Design of Software Systems (Ontwerp van SoftwareSystemen)

5 Unit Testing, Refactoring and Profiling

Roel Wuyts

OSS 2014-2015

A golden rule...

- Make it Work
- Make it Right
- Make it Fast

How does this work?

- First make sure the software does what you want
  - use unit tests
- Then rework the code until it speaks for itself
  - use refactorings
- Then optimize the performance, if needed
  - use profiling

Testing

Unit Testing	test individual components
Module Testing	test a collection of related components
Sub-System Testing	test sub-system interface mismatches
System Testing	• test interactions between sub-systems • tests that the complete system fulfils requirements
Acceptance Testing	test system with real rather than simulated data

Unit Testing

- How can I trust that changes did not destroy something?
- What is my confidence in the system ?
- How do I write tests?
- What is unit testing?

Tests

- Tests represent your trust in the system
- Build them incrementally
  - Do not need to focus on everything
  - When a new bug shows up: write a test
- Even better: test first!
  - Act as your first client
  - Helps finding proper interfaces
- Tests are active documentation: they are always in sync

Testing Style

- “The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run.”
  - write unit tests that thoroughly test a single class
  - write tests as you develop (even before you implement)
  - write tests for every new piece of functionality
- “Developers should spend 25-50% of their time developing tests.”

But I can’t cover anything!

- Sure! Nobody can but:
  - When someone discovers a defect in your code, first write a test that demonstrates the defect.
  - Then debug until the test succeeds.

“Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.”

Martin Fowler

Unit Testing

- Ensure that you get the specified behaviour of the public interface of a class
  - Normally tests a single class
- General setup of a test:
  - Create a context,
  - Send a stimulus,
  - Check the results

Example

```
public class SaleTest extends TestCase
{
  // …
  public void testMakeLineItem() {
    Sale fixture = new Sale();
    Money total = new Money(7.5);
    Money price = new Money(2.5);
    ItemID id = new ItemID(1);
    ProductDescription desc = new ProductDescription(id, price, “product 1”);

    sale.makeLineItem(desc, 1);
    sale.makeLineItem(desc, 2);

    assertTrue(sale.getTotal().equals(total));
}
```

About Failures and Errors

- A failure is a failed assertion
  - i.e., an anticipated problem that you test.
    - assertEquals(2, myContainer.nrOfElements())
- An error is a condition you didn't check for.
  - e.g. an exception being thrown you did expect

```
boolean isExceptionThrown = false;
try {
    myContainer.get(3);
} catch(IndexOutOfBoundsException e) {
    isExceptionThrown = true;
}
assertTrue(isExceptionThrown);
```

Good Unit Tests

- Are repeatable
  - have to be deterministic to be useful
- Require no human intervention
  - so that they can be automated
- Are "self-described" and tell a story
  - to serve as documentation
- Change less often than the system
  - they encode stable functionality

Designing tests

- Build simple tests
- Check that failures are caught
- Run tests frequently (every couple of minutes)
- Test Infrastructure code first, then application-specific code
- Reuse as much test code as you can (tests are code!)
- Write small tests that test one particular aspect
- Make sure the tests are deterministic

Why spending time testing?

- Find problems soon.
  - in context of what you were doing!
- Serve as documentation.
- Ease maintenance and evolution.
  - new developers jump in anytime..
- Have something to show all the time.

Testing Frameworks

- Tests have to be repeatable
- Unit Testing Frameworks implement necessary infrastructure so that you can set up your tests, run them frequently, and see the results
- SUnit is “the mother of all unit test frameworks”
  - started in Smalltalk
  - fanned out to all kinds of other languages
    - JUnit, NUnit, CppUnit, ...

JUnit overview

- Junit (inspired by Sunit) is a simple “testing framework” that provides:
  - classes for writing Test Cases and Test Suites
  - methods for setting up and cleaning up test data (“fixtures”)
  - methods for making assertions
  - textual and graphical tools for running tests

Testing Frameworks

- Key parts
  - TestCase: bundles test methods
  - Some mechanism to execute test code

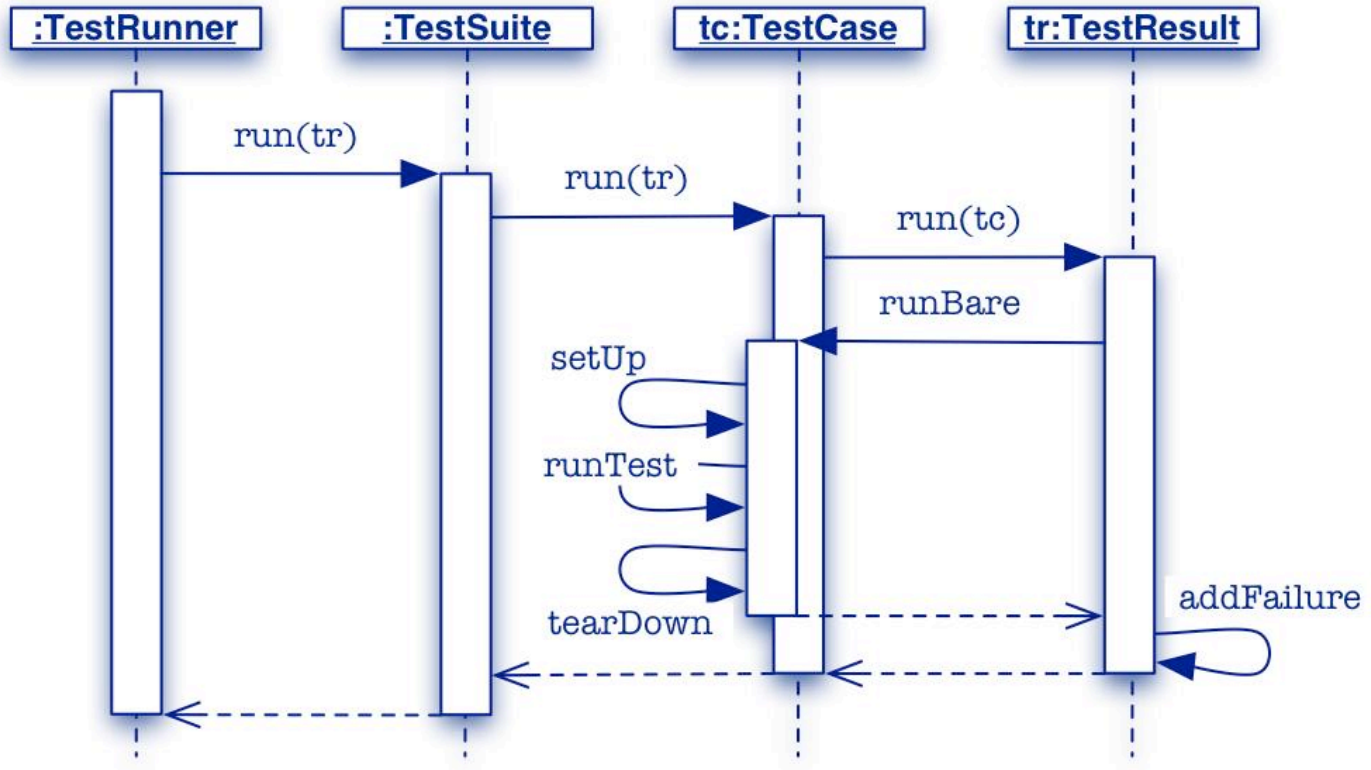
    (methods, macroes, ...)
  - Fixture (≈ Resource): known set of objects that serves as a base for a set of test cases
  - TestSuite: bundles testcases so that they can be run together
  - TestRunner: runs a testsuite, outputting results

A testing scenario

- The framework calls the test methods that you define for your test cases
  - You need to declare a TestRunner
  - You specify who will gather the results
  - You add the needed tests to the runner
  - You run the TestRunner
    - this automatically runs all tests, collecting the results
  - You pass the results to an Outputter

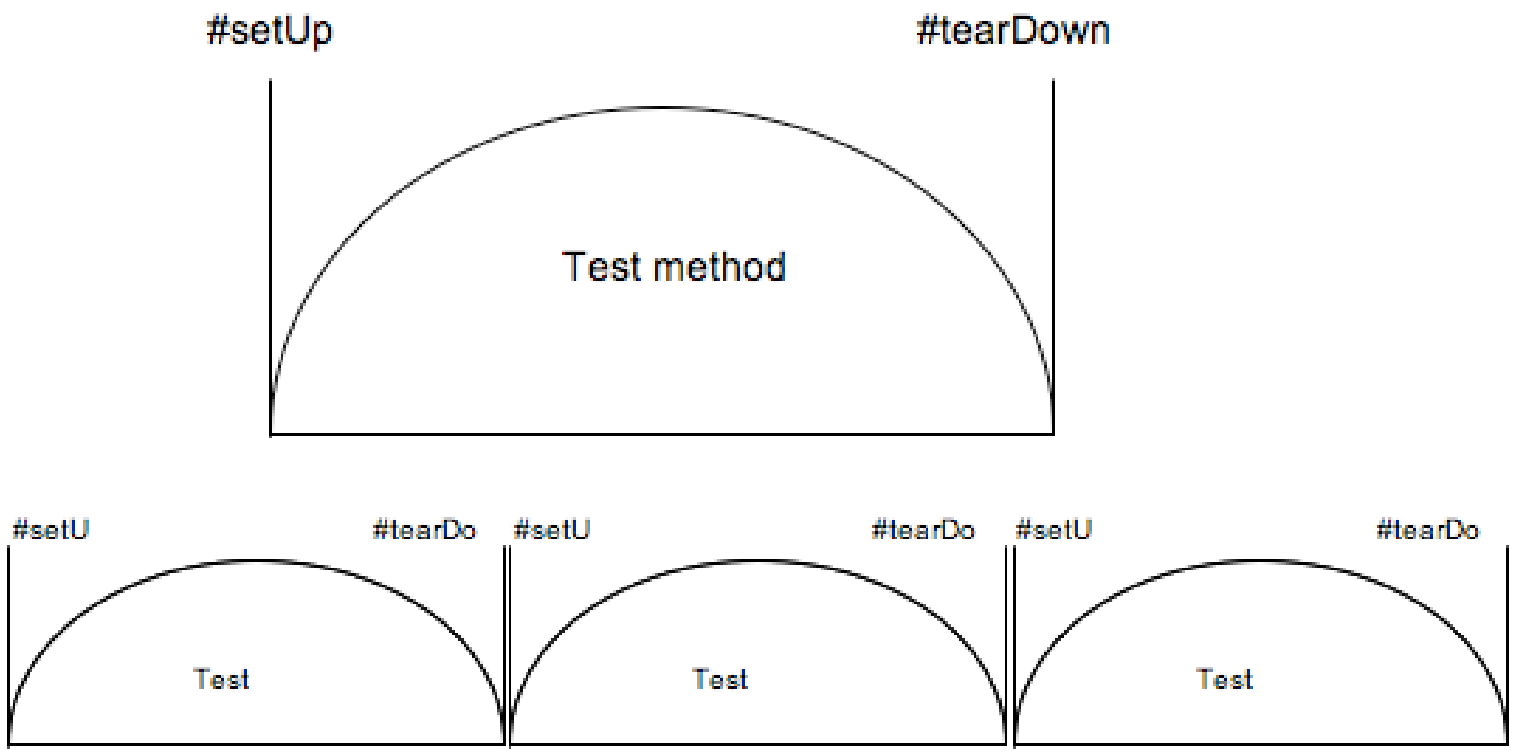
A testing scenario

- The framework calls the test methods that you define for your test cases



Setup and TearDown

- Executed before and after each test
  - setUp allows us to specify and reuse the context
  - tearDown makes us clean-up afterwards



- Example unit test for an online ordering system

Mocking & Stubbing

- Example unit test for an online ordering system

```
public class OrderStateTester extends TestCase {

      private static String TALISKER = "Talisker";
      private static String HIGHLAND_PARK = "Highland Park";
      private Warehouse warehouse = new WarehouseImpl();

      protected void setUp() throws Exception {
        warehouse.add(TALISKER, 50);
        warehouse.add(HIGHLAND_PARK, 25);
      }
      public void testOrderIsFilledIfEnoughInWarehouse() {
        Order order = new Order(TALISKER, 50);
        order.fill(warehouse);
        assertTrue(order.isFilled());
        assertEquals(0, warehouse.getInventory(TALISKER));
      }
      public void testOrderDoesNotRemoveIfNotEnough() {
        Order order = new Order(TALISKER, 51);
        order.fill(warehouse);
        assertFalse(order.isFilled());
        assertEquals(50, warehouse.getInventory(TALISKER));
      }
```

Mocking & Stubbing

- Example unit test for an online ordering system

```
public class OrderStateTester extends TestCase {

      private static String TALISKER = "Talisker";
      private static String HIGHLAND_PARK = "Highland Park";
      private Warehouse warehouse = new WarehouseImpl();

      protected void setUp() throws Exception {
        warehouse.add(TALISKER, 50);
        warehouse.add(HIGHLAND_PARK, 25);
      }
      public void testOrderIsFilledIfEnoughInWarehouse() {
        Order order = new Order(TALISKER, 50);
        order.fill(warehouse);
        assertTrue(order.isFilled());
        assertEquals(0, warehouse.getInventory(TALISKER));
      }
      public void testOrderDoesNotRemoveIfNotEnough() {
        Order order = new Order(TALISKER, 51);
        order.fill(warehouse);
        assertFalse(order.isFilled());
        assertEquals(50, warehouse.getInventory(TALISKER));
      }
```

collaborator

tested object
"system under test"

state verification

Mocking & Stubbing

- Using mocking (jMock library example)

```
public class OrderInteractionTester extends MockObjectTestCase {

      private static String TALISKER = "Talisker";

      public void testFillingRemovesInventoryIfInStock() {
        Order order = new Order(TALISKER, 50);
        Mock warehouseMock = new Mock(Warehouse.class);

        warehouseMock.expects(once()).method("hasInventory")
          .with(eq(TALISKER),eq(50))
          .will(returnValue(true));
        warehouseMock.expects(once()).method("remove")
          .with(eq(TALISKER), eq(50))
          .after("hasInventory");

        order.fill((Warehouse) warehouseMock.proxy());

        warehouseMock.verify();
        assertTrue(order.isFilled());
      }

  }
```

setup - data

setup - expectations

exercise

verify

More info: http://martinfowler.com/articles/mocksArentStubs.html

Refactorings

- Refactoring
  - What is it?
  - Why is it necessary?
  - Examples
  - Tool support
  - Obstacles to refactoring

What is 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 [Fowl99a]
- A behaviour-preserving source-to-source program transformation [Robe98a]
- A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ... [Beck99a]

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?

- “Grow, don’t build software” (Fred Brooks)
- “Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” (Fowler)
- 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

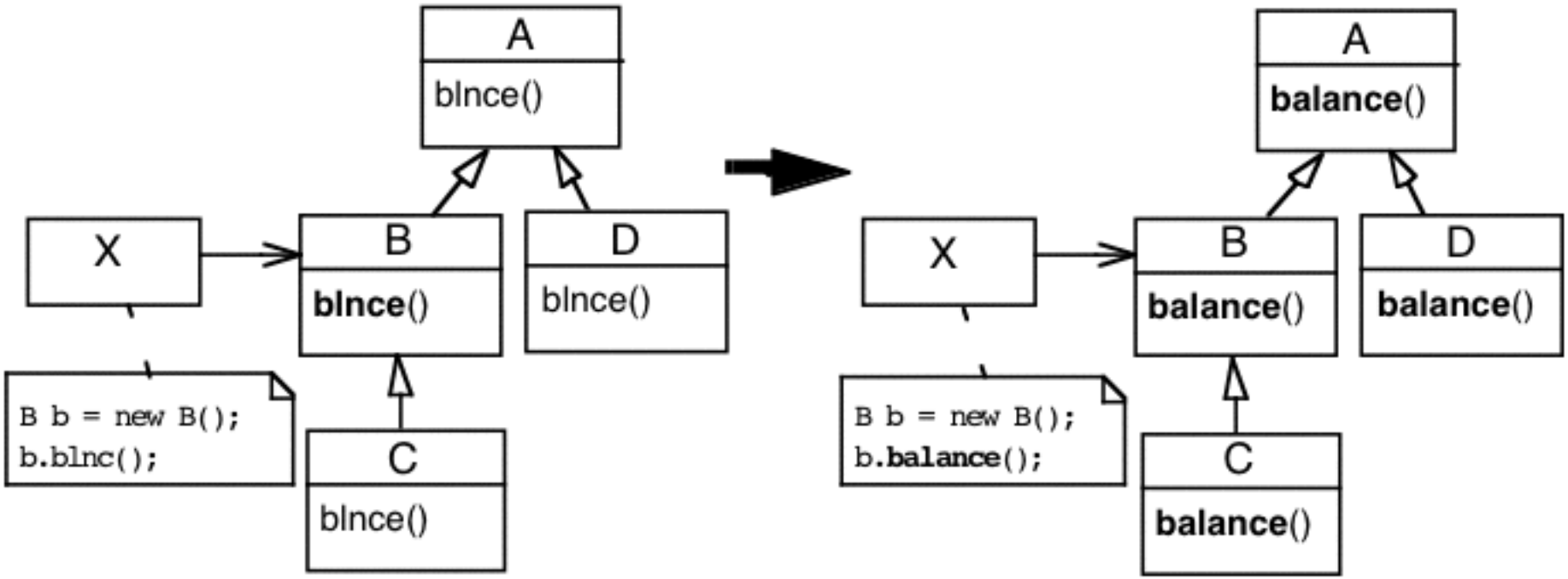
Why Refactoring?

- 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

Examples of Refactoring Analysis

- Rename Method
  - existence of similar methods
  - references of method definitions
  - references of calls
- AddClass
  - simple
  - namespace use and static references between class structure

Rename Method



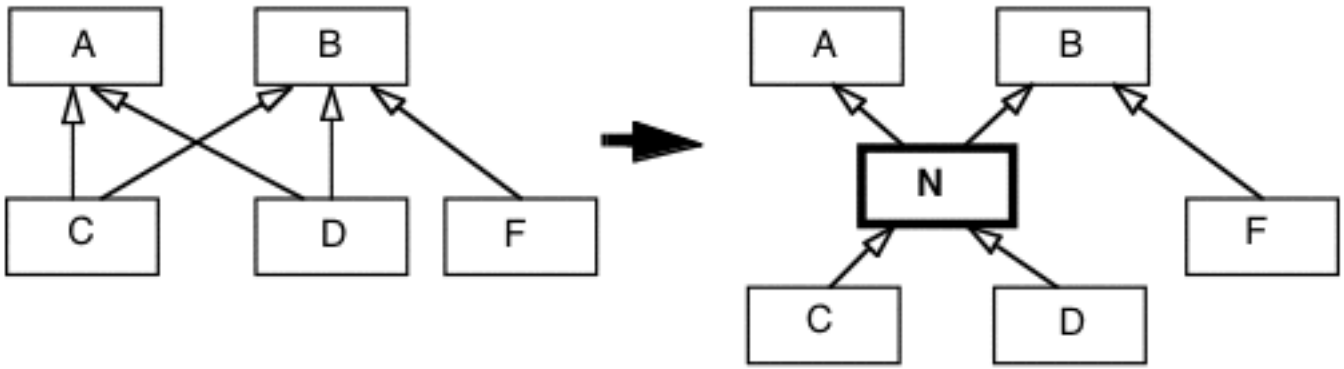
Rename Method: Do It Yourself

- Check if a method does not exist in the class and superclass/subclasses with the same “name”
- Browse all the implementers (method definitions)
- Browse all the senders (method invocations)
- Edit and rename all implementers
- Edit and rename all senders
- Remove all implementers
- Test

Rename Method

- Rename Method (method, new name)
- Preconditions
  - no method exists with the signature implied by new name in the inheritance hierarchy that contains method
  - [Smalltalk] no methods with same signature as method outside the inheritance hierarchy of method
  - [Java] method is not a constructor
- PostConditions
  - method has new name
  - relevant methods in the inheritance hierarchy have new name
  - invocations of changed method are updated to new name
- Other Considerations
  - Typed/Dynamically Typed Languages => Scope of the renaming

Add class



Add Class

- Preconditions
  - no class and global variable exists with classname in the same scope
  - subclasses are all subclasses of all superclasses
  - [Smalltalk] superclasses must contain one class
  - [Smalltalk] superclasses and subclasses cannot be metaclasses
- Postconditions
  - new class is added into the hierarchy with superclasses as superclasses and subclasses as subclasses
  - new class has name classname
  - subclasses inherit from new class and not anymore from superclasses
- Considerations: Abstractness

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 activities

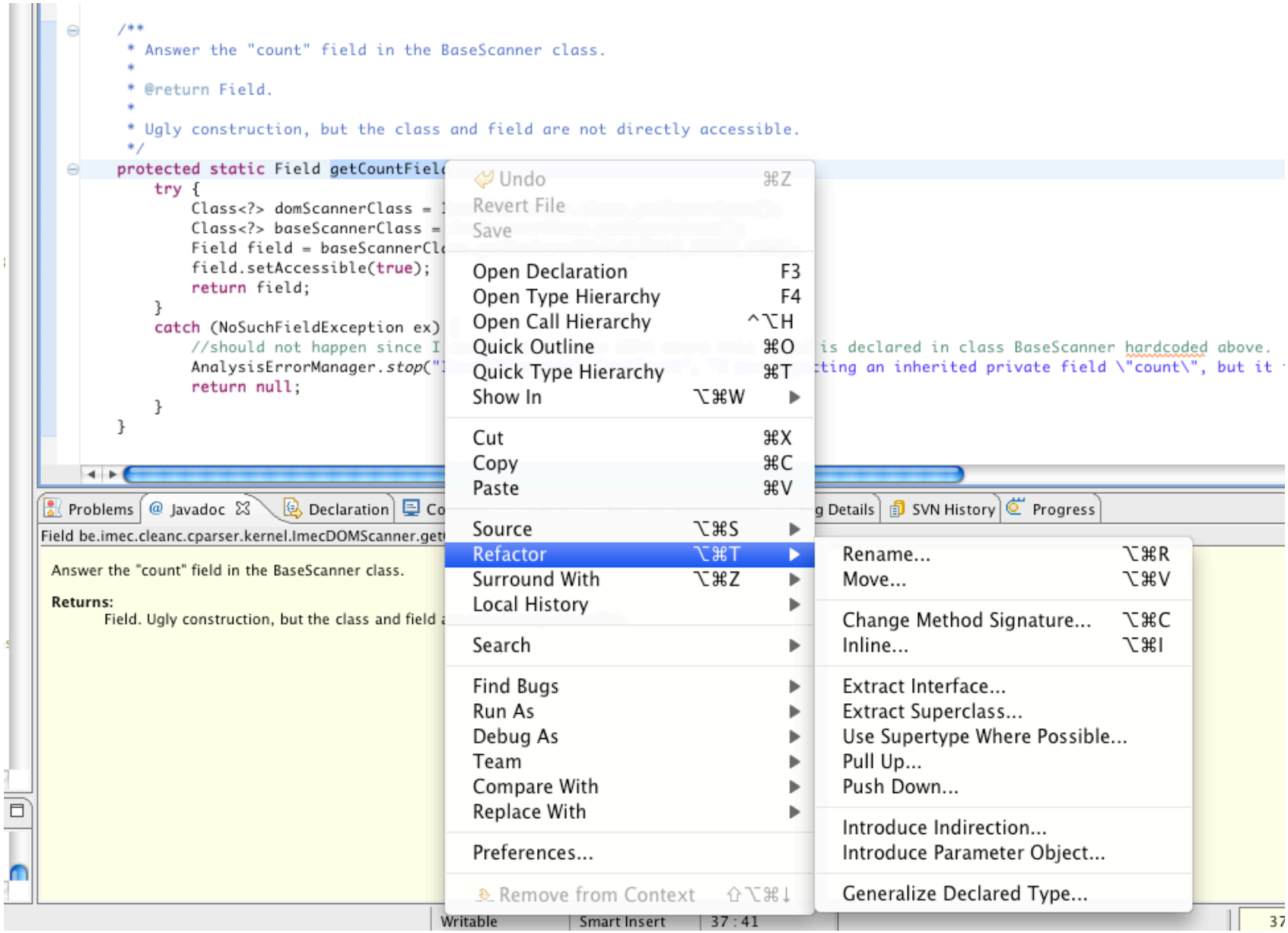
Change Efficiently	Failure Proof
Refactoring Tools - source-to-source program transformation - behaviour preserving ⇒ Improve Structure	Regression Testing - Repeating past tests - requires no user interaction - is deterministic ⇒ Verify damage to previous work
Development Environment - Fast edit-compile-run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

Conclusion: Tool Support

Note: Do not apply refactoring tools in isolation!

	Smalltalk	C++	Java
refactoring tools	++	- (?)	+
rapid edit-compile-run cycles	++	-	+-
reverse engineering facilities	+-	+-	+-
regression testing	+	+	+
version & configuration management	+	+	+

Refactoring in Eclipse



When to Refacctor ?

- When you add functionality
  - Helps you to understand the code you are modifying.
  - Sometimes the existing design does not allow you to easily add the feature.
- When you need to fix a bug
  - If you get a bug report, it’s a sign the code needs refactoring
  - because the code was not clear enough for you to see the bug in the first place
- When you do a code review
  - Code reviews help spread knowledge through the development team.
  - Works best with small review groups

When to Refactor

- You should refactor:
  - Any time that you see a better way of doing things
    - "Better" means making the code easier to understand and to modify in the future
  - You can do so without breaking the code
    - Unit tests are essential for this (remember: do not refactor in isolation)
- You should NOT refactor:
  - Stable code (code that won't ever need to change, code library)
  - Someone else's code
    - Unless you've inherited it (and now it's yours)

≠ XP practice!

- Rule of Thumb: 'Three strikes and you refactor'
  - 1st time: Write from scratch
  - 2nd time: Duplication eventually admissible
  - 3rd time: Refactor !!!

Example: Switch Statements

- Switch statements are very rare in properly designed object-oriented code
  - Therefore, a switch statement is a simple and easily detected "bad smell"
  - Of course, not all uses of switch are bad
  - A switch statement should *NOT* be used to distinguish between various kinds of object
- There are several well-defined refactorings for this case
  - The simplest is the creation of subclasses

Wuyts Roel

Example: Bad Smell

```
class Animal {
  final int MAMMAL = 0, BIRD = 1, REPTILE = 2;
  int myKind;  // set in constructor
  ...

  String getSkin() {
    switch (myKind) {
      case MAMMAL: return "hair";
      case BIRD: return "feathers";
      case REPTILE: return "scales";
      default: return "integument";
    }
  }
}
```

Example: Improved

```
class Animal {
        String getSkin() {
            return "integument";
        }
}

class Mammal extends Animal {
        String getSkin() {
            return "hair"; }
        }

class Bird extends Animal {
        String getSkin() {
            return "feathers";
        }
}

class Reptile extends Animal {
        String getSkin() {
            return "scales";
        }
}
```

JUnit Tests

- As we refactor, we need to run (JUnit) tests to ensure that we haven't introduced errors

```
public void testGetSkin() {
        assertEquals("hair", myMammal.getSkin());
        assertEquals("feathers", myBird.getSkin());
        assertEquals("scales", myReptile.getSkin());
        assertEquals("integument", myAnimal.getSkin());
}
```

- This should work equally well with either implementation
- The setUp() method of the test fixture may need to be modified
- Re-running unit tests proves that the refactoring succeeded (= external behavior remained unchanged)

Wuyts Roel

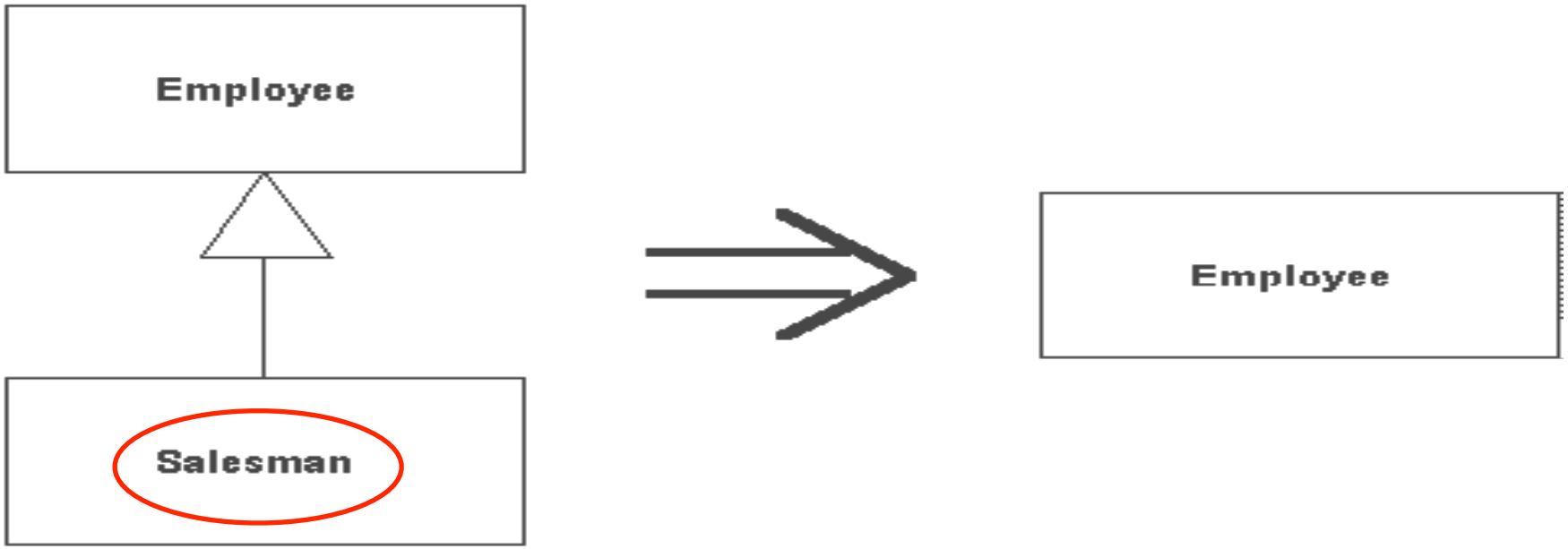
Refactoring Examples

- Add Parameter
- Change Association
- Change Reference to Value
- Change Value to Reference
- **Collapse Hierarchy**
- **Consolidate Conditional**
- Convert Procedures to Objects
- **Decompose Conditional**
- **Encapsulate Collection**
- **Encapsulate Downcast**
- Encapsulate Field
- **Extract Class**
- Extract Interface
- **Extract Method**
- Extract Subclass
- Extract Superclass
- Form Template Method
- Hide Delegate
- Hide Method
- **Inline Class**
- Inline Temp
- Introduce Assertion
- Introduce Explain Variable
- Introduce Foreign Method
- …

72 Refactorings identified by Fowler

Refactoring Example: Collapse Hierarchy

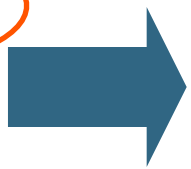
- When superclass and subclass are not very different: Merge them



Refactoring Example: Consolidate Conditional

- When the same fragment of code is in all branches: Move it out

```
double disabilityAmount()
 {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12)
       return 0;
    if (_isPartTime) return 0;
   // compute the disability amount
}
```



```
double disabilityAmount()
{
    if (isNotEligableForDisability())
       return 0;
    // compute the disability amount
}
```

Refactoring Example: Decompose Conditional

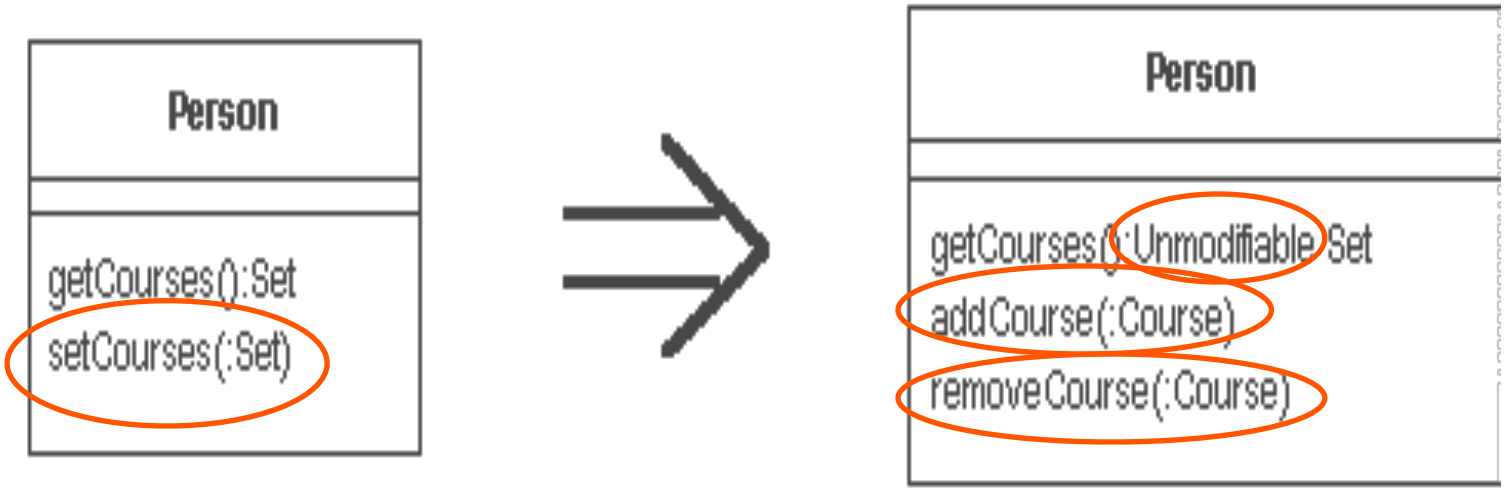
- When having a complicated conditional statement: Extract if/then/else parts

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
        charge = quantity * _winterRate + _winterServiceCharge;
else
        charge = quantity * _summerRate;
```

```
if (notSummer(date))
      charge = winterCharge (quantity);
else charge = summerCharge (quantity);
```

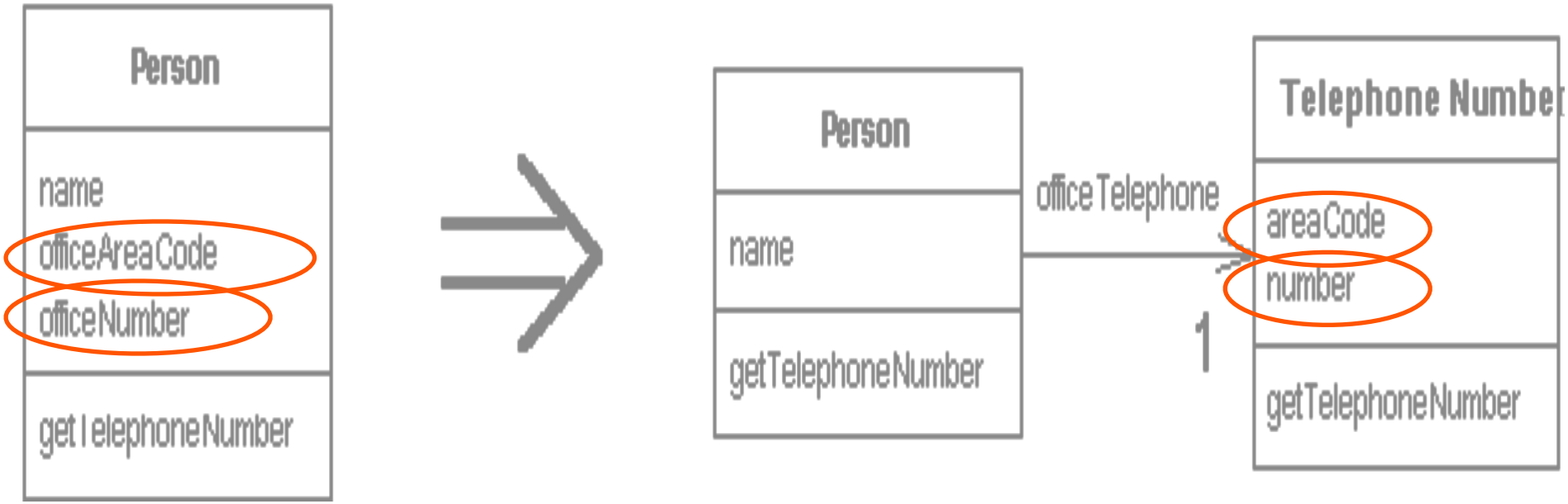
Refactoring Example: Encapsulate Collection

- When a method returns a collection: Provide Read-only view & add/remove methods



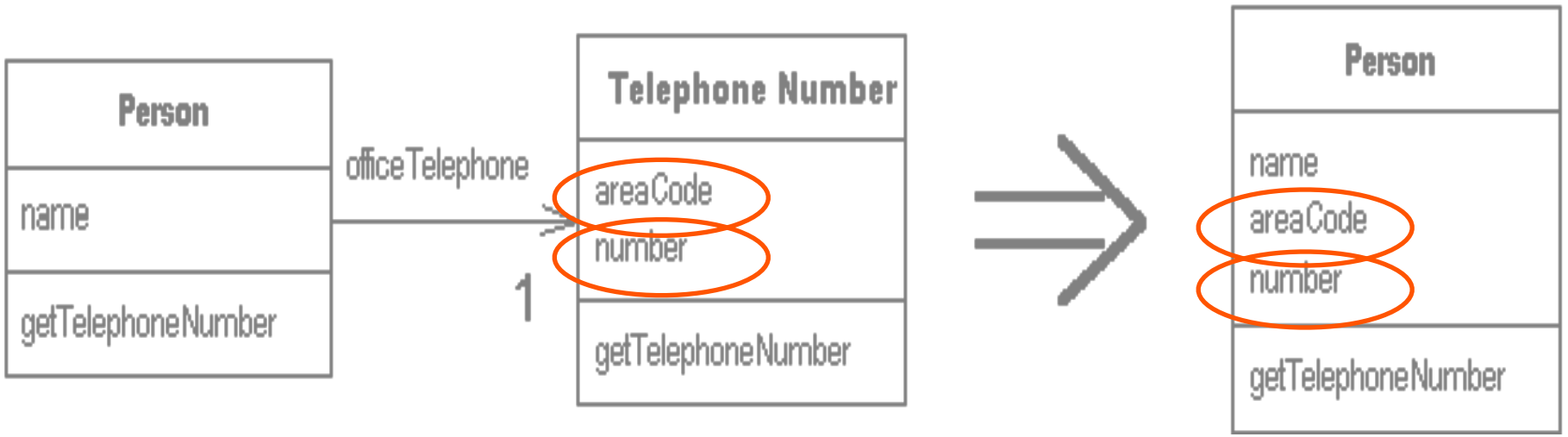
Refactoring Example: Extract Class

- When we have 1 class doing the work that should be done by 2: Create new class, move fields & methods
  - => GRASP High Cohesion



Refactoring Example: Inline Class

- When a class isn't doing very much: Merge with other class

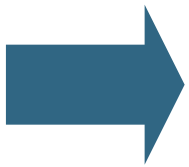


Refactoring Example: Encapsulate Downcast

- When a method returns an object that needs to be downcasted by its callers:
  - Move the downcast to within the method.
  - happens often when a class uses a collection or iterator

```
Object lastReading()  {
  return readings.lastElement();
}
```

```
Reading lastReading =
   (Reading) theSite.lastReading();
```



```
Reading lastReading()  {
  return (Reading) readings.lastElement();
}
```

```
Reading lastReading = theSite.lastReading();
```

Refactoring Example 9: Extract Method

- When we have a code fragment that can be grouped together: turn the fragment into a method with an explanative name

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

```
void printOwing() {
    printBanner();
    printDetails(getOutstanding());
}
```

Bad Smells in Code

- Duplicated Code
- **Long Method**
- **Large Class**
- **Long Parameter List**
- Divergent Change
- **Shotgun Surgery**
- **Feature Envy**
- Data Clumps
- Primitive Obsession
- Switch Statements
- Comments
- Parallel Inheritance/Interface Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Incomplete Library Class
- Data Class
- Refused Bequest
- Alternative Classes with Different Interfaces

Bad Smells

- Where did this term come from?

"If it stinks, change it."
--Grandma Beck

- The basic idea is that there are things in code that cause problems
  - Duplicated code
  - Long methods
  - …
- But any time you change working code, you run the risk of breaking it
  - A good test suite makes refactoring much easier and safer
- Bad smells gives inspiration, but are not designed as metrics
  - You have to decide yourself when something is "too much", …

Example: Duplicated Code

- If you see the same code structure in more than one place, find a way to unify them
- "Number one in the stink parade" !!!
- The usual solution is to perform
  - ExtractMethod: create a single method from the duplicated code
  - Invoke from all places: Use it wherever needed
  - You sometimes need additional refactorings (Add Parameter, …)
- This adds the overhead of method calls, thus the code could get a bit slower

Other Bad Smells

- Long Method
  - The longer a procedure is, the more difficult it is to understand.
  - Solution: perform EXTRACT METHOD or Decompose Conditional or Replace Temp with Query.
- Large class
  - When a class is trying to do too much, it often shows up as too many instance variables.
  - Solution: perform EXTRACT CLASS or EXTRACT SUBCLASS
- Long Parameter List
  - In OO, you don't need to pass in everything the method needs. Instead, you pass enough so the method can get to everything it needs
  - Solution: Use REPLACE PARAMETER WITH METHOD when you can get the data in one parameter by making a request of an object you already know about.

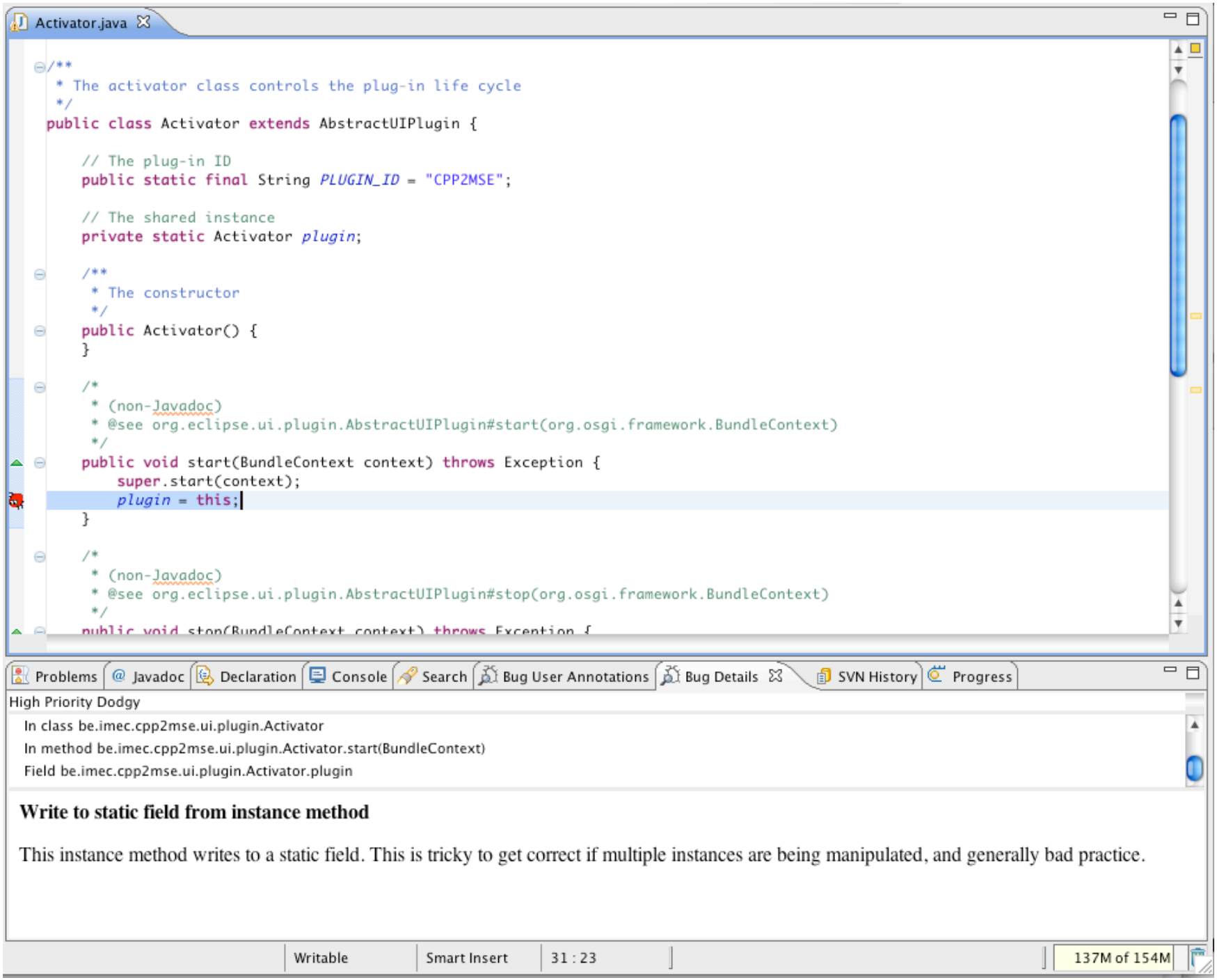
Other Bad Smells

- Shotgun Surgery
  - This situation occurs when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.
  - Solution: perform MOVE METHOD/FIELD or INLINE CLASS bring a whole bunch of behavior together.
- Feature Envy
  - A method that seems more interested in a class other than the one it is in.
  - Solution: perform MOVE METHOD or EXTRACT METHOD on the jealous bit and get it home.

Bad Smell/Sweet Smell: Comments

- Fowler says “comments often are used as a deodorant”
  - If you need a comment to explain what a block of code does, use Extract Method
  - If you need a comment to explain what a method does, use Rename Method
  - If you need to describe the required state of the system, use Introduce Assertion
- When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous
- The point is that code should be self-explanatory, so that comments are not necessary
- This should not discourage the use of comments (especially javadoc comments)
  - A comment is a good place to say *why* you did something

Java FindBugs



Practical information

- When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature
- Before you start refactoring, check that you have a solid suite of tests. These tests must be self-checking.
- Make sure all tests are fully automatic and that they check their own results.
- Run your tests frequently. Localize tests whenever you compile—every test at least every day.
- It is better to write and run incomplete tests than not to run complete tests
- Think of the boundary conditions under which things might go wrong and concentrate your tests there
- Don't forget to test that exceptions are raised when things are expected to go wrong
- When you get a bug report, start by writing a unit test that exposes the bug.
- Refactoring changes the programs in small steps. If you make a mistake, it is easy to find the bug.

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

Conclusion

- Refactoring is just a way of rearranging code
  - Refactorings are used to solve problems
  - If there's no problem, you shouldn't refactor
- The notion of "bad smells" is a way of helping us recognize when we have a problem
  - Familiarity with bad smells helps us avoid them in the first place
- Refactorings are mostly pretty obvious
  - Most of the value in discussing them is just to bring them into our "conscious toolbox"
  - Refactorings have names in order to crystalize the idea and help us remember it

Profiling

- What and how

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!

Profiling

- "Measure the behaviour of a program as it runs"
- Note: can profile different things
  - execution speed
  - memory usage
  - ...

Profiling concepts

- How does it work?
  - Sampling: gather information from time to time
    - Less accurate
    - Less performance overhead
  - Code instrumentation: modify program to analyze itself
    - Full instrumentation is very exact
    - Slower
    - Risc for Heisenbugs
    - Can be manual, static, dynamic, ...

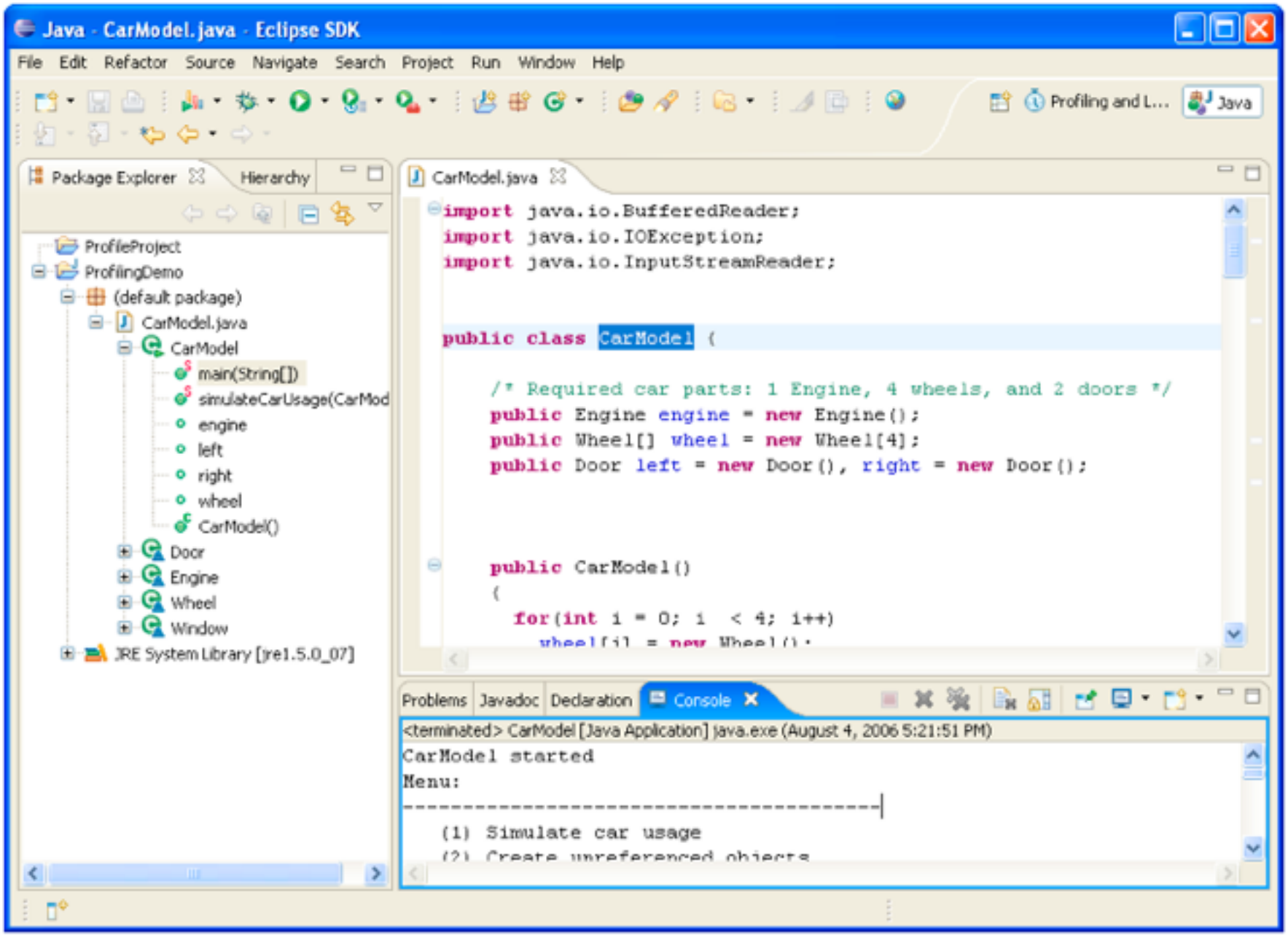
Profiler Tools

- Can be integrated in Development Environment
  - linked with code: can highlight slow methods, …
  - make profile data understandable and usable
- Can be stand-alone
  - no need to get project in IDE just to profile

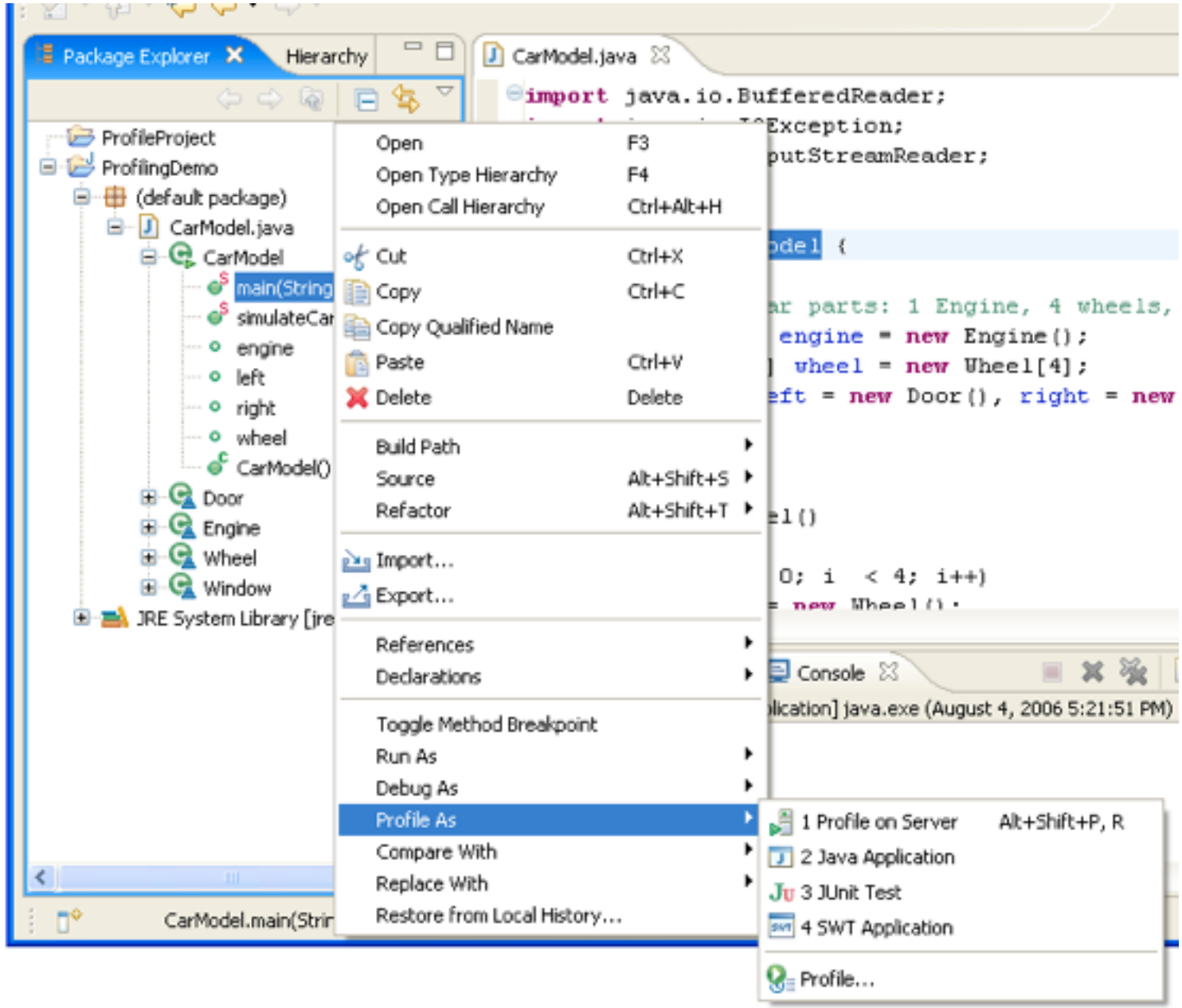
Example: Java Profiling in Eclipse

- Java profiling can be installed in Eclipse
  - Does Memory and Execution Time profiling
    - local or remote

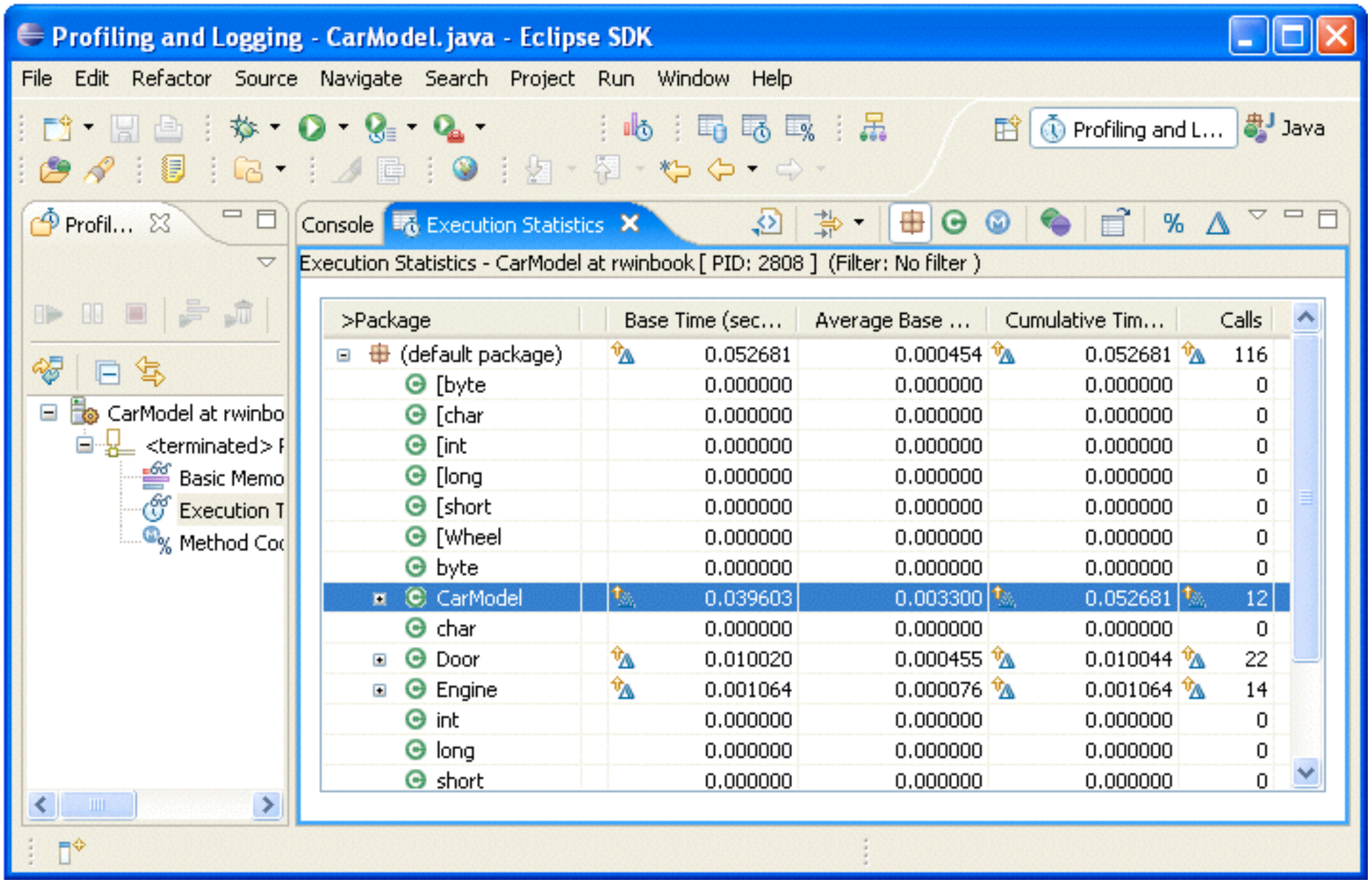
We have a Java project to profile...



Profile the main function

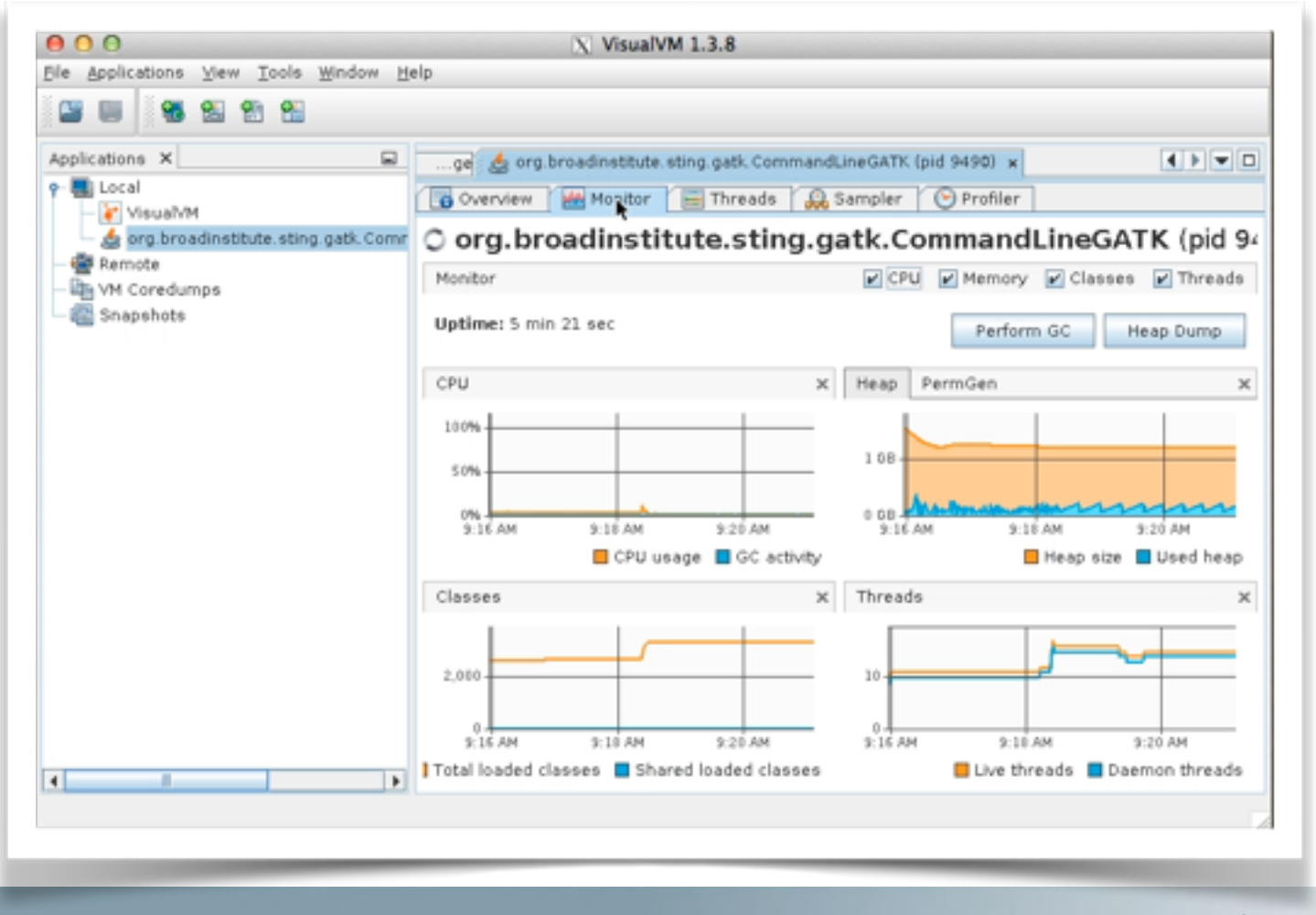


View results in Profiling perspective



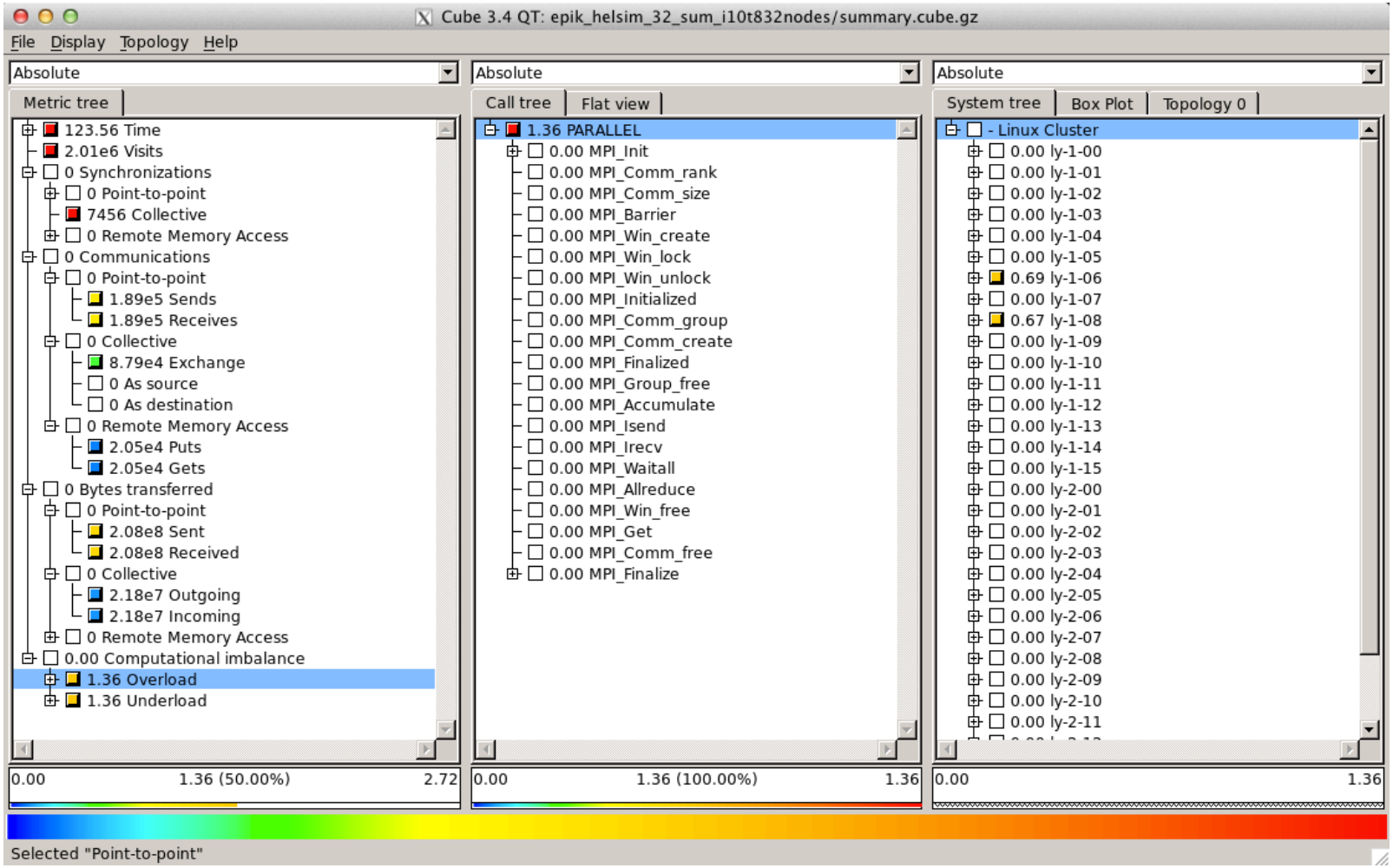
Example: VisualVM (http://visualvm.java.net/)

- monitor and/or sample CPU time and memory
- Easy to use, stand-alone
- See video



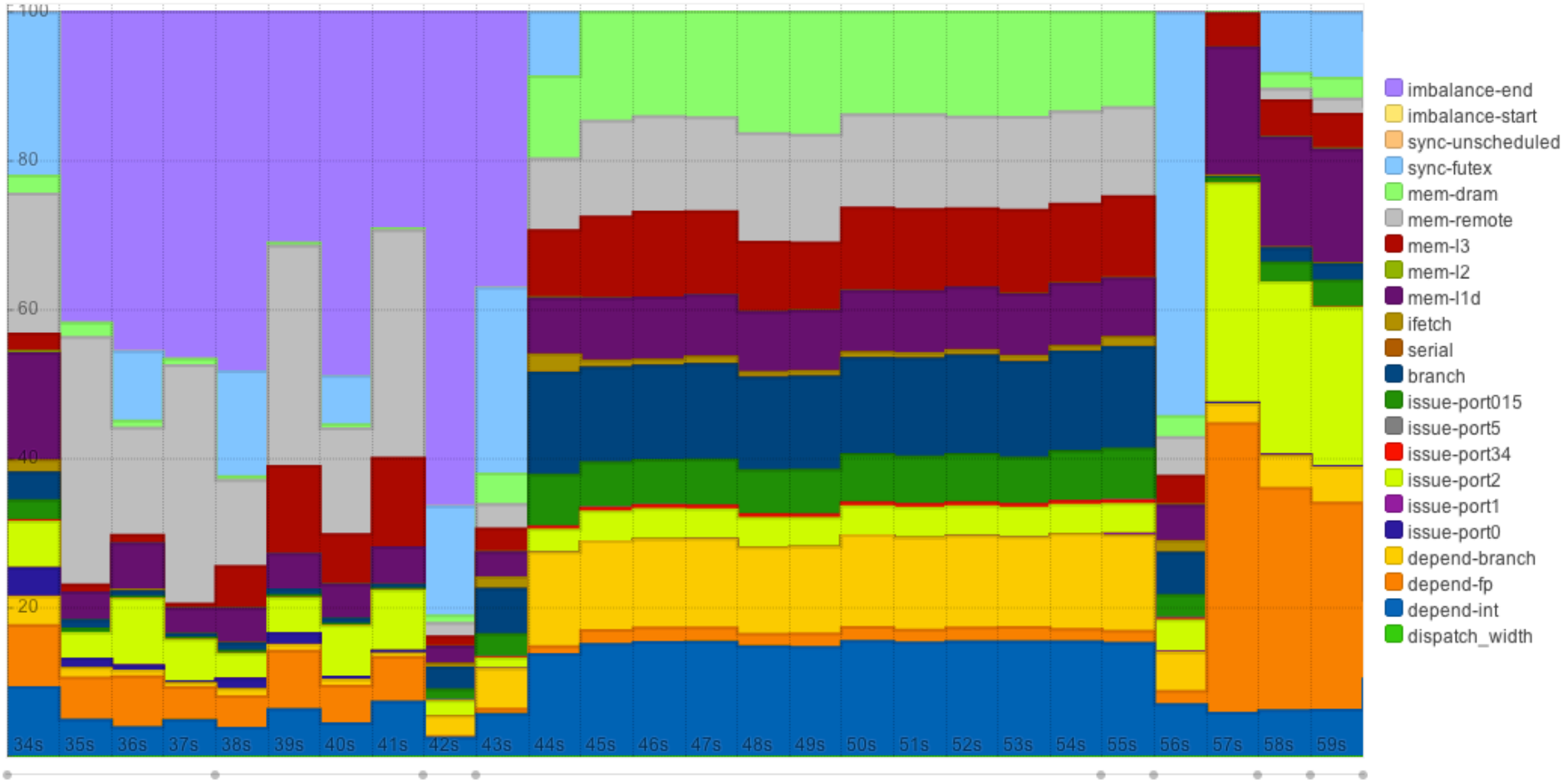
Other useful tools exist for profiling…

"Scalasca" : spot communication&synchronization imbalances in MPI programs (http://scalasca.org)

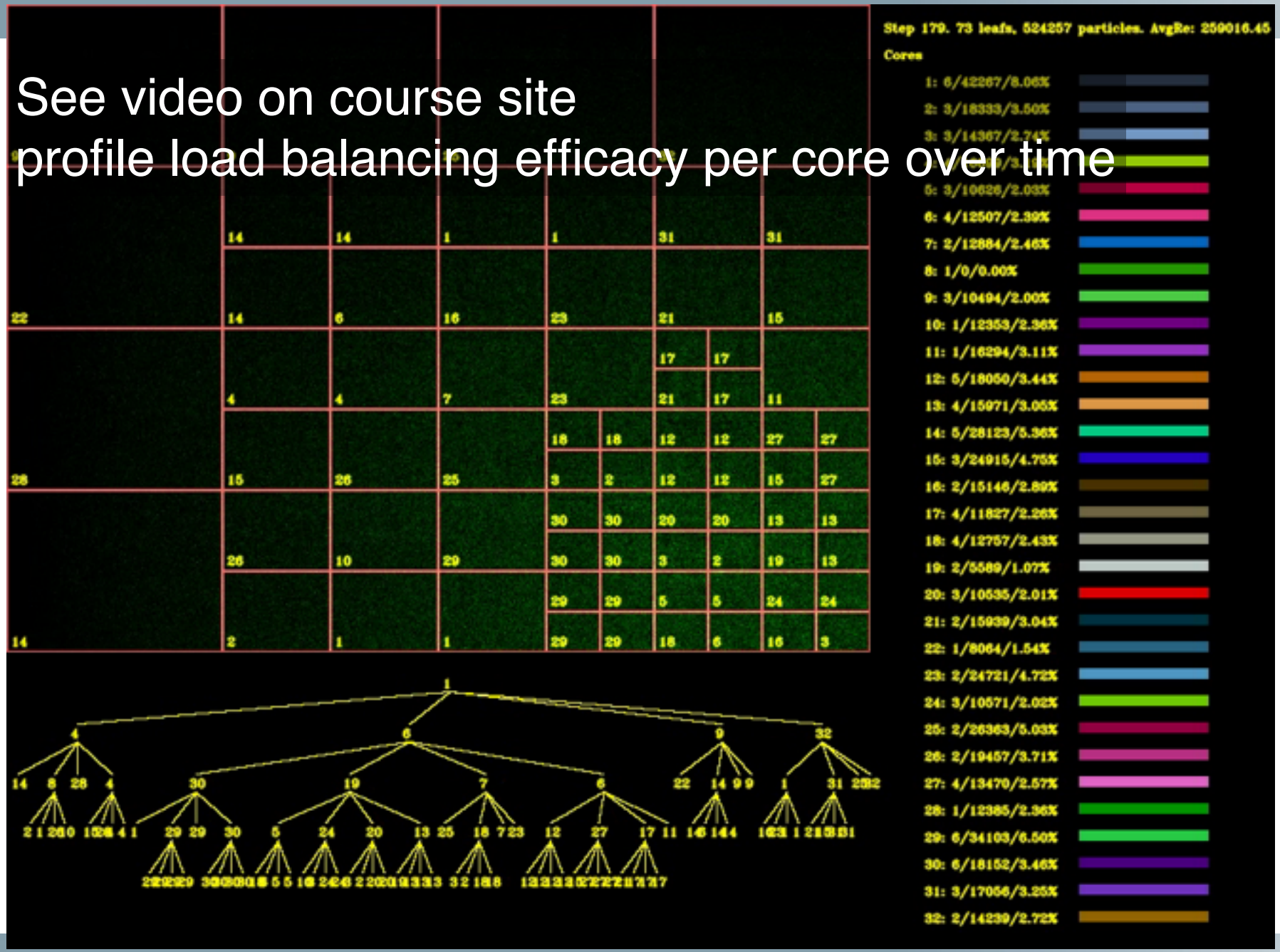


Other useful tools exist for profiling…

“Sniper” : fast hardware simulator for detailed analysis (http://snipersim.org)



Sometimes you should *roll your own*



Conclusion

- Make it Work, Make it Right, Make it Fast
- Unit testing remove fear of making changes
- Refactoring remove fear of making changes
- Profiling tells you where to make performance-related changes
  - focus your effort

License: Creative Commons 4.0

http://creativecommons.org/licenses/by-sa/4.0/

You are free to:

Share — copy and redistribute the material in any medium or format

Adapt — remix, transform, and build upon the material

for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give **appropriate credit**, provide a link to the license, and **indicate if changes were made**. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the **same license** as the original.

No additional restrictions — You may not apply legal terms or **technological measures** that legally restrict others from doing anything the license permits.