Programming Tools

David Rabinowitz

This Lecture

- Personal Productivity Tools
  - And how to use them
- Refactoring
- Static Analysis & Metrics
- Profiling

Refactoring

- Improving the design of existing code, without changing its observable behavior
- Here's the *Extract Method* refactoring:

Before:

```
void f(int[] a) {
    …
    // compute score
    score = initial_score;
    for (int i=0; i<a.length; i++)
        score += a[i] * delta;
}
```

After:

```
void f() {
    …
    computeScore();
}
computeScore(int[] a) {
    // code cut & pasted here
}
```


Why?

- Why Refactor?
  - Improve software design
  - Make software easier to understand
  - Help find bugs
  - Help program faster
- Preconditions
  - Working code
  - Good set of unit tests

When?

- When to refactor
  - Before adding functionality
  - Before fixing a bug
  - During code review
- When not to refactor
  - During adding functionality
  - During fixing a bug
  - No good set of unit tests
  - Small programs (usually)

Code Smells

- "If it stinks, change it"
  - Duplicate code
  - Switch statements
  - Long method
  - Data class
  - Long parameter list
  - Primitive obsession
  - Temporary field
  - …

Documented Refactorings

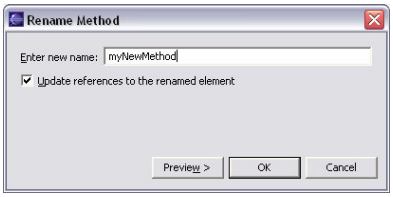
- There's a catalog
  - Fowler's book
  - www.refactoring.com/catalog
- There are many others
- Way to learn good OOD principles
- Pay attention to the mechanics

Automated Refactorings

- Eclipse's 'Refactor' menu automates thing
  - Undo & Redo
  - Physical Structure
  - Class Level Structure
  - Structure Inside a Class
- Wizards & Preview Windows Included
- Other tools exist: see refactorit.com

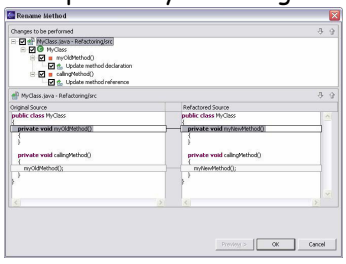
Automated Refactoring Example

- The 'Rename' Refactoring renames any Java element, and references to it:



Automated Refactoring Example II

- You can preview your changes:



Encapsulate Field

Before:

```
public String name;
```

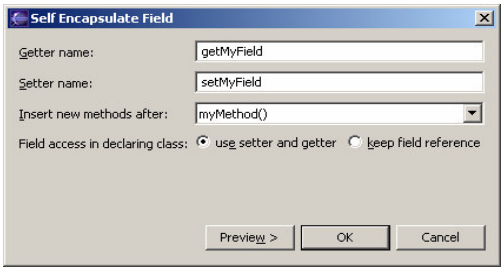
After:

```
private String name;

public String getName() {
    return name;
}

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


Encapsulate Field in Eclipse



Introduce Null Object

Before:

```
if (project == null)
    plan = Plan.default();
else
    plan = project.getPlan();
```

After:

```
class NullProject
  implements Project {

  public Plan getPlan() {
      return Plan.default();
  }

  // other Project methods
}
```

- This is the *Null Object Pattern*

Parameterize Method

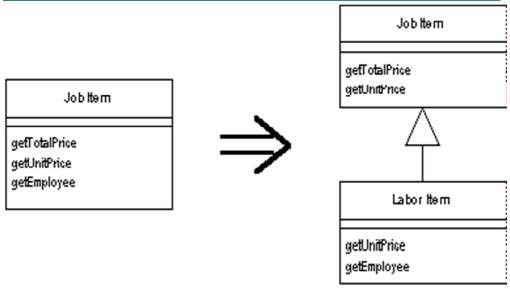
Before:

```
class Server {
    handleGet(…)
    handlePut(…)
    handleSet(…)
}
```

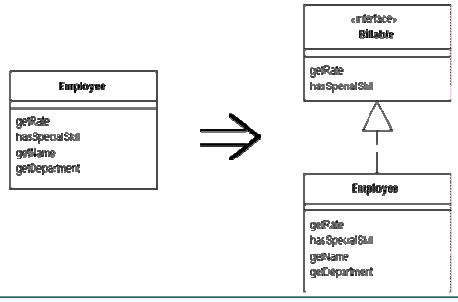
After:

```
class Server {
  handle(EventType et, …)
}
```

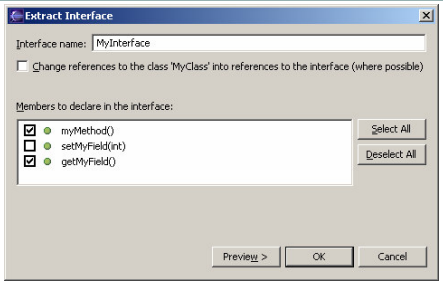

Extract Subclass



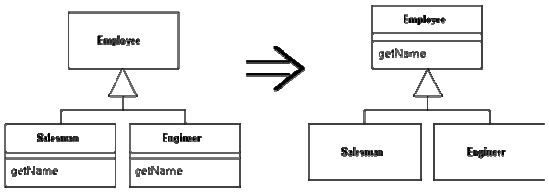
Extract Interface



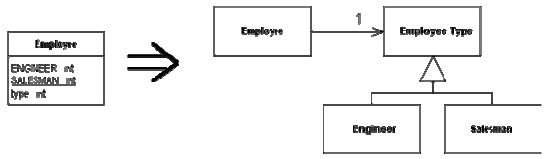
Extract Interface in Eclipse



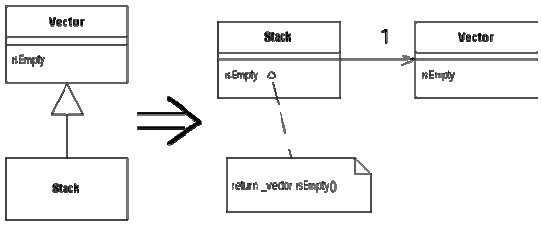
Pull Up Method



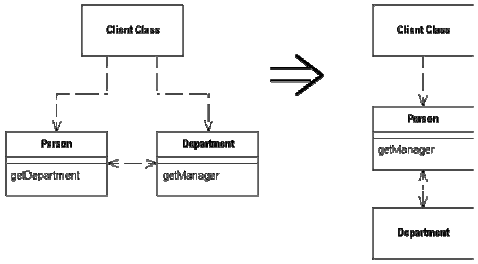
Replace Type Code with State/Strategy



Replace Inheritance with Delegation

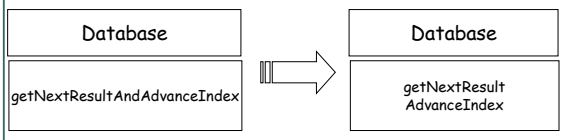


Hide Delegate



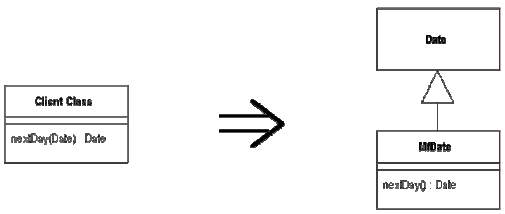
- Obeys the *Law of Demeter*

Separate Query from Modifier



- Obeys *Command-Query Separation*

Introduce Local Extension



- Alternative: Introduce Foreign Method

The opposites are there too

- Inline method (extract method)
- Replace Parameter with Explicit Methods (Parameterize Method)
- Collapse Hierarchy (Extract subclass)
- Remove middle man (Hide delegate)
- Push down method (pull up method)
- Replace delegation with inheritance

More useful Refactorings in Eclipse

- Rename
- Move
- Change Method Signature
- Use Supertype where possible
- Extract Constant
- Introduce Factory
- ...

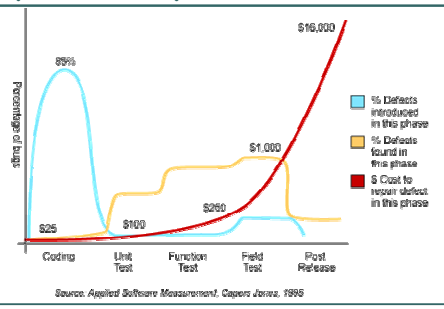
How to Refactor

- Recognize the smells
- Refactor in small discrete steps
- Test after each step
- Refactor in pairs
- Use documented refactorings
- Don't mix with adding functionality or fixing a bug

Static Code Analysis

- Programs that help gain understanding of your code
- Find areas in the code with
  - Possible Bugs
  - "Fishy" Design
  - Inconsistent Style
- It's no replacement for testing
  - Finding (non-trivial) bugs is undecidable

Why is it so important?



Source: Applied Software Measurement, Capers Jones, 1995

Available Tools

- Commercial
  - Lint for C and C++ (see gimpel.com)
  - JTest (parasoft.com)
- Free Eclipse Plugins
  - JLint (artho.com)
  - CPD - Copy Paste Detector
  - PMD
  - CheckStyle
  - JDepend - Metrics

Lint

- Looks for over 800 C/C++ Issues
- Things that compilers either miss or allow
- Specific C++ Errors, for example:
  - Throwing from a destructor
  - Not checking for NULL argument in 'delete'
  - Order of initializations / constructors
  - Non-virtual over-riden methods
- Macro scanning
  - Incorrect parameter passing, Side effects, ...

Lint II

- Value Tracking
  - Division by zero, null dereference, out-of-bounds, memory leaks, double deallocation, …
- Casting & Values
  - Loss of sign, truncations, Assignment in 'if', …
- Specific C Issues
  - printf() arguments, order of evaluation: a[i] = i++;
- Style
  - Indentation, suspicious semi-colons (a > b); , …
- Hundreds of other issues

JTest

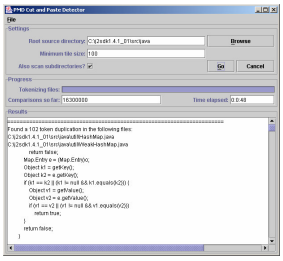
- Checks for 380 Java & Style Issues
- Can automatically correct 160 of these
- Extensible by user-defined issues
- Supports metrics as well
  - Number of bytes, classes, lines, methods, …
  - Issue = Deviation from acceptable metric range
- Some issues are shared with C/C++
  - Values, Casting, Unreachable code, Indentation, Comments, Initialization, Exceptions, …

JTest II

- Other Java Specific Issues
  - Portability
  - Security
  - Optimization
  - Garbage Collection
  - Threads and Synchronization
  - Internationalization
  - Servlets / EJBs
  - Naming Conventions
  - …

CPD – Copy Paste Detector

- Works with Java, C, C++ and PHP
- http://pmd.sourceforge.net/cpd.html
- From the examples:
  - A 307 lines(!) of duplicated code in Apache 2

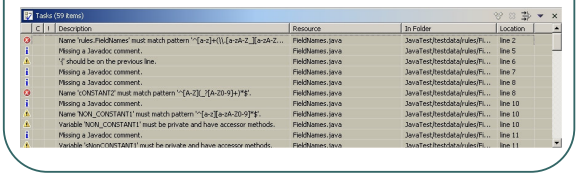


PMD

- For Java code
- Checks
  - Unused local variables / parameters / private methods
  - Empty catch blocks
  - Empty 'if' statements
  - Duplicate import statements
  - Classes which could be Singletons
  - Short/long variable and method names
  - And many many more …

CheckStyle

- Similar to PMD
- Javadoc Comments, Naming Conventions, Headers, Imports, Size Violations, Whitespace, Modifiers, Blocks, Coding Problems, Class Design, Duplicate Code



JDepend

- Calculates metrics for java packages
- Calculated metrics
- ***CC*** - Concrete Class Count
  - The number of concrete classes in this package.
- ***AC*** - Abstract Class Count
  - The number of abstract classes or interfaces in this package.

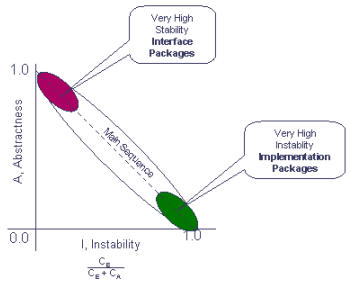
JDepend (2)

- **Ca** - Afferent Couplings
  - The number of packages that depend on classes in this package.
  - "*How will changes to me impact the rest of the project?*"
- **Ce** - Efferent Couplings
  - The number of other packages that classes in this package depend upon.
  - "*How sensitive am I to changes in other packages in the project?*"

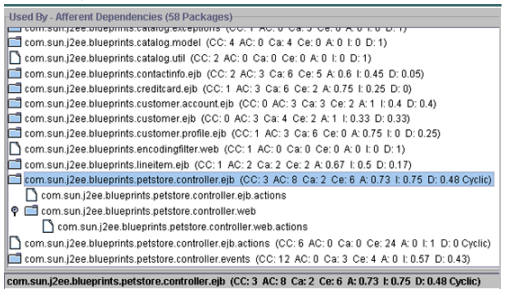
JDepend (3)

- **A** - Abstractness (0-1)
  - Ratio (0.0-1.0) of Abstract Classes (and interfaces) in this package.
  - *AC/(CC+AC)*
- **I** - Instability (0-1)
  - Ratio (0.0-1.0) of Efferent Coupling to Total Coupling.
  - *Ce/(Ce+Ca).*
- **D** - Distance from the Main Sequence (0-1)
- **Cyclic** - If the package contains a dependency cycle

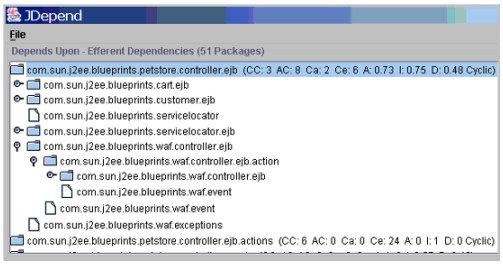
The main sequence



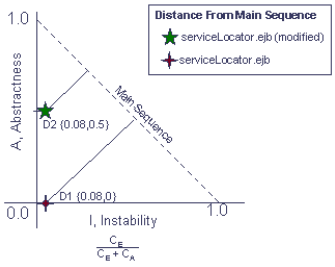
Examples – Pet Store



Examples – Pet Store (2)



How to improve the rating?



Profiling

- A profiler is a program that can track the performance of another program
- Used to solve performance problems
  - "How come a simple file viewer take 30 seconds to start, and over 2 minutes to find text in a medium text file?"
- Used to solve memory problems
  - "Why does my text editor take 50MB on startup, and 300MB after a hour of work?"

Performance Tuning

- How can I make my program faster?
- The 80 / 20 Principle
  - 80% of the time is spent in 20% of the code
  - Key Issue: Find the bottlenecks
- Classic Mistake: Assume the bottlenecks
  - You can't know where they'll be
- Classic Mistake II: Optimize in Advance
  - Start with the right design, then optimize

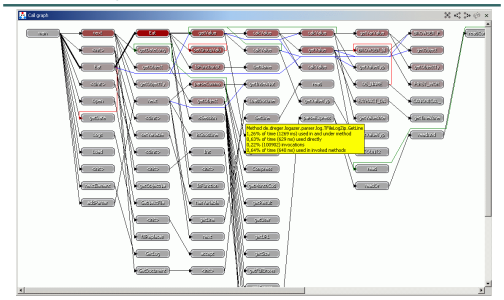
Performance Tuning Process

- Step 1: Identify the bottlenecks
  - Use a profiler!
  - Find & measure the bottlenecks
- Step 2: Decide how to solve bottlenecks
  - Make them faster (new algorithm, data str.)
  - Call them less often (caching, lazy execution)
- Step 3: Measure again
  - Only way to make sure improvement happened

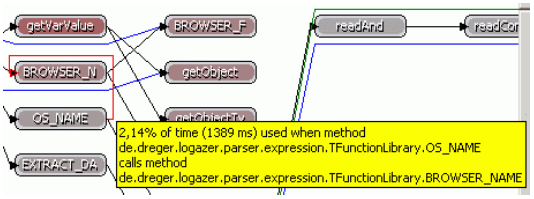
Eclipse Profiler Plugin

- We'll demonstrate on the (free!) Eclipse Profiler Plugin
- What is tracked
  - CPU
  - Memory usage
  - Number of objects
  - Object graph
  - Call graph

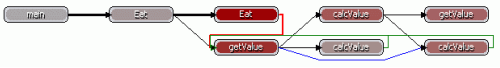
Call Graph



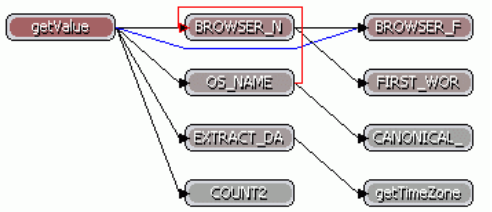
Call hint



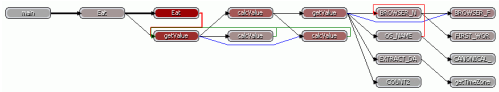
Callers



Callees



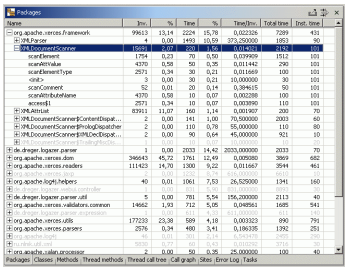
Callers and callees



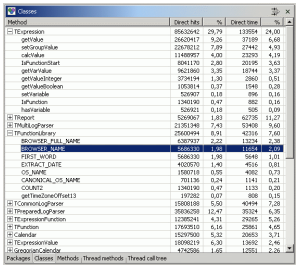
CPU Profiling

- How many invocations were?
- How much time have we spent in a package / class / method?
- Finds the bottlenecks
  - Just sort by time or number of invocations

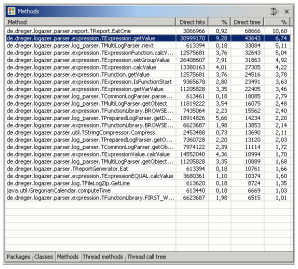
Packages



Classes

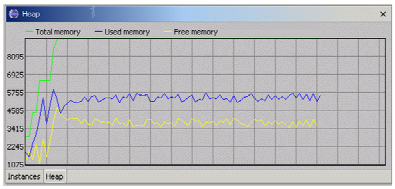


Methods

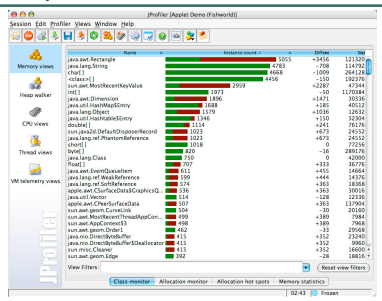


Memory

- How much memory does the program take?
- Are there memory leaks?



Memory Monitor



Profiling - summery

- How does my application behave?
- What are the critical paths?
- Where are the bottlenecks?
- Do I have memory leaks?
  - Java users – you are not exempted!

Summary

- Personal Productivity Tools
  - Refactoring
  - Static Analysis & Metrics
  - Profilers
- Use them!
- There's more – see Eclipse Plugins