SE Software Engineering
The University of Auckland

Quality Assurance
Coding Principles

Part II - Lecture 12

Horror Stories

“Lost Radio Contact Leaves Pilots On Their Own”

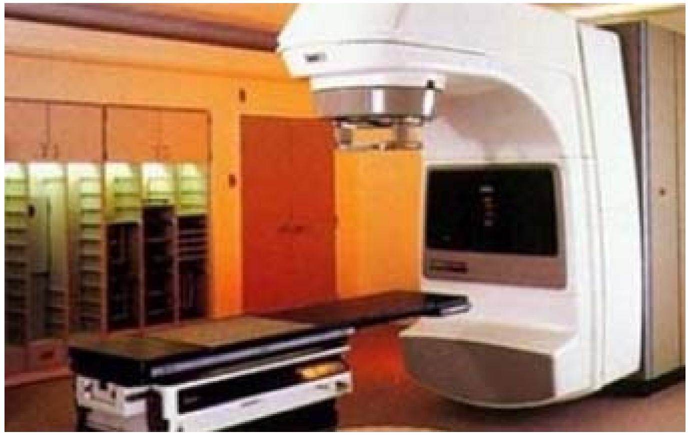
(2004)

http://spectrum.ieee.org/aerospace/aviation/lost-radio-contact-leaves-pilots-on-their-own

“Radiation Deaths linked to AECL Computer Errors”

(1985)

http://www.ccnr.org/fatal_dose.html



Today's Outline

SE Software Engineering The University of Auckland

- Common Java Mistakes
- Java Coding Guidelines
- Refactoring

Common Java Mistakes

CONGRATULATIONS! YOU HAVE JUST DISCOVERED A COMPLETELY NEW ERROR!
GOFF

To err is human,
but to really foul things up
you need a computer.
(Paul Ehrlich)

Accessing Non-Static Members from Static Methods

- **Non-static members** belong to objects
- **Static members** belong to a class
- If you don't have an object you cannot access a non-static member
- **this** refers to the object on which a non-static method is called

```
public class Demo {
  public int x = 1;
  public void m() { }
  public static void main
  (String[] args) {
    int y = x;
    m();
    Object o = this;
} }
```

```
public class Demo {
  public int x = 1;
  public void m() { }
  public static void main
  (String[] args) {
    Demo d = new Demo();
    int y = d.x;
    d.m();
    Object o = d;
} }
```

Mistyped Method Name when Overriding

SE Software Engineering The University of Auckland

- Java supports **method polymorphism** through overriding
  - A superclass **A** defines a method **m**
  - A subclass **B** of **A** can define its own **m**, overriding the definition in **A**
  - The type of the object on which **m** is called decides which version of **m** is used (the one of **A** or the one of **B**)
- Problem: when method definition in subclass uses method name different from method in superclass, overriding does not work
- Symptom: a method doesn't get called; no compiler warning
- Found by tracing control flow of a program or use **@Override**

```
public class Demo extends WindowAdapter {
  // This should be "windowClosed" !!!
  public void windowClose(WindowEvent e) {
      System.exit(0);
} }
```

Program does not stop after closing window!!!

Overriding with Different Semantics

- Make sure that any method that you override preserves the semantics of the original
- Otherwise: possibly strange behaviour in program parts that seemed to work all right
- Example: using NZ together with German GST code; incompatible semantics!!!

```
public class Product {
  public double grossPrice;
  public double netPrice() {
    return 1.125*grossPrice; // in NZ: 12.5% GST
} }
```

```
public class Food extends Product {
  public double netPrice() {
    // in Germany: only 7% GST on food
    return 1.07*grossPrice;
} }
```

Insufficient Exception Handling

- In Java: many exceptions must either be caught or declared
- Sometimes people catch them without actually handling them
- Problem: when the exception is thrown, it is not apparent
- The problem that caused the exception might cause trouble later

```
public double reciprocal(double x) {
  double y = 0;
  try {
    y = 1/x; // ArithmeticException for x==0
  } catch (Exception e) {} // no handling
  return y; // returns 0 for x==0
}
```

```
…
} catch (Exception e) {
  System.err.println(e);
  throws new MyException("Input error", e);
}
…
```

More Common Errors

1. Don't confuse == with **`equals`**
2. Array indices start with 0 (→ off-by-one error)
3. Distinguish primitive types, reference types and immutable reference types (call-by-value vs. call-by-reference)
4. Most common error: **`NullPointerException`**
   - Either improper object initialization (quite easy to find)
   - Or method that returns null
     (check the return value or use exceptions instead of returning **`null`**)

NullPointerException

```
Object o = null;
o.getClass();
```

ClassCastException

```
Integer i = (Integer)
  "hello";
```

ArithmeticException

```
int x = 4/0;
```

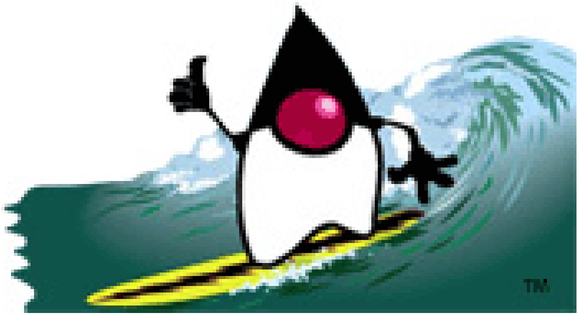
ArrayStoreException

```
Object x[] = new String[3];
x[0] = new Integer(0);
```

IndexOutOfBoundsException

```
Object x[] = new String[3];
x[3] = "hello";
```

Coding Style



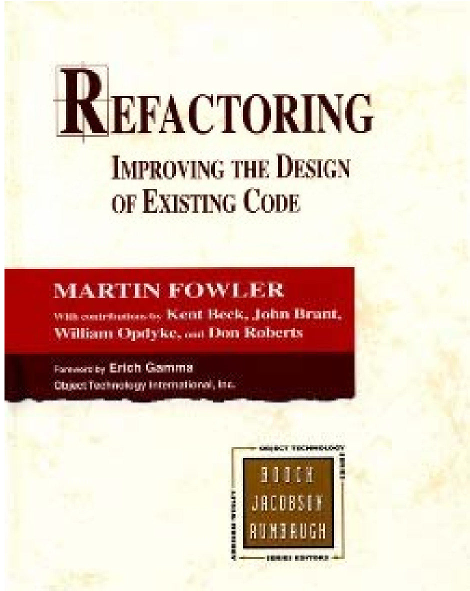
Naming Conventions

1. Begin class and interface names with **uppercase letter** e.g. **`Demo`**, **`Panel`**, **`GridbagLayout`**
2. Begin member names, method parameters and local variables with **lowercase letter**, e.g. **`getMax`**, **`start`**
3. Use **CamelCase**, i.e. a new word in a name is "separated" by an uppercase letter, e.g. **`getMainPanel`**
4. Package names are **lowercase**, e.g. **`java.awt.color`**
5. **`static final`** constants should be all **uppercase** with words separated by underscores ("_"), e.g. MIN_WIDTH
6. Type parameter names for generics should be a **single capital** letter, e.g. **`List<T>`**
7. Sometimes other conventions:
   - Name prefix for interfaces, e.g. **`ICollection`**
   - Name prefix for private variables, e.g. **`_size`**

Other Coding Guidelines

- **Comment your code**, particularly when doing something that is not straightforward
  - Comment at the beginning of a class/method/variable What is the class/method/variable for?
  - Also comment some statements
  - Empty line between logical groups of statement
- Code should be **machine-independent**,
  e.g. do not use absolute filenames in source code because different computers have different folders
  (e.g. **`"c:\myfolder\myfile.txt"`**)
  Use filenames relative to the application folder instead
  (e.g. **`"subfolder\myfile.txt"`**)
- **Handle error conditions** (e.g. throw and handle Exceptions)
- Use **`asserts`** to make sure errors do not propagate

Refactoring



REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER
With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International, Inc.

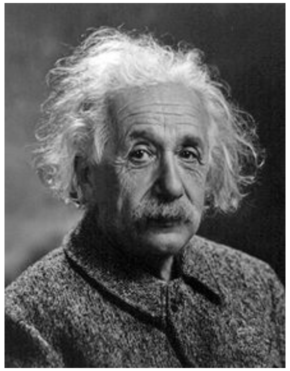
There's always room
for improvement, you know
- it's the biggest room
in the house
(L. H. Leber)

Refactoring

- "The art of improving the design of existing code safely"
  - **Rewriting** source code in order to improve its design or readability ("cleaning it up"; as we know it from XP)
- Refactoring may change HOW the code works but NOT WHAT it does (**preserving semantics**)
  - Neither fixes bugs nor adds new functionality
  - Changes may be very small or large (several files)
  - Encourages exploratory programming, rewriting of code, higher code quality
- **Test cases** help to ensure changes preserve semantics
- Refactoring literature describes indicators for common design problems ("smells") and possible solutions ("refactorings")
  - Fowler, Martin (1999). *Refactoring. Improving the Design of Existing Code*. Addison-Wesley.
  - Wake, William C. (2003). *Refactoring Workbook.* Addison-Wesley.

Simplicity

- KISS ("Keep it Short and Simple"), Occam's razor and Einstein: "everything should be made as simple as possible, but no simpler"
- Simplicity is an important principle for refactoring: can we rewrite the code so that it is simpler?
- Avoid unnecessary complexity, e.g.
  - Remove dead/unnecessary code
  - Use clear and simple names (in XP: system metaphor)
  - Not more coding than necessary (especially in XP)
- **Coding for humans**: clarity, readability, understandability
  "Clever hacks" are not worth it, they confuse people
- **Maintainability** more important than performance
  - "Premature optimization is the root of all evil"
  - Moore's law vs. incredibly high software maintenance costs



Smell: Long Method

- **Symptom**: many lines of code (LOC) in a single method (>> 10 LOC as a heuristic)
- **Cause**: a programmer keeps on writing in a single method
- **Solution**: find coherent groups of statements, extract meaningful methods
- **Payoff**: better readability, clearer structure, chances for abstraction and reuse
- Loss of performance is usually negligible

```
void m() {
  double[] data = {4.2, 6.4, 1.5, 9, 10.1};
  double avg = 0;
  for(double x : data) avg += x;
  avg /= data.length;
  double var = 0;
  for(double x : data) var += (x-avg) * (x-avg);
  var /= data.length;
}
```

double sum(double[] d)

double avg(double[] d)

double var(double[] d)

Smell: Large Class

- **Symptoms**: large number of instance variables, methods or LOC
- **Causes**:
  - Class gets "overweight" by incrementally adding more and more functionality without following a clear design
  - The underlying concept was misunderstood and is in fact a conglomerate of many concepts
- **Problem**: class looses its clear shape; it does not embody a single concept with a well-defined function anymore
- **Solutions**:
  - Extract classes embodying their own concepts
  - Extract subclasses that implement specialized functionality
  - Extract interfaces that clearly define feature subsets
- **Payoff**: simplicity & clarity of the parts, chances for abstraction & reuse
- **Example**: GUI is merged with underlying data model and/or application logic

Smell: Magic Number

- **Symptoms**: a constant value ("literal") appears in a method, possibly at several locations
- **Cause**: value is used ad hoc when it is needed; no further use anticipated
- **Problems**:
  - Hard to maintain
  - Easy to introduce bugs through incomplete changes
- **Solution**: replace literals with symbolic constants (**`static final`**) or enums: **`enum Gender { MALE, FEMALE }`**
- **Examples**:
  - Mathematical/physical constants (pi, e, conversion factors, ...)
  - Identification numbers (special data elements, errors, ...)
  - Configuration settings (e.g. file names, program behavior)

Duplicated Code

- **Symptoms**:
  - Two code fragments look (nearly) identical
  - Two code fragments do (nearly) the same
- **Causes**:
  - Several programmers working independently (duplication might not be obvious or is not anticipated)
  - Programmers copy, paste & adapt code that almost fits their needs
- **Solution**: extract method
  - If duplicates just do the same: choose and substitute the superior algorithm (or merge)
  - If duplicates in sibling classes: pull up method and fields into superclass; form template method
- **Examples**:
  - Small auxiliary tasks (e.g. sorting numbers, finding elements) are solved ad hoc
  - Overlapping requirements cause similar UI or logic

Template Methods

- Algorithm is mostly the same for several related types (siblings) but varies in the details
- Idea: describe the common, general steps of the algorithm in a method of the superclass (template method); put the details into helper methods of the subclasses

```
abstract class Game {
  int numPlayers;
  abstract void makeTurn(int player);
  abstract int getWinner();
  final void play() {
    while(getWinner()==0)
      for(int p=1;p<=numPlayers;p++)
        makeTurn(p);
    System.out.println(
      "Player "+getWinner()+" wins");
  }
}
```

```
class Chess
  extends Game
{ …
  Chess() {
    numPlayers = 2;
  }
  void makeTurn
  (int p) { … }

  int getWinner()
  { … }
}
```

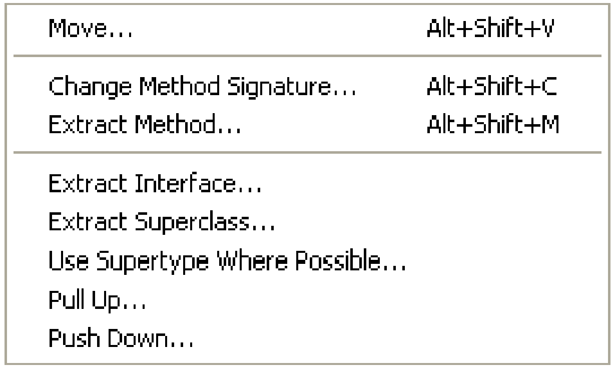
"Ask What Kind" Anti-Pattern (Simulated Inheritance)

- **Symptom**: method uses **`switch`** or several **`ifs`** (possibly with **`instanceof`**) to distinguish between different kinds of objects
- **Cause**: related but different concepts are not represented by different classes, lack of method polymorphism
- **Solution**:
  - Represent the different kinds by different subclasses
  - Implement subclass-specific behavior by overriding methods in the subclasses
  - Subclass-specific method is invoked automatically ("don't ask what kind")

```
class C {
  String type;
  void m() {
    if(type.equals("A")) m1();
    if(type.equals("B")) m2();
} }
```

```
class C {
  void m() { generic }
}
class A extends C {
  void m() { m1 }
}
class B extends C {
  void m() { m2 }
}
```

Refactoring in Eclipse


Move... Alt+Shift+V
Change Method Signature... Alt+Shift+C
Extract Method... Alt+Shift+M
Extract Interface...
Extract Superclass...
Use Supertype Where Possible...
Pull Up...
Push Down...

- Select source code,
  right-click and use *Refactor* submenu
- Refactoring with preview of changes
  (individual changes can be vetoed)
- When method name is selected:
  Rename, Inline, Pull Up, Push Down,
  Introduce Indirection, Change Method Signature, …
- When field name is selected:
  Rename, Pull Up, Push Down, Encapsulate Field,
  Generalize Declared Type, …
- When class name is selected:
  Rename, Move, Extract Interface, Extract Superclass,
  Use Supertype Where Possible, …
- When statements are selected: *Extract Method*
- When expression is selected: *Extract Constant*
- Nice summary of Eclipse refactoring
  http://www.cs.umanitoba.ca/~eclipse/13-Refactoring.pdf

Today's Summary

- Watch out: there are **common Java errors**. Avoiding them can save days of debugging.
- **Coding Style** guidelines are used/enforced by all serious projects (for code readability)
- **Refactoring**: "The art of improving the design of existing code safely"

References:

David Reilly. Top Ten Errors Java Programmers Make
http://www.javacoffeebreak.com/articles/toptenerrors.html

Oracle. Code Conventions for the Java Programming Language.
http://www.oracle.com/technetwork/java/codeconv-138413.html

Martin Fowler. http://refactoring.com/

Quiz

1. Why is it important to handle exceptions when they are caught?
2. What does refactoring mean for the functionality in a program?
3. Why do we have naming conventions?