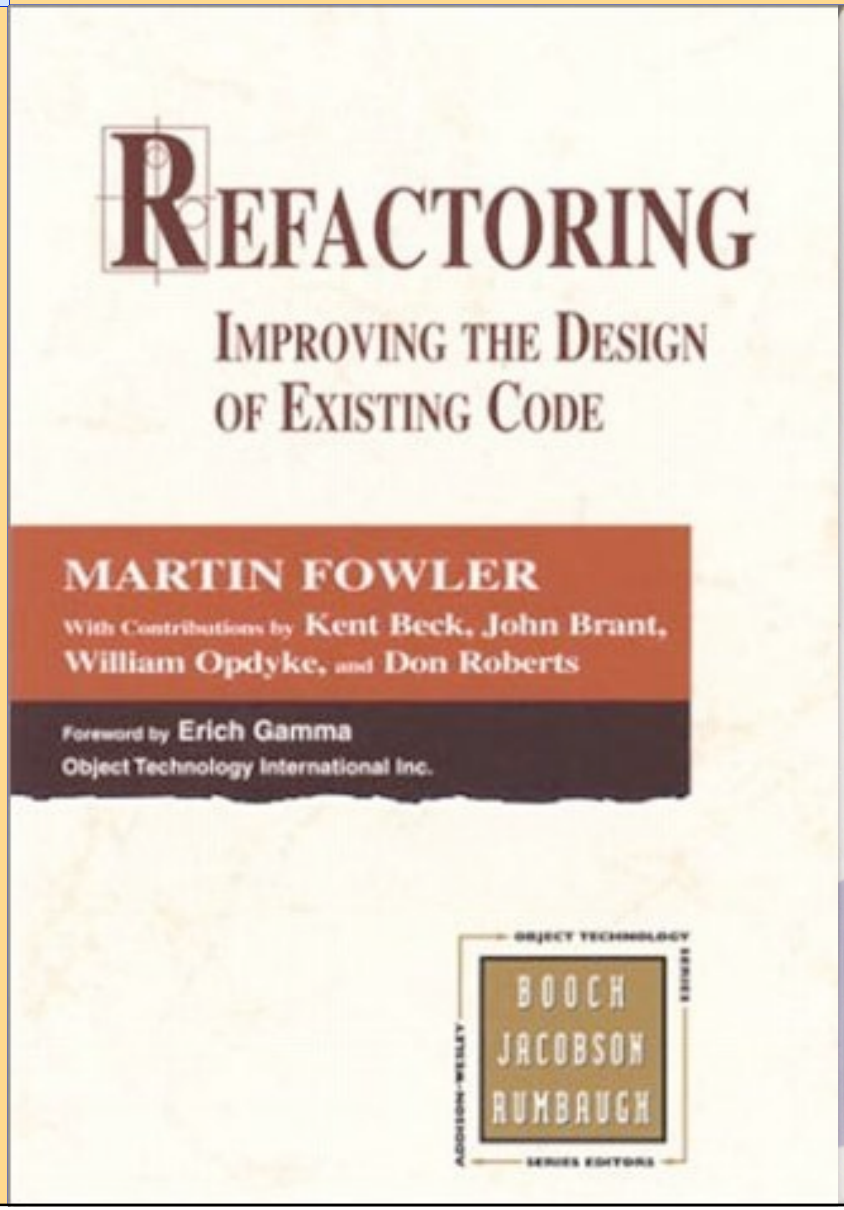
Tom Mens

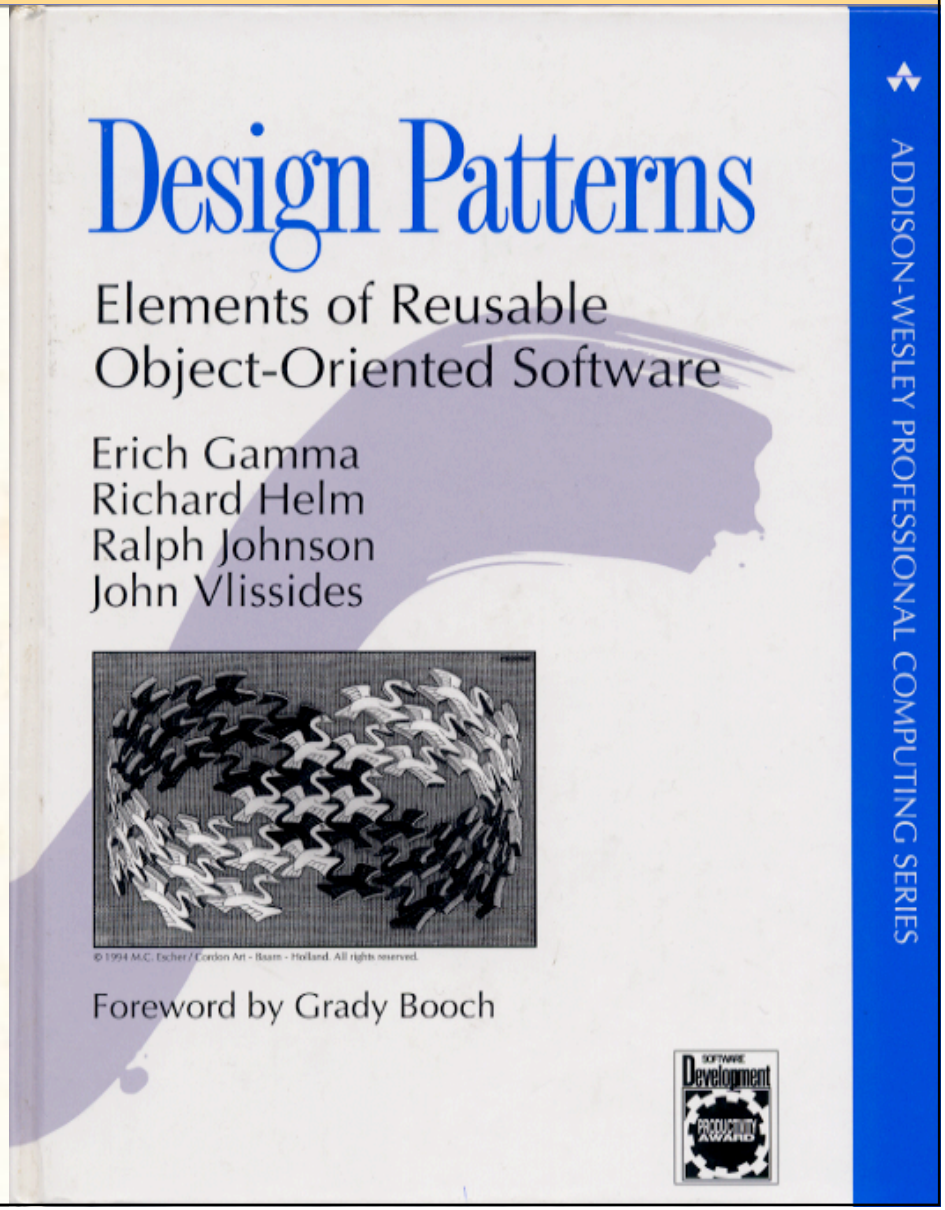
Software Engineering Lab
University ofMons-Hainaut
w3.umh.ac.be/genlog

Course Overview

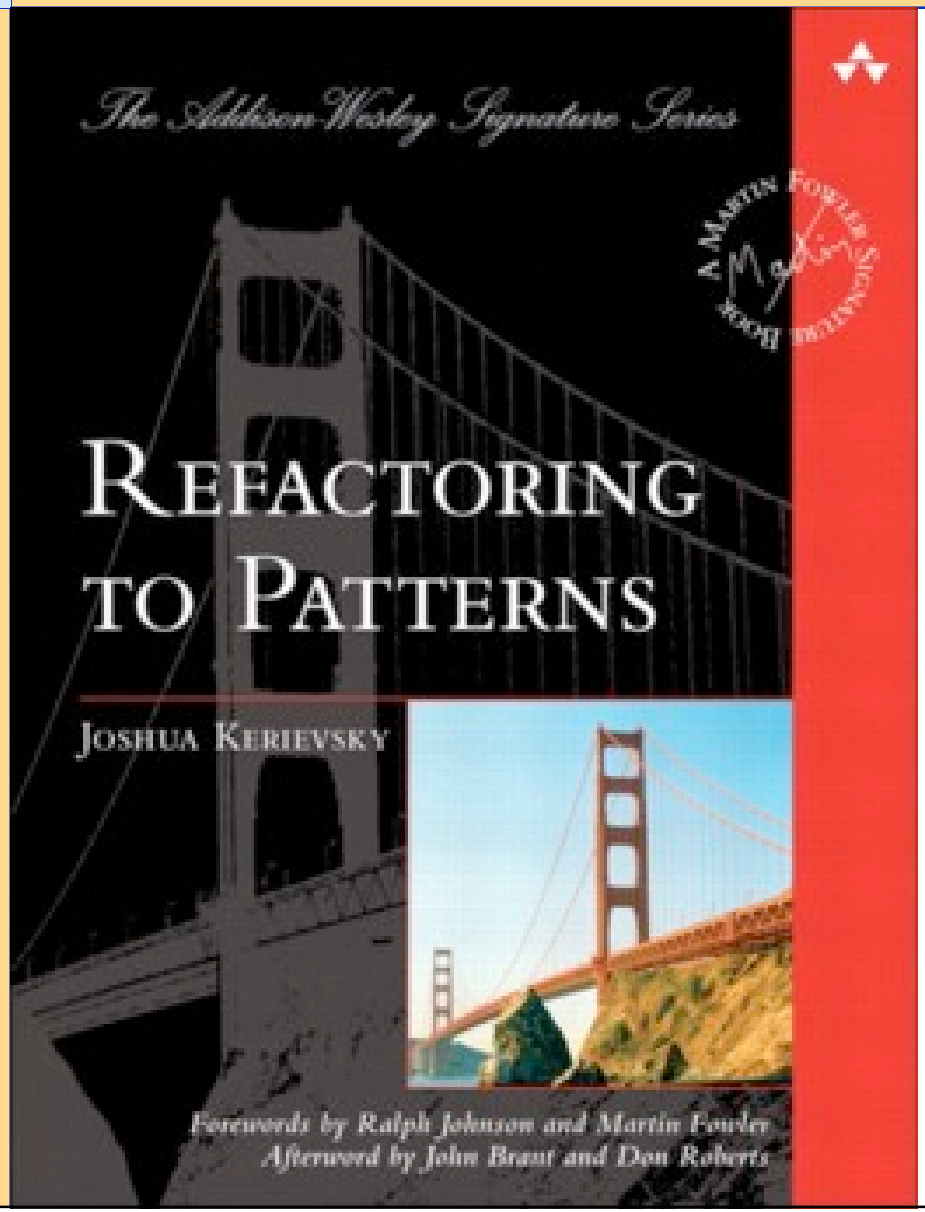
- Goal
  - To introduce and explain the technique of software refactoring and to explain its importance
- Overview
  - What is it?
  - Why is it needed?
  - When should we do it?
  - How can it be automated?
  - Categories of refactorings

References
Books





References
Books



- A marriage of refactoring with design patterns
  - using patterns to improve an existing design is better than using patterns early in a new design
  - existing designs can be improved by using specific refactorings that introduce design patterns

References
selected PhD dissertations

- W. G. Griswold. *Program restructuring as an aid to software maintenance*
  - University of Washington, USA, 1991
- W. F. Opdyke. *Refactoring object-oriented frameworks*
  - University of Illinois at Urbana-Champaign, USA, 1992
- I. Moore. *Automatic restructuring of object-oriented programs*
  - Manchester University, UK, 1996
- D. Roberts. *Practical Analysis for Refactoring*
  - University of Illinois at Urbana-Champaign, 1999
- S. Tichelaar. *Modeling object-oriented software for reverse engineering and refactoring*
  - University of Bern, Switzerland, 2001

References
selected journal articles

- W. G. Griswold, D. Notkin. *Automated assistance for program restructuring*
  - ACM Trans. Software Engineering and Methodology, 2(3): 228-269, July 1993.
- P. L. Bergstein. *Maintenance of Object-Oriented Systems During Structural Evolution*
  - TAPOS Journal 3(3): 185-212, 1997
- D. Roberts, J. Brant, R. E. Johnson. *A refactoring tool for Smalltalk*
  - TAPOS Journal 3(4): 253-263, 1997
- T. Mens, T. Tourwé. *A survey on software refactoring*
  - IEEE Transactions on Software Engineering, 30(2): 126-13, February 2004

Course Overview

- What is refactoring?
- Why is it needed?
- When do we need to refactor?
- How can it be automated
- Categories of refactoring

What is refactoring?

- A software transformation that
  - preserves the external software behaviour
  - improves the internal software structure

- Some definitions
  - [Fowler1999] a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour
  - [Roberts1998] A behaviour-preserving source-to-source program transformation
  - [Beck1999] A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ...

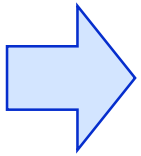
Motivating example Encapsulate Field

Fowler 1999, page 206

There is a public field

Make it private and provide accessors

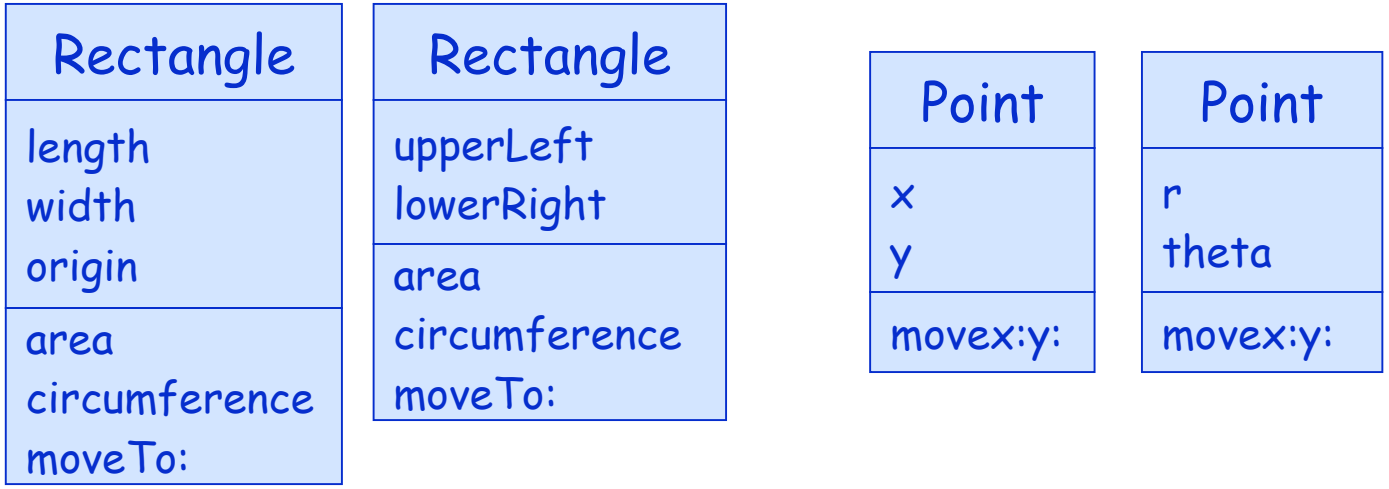
```
public class Node {
  public String name;
  public Node nextNode;
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
    nextNode.accept(p); }
  }
```



```
public class Node {
  private String name;
  private Node nextNode;
  public String getName() {
    return this.name; }
  public void setName(String s) {
    this.name = s; }
  public Node getNextNode() {
    return this.nextNode; }
  public void setNextNode(Node n) {
    this.nextNode = n; }
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
    this.getNextNode().accept(p); }
  }
```

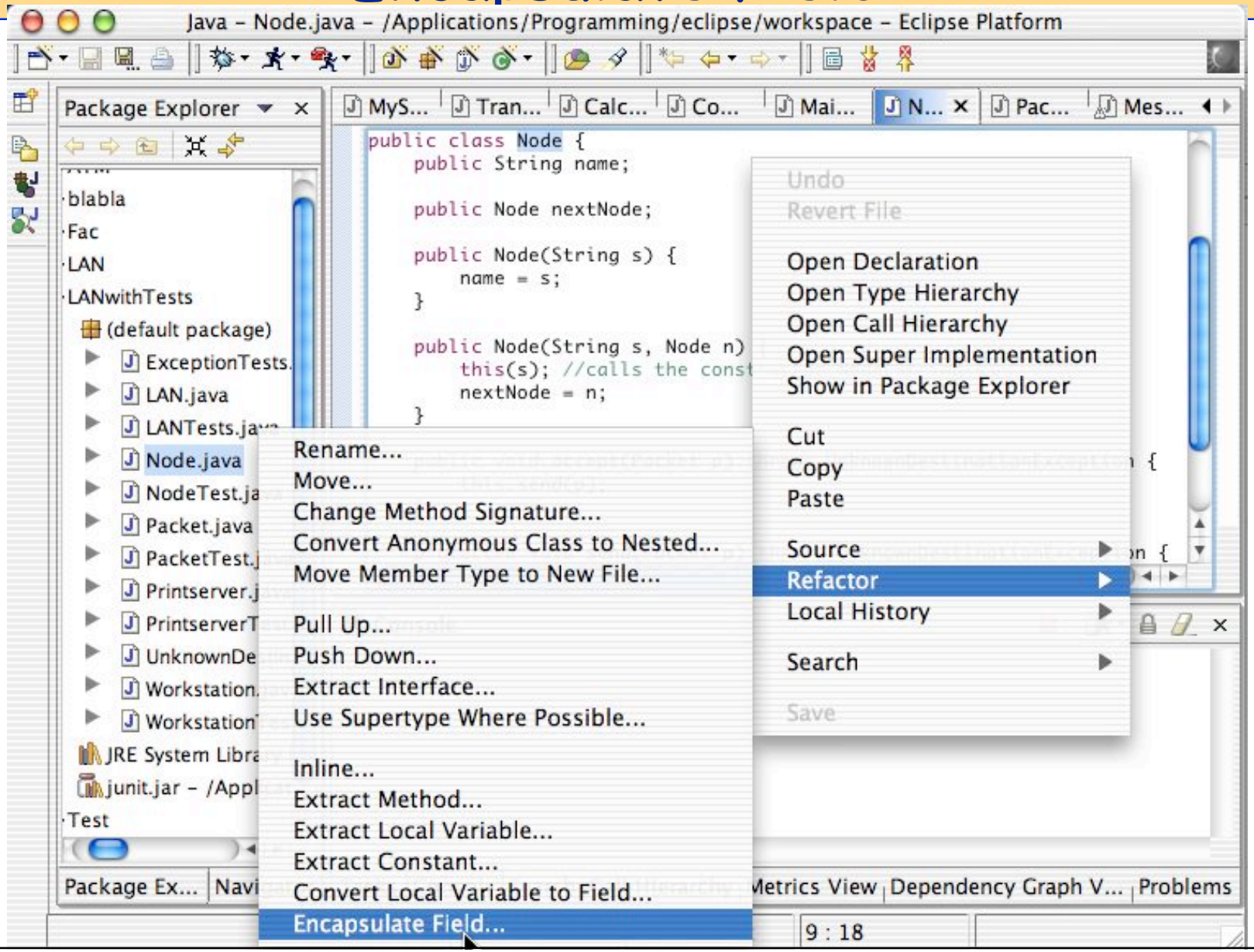
Motivating example
Encapsulate Field

- Why should we apply this refactoring?
  - Encapsulating the 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

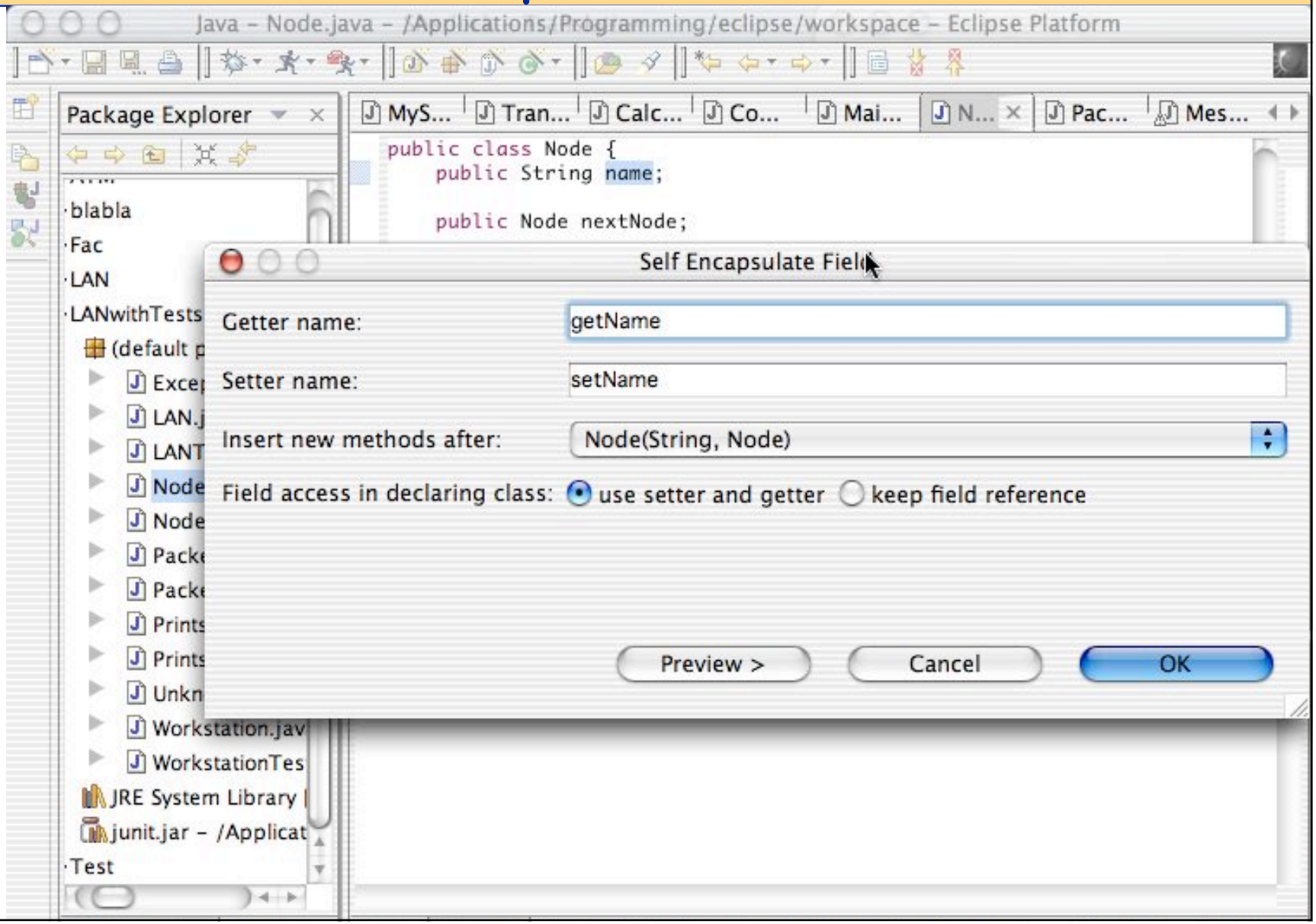


Different internal representation but same interface!

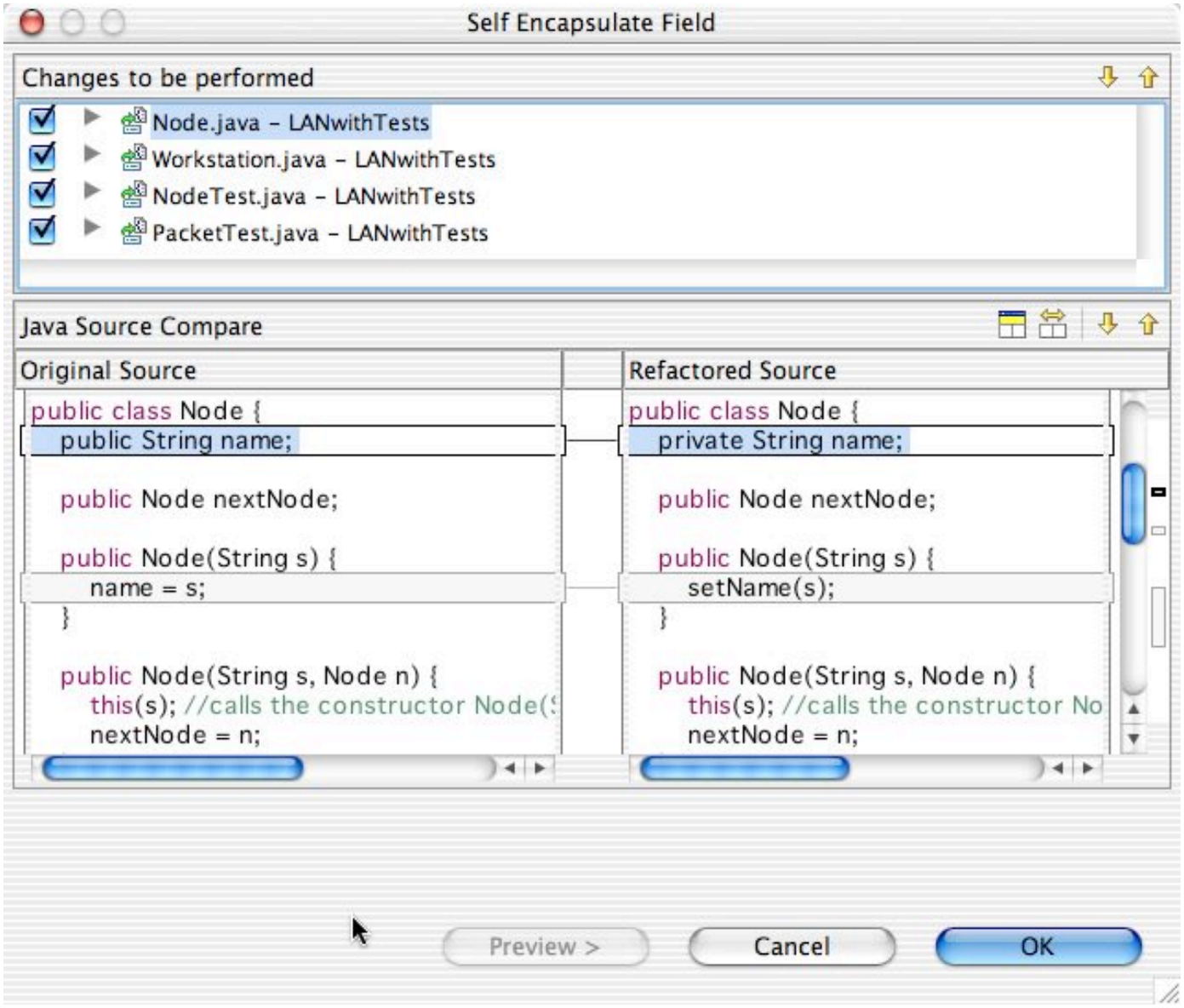
Motivating example
Encapsulate Field



Motivating example
Encapsulate Field



Motivating example
Encapsulate Field

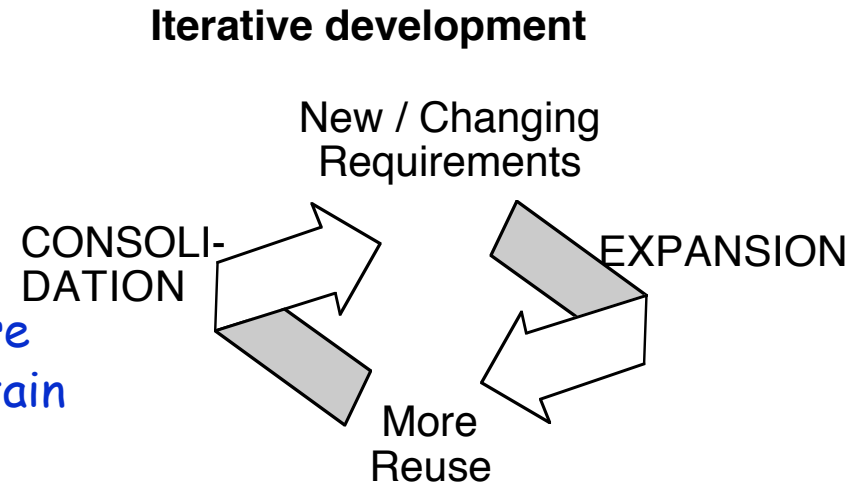


Why do we need refactoring?

- To improve the software design
- To reduce
  - software decay / software aging
  - software complexity
  - software maintenance costs
- To increase
  - software understandibility
    - e.g., by introducing design patterns
  - software productivity
    - at long term, not at short term
- To facilitate future changes
- ...

When do we need to refactor?

- Two essential phases in the iterative software development approach
  - (According to Foote and Opdyke, 1995)
  - expansion
    - adding new functionality
  - consolidation
    - reorganise and restructure the software to make it more reusable and easier to maintain
    - introduce design patterns
    - apply refactorings
  - fits naturally in spiral process model



Consolidation is necessary to ensure next expansion success

When do we need to refactor?

- Refactoring also fits naturally in the agile methods philosophy
  - e.g. Extreme Programming
- Is needed to address the principle "Maintain simplicity"
  - Wherever possible, actively work to eliminate complexity from the system
  - By refactoring the code

When do we need to refactor? Some guidelines

- When you think it is necessary
  - Not on a periodical basis
- Apply the rule of three
  - first time: implement solution from scratch
  - second time: implement something similar by duplicating code
  - third time: do not reimplement or duplicate, but factorise!
- Consolidation before adding new functionality
  - Especially when the functionality is difficult to integrate in the existing code base
- During debugging
  - If it is difficult to trace an error, refactor to make the code more comprehensible
- During formal code inspections (code reviews)

How do we know what to refactor?

- Identify bad smells in the source code [Beck1999]
  - "structures in the code that suggest (sometimes scream for) the possibility of refactoring"
- Examples
  - Duplicated Code
  - Long Method
    - The longer the method the harder it is to see what it's doing
  - Large Class
  - Case Statement
    - replace procedural code by object-oriented code
  - Feature Envy
    - Often a method that seems more interested in a class other than the one it's actually in.
  - Lazy Class
  - ...

How do we know what to refactor?

Bad smell	Proposed refactoring
duplicated code	extract method pull up variable form template method substitute algorithm
long method	extract method replace temp with query introduce parameter object preserve whole object replace method with method object
large class	extract class extract subclass
feature envy	move method
lazy class	collapse hierarchy inline class

Refactoring tool support

- Available for all major OO programming languages (see http://www.refactoring.com)
  - Java
    - Xrefactory, RefactorIT, jFactor, IntelliJ IDEA, Eclipse
  - Smalltalk
    - Refactoring Browser [Roberts et al. 1997]
  - C++
    - CppRefactory, Xrefactory
  - C#
    - C# Refactoring Tool, C# Refactory
  - Delphi
    - Modelmaker Tool, Castalia
  - Eiffel, and many more

Refactoring tool support

- Available for all major operating systems
  - Windows
  - Generic UNIX
  - Mac OS X
  - Linux

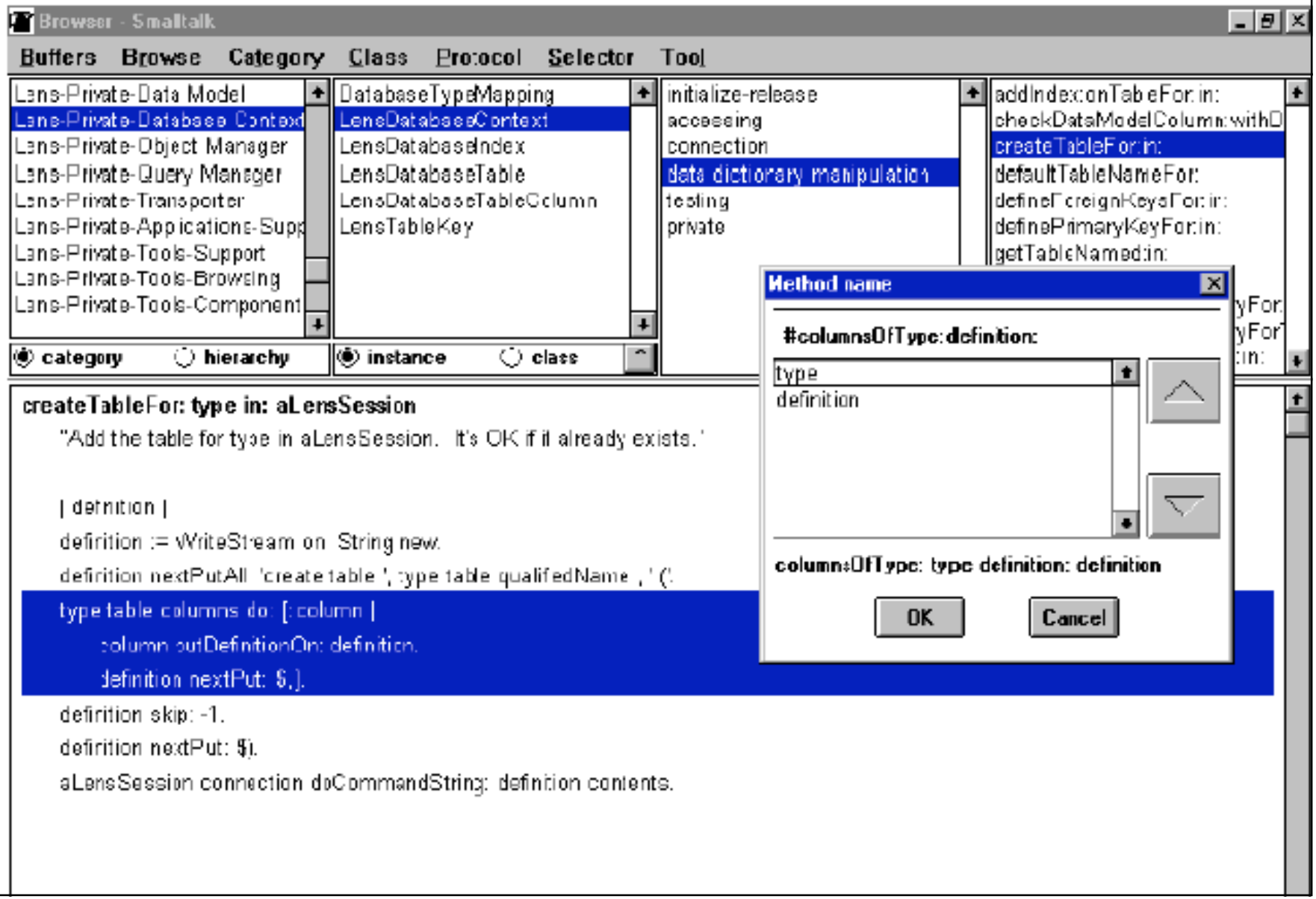
Refactoring tool support

- Available for all major software development environments (IDEs)
  - NetBeans
    - RefactorIT
  - Oracle Jdeveloper
    - RefactorIT
  - Borland JBuilder
    - RefactorIT
  - Eclipse
    - built-in
  - IntelliJ IDEA
    - built-in
  - Emacs
    - Xrefactory
  - Visual Studio .NET
    - C# Refactory

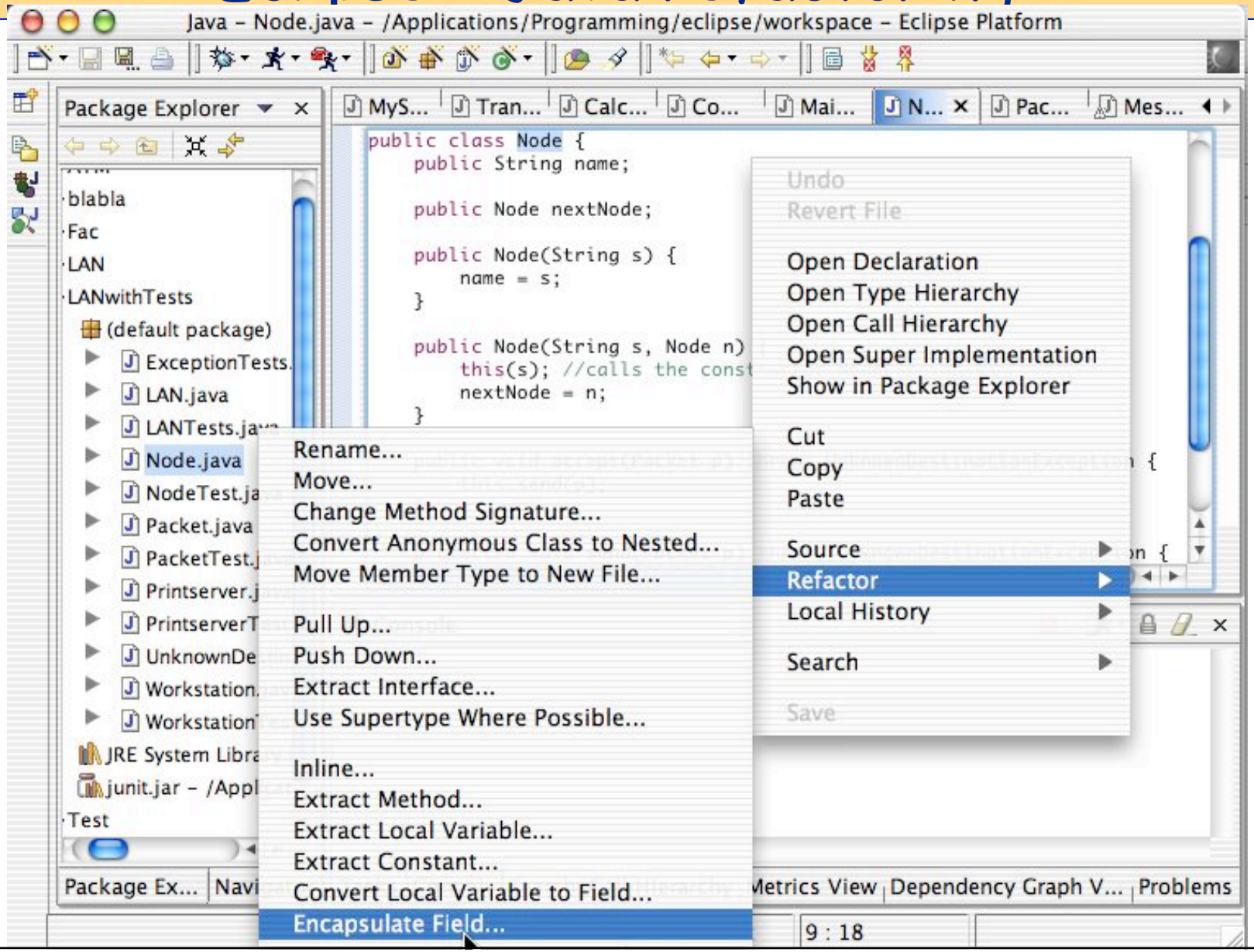
Refactoring tool support

- Current limitations
  - 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
  - no support for high-level refactorings

Refactoring tool support
Smalltalk - refactoring browser



Refactoring tool support
Eclipse - Java refactoring



categories of refactoring

categories of refactoring techniques

- based on the granularity
  - primitive versus composite / small versus big refactorings
- based on the programming language
  - language-specific (e.g. Java, Smalltalk, ...)
  - language-independent (e.g. [Tichelaar&al 2000])
- based on the degree of formality
  - formal (e.g. [Bergstein 1997])
  - not formal (e.g. [Fowler 1999])
  - semi-formal (e.g. Opdyke)
- based on the degree of automation
  - fully automatic (e.g. [Moore 1996])
  - interactive (e.g. Refactoring Browser [Roberts&al 1997])
  - fully manual (e.g. [Fowler 1999])

categories of refactoring according to Demeyer

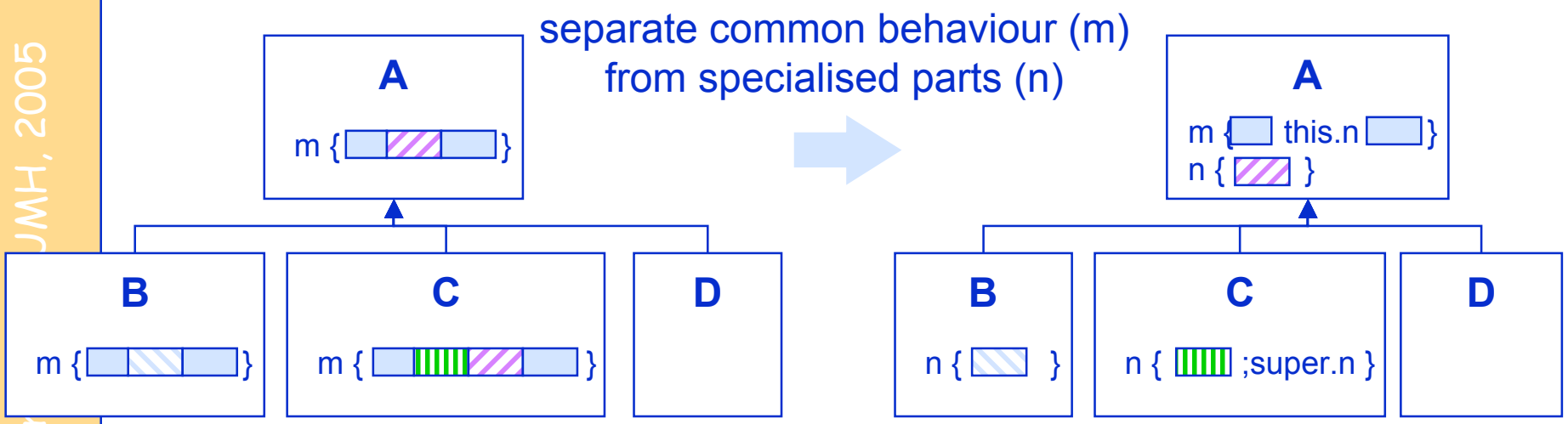
- 3 categories that correspond to generic design evolutions occuring frequently in object-oriented software systems
  [Demeyer&al 2000]

  - Creating template methods

  - Optimising class hierarchies

  - Incorporating composition relationships

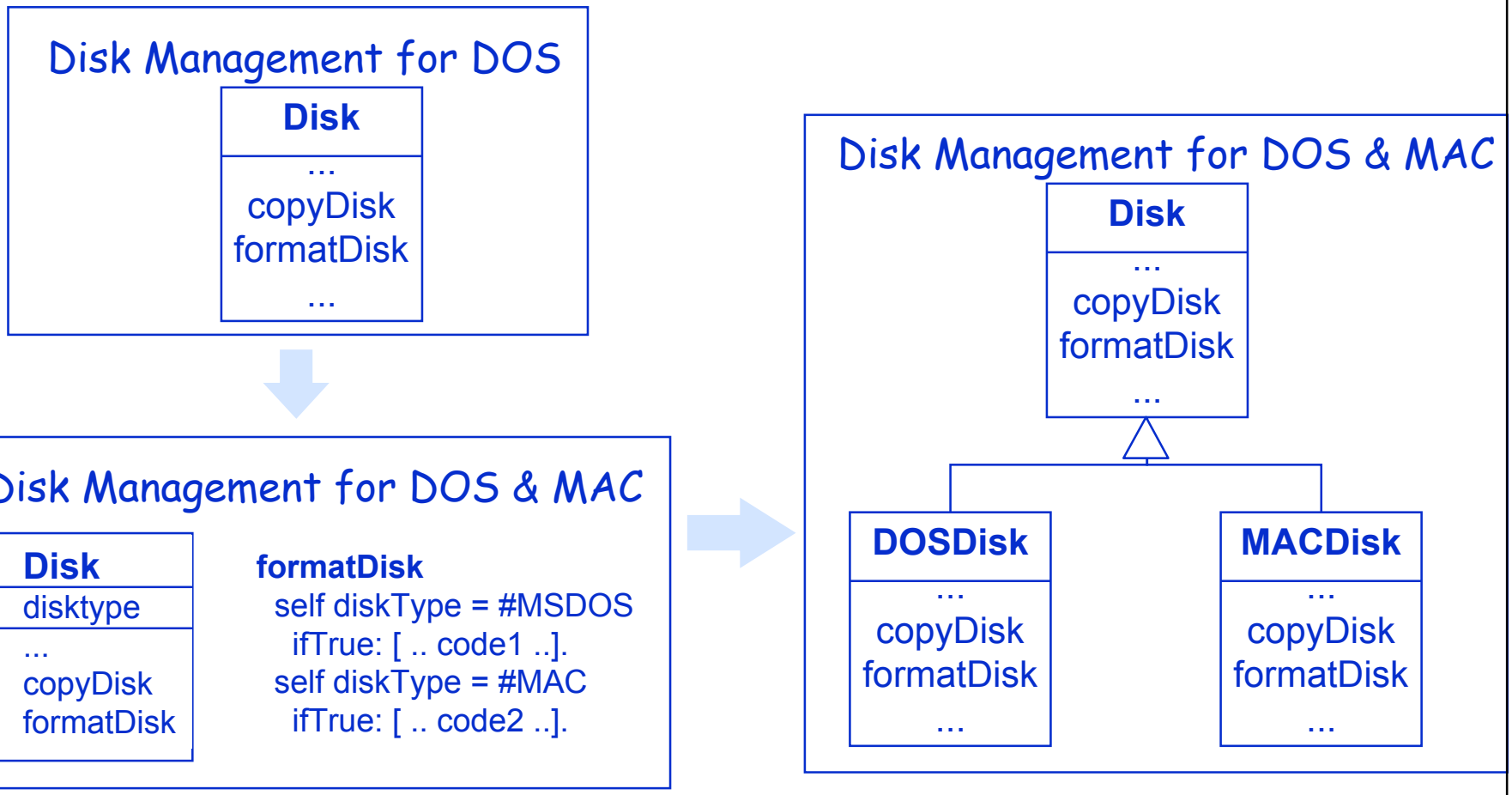
categories of refactoring
1. Creating template methods



categories of refactoring
2.a Refactor to specialise

- Refactor to specialise
  - improve class hierarchy structure by decomposing a large, complex class into several smaller classes
  - the complex class usually embodies both a general abstraction and several different concrete cases that are candidates for specialisation
    - Specialise the class by adding subclasses corresponding to the conditions in a conditional expression

categories of refactoring
2.a Refactor to specialise



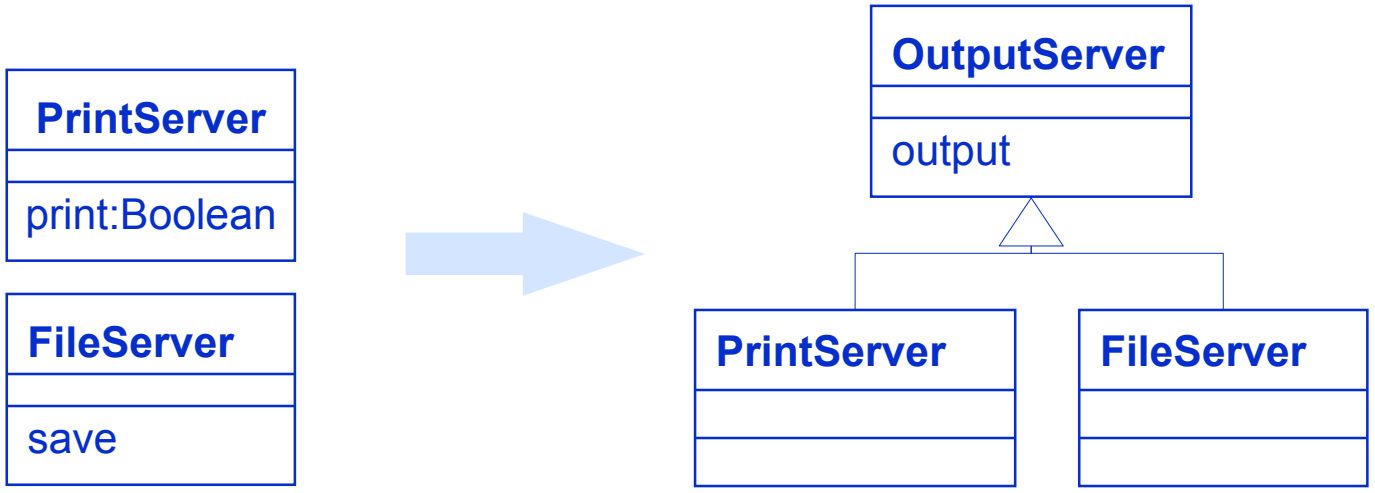
categories of refactoring
2.b Refactor to generalise

- Refactor to generalise
  - identify proper abstractions by examining concrete examples and generalising their commonalities
  - e.g. abstract classes can be generalised from concrete ones using bottom up analysis of the class hierarchy

- Generalisation proceeds by:
  - finding things that are given different names but are really the same (and thus renaming them)
  - parameterising to eliminate differences
  - break up large things into small things so that similar components can be found

categories of refactoring
2.b Refactor to generalise

- Example



categories of refactoring
2.b Refactor to generalise

- Steps to create an abstract superclass
  - Create a common superclass
  - Make method signatures compatible
  - Add method signatures to the superclass
  - Make method bodies compatible
  - Make instance variables compatible
  - Move instance variables to the superclass
  - Move common code to the abstract superclass

categories of refactoring

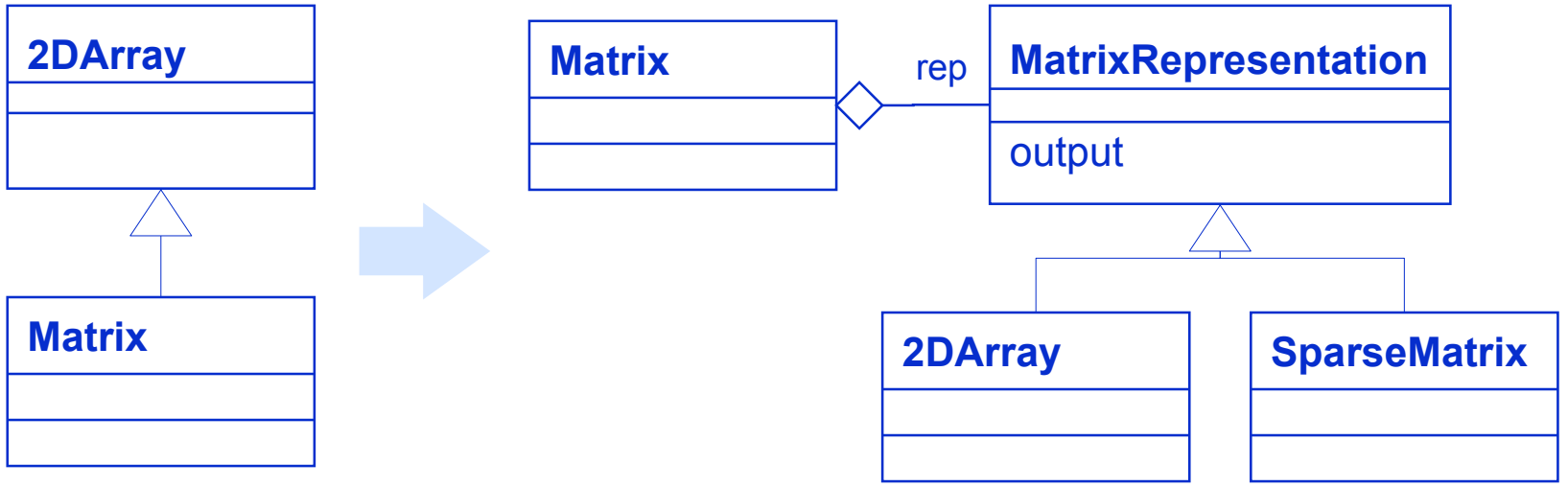
3. Incorporating composition relationships

- Motivation
  - Inheritance is sometimes overused and incorrectly used in modelling the relationships among classes
  - Aggregations are another way to model these relationships
- Refactorings regarding aggregations
  - move instance variables/methods from an aggregate class to the class of one of its components
  - move instance variables/methods from a component class to the aggregate classes that contain components which are instances of the component class
  - convert a relationship, modelled using inheritance, into an aggregation and vice versa [Johnson&Opdyke1993]

categories of refactoring

3. Incorporating composition relationships

- Example
  - Convert inheritance into aggregation



categories of refactoring according to Fowler

- 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]
- big refactorings
  - tease apart inheritance
  - extract hierarchy
  - convert procedural design to objects
  - separate domain from presentation

categories of refactoring
small refactorings

- small refactorings
  - (de)composing methods [9 refactorings]
    - Extract Method
    - Inline Method
    - Inline Temp
    - Replace Temp With Query
    - Introduce Explaining Variable
    - Split Temporary Variable
    - Remove Assignments to Parameter
    - Replace Method With Method Object
    - Substitute Algorithm

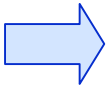
categories of refactoring
small refactorings

- (de)composing methods
  - 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:

Mind local vars!!!

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



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

categories of refactoring
small refactorings

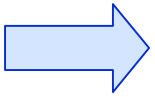
- (de)composing methods
  - Inline Method (opposite of Extract Method)

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:

```
int getRating(){
return (moreThanFiveLateDeliveries());
}
boolean more Than FiveLateDeliveries(){
return _numberOfLateDeliveries > 5;
}
```



```
int getRating(){
return (_numberOfLateDeliveries > 5);
}
```

categories of refactoring
small refactorings

- (de)composing methods
  - Inline Temp

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***)

example:

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

```
return (anOrder. basePrice() > 100)
```

Tom Mens, UMH, 2005

categories of refactoring
small refactorings

- (de)composing methods
  - 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:

```
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;
}
```

categories of refactoring
small refactorings

- (de)composing methods
  - 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:

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

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

if (isMacOs && isIEBrowser && wasInitialized() && wasResized){
//ACTION
}
```

categories of refactoring
small refactorings

- (de)composing methods
  - Split Temporary Variable

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:

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

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

categories of refactoring
small refactorings

- (de)composing methods
  - Remove Assignments To Parameter

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:

```
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;
```

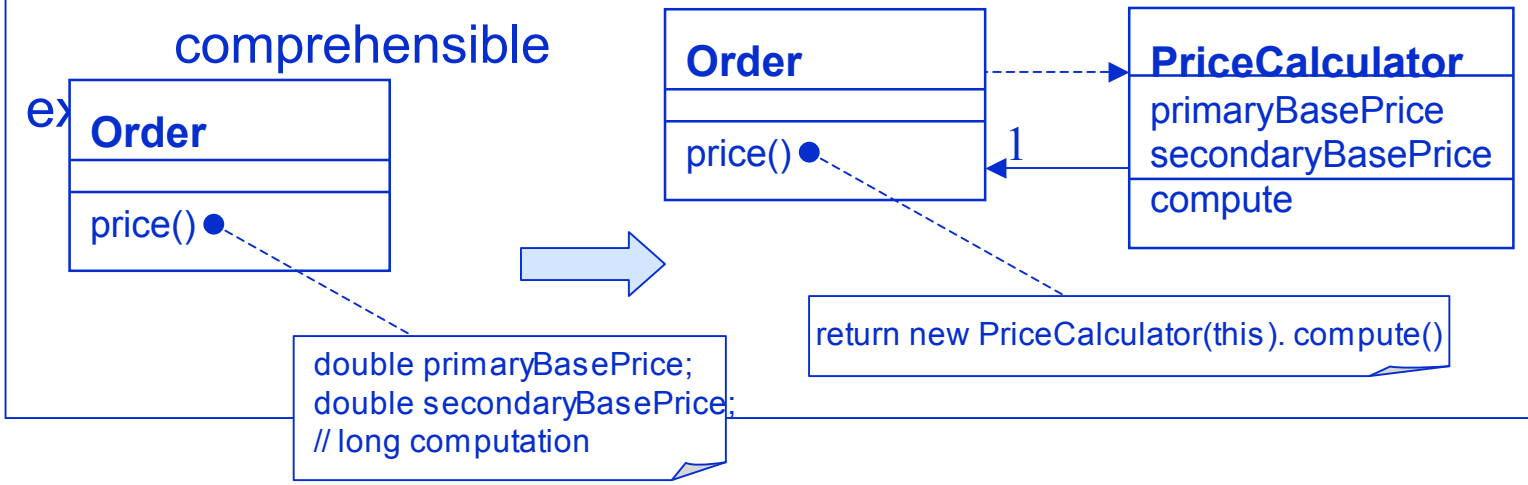
categories of refactoring
small refactorings

- (de)composing methods
  - 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

ex



categories of refactoring
small refactorings

- (de)composing methods
  - 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:

```
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”;
      }
  }
}
```

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

categories of refactoring
small refactorings

- small refactorings
  - moving features between objects [8 refactorings]
    - Move Method
    - Move Field
    - Extract Class
    - Inline Class
    - Hide Delegate
    - Remove Middle Man
    - Introduce Foreign Method
    - Introduce Local Extension

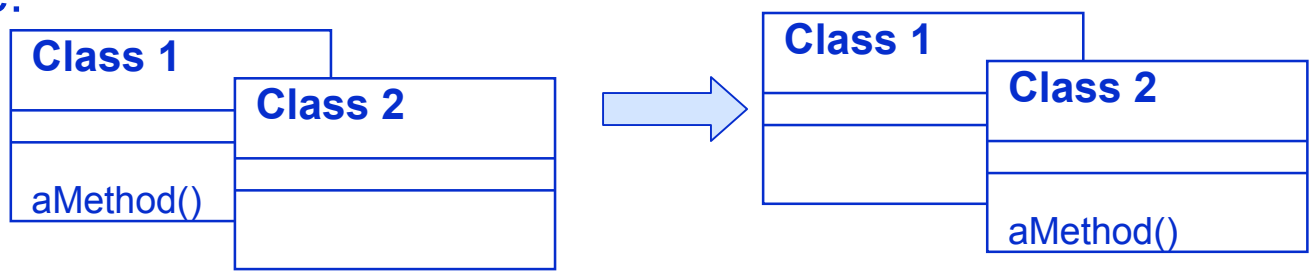
categories of refactoring
small refactorings

- moving features between objects
  - 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

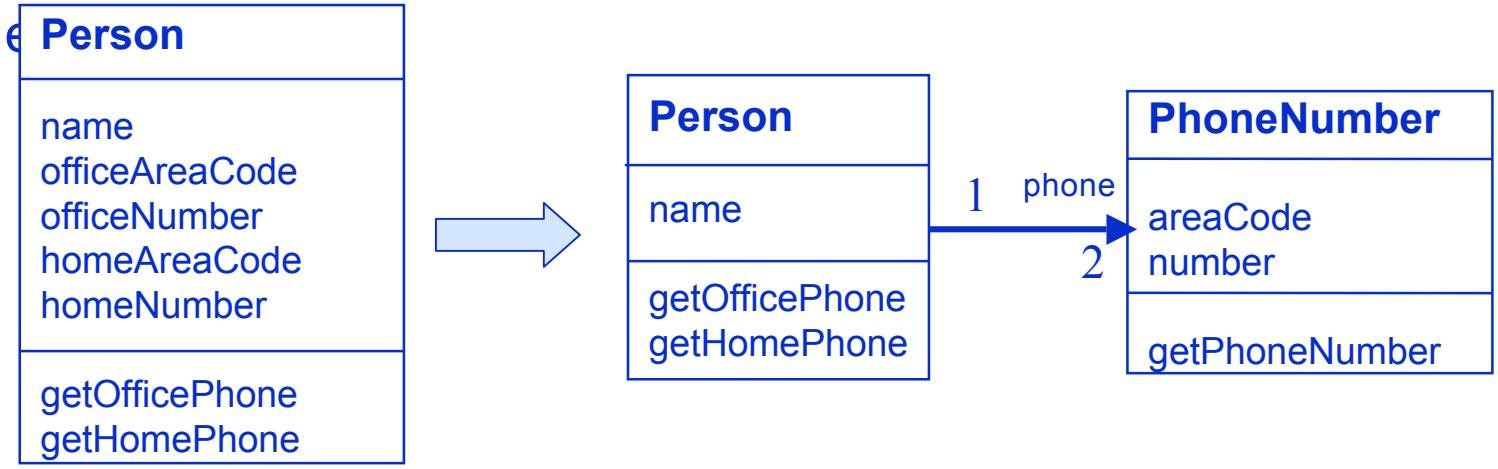
example:



categories of refactoring
small refactorings

- moving features between objects
  - Extract Class

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



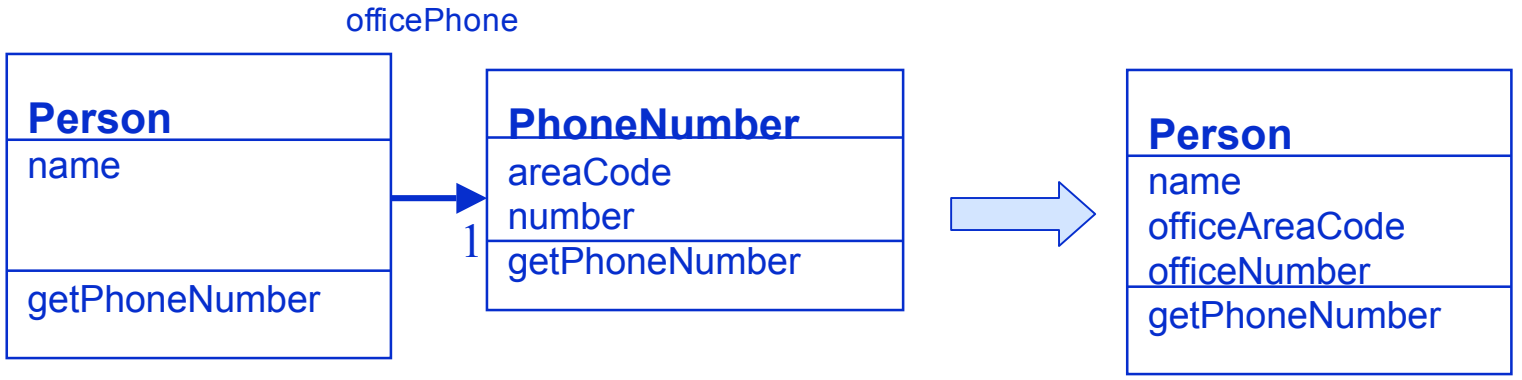
categories of refactoring
small refactorings

- moving features between objects
  - 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:



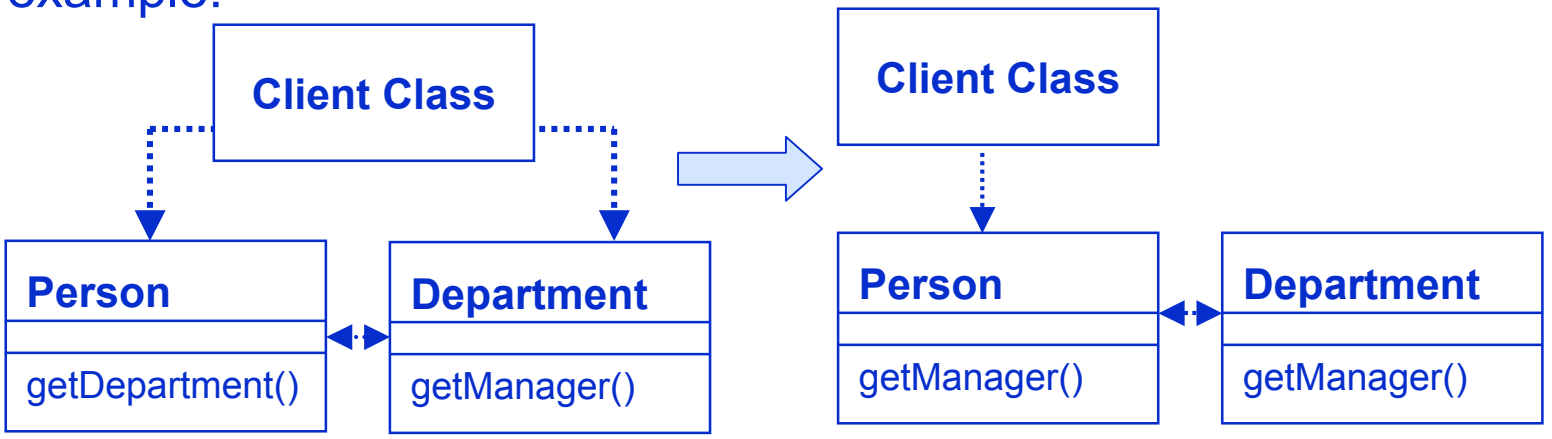
categories of refactoring
small refactorings

- moving features between objects
  - 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

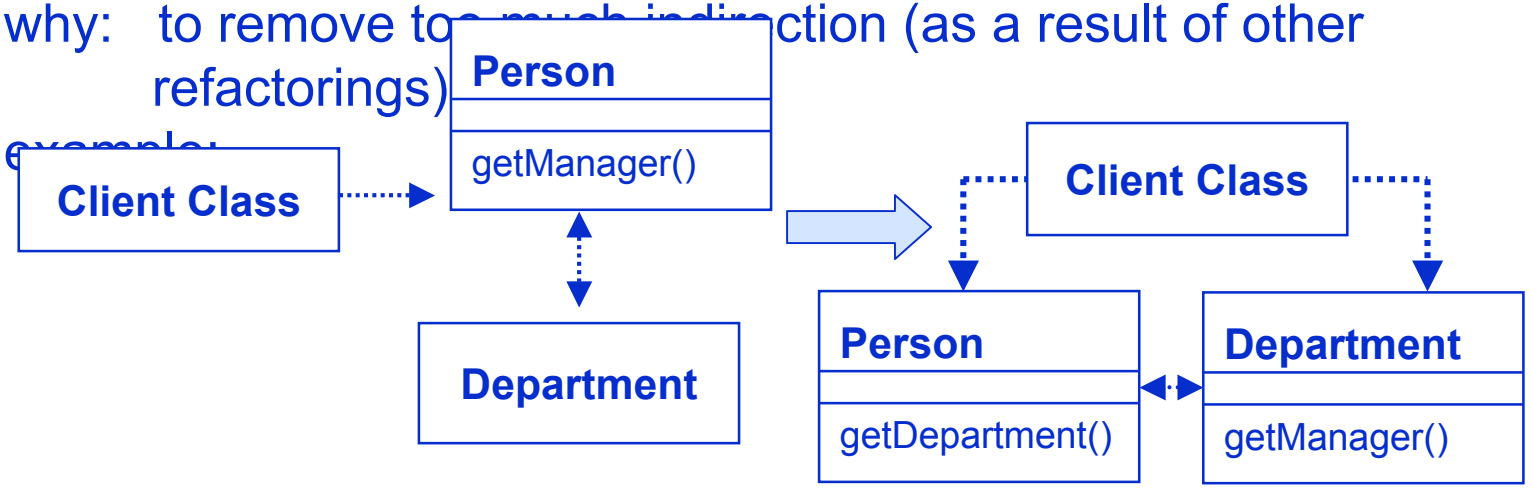
example:



categories of refactoring
small refactorings

- moving features between objects
  - 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:



categories of refactoring
small refactorings

- moving features between objects
  - 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:

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

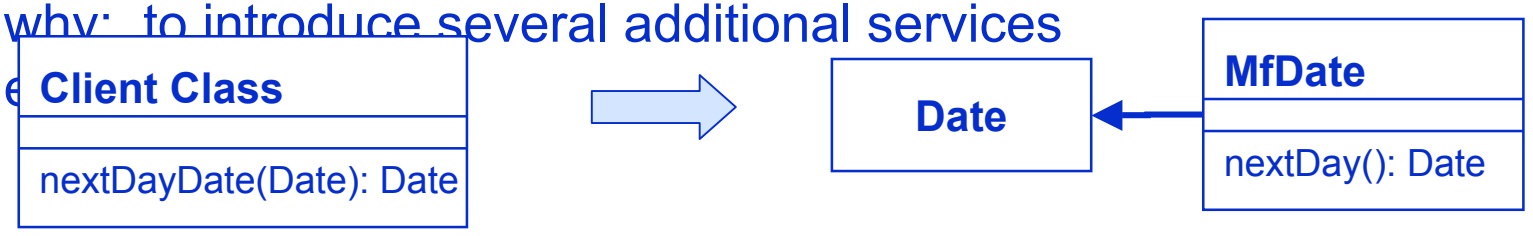
```
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date arg) {
  return new Date (arg.getYear(),
    arg.getMonth(), arg.getDate() + 1);
}
```

categories of refactoring
small refactorings

- moving features between objects
  - Introduce Local Extension

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
e



categories of refactoring
small refactorings

- small refactorings
  - organising data [16 refactorings]
    - encapsulate field (see motivating example)
    - replace data value with object
    - change value to reference / change reference to value
    - replace array with object
    - duplicate observed data
    - change unidirectional association to bidirectional / change bidirectional association to unidirectional
    - replace type code with class/subclass/state/strategy
    - replace magic number with symbolic constant
    - encapsulate collection
    - replace record with data class
    - replace subclass with fields

categories of refactoring
small refactorings

- organising data
  - Encapsulate field

what:
why:
example:

```
public String name;
```

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

categories of refactoring
small refactorings

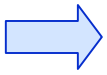
- organising data
  - Replace data value with object

what:

why:

example:

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



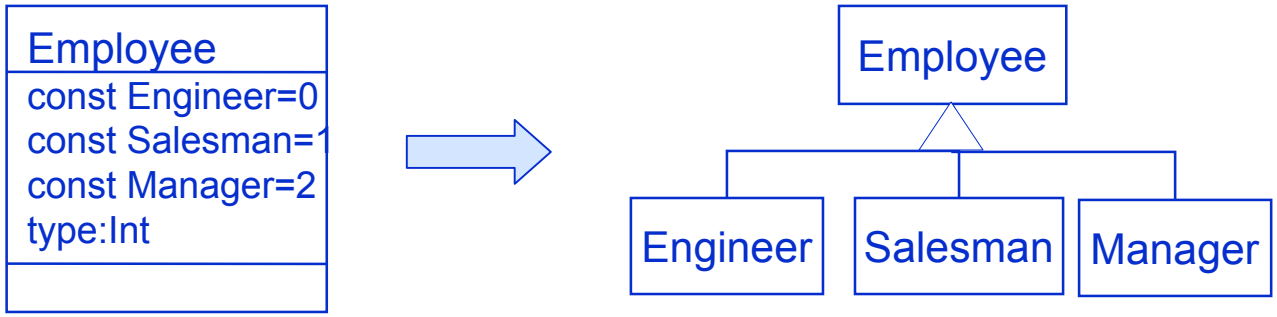
```
private Document doc;
public String getContents() {
  return this.doc.contents; }
public void setContents(String s) {
  this.doc.contents = s; }

public class Document {
  ...
  public String contents;
}
```

categories of refactoring
small refactorings

- organising data
  - Replace type code with subclasses

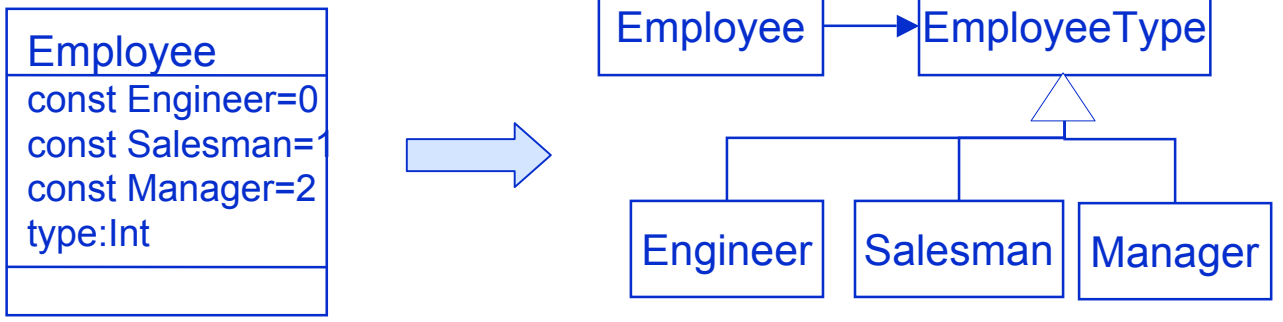
what: an immutable type code affects the behaviour of a class
example:



categories of refactoring
small refactorings

- organising data
  - Replace type code with state/strategy
    - if subclassing cannot be used, e.g., because of dynamic type changes during object lifetime (e.g. promotion of employees)

example:

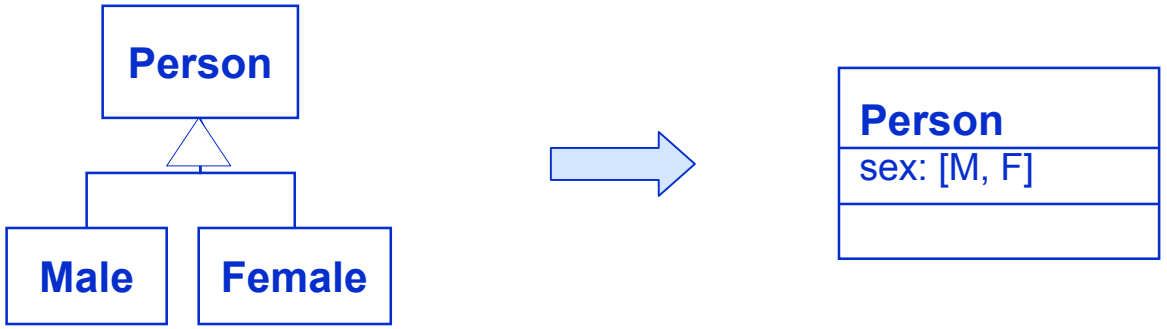


Makes use of **state design pattern** or **strategy design pattern**

categories of refactoring
small refactorings

- organising data
  - Replace subclass with fields

what: subclasses vary only in methods that return constant data
solution: change methods to superclass fields and eliminate subclasses
example:



similar to ***replace inheritance with aggregation***

categories of refactoring
small refactorings

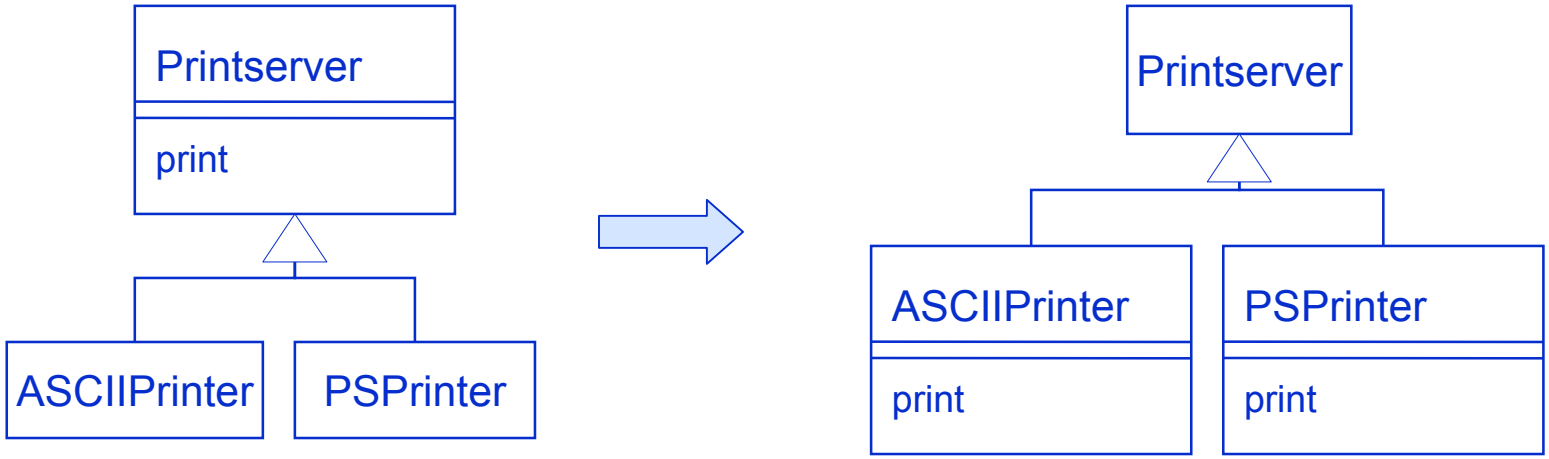
- small refactorings
  - simplifying conditional expressions [8 refactorings]
    - decompose conditional
    - consolidate conditional expression
    - consolidate duplicate conditional fragments
    - remove control flag
    - replace nested conditional with guard clauses
    - replace conditional with polymorphism
    - introduce null objects
    - introduce assertion

categories of refactoring
small refactorings

- small refactorings
  - dealing with generalisation [12 refactorings]
    - push down method / field
    - pull up method / field / constructor body
    - extract subclass / superclass / interface
    - collapse hierarchy
    - form template method
    - replace inheritance with delegation (and vice versa)
      - see earlier example of matrices

categories of refactoring
small refactorings

- dealing with generalisation
  - Push Down Method

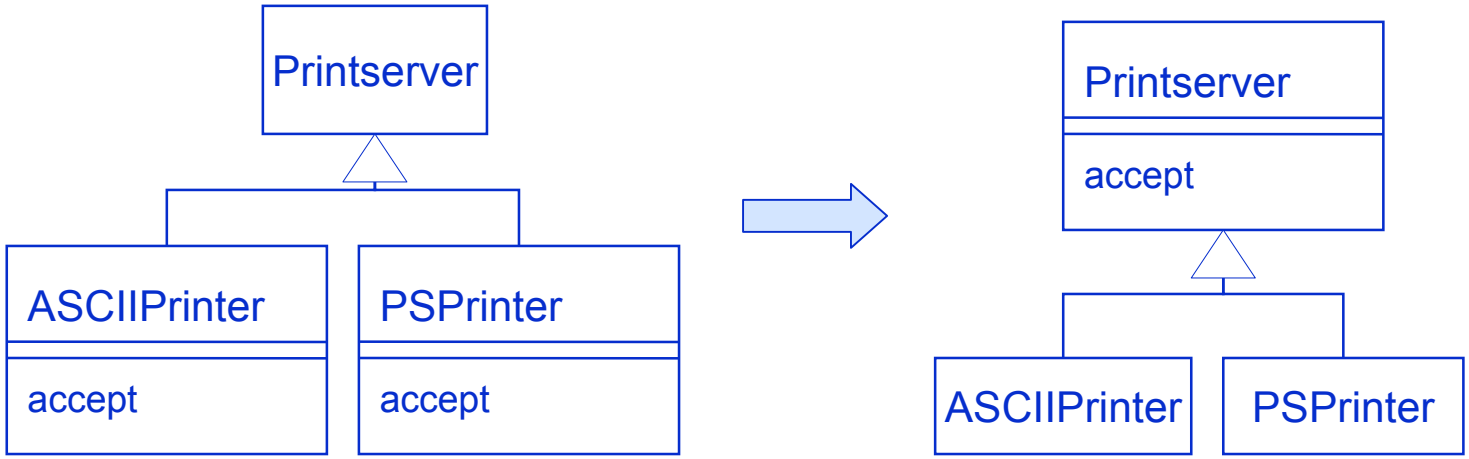


categories of refactoring
small refactorings

- Pull Up Method
  - simplest 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
    - e.g. if a method in each sibling class makes a send to the same 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

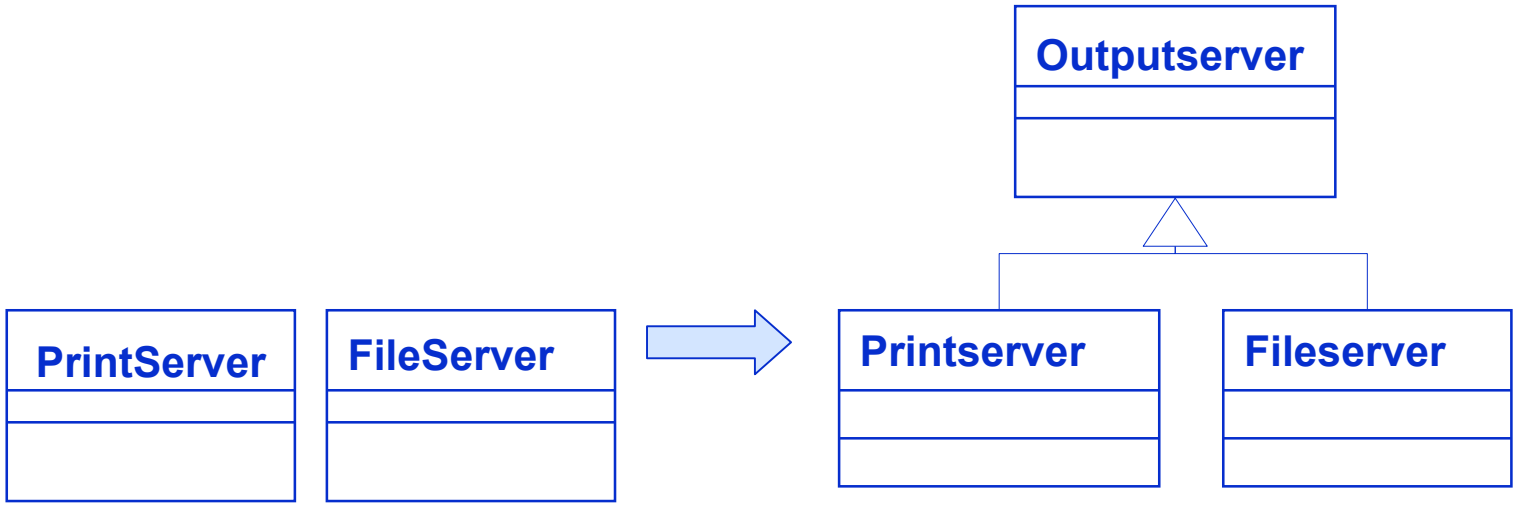
categories of refactoring
small refactorings

- dealing with generalisation
  - Pull Up Method



categories of refactoring
small refactorings

- dealing with generalisation
  - Extract Superclass
    - when you have 2 classes with similar features

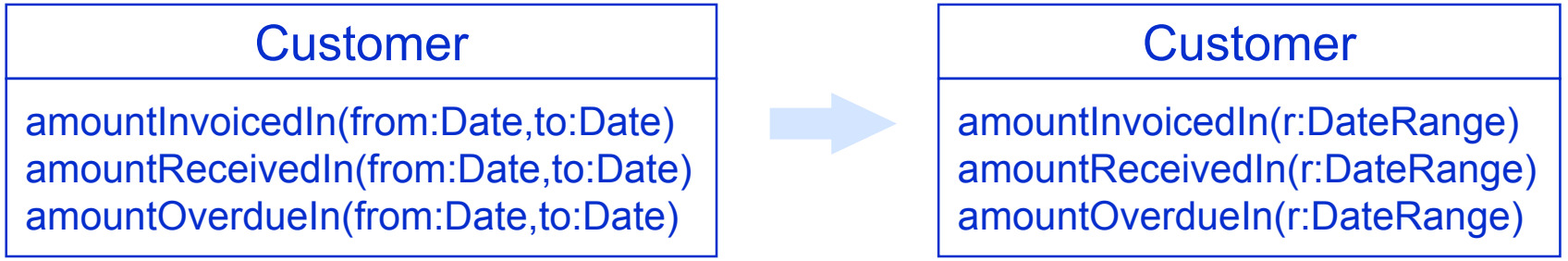


categories of refactoring
small refactorings

- small refactorings
  - simplifying method calls [15 refactorings]
    - rename method
    - add/remove parameter
    - separate query from modifier
    - parameterize method
    - replace parameter with method
    - preserve whole object
    - introduce parameter object
    - remove setting method
    - hide method
    - replace constructor with factory method
    - encapsulate downcast
    - replace error code with exception
    - replace exception with test

categories of refactoring
small refactorings

- simplifying method calls
  - Introduce Parameter Object
    - group parameters that belong together in a separate object



categories of refactoring
small refactorings

- simplifying method calls
  - 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:

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

```
void withdraw(int amount) throws BalanceException {
  if (amount > balance) throw new BalanceException();
  balance -= amount;
}
```

categories of refactoring
big refactorings

- characteristics
  - require a large amount of time (> 1 month)
  - require a degree of agreement among the development team
  - no instant satisfaction, no visible progress
- different kinds
  - tease apart inheritance
  - extract hierarchy
  - convert procedural design into objects
  - separate domain from presentation

categories of refactoring
big refactorings

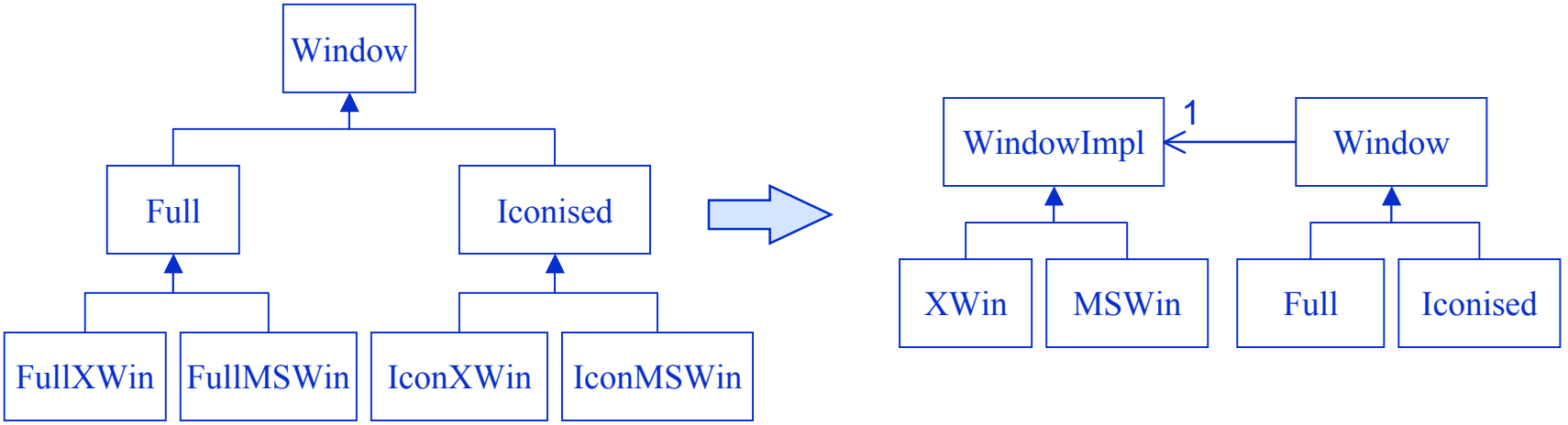
Tom Mens, UMH, 2005

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
- 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
    - eliminate unnecessary (empty) subclasses in original hierarchy
    - apply further refactorings (e.g. pull up method/field)

categories of refactoring
big refactorings

Tease apart inheritance



- Related design patterns
  - Bridge
    - decouples an abstraction from its implementation so that the two can vary independently
      - e.g. see above; matrices – arrays; …
  - Strategy / Visitor / Iterator / State

categories of refactoring
big refactorings

Tom Mens, UMH, 2005

Extract hierarchy

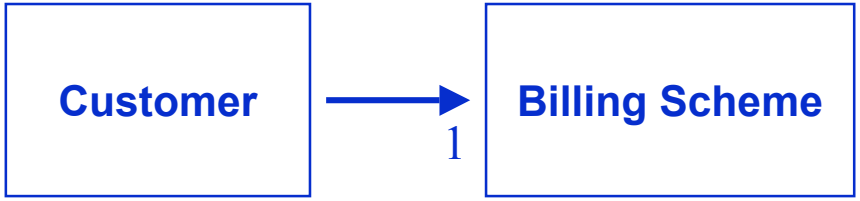
- Problem
  - an overly-complex class that is doing too much work, at least in part through many conditional statements
- Solution
  - turn class into a hierarchy where each subclass represents a special case
- Approach
  - create a subclass for each special case
  - use one of the following refactorings 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

categories of refactoring
big refactorings

Tom Mens, UMH, 2005

Extract hierarchy

- Example
  - calculating electricity bills
  - lots of conditional logic to cover many cases:
    - different charges for summer/winter
    - different billing plans for personal/business/government/…
    - different tax rates
    - reduced rates for persons with disabilities or social security



categories of refactoring
big refactorings

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
- Used small refactorings
  - extract method
  - move method
  - …

categories of refactoring
big refactorings

Tom Mens, UMH, 2005

Separate domain from presentation

- Goal
  - change a two-tier design (user interface/database) into a a three-tier one (UI/business logic/database)
- Solution
  - separate domain logic into separate domain classes
- Used small refactorings
  - extract method
  - move method/field
  - duplicate observed data
  - ...

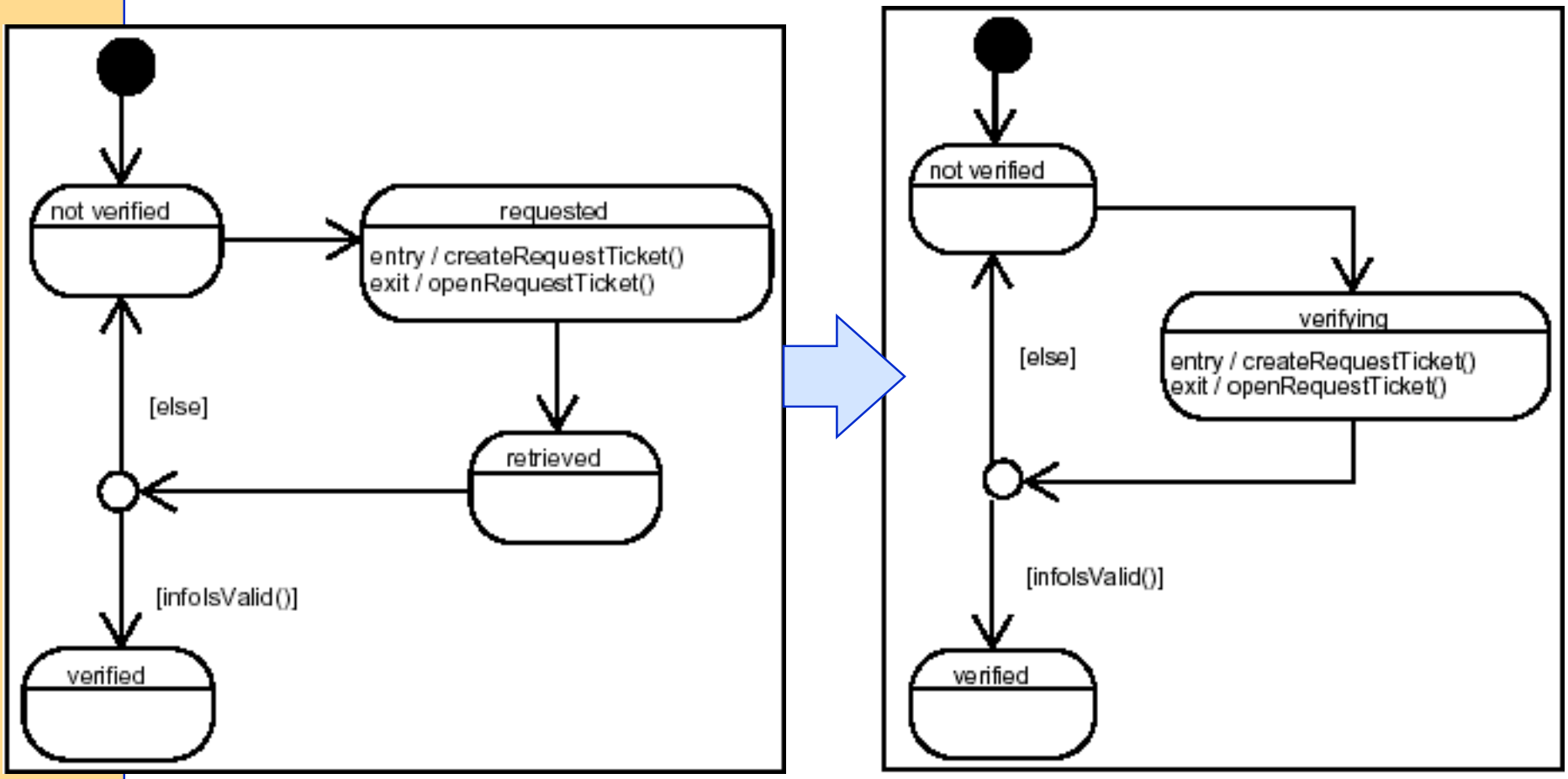
Refactoring of UML design models

refactoring of UML design models

- Refactoring techniques can also be applied at the level of design models
- Reference
  - M. Boger, T. Sturm, P. Fragemann. *Refactoring Browser for UML.*
    - Proc. 3rd Int'l Conf. on eXtreme Programming, pp. 77-81, 2002
  - Proposes refactorings for
    - class diagrams
    - sequence diagrams
    - state-transition diagrams
    - activity diagrams
  - Integated as plug-in for *Poseidon for UML*

refactoring of UML design models state-transition diagrams

- Example refactoring: Merge State



refactoring of UML activity diagrams

- Identified and implemented by [Boger&al2002]

- Make actions concurrent
  - Create a fork and a join pseudostate, and move several sequential groups of actions between them, thus enabling their concurrent execution
- Sequentialize concurrent actions
  - Removes a pair of fork and join pseudostates, and links the enclosed group of actions to another

refactoring of UML activity diagrams
Make Actions Concurrent