Software Reengineering Testing & Refactoring

Martin Pinzger
Delft University of Technology

TU Delft

SERG

Outline

Legacy code dilemma

Testing patterns

Refactoring

- Why, when, and how to refactor?

Refactoring examples

Conclusions

What we typically do in reengineering?

We are improving the software in some way:

- Improve performance
- Improve internal structure
  - As to make future feature requests easier
- Improve technologies under the hood
  - New database, new transaction manager, …

Keyword:
behavior preserving

In essence, the end-user experience remains the same

Hey, version 3.0.1 didn't bring on any changes! ☹

What we (don't) want

Improve the software internally

No problem! We just use the existing test suite!

Introduce bugs in well working software

I'm sorry, we don't have any tests ☹

Strategies

Edit and Pray

Regression tests? Great!

- But not if they are at the application level

Unit testing is more efficient

The Legacy Code Dilemma

When we change code, we should have tests in place

To put tests in place, we often need to change code

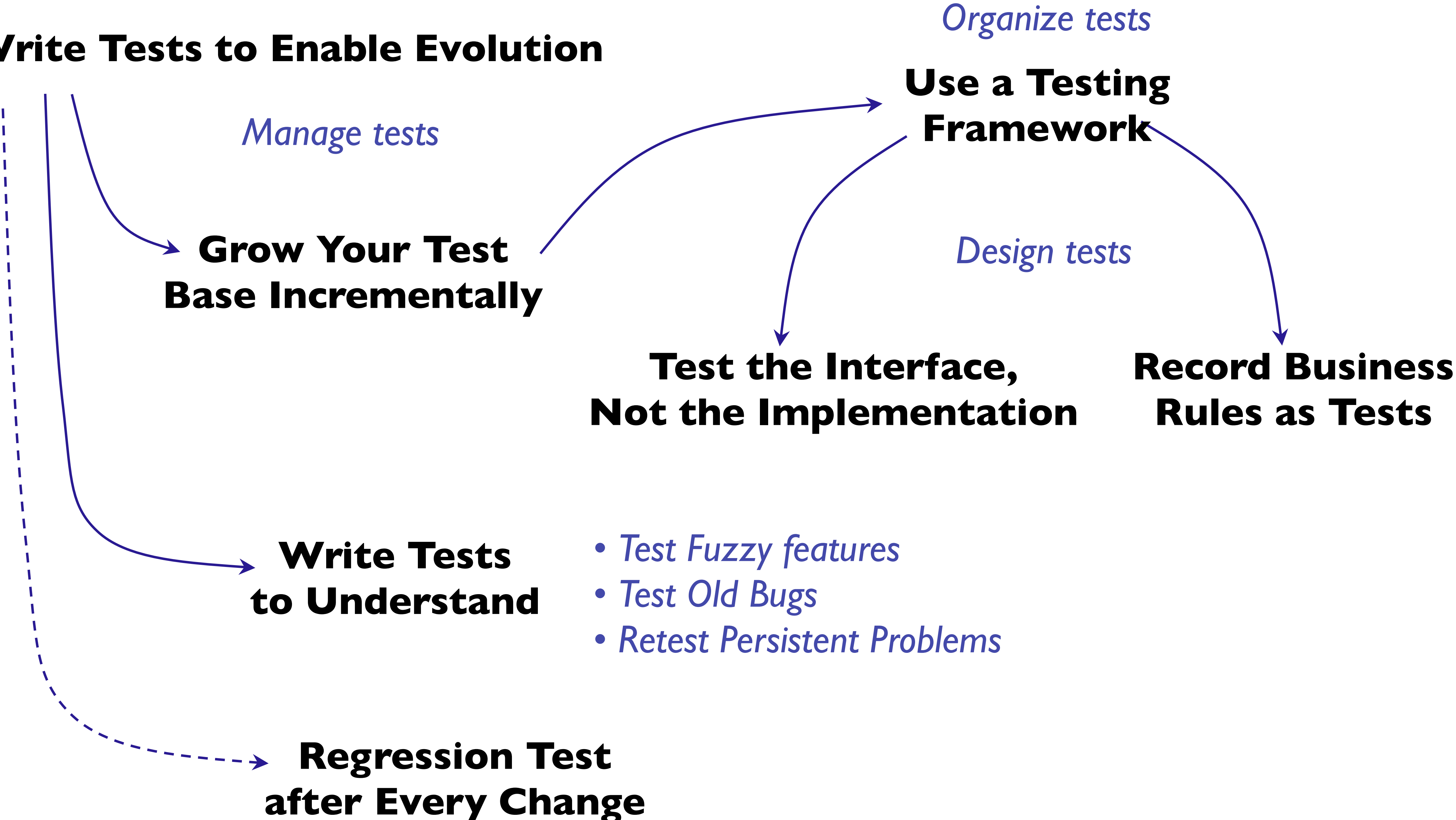
More info on how to handle this dilemma in the next lecture
"Working Effectively with Legacy Code"

Testing patterns

Tests: Your Life Insurance

Reduce risks posed by reengineering

Testing Patterns



Write Tests to Enable Evolution

Problem: How do you minimize the risks of change?

Solution: Introduce automated, repeatable, stored tests

xUnit Tests

Example: JUnit Test Case

```
public class TestPerson {
	private Person p;

	@BeforeClass
	public static void setUpBeforeClass() { ... }

	@AfterClass
	public static void tearDownAfterClass() { ... }

	@Before
	protected void setUp() throws Exception {
		super.setUp();
		p = new Person("Huga Bimbo");
	}

	@After
	protected void tearDown() throws Exception {
		super.tearDown();
	}

	@Test
	public void testGetName() {
		assertEquals("Name must be Huga Bimbo", p.getName(), "Huga Bimbo");
	}
}
```

Write Tests to Understand

Problem: How to decipher code without adequate tests or documentation?

Solution: Encode your hypotheses as test cases

- Exercise the code
- Formalize your reverse-engineering hypotheses
- Develop tests as a by-product

Grow Your Test Base Incrementally

Problem: When to start and when to stop writing tests?

Solution: Grow Your Test Base Incrementally

- First test critical components
  - Business value, likely to change, etc.
- Test bugs that have been reported
- Keep a snapshot of old system
  - Run new tests against old system

Test the Interface

Problem: How do you protect your investment in tests?

Solution: Apply black-box testing

- Test interfaces, not implementations
  - Be sure to exercise the boundaries
- Test scenarios
  - Use tools to check for coverage
- Beware:
  - Enabling testing will influence your design!

Other Testing Patterns

Retest Persistent Problems

Always tests these, even if you are making no changes to this part of the system

Test Fuzzy Features

Identify and write tests for ambiguous or ill-defined parts of the system

Test Old Bugs

Examine old problems reports, especially since the last stable release

— DeLano and Rising, 1998

Well-Designed Tests

Automation

Tests should run without human intervention

Persistence

Each test documents its test data, actions, and expected results

Repeatability

Tests can run after any change

Unit testing

Tests should be associated with software components

Independence

Each test should minimize its dependencies on other tests (avalanche effects)

Unit vs. Integration Tests

A test is not a unit test if:

- It talks to a database
- It communicates across the network
- It touches the file system
- You have to do things to your environment to run it
  - e.g., change config files

Tests that do this are integration tests

How much do your tests cover?

Java - CursorableLinkedList.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

JUnit

Finished after 34,898 seconds

Runs: 13009/13009 Errors: 0 Failures: 0

Failures | Hierarchy

- junit.framework.TestSuite
  - junit.framework.TestSuite
    - TestBagUtils
    - org.apache.commons.collections.TestClos
    - org.apache.commons.collections.TestColle
    - TestBufferUtils
    - TestEnumerationUtils
    - org.apache.commons.collections.TestFact
    - TestListUtils
    - TestMapUtils
    - org.apache.commons.collections.TestPred
    - TestSetUtils
    - org.apache.commons.collections.TestTrar
    - TestArrayStack
    - TestBeanMap
    - org.apache.commons.collections.TestBina
    - TestBoundedFifoBuffer
    - TestBoundedFifoBuffer2
    - TestCursorableLinkedList
    - TestDoubleOrderedMap
    - org.apache.commons.collections.TestExte
    - TestFastArrayList
    - TestFastArrayList1
    - TestFastHashMap
    - TestFastHashMap1
    - TestFastTreeMap
    - TestFastTreeMap1

Failure Trace

CursorableLinkedList.java

```java
public boolean addAll(int index, Collection c) {
    if(c.isEmpty()) {
        return false;
    } else if( size == index ||  size == 0) {
        return addAll(c);
    } else {
        Listable succ = getListableAt(index);
        Listable pred = (null == succ) ? null : succ.prev();
        Iterator it = c.iterator();
        while(it.hasNext()) {
            pred = insertListable(pred,succ,it.next());
        }
        return true;
    }
}
```

Problems | Javadoc | Declaration | Console | Coverage

TestAllPackages (31.10.2006 15:04:14)

Element	Coverage	Covered Lines	Total Lines
java - commons-collections	79,5 %	10927	13738
org.apache.commons.collections	74,1 %	3842	5183
ArrayStack.java	86,5 %	32	37
BagUtils.java	86,7 %	13	15
BeanMap.java	72,4 %	155	214
BinaryHeap.java	87,6 %	127	145
BoundedFifoBuffer.java	93,2 %	82	88
BufferOverflowException.java	55,6 %	5	9
BufferUnderflowException.java	88,9 %	8	9
BufferUtils.java	30,8 %	4	13
ClosureUtils.java	93,9 %	31	33
CollectionUtils.java	92,4 %	293	317
ComparatorUtils.java	8,6 %	3	35
CursorableLinkedList.java	85,4 %	444	520

Writable | Smart Insert | 149 : 28

Debugging vs. Testing

Debugging Sucks

Testing Rocks

Refactoring

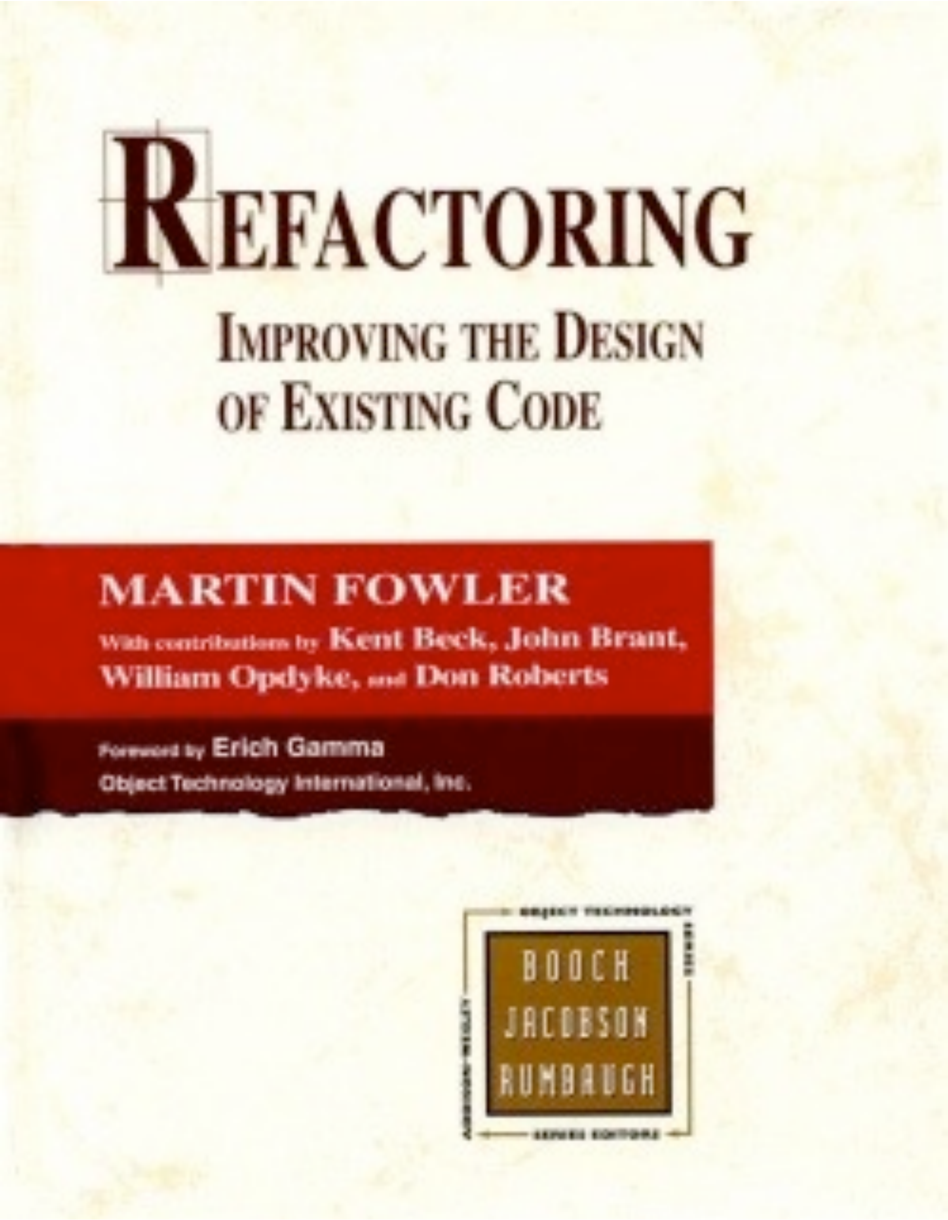
What is Refactoring?

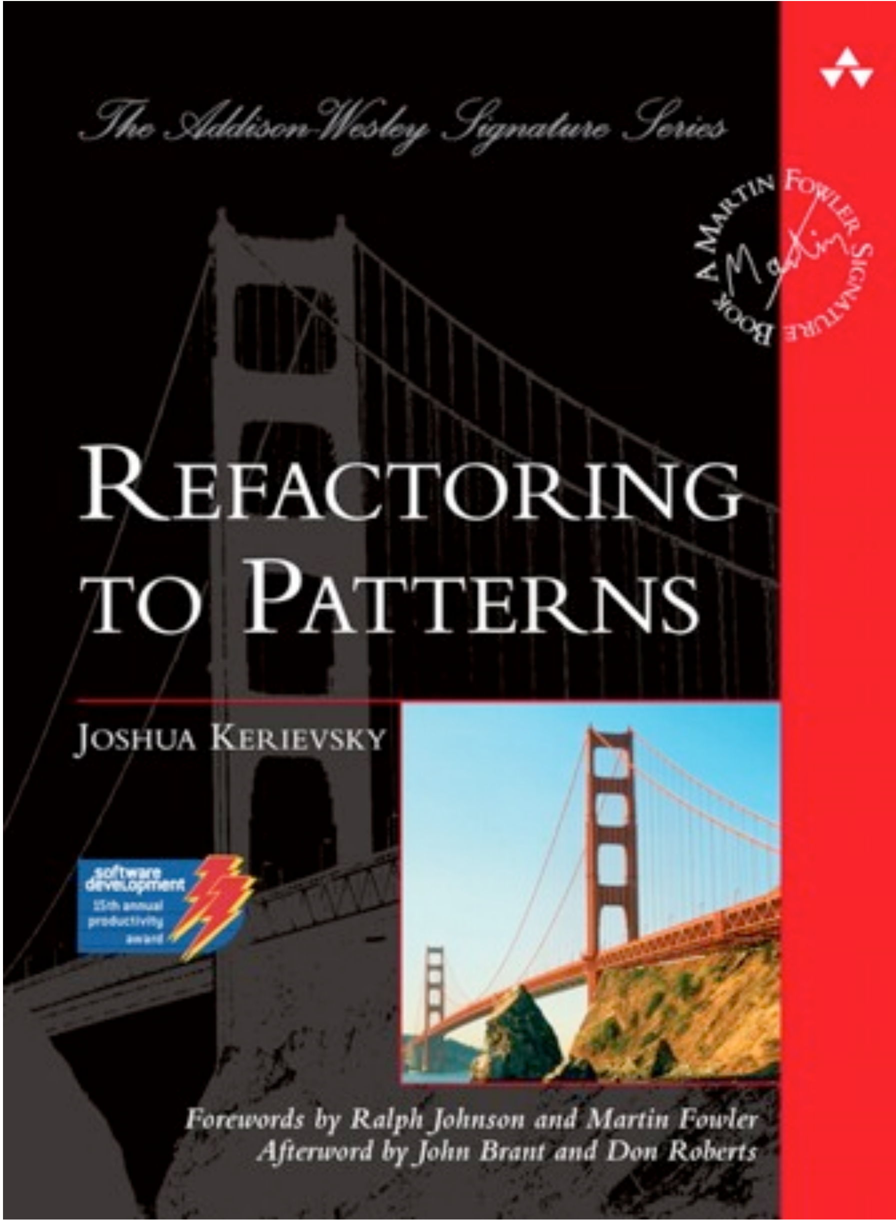
"The process of changing a software system without altering the external behavior of the code, yet improving its internal structure."

Which one is refactoring?

- Fixing a bug
- Adding threading to improve performance
- Renaming method identifiers to improve readability

Refactoring Literature





Why to Refactor?

Prevent “design decay”

Clean up mess in the code

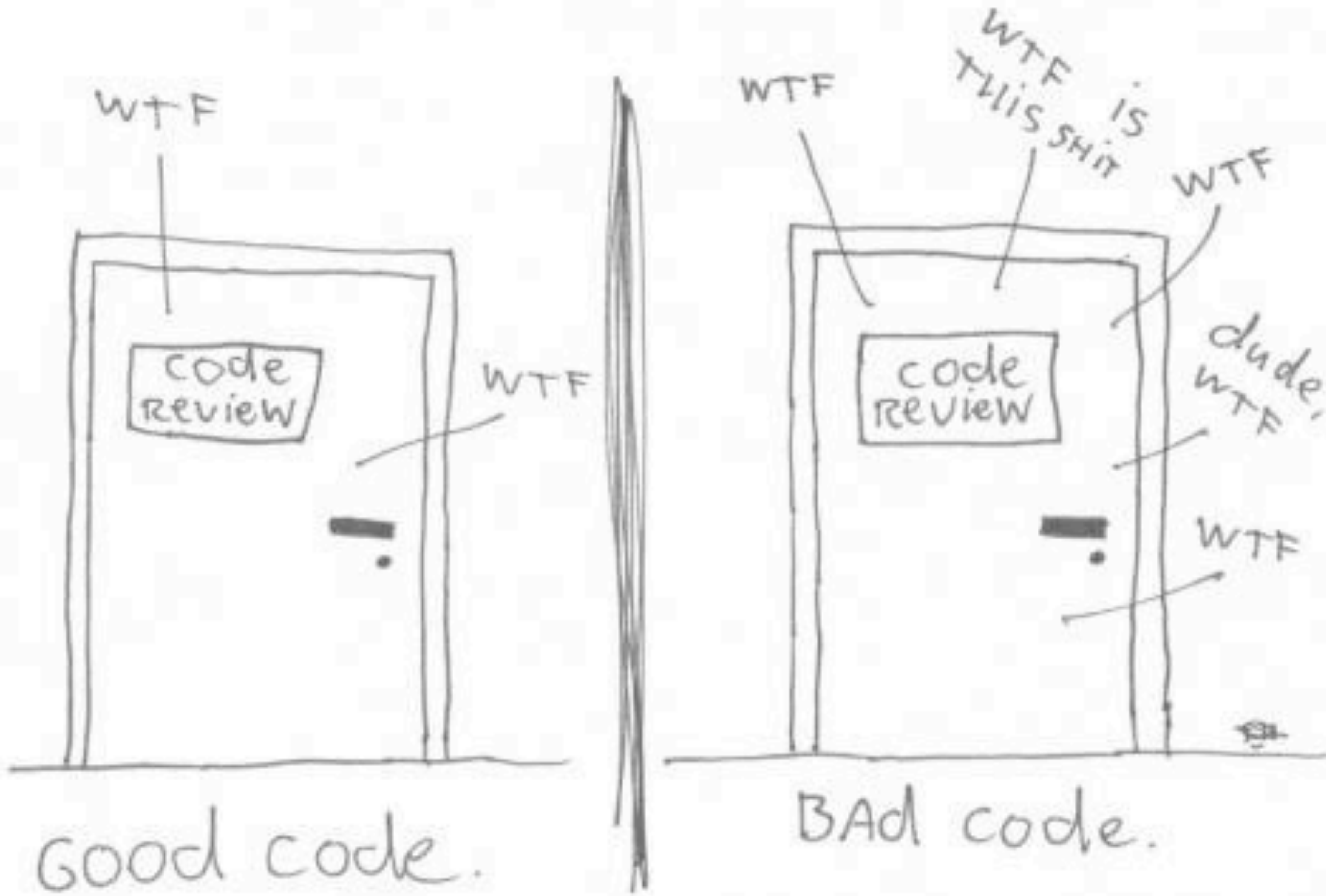
Simplify the code

Increase readability and understandability

Find bugs

...

When to Refactor?



When to Refactor?

Rule of Three

- If code is replicated three times, it should be extracted into a new procedure

When you add functionality

When you learn something about the code

When you fix a bug

When the code smells

-> “All the time”

When Not to Refactor?

When the tests are **not** passing

When you have impending deadlines

- Cunningham's idea of unfinished refactoring as debt

How to Refactor? - Refactoring Workflow

1. Make sure your tests pass

2. Find some code that “smells”

3. Determine how to simplify this code

4. Make the simplifications

5. Run tests to ensure things still work correctly

6. Repeat the simplify/test cycle until the smell is gone

Refactorings

Composing Methods

Extract Method, Inline Method, ...

Moving Features Between Objects

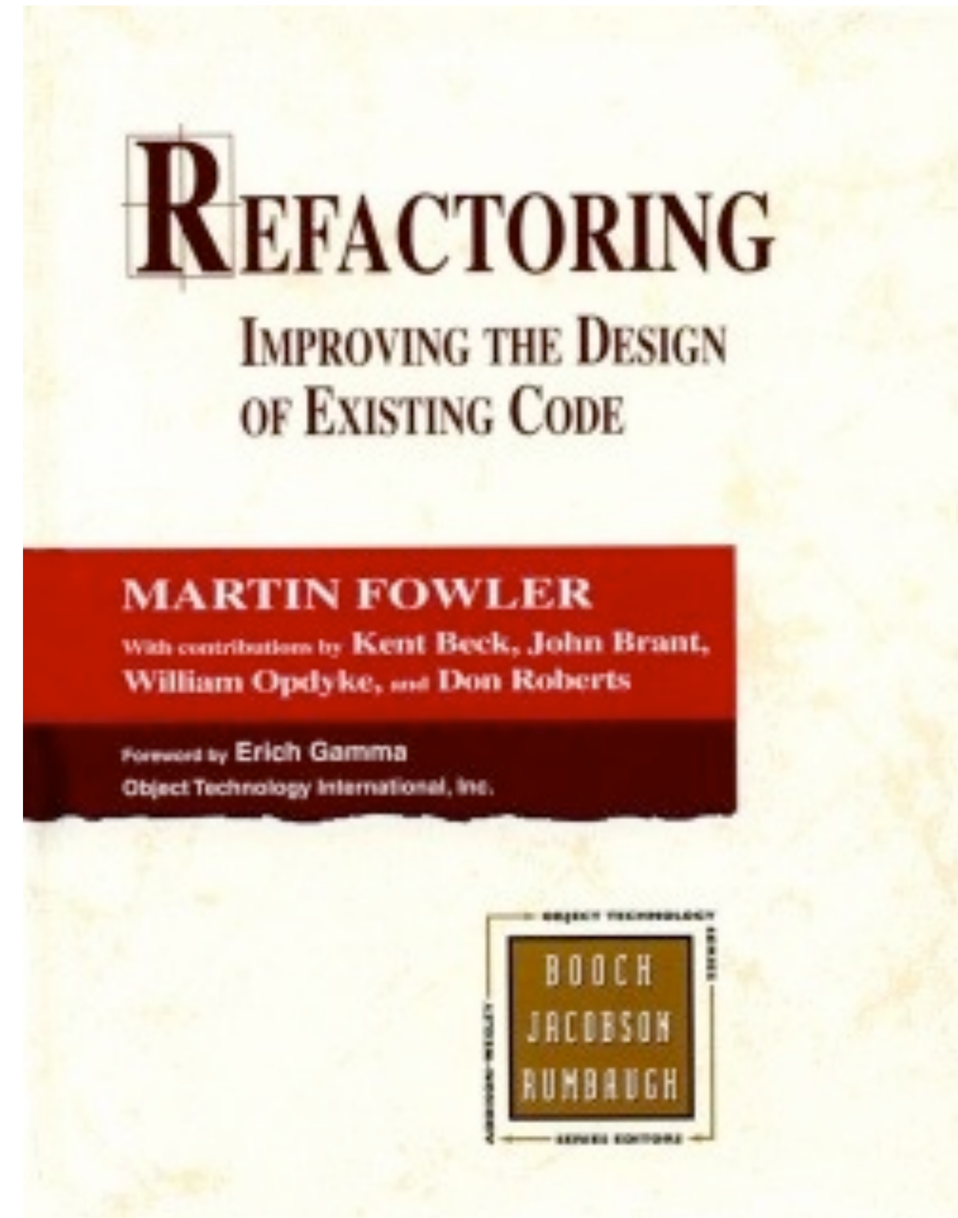
Move Method, Move Field, Hide Delegate, ...

Organizing Data

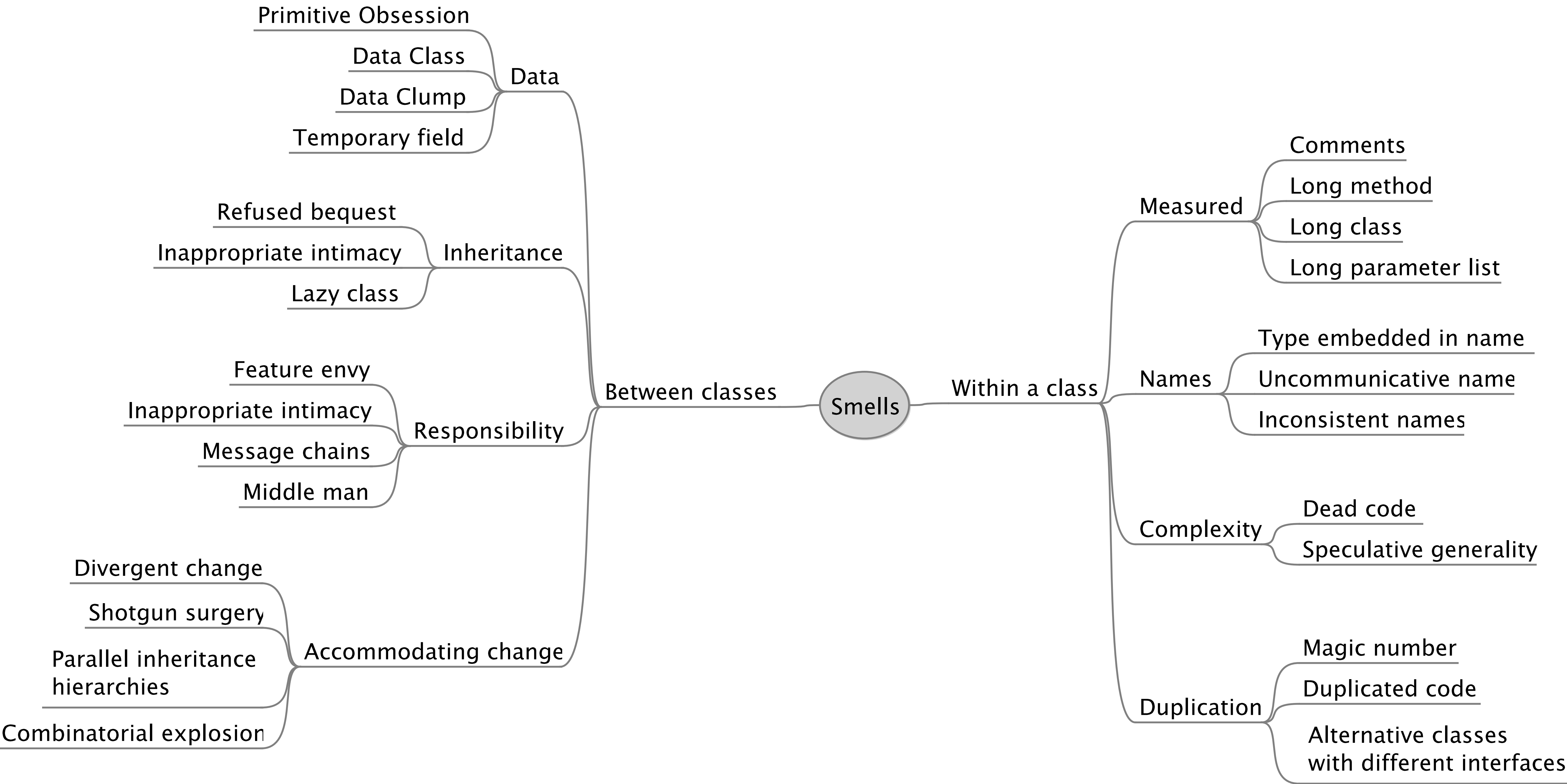
Replace Data Value with Object, ...

Simplifying Conditional Expressions

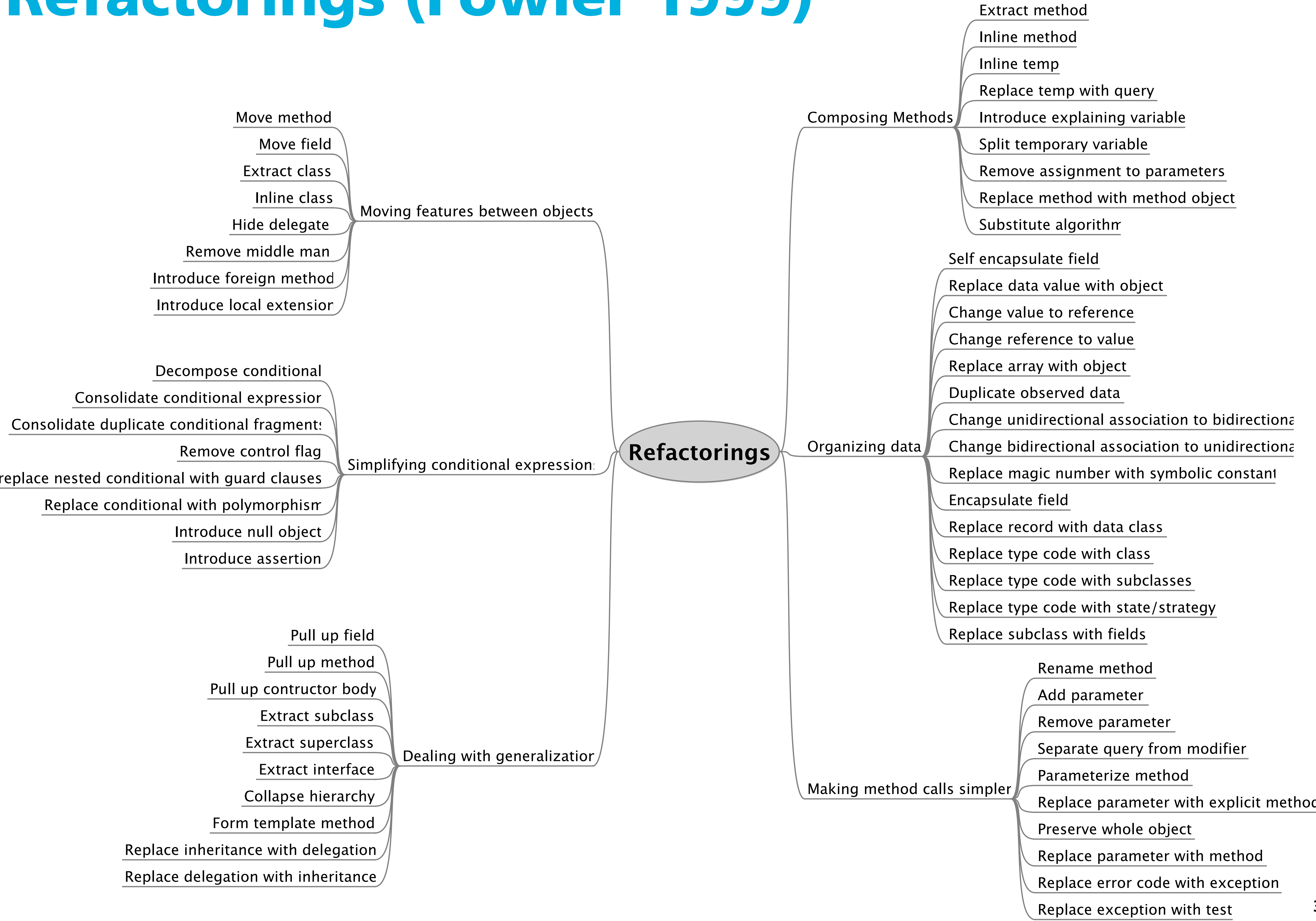
Decompose Conditionals, ...



Code Smells (Fowler 1999)



Refactorings (Fowler 1999)



Refactoring Examples

Smell 1: Duplicated Code

Extract Method

- Gather duplicated code

Pull Up Field

- Move to a common parent

Form Template Method

- Gather similar parts, leaving holes

Extract Class

- For unrelated classes, create a new class with functionality

Smell 2: Long Method

Extract Method

Extract related behavior

Replace Temp with Query

Remove temporaries when they obscure meaning

Introduce Parameter Object

Slim down parameter lists by making them into objects

Decompose Conditionals

Conditional and loops can be moved to their own methods

Example: Long Method

```
public double computePrice() {
    double totalAmount = 0;
    foreach (Rental each : rentals.elements()) {
        double thisAmount = 0;
        // comp. amounts for each line
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
        }
        totalAmount += thisAmount;
    }
    return totalAmount;
}
```

Example: Apply Extract Method

```
public double computePrice() {
    double totalAmount = 0;
    foreach (Rental each : rentals.elements()) {
        totalAmount += computePricePerRental(each);
    }
    return totalAmount;
}

public double computePricePerRental(Rental aRental) {
    double thisAmount = 0;
    switch (aRental.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount += 2;
        if (aRental.getDaysRented() > 2)
           thisAmount += (aRental.getDaysRented() - 2) * 1.5;
        break;
    case Movie.NEW_RELEASE:
        thisAmount += aRental.getDaysRented() * 3;
        break;
    }
    return thisAmount;
}
```

Example: Apply Extract Method 2nd

```
public double computePrice() { ... }

public double computePriceForRental(Rental aRental) {
    double thisAmount = 0;
    switch (aRental.getMovie().getPriceCode()) {
    case Movie.REGULAR:
        thisAmount = computePriceRentalRegularMovie(aRental); break;
    case Movie.NEW_RELEASE:
        thisAmount = computePriceRentalNewRelease(aRental); break;
    }
    return thisAmount;
}

public double computePriceRentalRegularMovie(Rental aRental) {
   double thisAmount = 2;
   if (aRental.getDaysRented() > 2)
       thisAmount += (aRental.getDaysRented() - 2) * 1.5;
   return thisAmount;
}

public double computePriceRentalNewRelease(Rental aRental) {
   return aRental.getDaysRented() * 3;
}
```

Smell 3: Divergent Change

“If you find yourself repeatedly changing the same class then there is probably something wrong with it.“

Extract Class

- Group functionality commonly changed into a class

Smell 4: Feature Envy

"If a method seems more interested in a class other than the class it actually is in."

Move Method

- Move the method to the desired class

Extract Method

- If only part of the method shows the symptoms

Example: Feature Envy

```
public class CapitalStrategy…
   public double capital(Loan loan) {
      if (loan.getExpiry() == null && loan.getMaturity() != null) {
         return loan.getCommitment() * loan.duration() * loan.riskFactor();
      }
      if (loan.getExpiry() != null && loan.getMaturity() == null) {
         if (loan.getUnusedPercentage() != 1.0) {
            return loan.getCommitment() * loan.getUnusedPercentage() *
               loan.duration() * loan.riskFactor();
         } else {
            return (loan.outstandingRiskAmount() * loan.duration() * loan.riskFactor())
              + (loan.unusedRiskAmount() * loan.duration() * loan.unusedRiskFactor());
         }
      }
      return 0.0;
   }
    ...
}
```

Comments

Often are a sign of unclear code (smell)...

Not necessarily bad but may indicate areas where the code is not as clear as it should be

- Extract Method
- Introduce Assertion

More Smells & Refactorings

Smell	Refactorings
Large Class	Extract Class, Extract Subclass, Extract Interface, Replace Data Value with Object
Shotgun Surgery	Move Method, Move Field, Inline Class
Long Parameter List	Replace Parameter with Method, Introduct Parameter Object, Preserve Whole Object
Data Class	Move Method, Encapsulate Field, Encapsulate Collection

Refactoring Exercise

Refactoring the Movie Rental Application

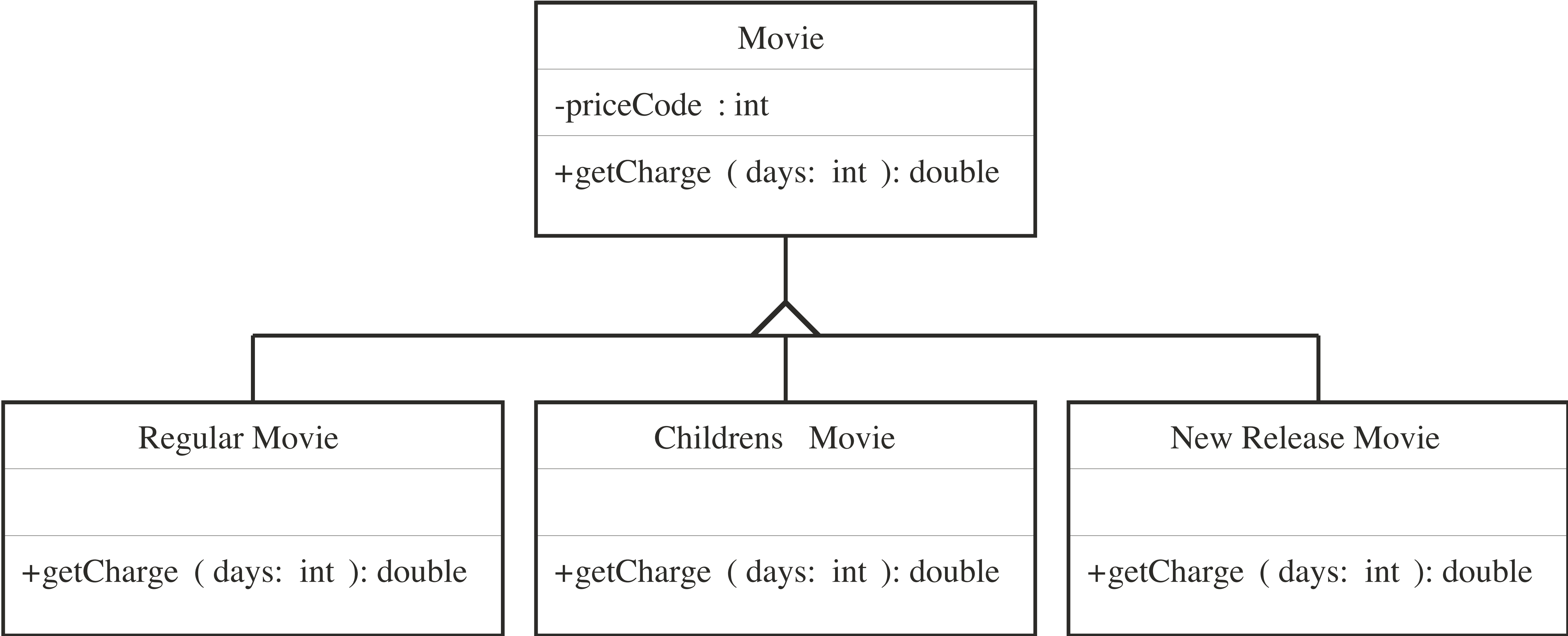
Download the source code from the Reengineering web-site

Import the Eclipse project

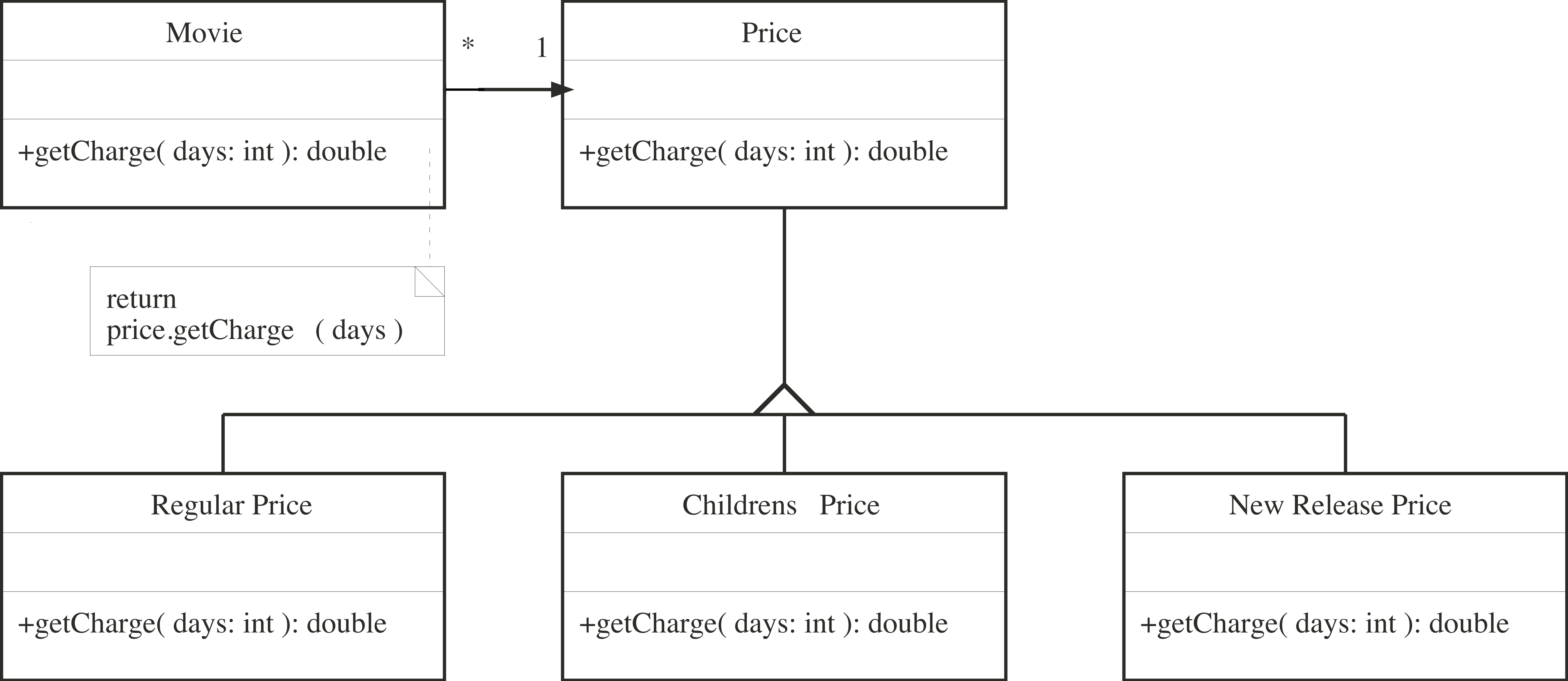
Refactor the “Bad Smells” that you find in the current release

Remember to add tests first!

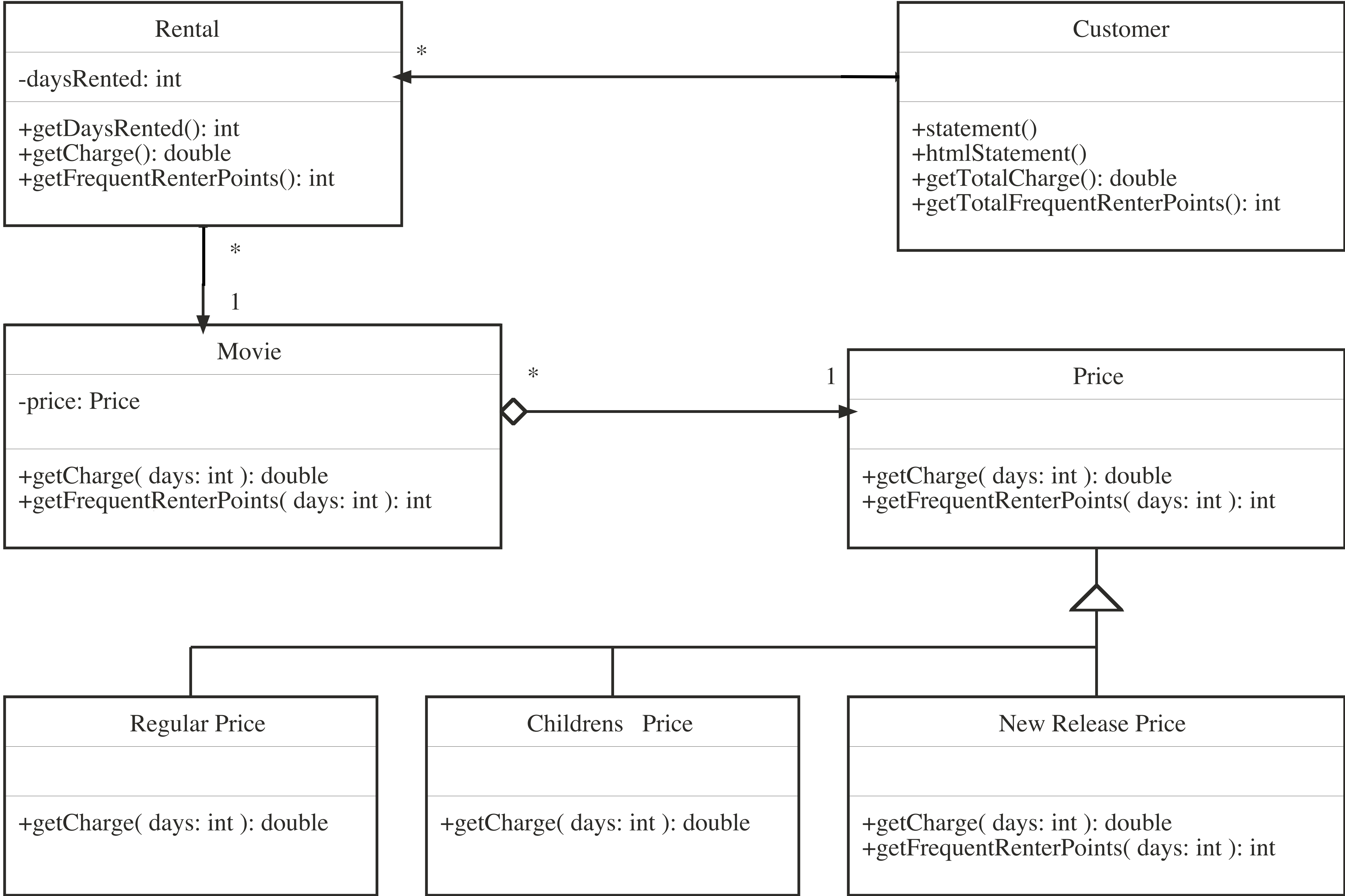
Potential Solution?



Solution with State Pattern



Solution after Refactoring



Problems with Refactoring

Taken too far

- Refactoring can lead to incessant tinkering with the code, trying to make it perfect

Refactoring code when the tests don’t work

- Leads to potentially dangerous situations

Databases can be difficult to refactor

Refactoring published API can break client code

Why Developers are Reluctant to Refactor?

Lack of understanding

Short-term focus

Not paid for overhead tasks like refactoring

Fear of breaking current program

Rules of Thumb to Refactoring

Refactoring may slow down execution

- But: “First do it, then do it right, then do it fast”

Clean first, then add new functionality

Do not meddle with things you do not understand to a large extent

Summary

Refactoring is improving the source code without changing the behavior of the system

Refactor all the time

- Make sure you have tests in place