Génie Logiciel et Gestion de Projets

Refactoring

Make it Work, Make it Right, Make it Fast

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

Roadmap

- What is refactoring?
- Why is it necessary?
- Some Examples
- When to refactor?
  - Bad Smells
- Catalog of Refactorings
- Tool Support
- Obstacles to refactoring.

Refactoring??

Refactoring??

- The process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure. [Fowl99]

- A behaviour-preserving source-to-source program transformation. [Robe99]

- A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ... [Beck99]

Typical Refactorings

Class Refactorings	**Method Refactorings**	**Attribute Refactorings**
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
	push method down	push variable down
	push method up	pull variable up
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

Why Refactoring?

Why Refactoring?

- "Grow, don't build software" (Fred Brooks)
- Some argue that good design does not lead to code needing refactoring ...

Why Refactoring?

- In reality
  - Extremely difficult to get the design right the first time.
  - You cannot fully understand the problem domain.
  - You cannot fully understand user requirements.
  - You cannot really plan how the system will evolve.
  - Original design is often inadequate.
  - System becomes brittle, difficult to change.

Why Refactoring?

- Refactoring helps you to
  - Manipulate code in a safe environment
    - Behaviour preserving
  - Recreate a situation where evolution is possible
  - Understand existing code
- Remember: software needs to be maintained
  - This is one way to do it safely

Some Examples

A First Example

```
void CVideoAppUi::HandleCommandL(TInt aCommand)
    {
    switch (aCommand){
       case EAknSoftkeyExit:
        case EAknSoftkeyBack:
        case EEikCmdExit:{Exit();  break;}
          // Play command is selected
       case EVideoCmdAppPlay:{DoPlayL(); break; }
         // Stop command is selected
       case EVideoCmdAppStop:{DoStopL(); break; }
         // Pause command is selected
       case EVideoCmdAppPause:{DoPauseL(); break; }
        // DocPlay command is selected
       case EVideoCmdAppDocPlay:{DoDocPlayL(); break; }
         // File info command is selected
       case EVideoCmdAppDocFileInfo:{DoGetFileInfoL();
              break; }
  ......
```

Replace case by Polymorphism

```
void CVideoAppUi::HandleCommandL(Command aCommand){
        aCommand.execute();
}
```

- Create a Command class hierarchy, consisting of a (probably) abstract class AbstractCommand, and subclasses for every command supported. Implement execute on each of these classes:

```
virtual void AbstractCommand::execute() = 0;
virtual void PlayCommand::execute() { ... do play
command ...};
virtual void StopCommand::execute() { ... do stop
command ...};
virtual void PauseCommand::execute() { ... do pause
command ...};
virtual void DocPlayCommand::execute() { ... do docplay
command ...};
virtual void FileInfoCommand::execute() { ... do file
info command ...};
```

Added advantage

- These case statements occur wherever the command integer is used in the original implementation
  - So you will quickly assemble a whole set of useful methods for these commands
    - Moreover, commands are then full-featured classes so they can share code, be extended easily without impacting the client, ...
    - They can also be used when adding more advanced functionalities such as undo etc.
- Have you noticed that the methods are shorter ?

Methods Returning Multiple Values

- Methods can only have a single return value
  - ever felt the need to return multiple values?
- Example

```
public class Text{
    public ?? fontAndColourOfSelection(Position
                                       position)
  " Want to return both font and colour!
    How to do this ?! "
```

Solution 1: Multiple methods

```
public class Text{
  public Font fontOfSelection(Positon position){
      “Return font of current selection”
  }
 public Colour colourOfSelection(Position position)
      “Return colour of current selection”
  }
}
public class Paragraph{
  public void copyStyleFrom(Text aText){
      this.setFont(aText.fontOfSelection(1));
      this.setColour(aText.colourOfSelection(1));
}
```

So now client needs to know that font and colour need to be set, and this needs to be done through individual methods

Solution 2 : use temp value

- "Can use an enum type, or an array, or a collection, etc."

```
public class Text{

  public Array fontAndColourOfSelection(Position
                                        position)
    "Return array with font as first element,
    and colour as second element"
}

public class Paragraph{

  public void copyStyleFrom(Text aText){
    Array result = aText.fontAndColourOfSelection(1);
    this.setFont(result[1]);
    this.setColour(result[2]);
}
```

Even worse: coupling between Text and Paragraph

In the case of using a data structure that relies on order (like an Array), the order information is implicit!

Adding a Missing Object

- When you need methods that return multiple values, it means you are missing appropriate objects
  - Create new class,
  - instance variables are the different values you would like to return.

Creating Extra Class

```
public class Style{
  private Colour colour;
  private Font font;

  public Style(Font aFont, Colour aColour){
     font = aFont;
     colour = aColour;
....}

public class Text{
  public Style styleAt(Position position){
   return new Style(this.fontAt(position),
                          this.colourAt(position));
}
```

Creating Extra Class

```
public class Paragraph{

  public void copyStyleFrom(Text aText){
     Style textStyle = aText.styleAt(1);
     this.setFont(textStyle.getFont());
     this.setColour(textStyle.getColour());
  }
```

Guardian Code Example

```
void CVideoAppUi::DynInitMenuPaneL(
    TInt aResourceId,CEikMenuPane* aMenuPane)
    {
    if ( aResourceId == R_VIDEO_MENU )
            {
            // Check whether the database has been created or not
            if ( iEngine->GetEngineState() != EPPlaying )
                {
                // The video clip is not being played
                aMenuPane->SetItemDimmed( EVideoCmdAppStop, ETrue );
                aMenuPane->SetItemDimmed( EVideoCmdAppPause, ETrue );
                }

            // If there is no item in the list box, hide the play, docplay
            // and file info menu items
            if ( !iAppContainer->GetNumOfItemsInListBox() )
                {
                aMenuPane->SetItemDimmed( EVideoCmdAppDocFileInfo, ETrue );
                aMenuPane->SetItemDimmed( EVideoCmdAppDocPlay, ETrue );
                aMenuPane->SetItemDimmed( EVideoCmdAppPlay, ETrue );
                }
        }
```

Guardian statement

} http://www.forum.nokia.com/info/sw.nokia.com/id/ee56cba2-0b78-45c9-831f-69c5007652fe/Video_Example_v1_0.zip.html

Guardian Code: Switching it around...

return when condition not met

```
void CVideoAppUi::DynInitMenuPaneL(
    TInt aResourceId,CEikMenuPane* aMenuPane)
    {
    if ( aResourceId != R_VIDEO_MENU ) {return };

   // Check whether the database has been created or not
   if ( iEngine->GetEngineState() != EPPlaying )
          {
          // The video clip is not being played
          aMenuPane->SetItemDimmed( EVideoCmdAppStop, ETrue );
          aMenuPane->SetItemDimmed( EVideoCmdAppPause, ETrue );
          }

  // If there is no item in the list box, hide the play, docplay
  // and file info menu items
  if ( !iAppContainer->GetNumOfItemsInListBox() )
    {
    aMenuPane->SetItemDimmed( EVideoCmdAppDocFileInfo, ETrue );
    aMenuPane->SetItemDimmed( EVideoCmdAppDocPlay, ETrue );
    aMenuPane->SetItemDimmed( EVideoCmdAppPlay, ETrue );
    }
   }
```

Split Using the Comments

```
void CVideoAppUi::DynInitMenuPaneL(
    TInt aResourceId,CEikMenuPane* aMenuPane)
    {
    if ( aResourceId != R_VIDEO_MENU ) {return };
```

dimButtonsWhenNotPlaying

```
// Check whether the database has been created or not
if ( iEngine->GetEngineState() != EPPlaying )
        {
        // The video clip is not being played
        aMenuPane->SetItemDimmed( EVideoCmdAppStop, ETrue );
        aMenuPane->SetItemDimmed( EVideoCmdAppPause, ETrue );
        }
```

```
// If there is no item in the list box, hide the play, docplay
// and file info menu items
if ( !iAppContainer->GetNumOfItemsInListBox() )
  {
  aMenuPane->SetItemDimmed( EVideoCmdAppDocFileInfo, ETrue );
  aMenuPane->SetItemDimmed( EVideoCmdAppDocPlay, ETrue );
  aMenuPane->SetItemDimmed( EVideoCmdAppPlay, ETrue );
  }
 }
```

dimButtonsWhenNoItem

Splitting results

```
void CVideoAppUi::DynInitMenuPaneL(TInt aResourceId,CEikMenuPane* aMenuPane)
{
   if ( aResourceId != R_VIDEO_MENU ) {return };
   dimButtonsWhenNotPlaying(aMenuPane);
   dimButtonsWhenNoItem(aMenuPane);
};

void CVideoAppUi::dimButtonsWhenNotPlaying(CEikMenuPane* aMenuPane)
{
   if ( iEngine->GetEngineState() != EPPlaying ) {
        aMenuPane->SetItemDimmed( EVideoCmdAppStop, ETrue );
        aMenuPane->SetItemDimmed( EVideoCmdAppPause, ETrue );
    }
}

void CVideoAppUi::dimButtonsWhenNoItem(CEikMenuPane* aMenuPane)
{
  if ( !iAppContainer->GetNumOfItemsInListBox() ) {
        aMenuPane->SetItemDimmed( EVideoCmdAppDocFileInfo, ETrue );
        aMenuPane->SetItemDimmed( EVideoCmdAppDocPlay, ETrue );
        aMenuPane->SetItemDimmed( EVideoCmdAppPlay, ETrue );
  }
}
```

More things that could be done

```
void CVideoAppUi::DynInitMenuPaneL(TInt aResourceId,CEikMenuPane* aMenuPane)
{
   if ( aResourceId != R_VIDEO_MENU ) {return };
   dimButtonsWhenNotPlaying(aMenuPane);
   dimButtonsWhenNoItem(aMenuPane);
};

void CVideoAppUi::dimButtonsWhenNotPlaying(CEikMenuPane* aMenuPane)
{
   if ( iEngine->GetEngineState() != EPPlaying ) {
                aMenuPane->DimItem( EVideoCmdAppStop);
                aMenuPane->DimItem( EVideoCmdAppPause);
   }
}

void CVideoAppUi::dimButtonsWhenNoItem(CEikMenuPane* aMenuPane)
{
  if ( !iAppContainer->GetNumOfItemsInListBox() ) {
                aMenuPane->DimItem( EVideoCmdAppDocFileInfo);
                aMenuPane->DimItem( EVideoCmdAppDocPlay);
                aMenuPane->DimItem( EVideoCmdAppPlay);
  }
}
```

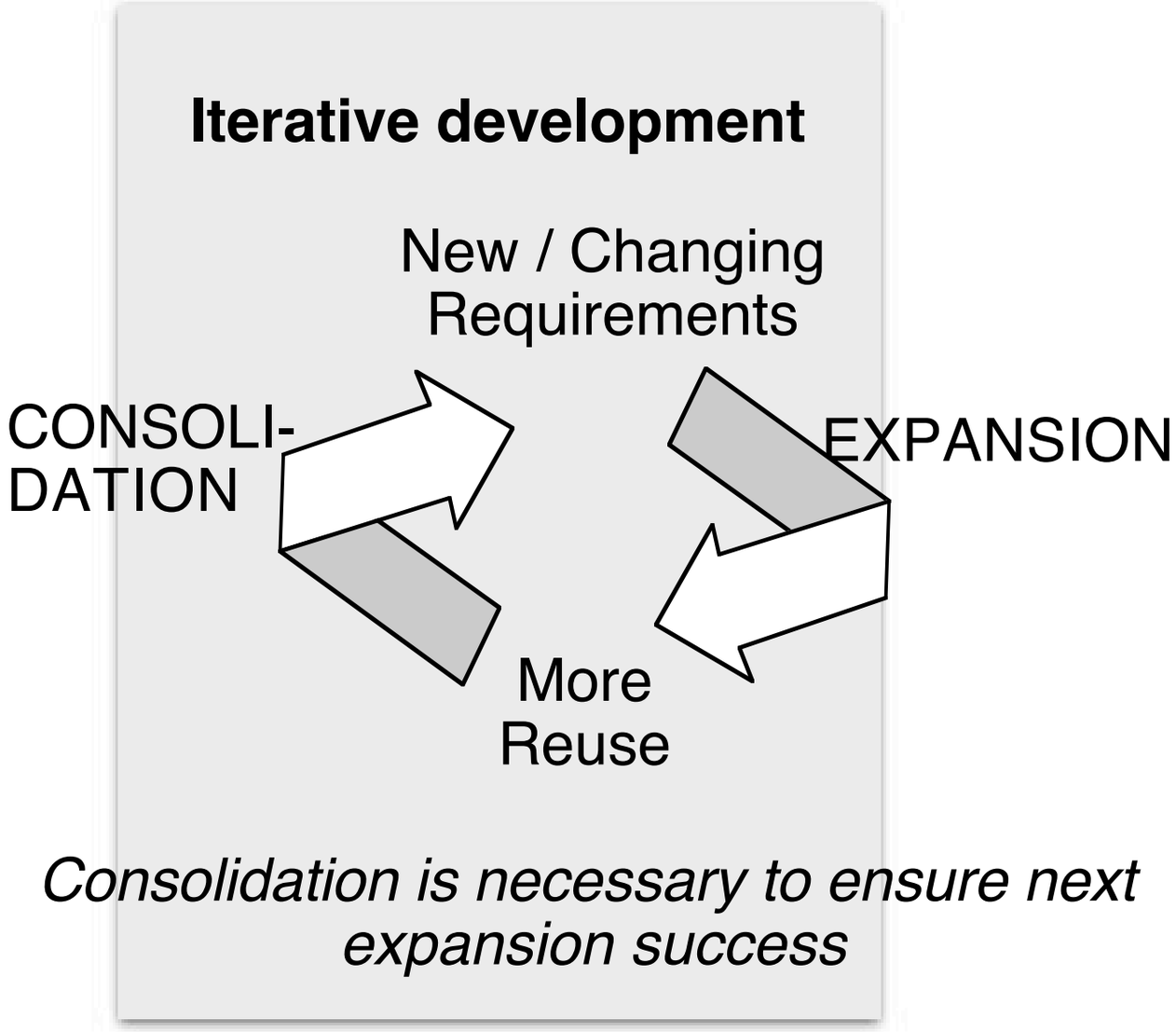
Removing Duplicated Code

- In the same class:
  - Extract Method
- Between two sibling subclasses:
  - Extract Method
  - Push identical methods up to common superclass
  - Form Template Method
- Between unrelated class:
  - Create common superclass
  - Move to Component
  - Extract Component (e.g., Strategy)

When to Refactor?

When to refactor?

- Two essential phases in the iterative software development approach:
  - 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 model



When 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.

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

Some Guidelines

- During debugging
  - If it is difficult to trace an error, refactor to make the code more comprehensible.
- During formal code inspections (code reviews)
- Identify bad smells in the source code [Beck99a]
  - “structures in the code that suggest (sometimes scream for) the possibility of refactoring”

Bad Code Smells

- "If it stinks, change it" [Grandma Beck]
  - Duplicated Code
  - Long Method
  - Large Class (Too many responsibilities)
  - Long Parameter List (Object is missing)
  - Case Statement (Missing polymorphism)
  - Shotgun Surgery (Little changes distributed over too much objects)

Bad Code Smells (ctd)

- Some more...
  - Feature Envy (Method needing too much information from another object)
  - Data Classes (Only accessors)
  - Data Clumps (Data always use together (x,y -> point))
  - Parallel Inheritance Hierarchies (Changes in one hierarchy require change in another hierarchy)
  - Middle Man (Class with too much delegating methods)
  - Temporary Field

Catalog of Refactorings [Fowler]

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

Catalog of Refactorings [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]

Catalog of Refactorings [Fowler]

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

Format of Each Refactoring

- name,
- summary,
- motivation,
- mechanics,
- examples.

Small Refactoring: (de)composing methods

(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

Extract Method

- You have a code fragment that can be grouped together

```
void printOwing(double amount){
  printBanner();
 //print details
 System.out.println(“name:” + _name);
 System.out.println(“amount” + amount);
}

void printOwing(double amount){
  printBanner();
  printDetails(amount);
}
void printDetails(double amount){
 System.out.println(“name:” + _name);
 System.out.println(“amount” + amount);
}
```

Extract Method

- Motivation
  - short, well-named methods
    - increases reuse,
    - easier to override.

Mechanics Extract Method

- create a new method, name it after the intention of the method
- copy extracted code from source method into the new target method
- scan the extracted code for references to any variables that are local in the scope of the source method. These are local variables and parameters to the method.
- see whether any temporary variables are used only within this extracted code. If so, declare them in the target method.

Mechanics Extract Method

- you may need to split temporary variable. You can eliminate temporary variables with *Replace Temp with Query* (see example)
- pass into the target method as parameters local-scope variables that are read from the extracted code.
- compile
- replace extracted code in the source method with a call to the target method. Remove any temporary variables if necessary
- compile and test

Example Extract Method

```
void printOwing(){
Enumeration e = orders.elements();
double outstanding = 0.0;
printBanner();

//calculate outstanding
while(e.hasMoreElements()){
Order each = (Order) e.nextElement();
outstanding += each.getAmount();
}

//print details
System.out.println("name" + name);
System.out.println("amount" + outstanding);

}
```

Example Extract Method

```
void printOwing(){
Enumeration e = orders.elements();
double outstanding = 0.0;
printBanner();

//calculate outstanding
while(e.hasMoreElements()){
Order each = (Order) e.nextElement();
outstanding += each.getAmount();
}

printDetails(outstanding);
}
```

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

Example Extract Method

```
void printOwing(){
printBanner();
double outstanding = getOutstanding();
printDetails(outstanding);
}

//calculate outstanding
Enumeration e = orders.elements();
while(e.hasMoreElements()){
Order each = (Order) e.nextElement();
outstanding += each.getAmount();
}
```

```
double getOutstanding(){
Enumeration e = orders.elements();
double outstanding = 0.0;
while(e.hasMoreElements()){
Order each = (Order) e.nextElement();
outstanding += each.getAmount();
}
return outstanding;
}
```

Example Extract Method

```
void printOwing(double previousAmount){
Enumeration e = orders.elements();
double outstanding = previousAmount * 1.2;
printBanner();

//calculate outstanding
while(e.hasMoreElements()){
Order each = (Order) e.nextElement();
outstanding += each.getAmount();
}

printDetails(outstanding);
}
```

Example Extract Method

```
void printOwing(double previousAmount){
double outstanding = previousAmount * 1.2;
printBanner();

outstanding = getOutstanding(outstanding);

printDetails(outstanding);
}
```

```
double getOutstanding(double intialValue){
double result = initialValue;
Enumeration e = orders.elements();
//calculate outstanding
while(e.hasMoreElements()){
Order each = (Order) e.nextElement();
result += each.getAmount();
}
return result;
}
```

Inline Method

- 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.
- Motivation: to remove too much indirection and delegation.
- Example:

```
int getRating(){
  return moreThanFiveLateDeliveries();
}

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

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

Split Temporary Variable

- 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.
- Motivation: 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);
```

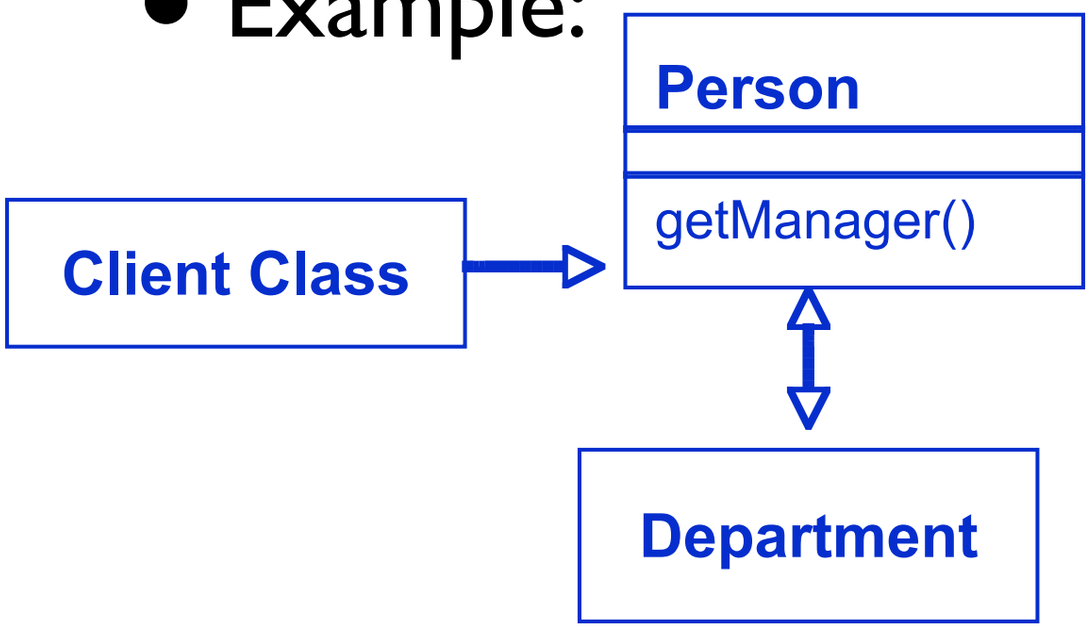
Small Refactorings: Moving features between objects

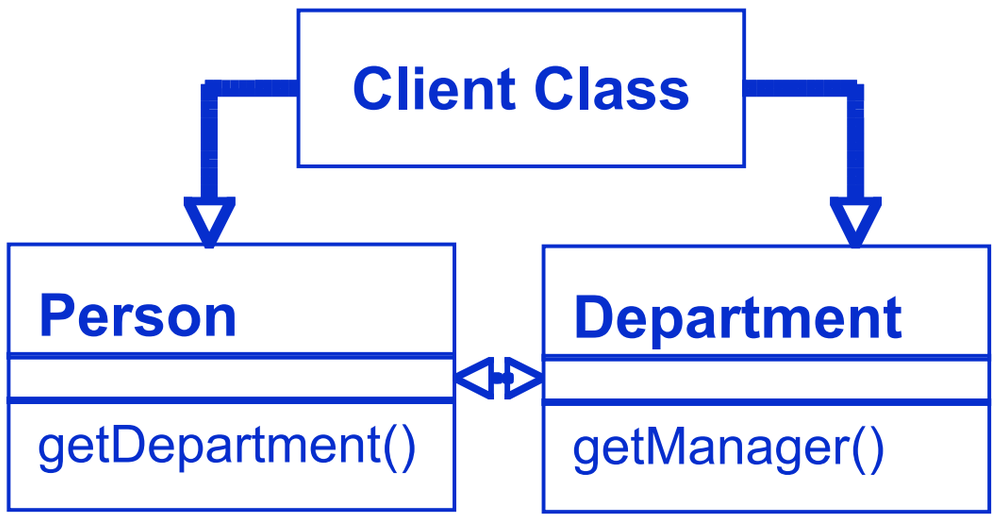
Moving features between objects

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

Remove Middle Man

- When a class is doing too much simple delegation, get the client to call the delegate directly.
- Motivation: to remove too much indirection (as a result of other refactorings).
- Example:





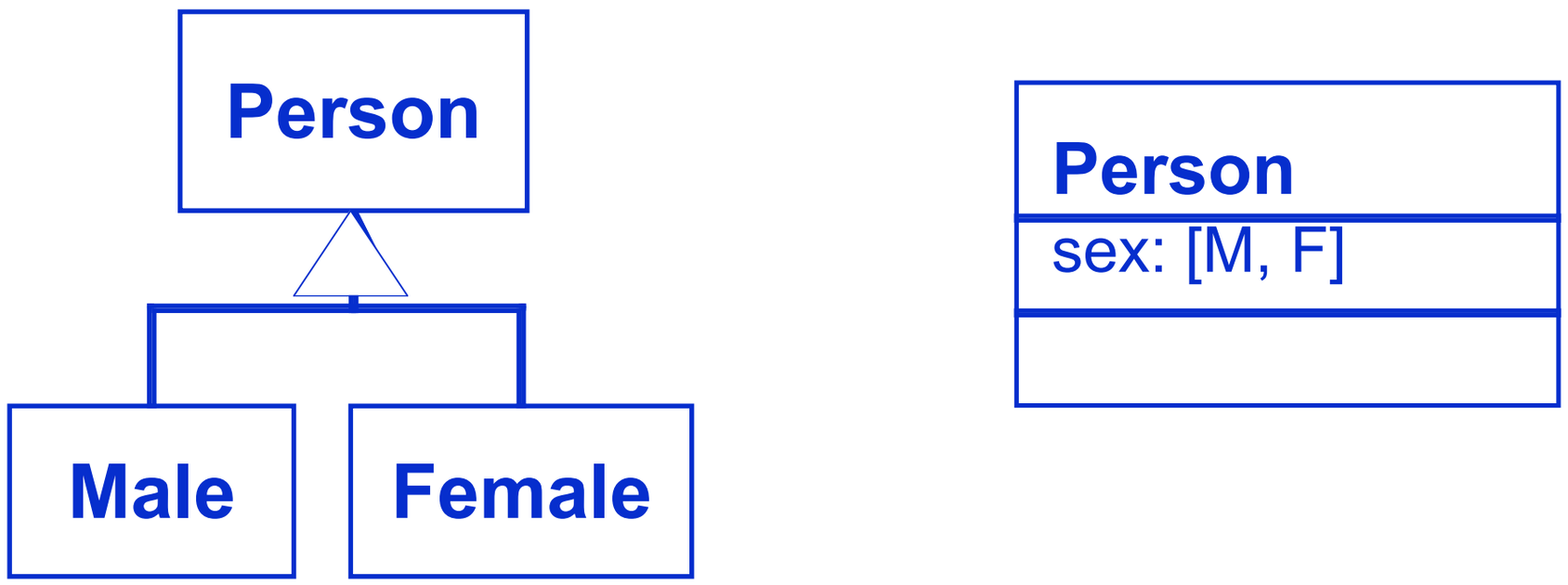
Small Refactorings: Organizing data

Organizing data

- 16 refactorings
  - encapsulate field
  - 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

Replace Subclass with Fields

- Subclasses vary only in methods that return constant data.
- Solution: change methods to superclass fields and eliminate subclasses
- Example:



Small Refactorings: Simplifying Conditional Expressions

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

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 (cf. A First Example)
  - introduce null objects
  - introduce assertion

Small Refactorings: Dealing with generalization

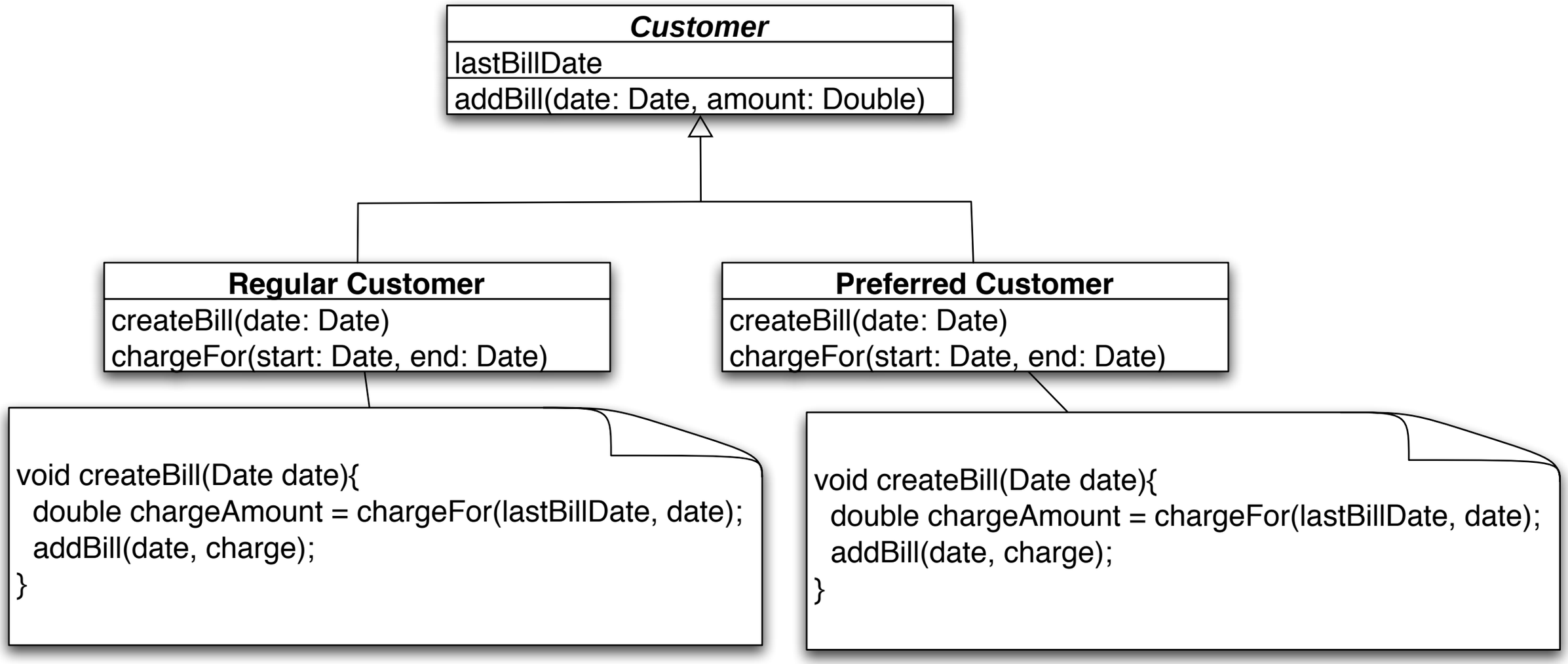
Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

Dealing with generalization

- 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)

Pull Up Method

- You have methods with identical results on subclasses
  - solution: move them to the superclass



Pull Up Method

- Preconditions
  - Source method does not refer to any variables declared in subclass
  - No method with same signature as source method may exist in target class
  - Ensure methods are identical.
  - If methods have different signature, change the signature to the one you want to use in the superclass

Mechanics Pull Up Method

- Mechanism
  - create a new method in the superclass, copy the body of one of the methods to it, adjust and compile
  - Delete one subclass method
  - compile and test
  - keep deleting and testing until only the superclass method remains
  - Take a look at the callers of this method to see whether you can change a required type to the superclass

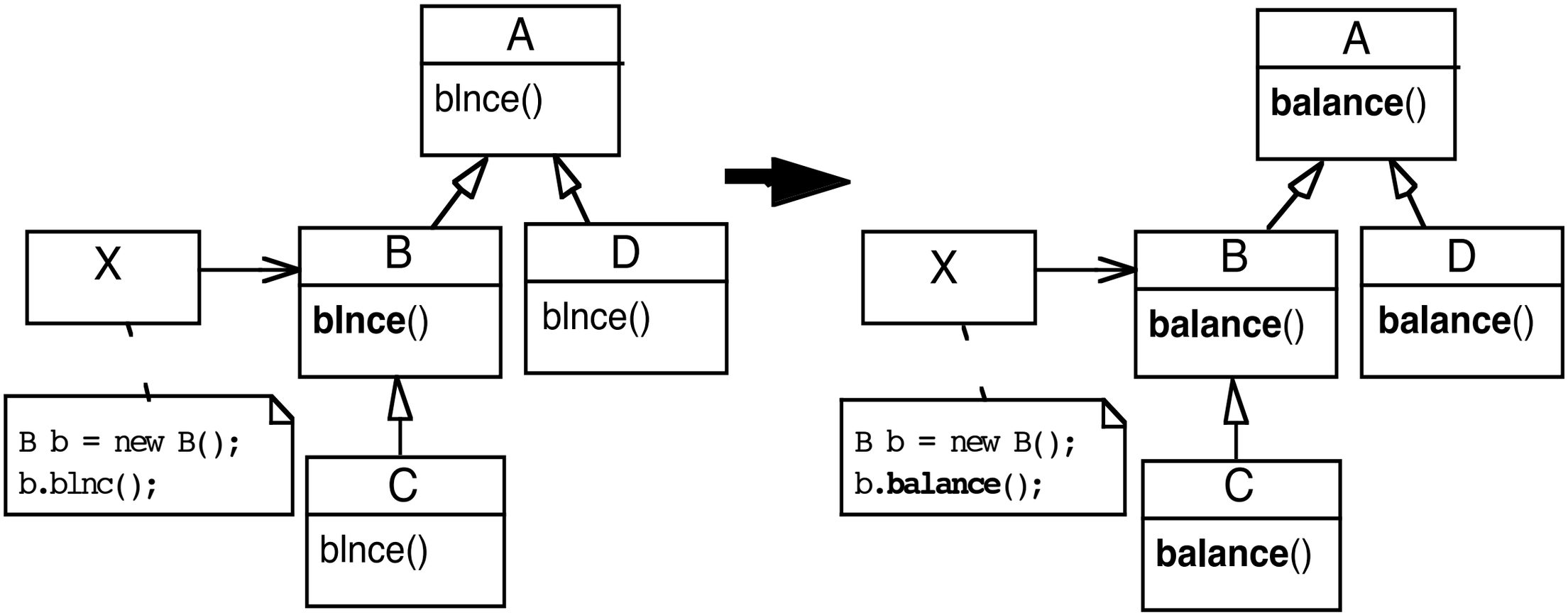
Small Refactorings
Simplifying method calls

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

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

Rename Method



Mechanics Rename Method

- Preconditions
  - no method exists with the signature implied by new name in the inheritance hierarchy that contains method.
- Mechanism
  - Declare new method with new name. Copy the old body of code and make any alterations to fit
  - compile
  - Change the body of the old method so that it calls the new one
  - compile and test

Mechanics Rename Method

- Mechanism (continued)
  - find all references to the old method name and change them to refer to the new one. Compile and test after every change.
  - remove old method.
  - compile and test
- PostConditions
  - method has new name
  - relevant methods in the inheritance hierarchy have new name
  - invocations of changed method are updated to new name

Big Refactorings

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

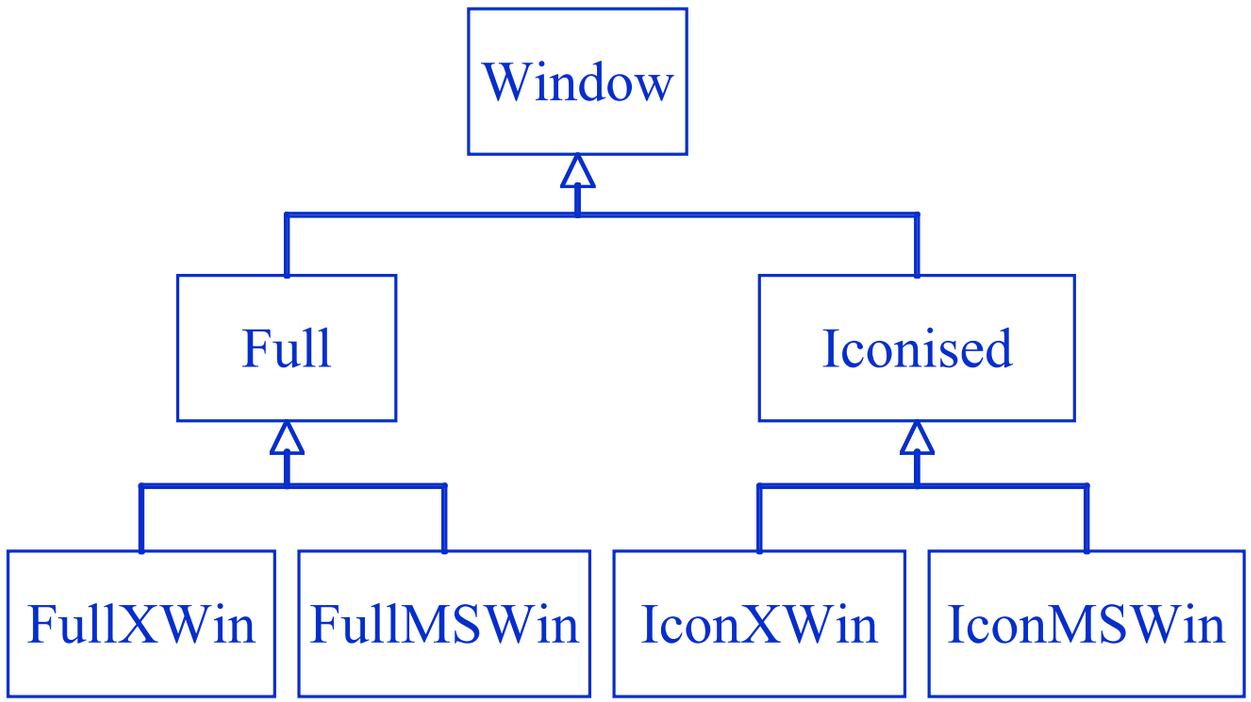
Big Refactorings

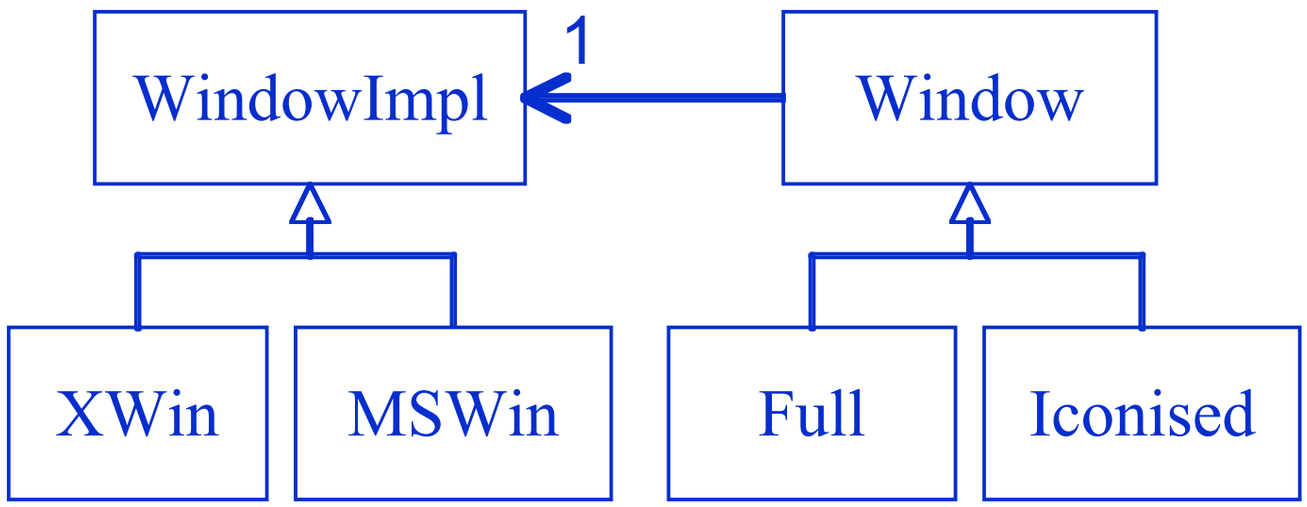
- characteristics
  - require a large amount of time (> 1 month)
  - require a degree of agreement among the development team
  - no instant satisfaction, no visible progress
- different kinds
  - tease apart inheritance
  - extract hierarchy
  - convert procedural design into objects
  - separate domain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

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)

Tease apart inheritance





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
- Use small refactorings:
  - extract method
  - move method
  - …

Refactorings due to Language Evolution

[Kiezunetal]

Context

- Bad code smells are not the only reason to refactor
- Another reason is:
  - Programming language evolution
- For example:
  - introduction of generics in Java 5.

Upgrade to use Generics: Technical Problems

- parameterization problem,
  - adding type parameters to an existing class definition so that it can be used in different context without loss of type information.

```
class ArrayList {...} => class ArrayList<T> {...}
```

- instantiation problem
  - the task of determining the type arguments that should be given to instances of the generic class in client code.

```
ArrayList names => ArrayList<String> names
```

```
public class MultiSet{
 private Map counts = new HashMap();
 public void add(Object t1){
    counts.put(t1, new Integer
          (getCount(t1) + 1));
 }
 public Object getMostCommon(){
   return new SortSet(this).getMostCommon();
 }
 public void addAll(Collection c1){
  for (Iterator it = c1.iterator();
              it.hasNext();){
       add(it.next());
 }
 public boolean contains(Object o1){
   return counts.containsKey(o1);
 }
 public boolean containsAll(Collection c)
   {
 return getAllElements().containsAll(c);
 }
 public int getCount(Object o2){
   return (! contains(o2))? 0 :
     ((Integer)counts.get
        (o2)).intValue;
 }
 public Set getAllElements(){
   return counts.keySet();
 }
}
```

```
class SortSet extends TreeSet{

  public SortSet(final MultiSet m){
   super(new Comparator(){
    public int compare(Object o3,
        Object o4){
    return m.getCount(o3) -
       m.getCount(o4);
   }});
   addAll(m.getAllElements());
  }

  public boolean addAll(Collection
                        c3){
   return super.addAll(c3);
  }

  public Object getMostCommon(){
   return isEmpty() ? null :
                      first();
  }
}
```

```
public class MultiSet{
 private Map counts = new HashMap();
 public void add(Object t1){
    counts.put(t1, new Integer
          (getCount(t1) + 1));
 }
 public Object getMostCommon(){
   return new SortSet(this).getMostCommon
        ();
 }
 public void addAll(Collection c1){
  for (Iterator it = c1.iterator();
              it.hasNext();){
       add(it.next());
 }
 public boolean contains(Object o1){
   return counts.containsKey(o1);
 }
 public boolean containsAll(Collection c)
   {
 return getAllElements().containsAll(c);
 }
 public int getCount(Object o2){
   return (! contains(o2))? 0 :
     ((Integer)counts.get
        (o2)).intValue;
 }
 public Set getAllElements(){
   return counts.keySet();
 }
}
```

```
public class MultiSet<T1>{
 private Map<T1, Integer> counts = new
                HashMap<T1, Integer>();
 public void add(T1 t1){
    counts.put(t1, new Integer
          (getCount(t1) + 1));
 }
 public T1 getMostCommon(){
  return new SortSet<T1>
     (this).getMostCommon();
 }
 public void addAll(Collection<? extends
                              T1> c1){
   for (Iterator<? extends T1> it =
        c1.iterator(); it.hasNext();){
       add(it.next());
  }
public boolean containsAll(Collection<?>
                              c){
 return getAllElements().containsAll(c);
 }
public boolean contains(Object o1){
    return counts.containsKey(o1);
 }
public int getCount(Object o2){
   return (! contains(o2))? 0 :
    ((Integer)counts.get(o2)).intValue;
 }
public Set<T1> getAllElements(){
  return counts.keySet();}
}
```

```
class SortSet extends TreeSet{

  public SortSet(final MultiSet m){
   super(new Comparator(){
    public int compare(Object o3,
        Object o4){
    return m.getCount(o3) -
       m.getCount(o4);
   }});
   addAll(m.getAllElements());
  }

  public boolean addAll(Collection
                        c3){
   return super.addAll(c3);
  }

  public Object getMostCommon(){
   return isEmpty() ? null :
                      first();
  }
}
```

```
class SortSet<T2> extends TreeSet<T2>{

  public SortSet(final MultiSet<? extendsT2> m){
   super(new Comparator<T2>(){
    public int compare(T2 o3, T2 o4){
   return m.getCount(o3) - m.getCount(o4);
   }});
   addAll(m.getAllElements());
  }

  public boolean addAll(Collection<? extends T2> c3){
   return super.addAll(c3);
  }

  public T2 getMostCommon(){
   return isEmpty() ? null : first();
  }
}
```

Remarks

- the desired parametrization requires 19 non-trivial changes to the program's type annotatons,
- involves subtle reasoning.

  => class parametrization is a complex process and automated tool assistance is highly desirable

Solution

- The behaviour of any client of the parametrized classes is preserved.
- The translation produces a result similar to that which would be produced manually by a skilled programmer.
- approach admits an efficient implementation that is easy to use.

Tool Support

Tool Support

- Could do refactoring by hand (see Rename Method example)
- But much better if automated:
  - easier
  - safer
- Which tools are needed to support refactoring?

Tool Support for Refactoring

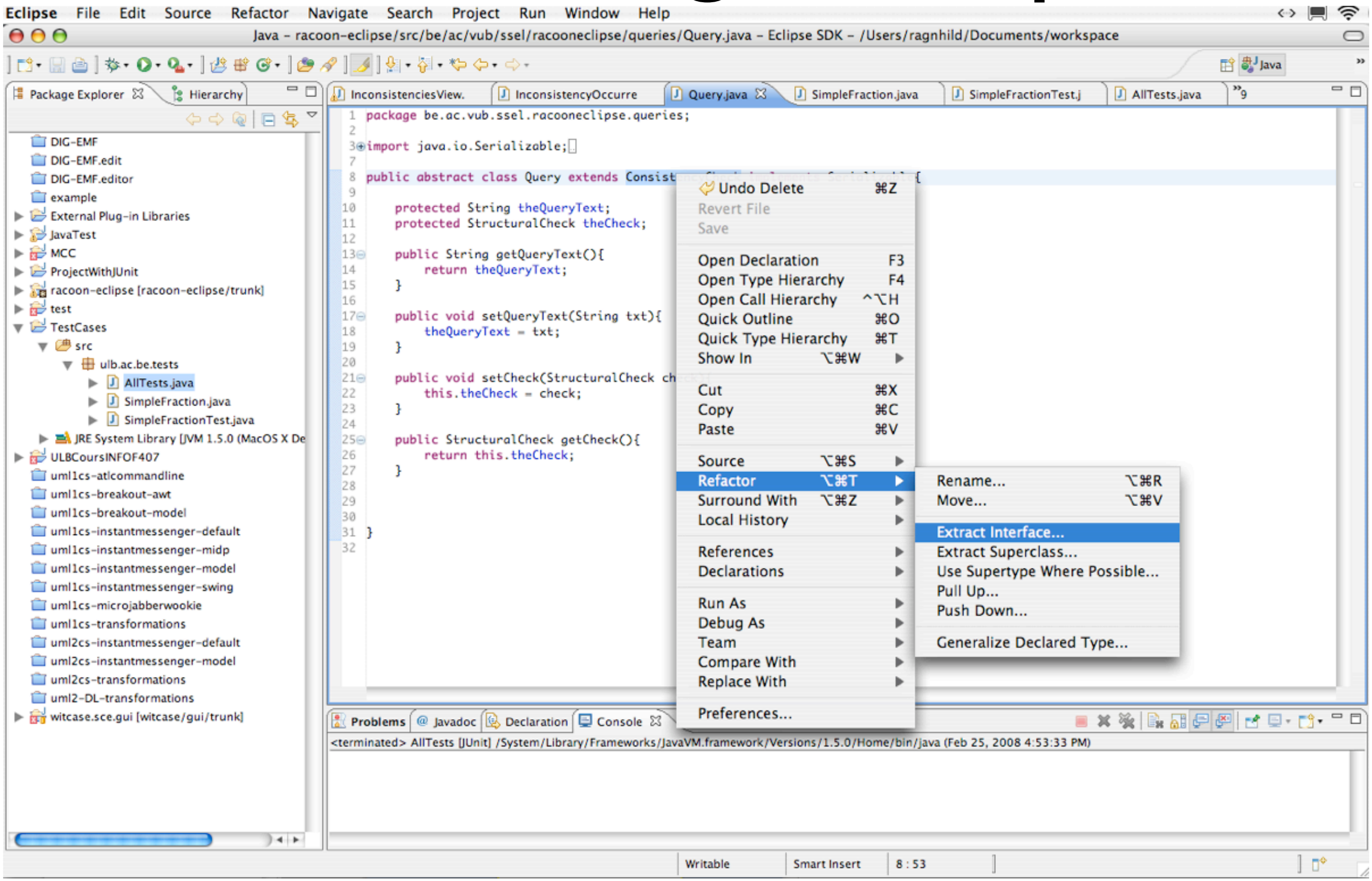
Change Efficiently	**Failure Proof**
Refactoring Tools - source-to-source program transformation - behaviour preserving ⇒ Improve Structure	Regression Testing - Repeating past tests - requires no user interaction - is deterministic ⇒ Verify damage to previous work
Development Environment - Fast edit-compile-run - Integrated in environment ⇒ Convenient	Configuration&Version Management - track different versions - track who did what ⇒ can revert to earlier versions

Tool Support for Refactoring

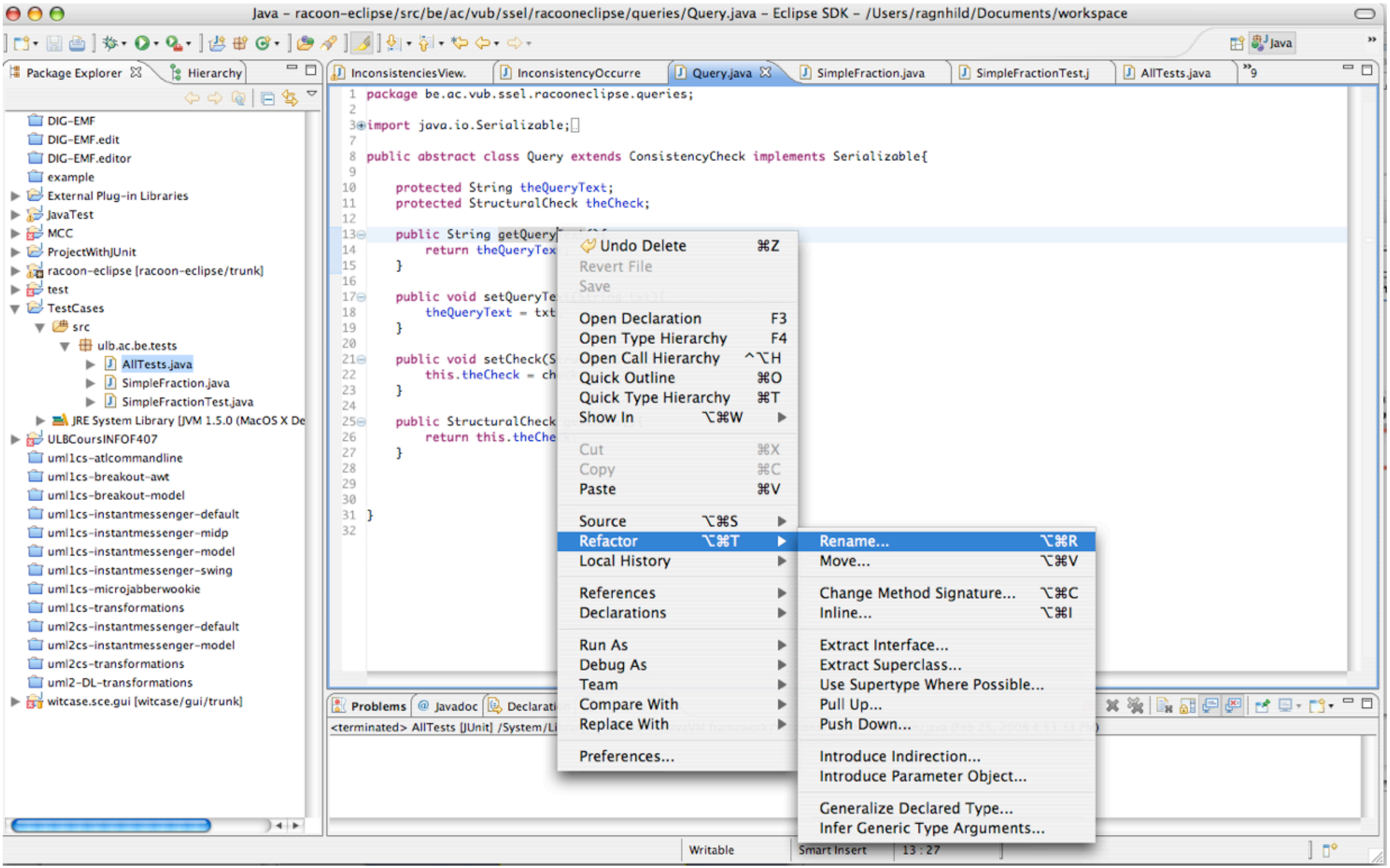
Do not apply refactoring tools in isolation.

- Refactoring activities
  - determine when the software needs to be restructured
  - identify which refactorings need to be applied and where
  - apply the refactorings (including testing and change propagation)
  - verify the intended effect of the refactoring
- Refactoring tool support is often limited to only some of these activities

Refactoring in Eclipse



Refactoring in Eclipse



Obstacles to Refactoring

Obstacles to Refactoring

- Performance issue
  - "Refactoring will slow down the execution"
- Cultural Issues
  - "We pay you to add new features, not to improve the code!"
- If it doesn't break, do not fix it
  - "We do not have a problem, this is our software!"
- Development is always under time pressure
  - Refactoring takes time
  - Refactoring better after delivery
  - Process should take it into account, like testing

Performance Myth

- Don’t think that clean software is slow!
- Normally only 10% of your system consumes 90% of the resources so just focus on 10 %.
  - Refactorings help to localise the part that need change
  - Refactorings help to concentrate the optimisations
- Always use a profiler on your “slow” system to guide your optimisation effort
  - Never optimise first!

Remember This

"Make it Work, Make it Right, Make it Fast"

Tutorial

Ragnhild Van Der Straeten - ULB - Génie logiciel et gestion de projets - 2007/2008

References

- [Fowl99] Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts. Refactoring. Improving the Design of Existing Code. 1999. Addison-Wesley. ISBN 0201485672
- [Beck99a] Kent Beck. Smalltalk Best Practice Patterns. Prentice Hall. 1996. ISBN 013476904X
- [Robe99] D. Roberts. Practical Analysis for Refactoring. Ph.D. Thesis University of Illinois at Urbana-Champaign, 1999

References

- [Kiezunetal] Adam Kiezun, Michael Ernst, Frank Tip and Robert Fuhrer. Refactoring for Parametrizing Classes in the Proceedings of ICSE2007, Minneapolis, USA.
- Refactoring Home Page. www.refactoring.com