# Code Refactoring

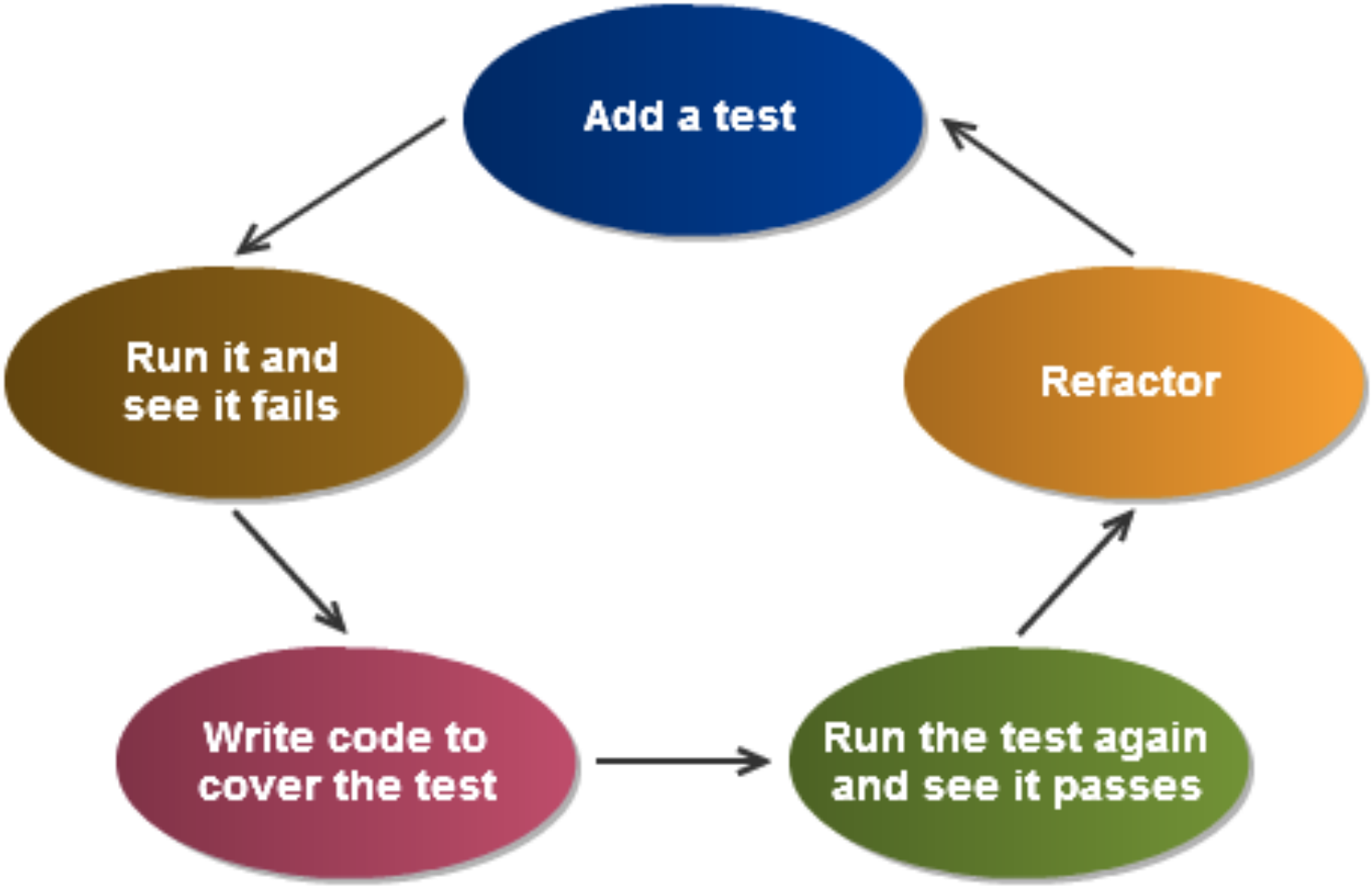
CS580 Advanced Software Engineering
http://cs580.yusun.io
November 10, 2014

Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

CAL POLY POMONA

# Test-Driven Development

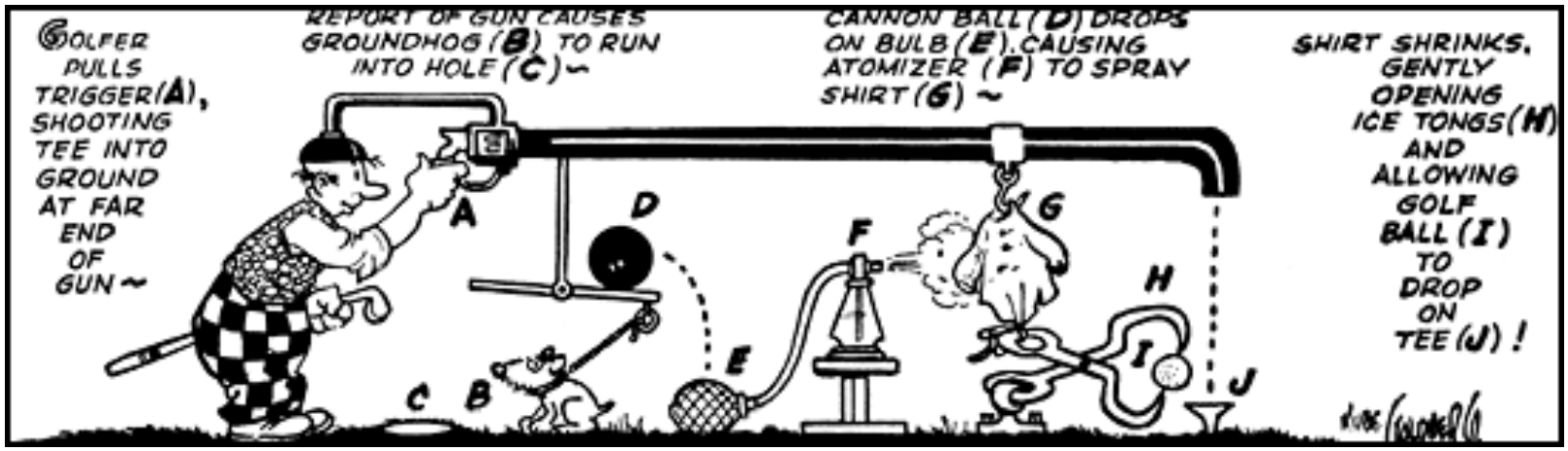
# Test-Driven Development



# The Problem: Software Drift

- Over many phases of maintenance, structure begins to decay (entropy)

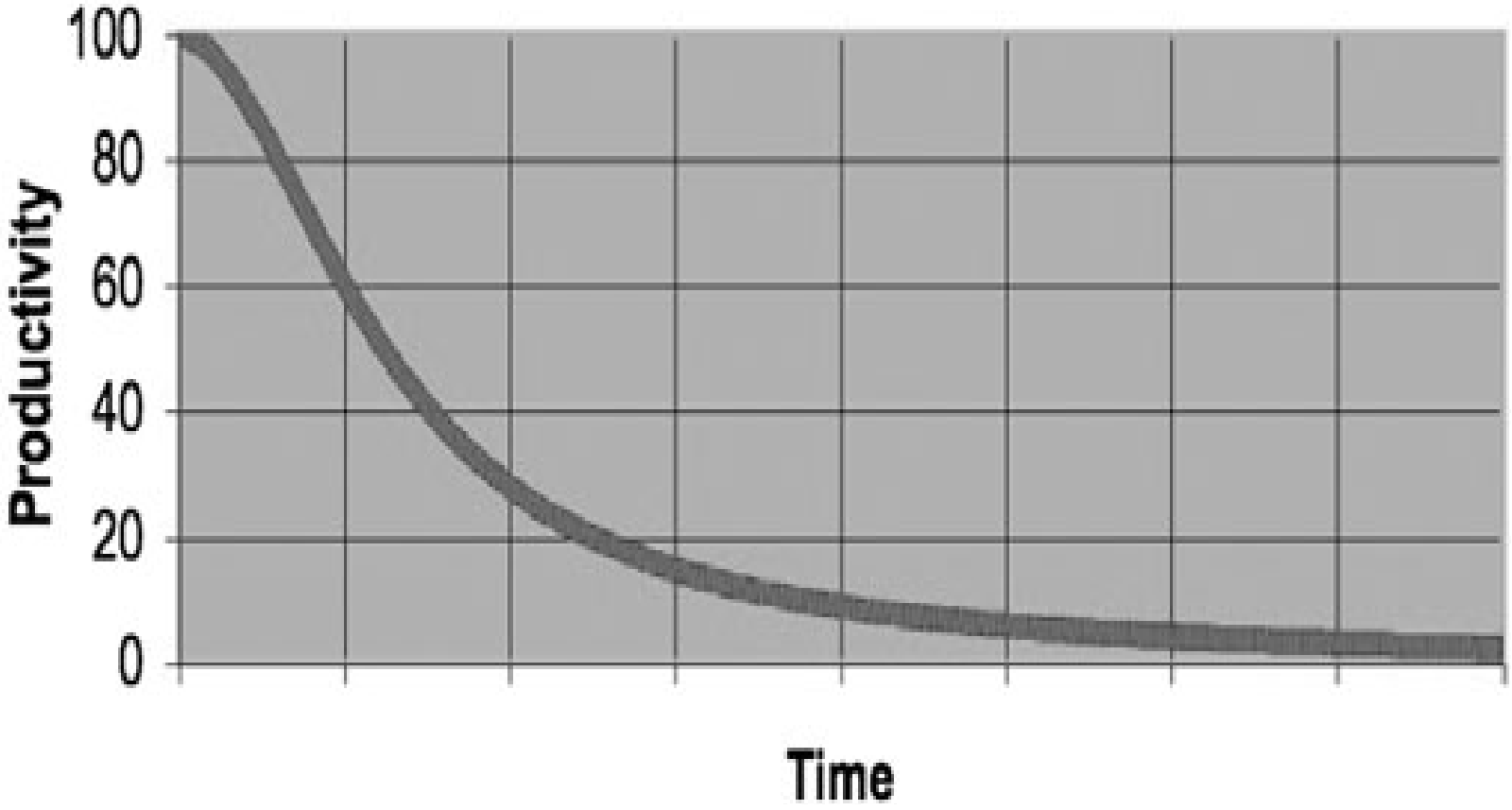
- Begins to resemble Rube Goldberg machine



# Software Nature – Software Entropy

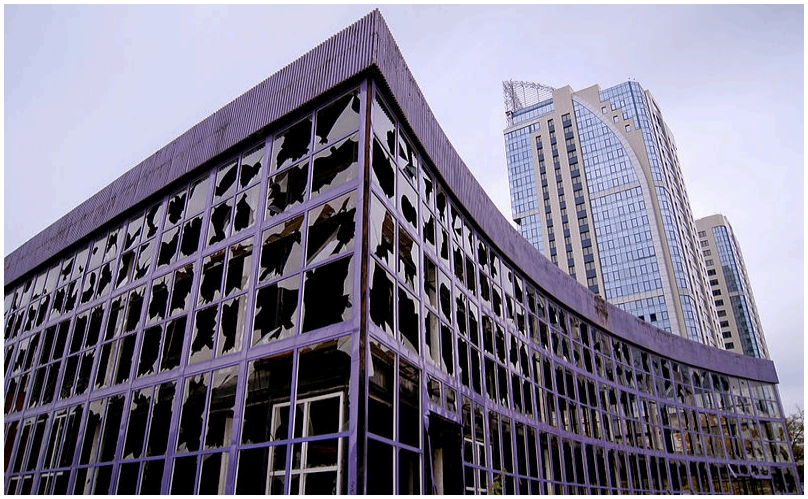
- Software tends to degrade / decay
- Software rot – like a piece of bad meat

# Developers Productivity vs. Time

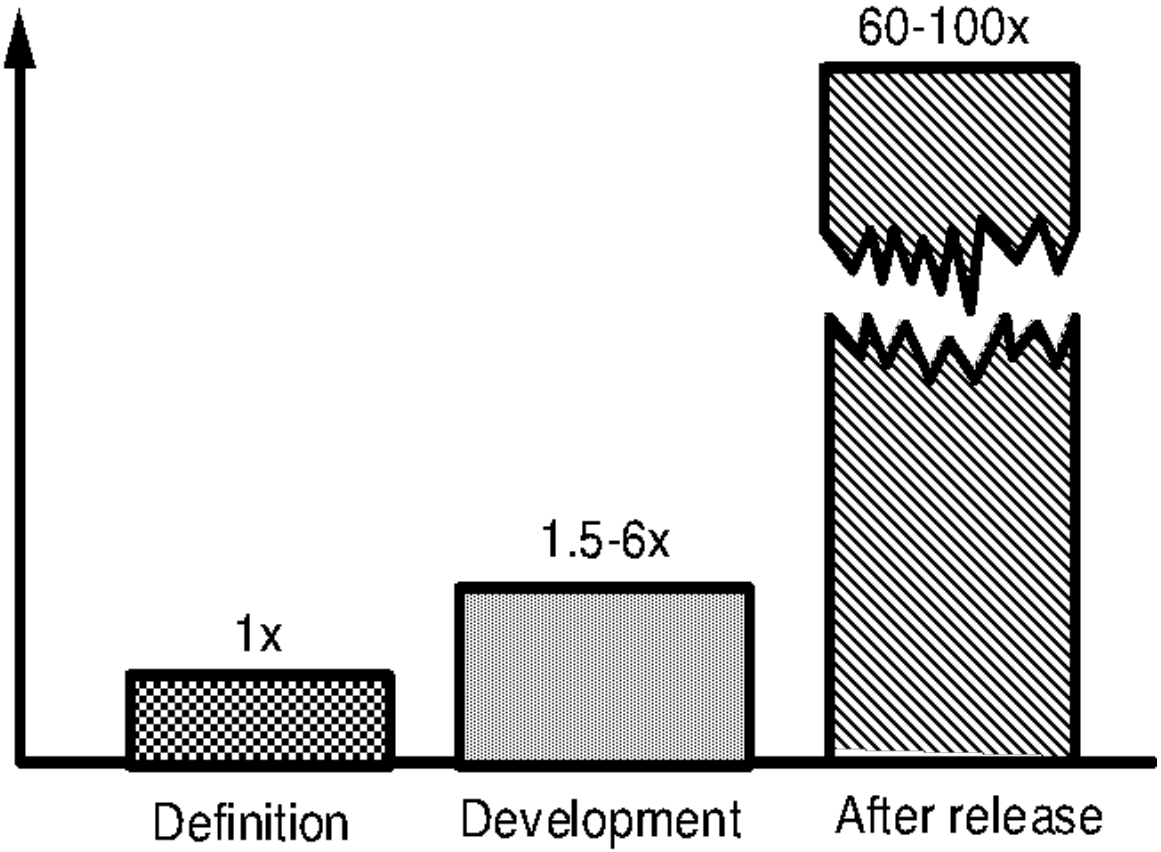


# Psychology Reason: Broken Window Theory

- Came from city crime researcher
- A broken window will trigger a building into a smashed and abandoned derelict
- So does the software
- Don't live with the Broken window



# The Cost of Change



# How to Prevent Software from Rotting?

- Applies OO design principles
- Uses design patterns
- Follows agile practices
- Refactoring will reduce the software entropy

# What is Refactoring?

- "*The process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure*" – [Fowler]

- "*A behavior-preserving source-to-source program transformation*" – [Roberts]

- "*A change to the system that leaves its behavior unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability*" – [Beck]

# Refactor to Understand

- The Obvious
  - Programs hard to read
    - Programs hard to understand
      - Programs hard to modify
  - Programs with duplicated logic are hard to understand
  - Programs with complex conditionals are hard to understand

# Refactor to Understand

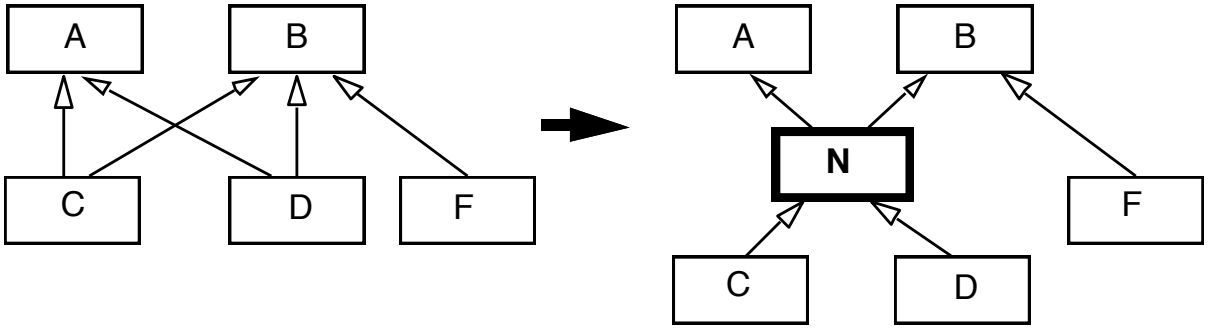
- Refactoring code creates and supports understanding
  - Renaming instance variables helps understanding methods
  - Renaming methods helps understanding responsibility
  - Iterations are necessary

# 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 | push variable up |
| | add parameter to method | create accessors |
| | move method to component | abstract variable |
| | extract code in new method | |

These simple refactorings can be combined to provide bigger restructurings such as the introduction of design patterns

# Example: Add Class




- Add Class (new name, package, superclasses, subclasses)
  - Preconditions
    - No class or global variable exists with new name in the same scope
    - Subclasses are all subclasses of all superclasses
  - Postconditions
    - New class is added into the hierarchy with superclasses as superclasses and subclasses as subclasses
    - New class has name new name
    - Subclasses inherit from new class and not anymore from superclasses

# Example: Rename Method

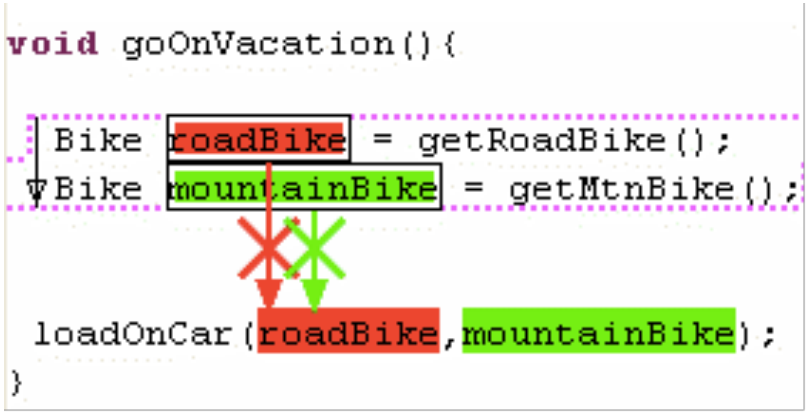
- Rename Method (new name, method)
- Preconditions
  - No method exists with the signature implied by new name in the inheritance hierarchy that contains method
- Postconditions
  - Method has new name
  - Relevant methods in the inheritance hierarchy have new name
  - Invocations of changed method are updated to new name

# Rename Method: Manual Steps

- Do it yourself approach
  - 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
- Automated refactoring is better!

# Refactoring Tools

- Based on provable transformations
  - Build parse tree of programs
  - Research – mathematical proof that refactoring does not change semantics; graph transformation
  - Embed refactoring in tool
- Speeds up refactoring
  - Extract method: select code, type in method name
- In Eclipse for Java and other languages

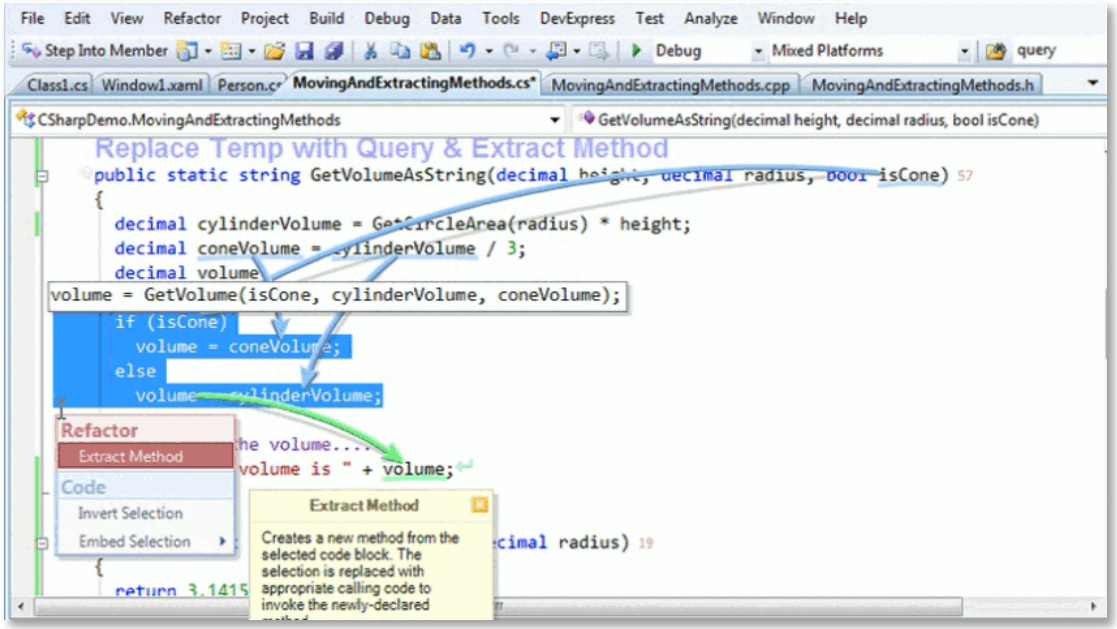


# Refactoring Tool Usage

- "Refactoring Tools: Fitness for Purpose" [Murphy-Hill and Black]
  - Includes survey of 112 people at the Agile Open Northwest 2007
  - On average, when a refactoring tool is available for a refactoring that programmers want to perform, they choose to use the tool 68% of the time; the rest of the time they refactor by hand

# Refactoring Tool Usage

- Refactoring activity requires multiple modal dialog boxes
  - Separation between program editing and refactoring tasks
  - A solution: visualize refactoring changes directly in the source editor



Screen shot of Refactor! Pro

# Typical Refactorings

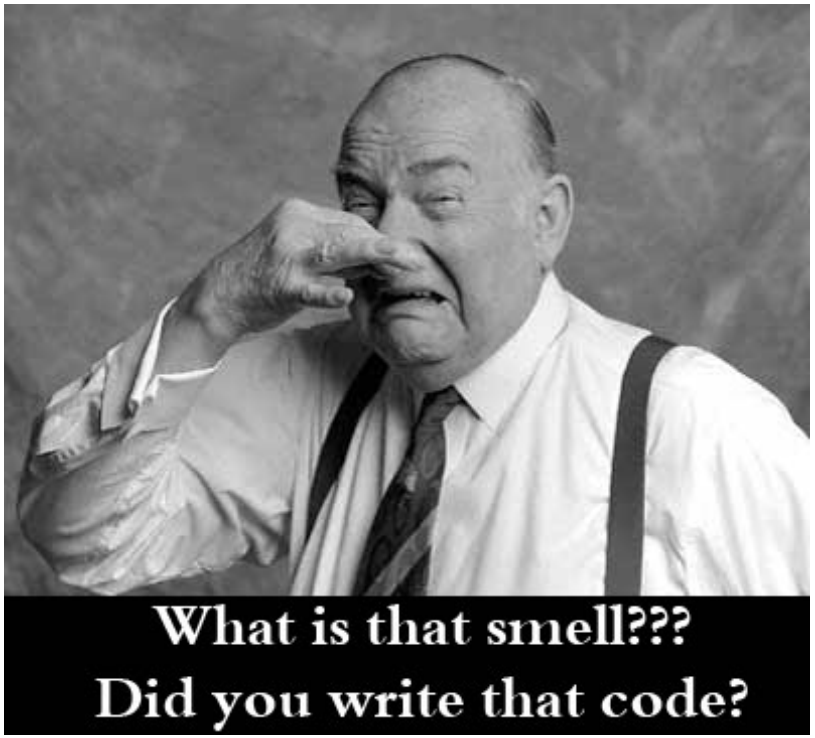
| Class Refactorings | Method Refactorings | Attribute Refactorings |
|---|---|---|
| add (sub)class to hierarchy | add method to class | add variable to class |
| rename class | rename method | rename variable |
| remove class | remove method | remove variable |
| | push method down | push variable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

These simple refactorings can be combined to provide bigger restructurings such as the introduction of design patterns

# When to Apply Refactoring?


# Bad Smells in Code

- Duplicated code
- Long method
- Divergent change
- Shotgun surgery
- Data clumps
- Switch statements
- Lazy class
- Inappropriate intimacy



# Duplicated Code

- "The #1 bad smell"

```
if (!delete(file)) {
    String message = "Unable to delete file "
        + file.getAbsolutePath();
    if (failonerror) {
        throw new BuildException(message);
    } else {
        log(message, quiet ? Project.MSG_VERBOSE
            : Project.MSG_WARN);
    }
}
```

```
if (!delete(f)) {
    String message = "Unable to delete file "
        + f.getAbsolutePath();
    if (failonerror) {
        throw new BuildException(message);
    } else {
        log(message, quiet ? Project.MSG_VERBOSE
            : Project.MSG_WARN);
    }
}
```

```
if (!delete(f)) {
    String message = "Unable to delete file "
        + f.getAbsolutePath();
    if (failonerror) {
        throw new BuildException(message);
    } else {
        log(message, quiet ? Project.MSG_VERBOSE
            : Project.MSG_WARN);
    }
}
```

```
if (!delete(dir)) {
    String message = "Unable to delete directory "
        + dir.getAbsolutePath();
    if (failonerror) {
        throw new BuildException(message);
    } else {
        log(message, quiet ? Project.MSG_VERBOSE
            : Project.MSG_WARN);
    }
}
```

# Duplicated Code

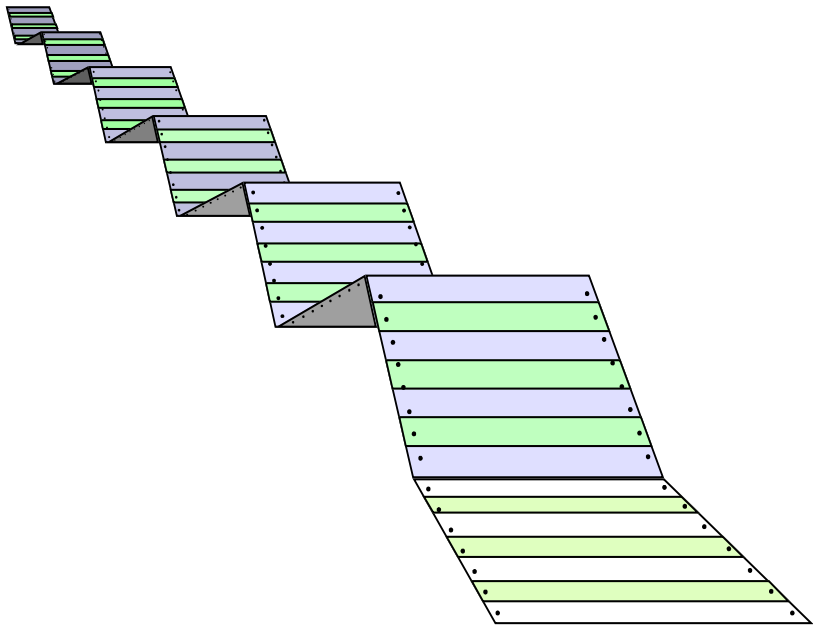
- Same expression in two methods in the same class?
  - Solution – make it a private ancillary routine and parameterize it (Extract method)
- Same code in two related classes?
  - Solution – push commonalities into closest mutual ancestor and parameterize (Pull up method)
- Same code in two unrelated classes?
  - Ought they be related?
    - Solution – introduce abstract parent (Extract class, Pull up method)
  - Does the code really belong to just one class?
    - Solution – make the other class into a client (Extract method)

# Not All Clones are Harmful

- "'Clones Considered Harmful' Considered Harmful" [Kapser and Godfrey]
  - Forking – used to bootstrap development of similar solutions, with the expectation that evolution of the code will occur somewhat independently
  - Templating – directly copy behavior of existing code but appropriate abstraction mechanisms are unavailable
  - Customization – currently existing code does not adequately meet a new set of requirements

# Long Method

- Often a sign of
  - Bad cohesion
  - Trying to do too many things
  - Poorly thought out abstractions and boundaries



# Long Method

- Best to think carefully about the major tasks and how they inter-relate
  - Solution – break up into smaller private methods within the class (Extract method)
- Fowler's heuristic
  - When you see a comment, make a method
  - Often, a comment indicates
    - The next major step
    - Something non-obvious whose details detract from the clarity of the routine as a whole.
  - In either case, this is a good spot to "break it up"

# Single Responsibility Principle

# Can’t you do anything right?

# OVERLOAD Kills

# What is SRP?

> Every software module should have only one reason to change
>
> **R. Martin**

- Software Module – Class, Function, etc.
- Reason to Change – Responsibility

# Divergent Change

- If, over time, you make changes to a class that touch completely different parts of the class
  - Likely, this class is trying to do too much and contains too many unrelated subparts
- Over time, some classes develop a "God complex"
  - They acquire details/ownership of subparts that rightly belong elsewhere
- This is a sign of poor cohesion
  - Unrelated elements in the same container
- Solution – break it up, reshuffle, reconsider relationships and responsibilities (Extract class)

# Shotgun Surgery

- … the opposite of divergent change
  - Each time you want to make a single, seemingly coherent change, you have to change lots of classes in little ways
- Also a classic sign of poor cohesion
  - Related elements are not in the same container!
- Solution – look to do some gathering, either in a new or existing class (Move method/field)

# Data Clumps

- You see a set of variables that seem to "hang out" together
  - e.g., passed as parameters, changed/accessed at the same time
- Usually, this means that there's a coherent subobject just waiting to be recognized and encapsulated

```
void Scene::setTitle (string titleText,
        int titleX, int titleY,
        Colour titleColour){…}

void Scene::getTitle (string& titleText,
        int& titleX, int& titleY,
        Colour& titleColour){…}
```

# Data Clumps

- In the example, a Title class is dying to be born
- If a client knows how to change a title's x, y, text, and colour, then it knows enough to be able to "roll its own" Title objects
  - However, this does mean that the client now has to talk to another class
- This will greatly shorten and simplify your parameter lists (which aids understanding) and makes your class conceptually simpler too.

# Switch Statements

```
Double getSpeed () {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() –
                getLoadFactor() * _numCoconuts;
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0
                : getBaseSpeed(_voltage);
    }
}
```

# Switch Statements

- This is an example of a lack of understanding polymorphism and a lack of encapsulation
- Solution – redesign as a polymorphic method in a hierarchy (Replace conditional with polymorphism, replace type code with subclasses)

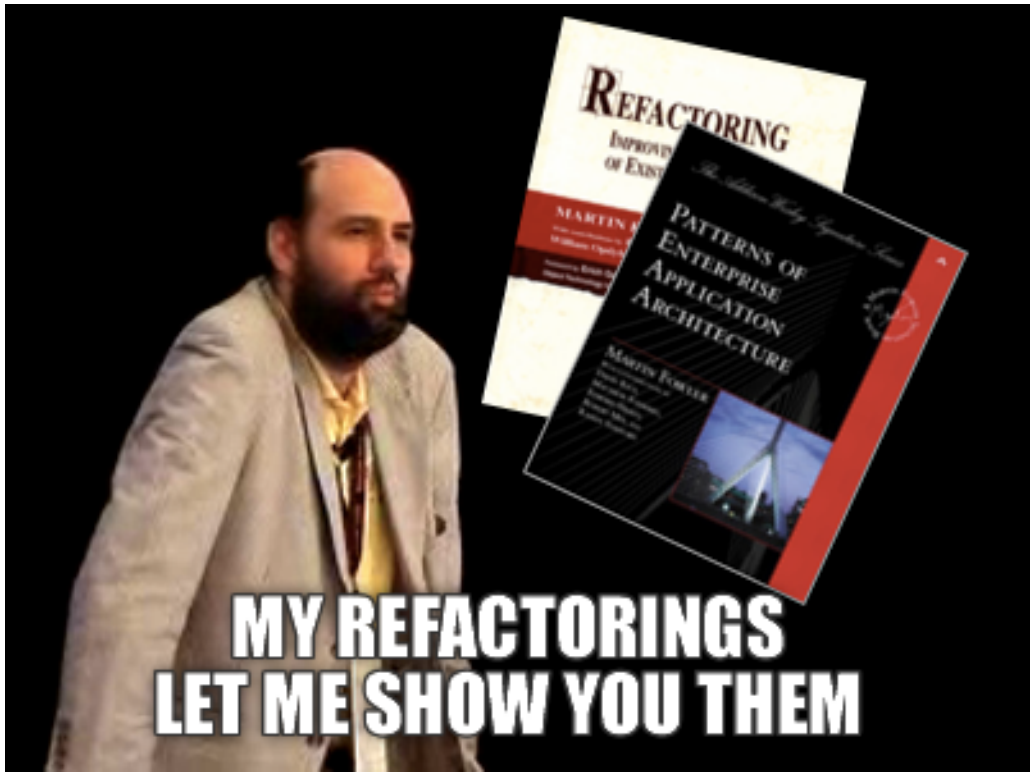
# Lazy Class

- Classes should pull their weight
  - Every additional class increases the complexity of a project
  - If you have a class that isn't doing enough to pay for itself, can it be collapsed or combined into another class?
- If there are several sibling classes that don't exhibit polymorphic behavioural differences, then consider just collapsing them back into the parent and add some parameters
- Often, lazy classes are legacies of ambitious design or a refactoring that gutted the class of interesting behaviour
- Solution – (Collapse hierarchy, Inline class)

# Inappropriate Intimacy

- Sharing of secrets between classes
  - e.g., public variables, indiscriminate definitions of get/set methods
- Leads to data coupling, intimate knowledge of internal structures and implementation decisions
- Solution
  - Appropriate use of get/set methods
  - Rethink basic abstraction
  - Merge classes if you discover "true love"
  - (Move/extract method/field, Change bidirectional association to unidirectional, Hide delegate)

# Case Study – from Martin Fowler



# Sample Output

```
Rental Record for Dinsdale Pirhana
  Monty Python and the Holy Grail   3.5
  Ran                               2
  Star Trek 27                      6
  Star Wars 3.2                     3
  Wallace and Gromit                6
Amount owed is 20.5
You earned 6 frequent renter points
```

# Class Movie

```
public class Movie {
  public static final int CHILDRENS = 2;
  public static final int REGULAR = 0;
  public static final int NEW_RELEASE = 1;

  private String title;
  private int priceCode;

  public Movie(String _title, int _priceCode) {
    title = _title;
    priceCode = _priceCode;
  }

  public int getPriceCode() {
    return priceCode;
  }

  public void setPriceCode(int arg) {
    priceCode = arg;
  }

  public String getTitle () {
    return title;
  }
}
```

# Class Rental

```
public class Rental {
  private Movie movie;
  private int daysRented;

  public Rental(Movie _movie, int _daysRented) {
    movie = _movie;
    daysRented = _daysRented;
  }

  public int getDaysRented() {
    return daysRented;
  }

  public Movie getMovie() {
    return movie;
  }
}
```

# Class Customer

```
public class Customer {
  private String name;
  private Vector rentals = new Vector();

  public Customer (String _name) {
    name = _name;
  }

  public void addRental(Rental arg) {
    rentals.addElement(arg);
  }

  public String getName ()    {
    return _name;
  }

// see next slide
...
```

# Class Customer (continued)

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";
  while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();

    //determine amounts for each line
    double thisAmount = 0;
    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;
      case Movie.CHILDRENS:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) * 1.5;
        break;
    }

// see next slide
...
```

# Class Customer (continued)

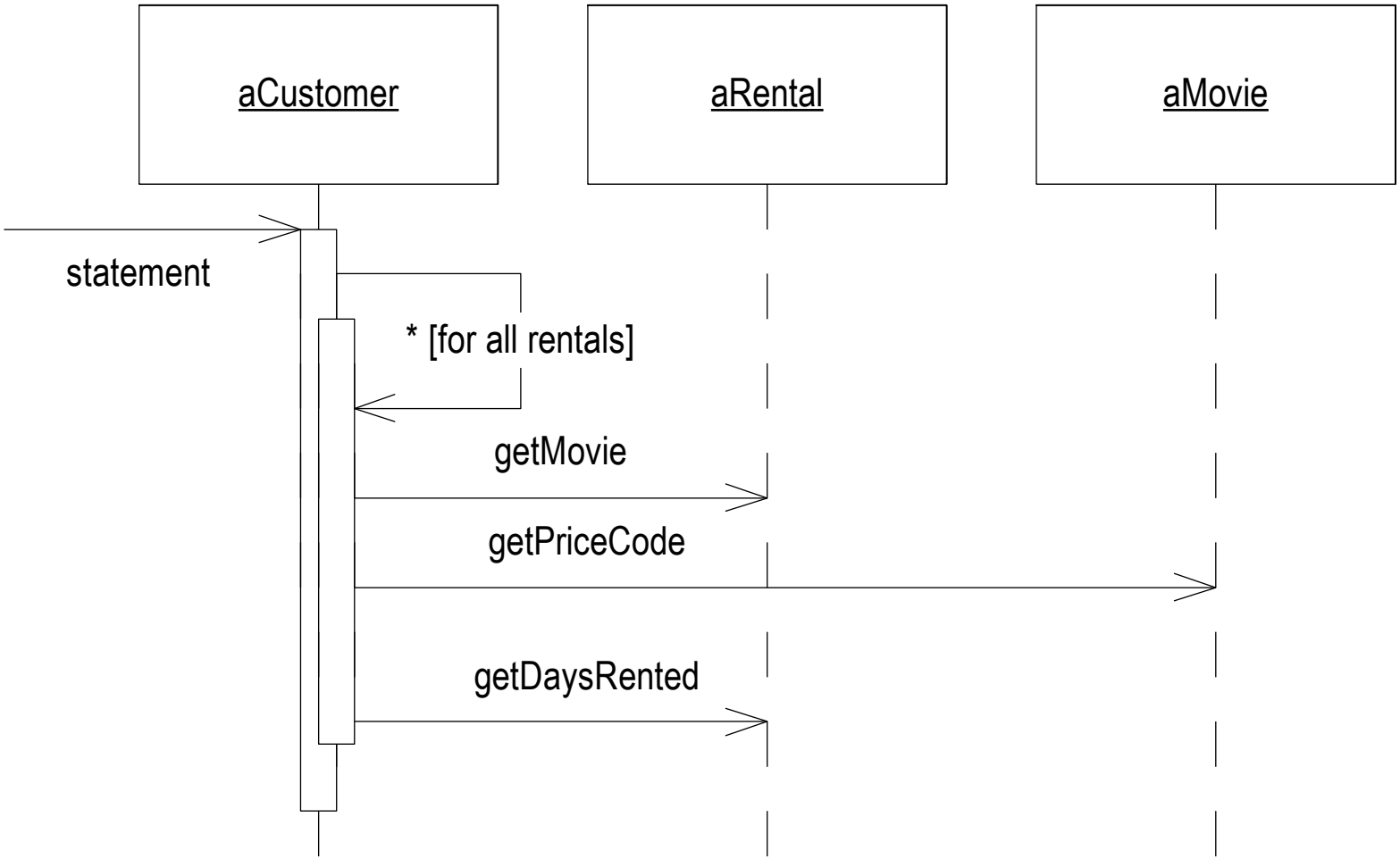
```
    // add frequent renter points
    frequentRenterPoints ++;

    // add bonus for a two day new release rental
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
      each.getDaysRented() > 1) frequentRenterPoints ++;

    //show figures for this rental
    result += "\t" + each.getMovie().getTitle()+ "\t" +
    String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;
  }

  //add footer lines
  result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
  result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
  return result;
}
```

# Interactions for Method `statement`



# Steps for Extract Method

- Create method named after intention of code
- Copy extracted code
- Look for local variables and parameters
  - Turn into parameter
  - Turn into return value
  - Declare within method
- Compile
- Replace code fragment with call to new method
- Compile and test

# Candidate Extraction

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";
  while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();

    //determine amounts for each line
    double thisAmount = 0;
    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;
      case Movie.CHILDRENS:
        thisAmount += 1.5;
        if (each.getDaysRented() > 3)
          thisAmount += (each.getDaysRented() - 3) * 1.5;
        break;
    }
```

# Extracting the Amount Calculation

```
private double amountFor(Rental each) {
  double thisAmount = 0;
  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;
    case Movie.CHILDRENS:
      thisAmount += 1.5;
      if (each.getDaysRented() > 3)
        thisAmount += (each.getDaysRented() - 3) * 1.5;
      break;
  }
  return thisAmount;
}
```

# Method `statement` After Extraction

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";
  while (rentals.hasMoreElements()) {
    Rental each = (Rental) rentals.nextElement();

    thisAmount = amountFor(each);

    // add frequent renter points
    frequentRenterPoints ++;

    // add bonus for a two day new release rental
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
      each.getDaysRented() > 1) frequentRenterPoints ++;

    //show figures for this rental
    result += "\t" + each.getMovie().getTitle()+ "\t" +
    String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;
  }

  //add footer lines
  result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
  result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
  return result;
}
```

# Extracting the Amount Calculation

- Is this important?
- Is this method in the right place?

```
private double amountFor(Rental each) {
  double thisAmount = 0;
  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;
    case Movie.CHILDRENS:
      thisAmount += 1.5;
      if (each.getDaysRented() > 3)
        thisAmount += (each.getDaysRented() - 3) * 1.5;
      break;
  }
  return thisAmount;
}
```

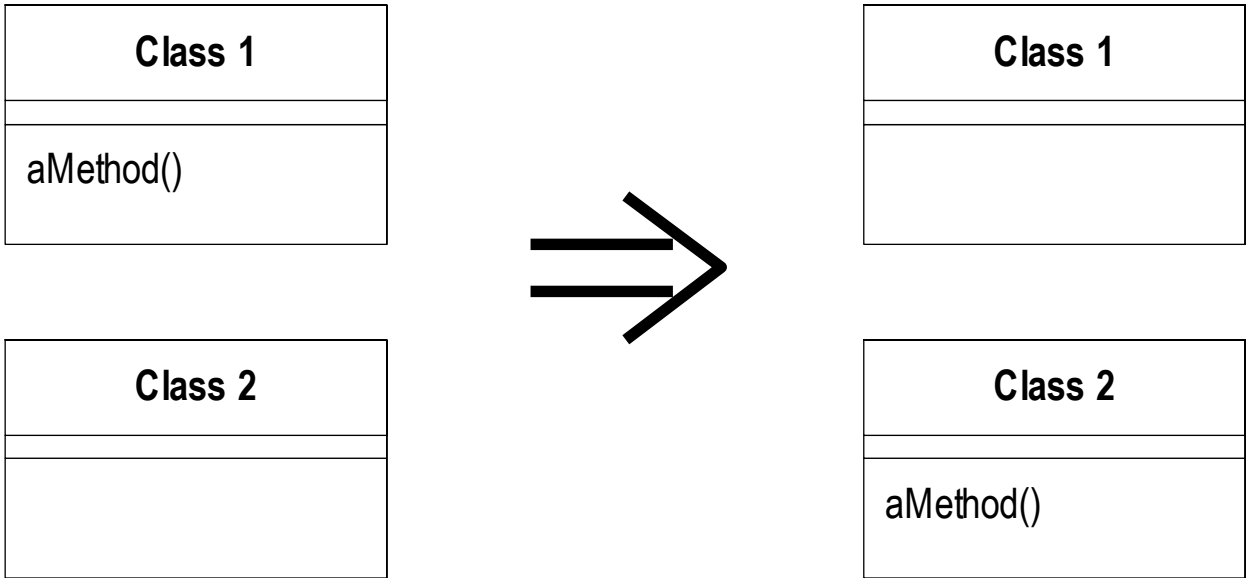
# Change Names of Variables

- Should this method really be in the Customer class?

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

# Move Method

- A method is, or will be, using or used by more features of another class than the class it is defined on
- Create a new method with a similar body in the class it uses most
- Either turn the old method into a simple delegation, or remove it altogether

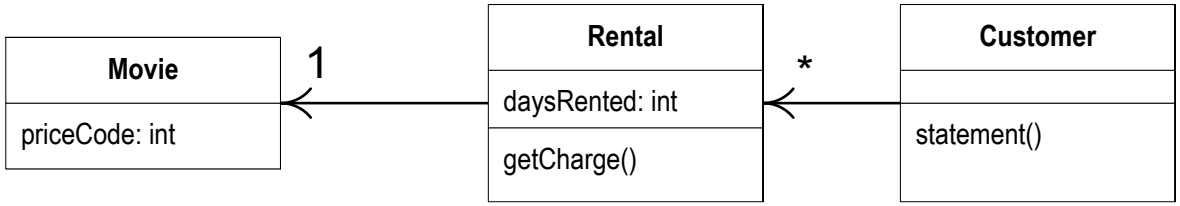


# Steps for Move method

- Declare method in target class
- Copy and fit code
- Set up a reference from the source object to the target
- Turn the original method into a delegating method
  - `amountFor(Rental aRental) {return aRental.getCharge();}`
  - Check for overriding methods
- Compile and test
- Find all users of the method
  - Adjust them to call method on target
- Remove original method
- Compile and test

# Moving Method amount to Rental

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



# Altered `statement`

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rentals.elements();
  String result = "Rental Record for " + getName() + "\n";
  while (rentals.hasMoreElements()) {
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();

    thisAmount = each.getCharge();

    // add frequent renter points
    frequentRenterPoints ++;
    // add bonus for a two day new release rental
    if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
      each.getDaysRented() > 1) frequentRenterPoints ++;

    //show figures for this rental
    result += "\t" + each.getMovie().getTitle()+ "\t" + String.valueOf(thisAmount) + "\n";
    totalAmount += thisAmount;
  }

  // add footer lines
  result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
  result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
  return result;
}
```

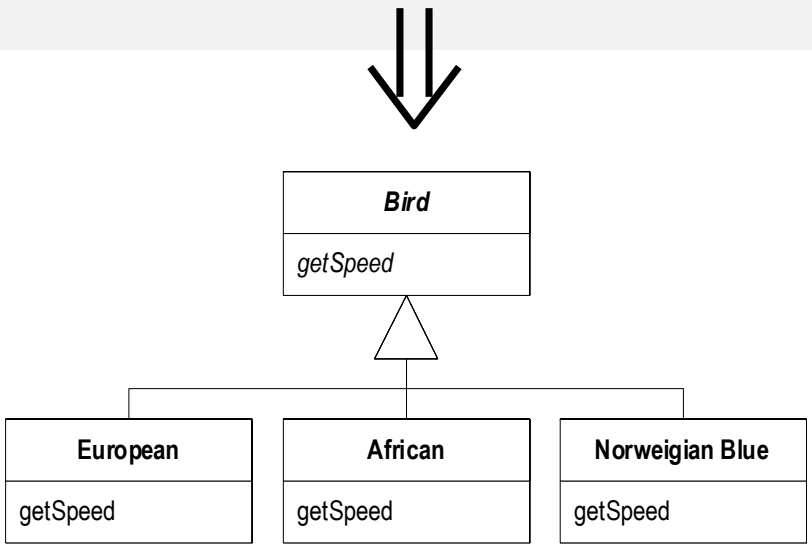
# Replace Conditional With Polymorphism

**You have a conditional that chooses different behavior depending on the type of an object**

***Move each leg of the conditional to an overriding method in a subclass***

***Make the original method abstract***

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEIGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw new RuntimeException ("Should be unreachable");
}
```



# Steps for Replace Conditional with Polymorphism

- Move switch to superclass of inheritance structure
- Copy one leg of case statement into subclass
- Compile and test
- Repeat for all other legs
- Replace case statement with abstract method

# Move getCharge to Price

```
public class Movie...
  public double getCharge(int daysRented) {
    return _price.getCharge(daysRented);
  }

public class Price...
  public double getCharge(int daysRented) {
    double result = 0;
    switch (getPriceCode()) {
      case Movie.REGULAR:
        result += 2;
        if (daysRented > 2)
          result += (daysRented - 2) * 1.5;
        break;
      case Movie.NEW_RELEASE:
        result += daysRented * 3;
        break;
      case Movie.CHILDRENS:
        result += 1.5;
        if (daysRented > 3)
          result += (daysRented - 3) * 1.5;
        break;
    }
    return result;
}
```

# Override getCharge in the subclasses

```
Class Price...
abstract public double getCharge(int daysRented);
```

```
Class RegularPrice...
  public double getCharge(int daysRented){
    double result = 2;
    if (daysRented > 2)
      result += (daysRented - 2) * 1.5;
    return result;
  }

Class ChildrensPrice...
  public double getCharge(int daysRented){
    double result = 1.5;
    if (daysRented > 3)
      result += (daysRented - 3) * 1.5;
    return result;
  }

Class NewReleasePrice...
  public double getCharge(int daysRented){
    return daysRented * 3;
  }
```

Do each leg one at a time then...

# Obstacles to Refactoring

- Complexity
  - Changing design is hard
  - Understanding code is hard
- Possibility to introduce errors
  - Run tests if possible
  - Build tests
- Clean first, then add new functionality
- Cultural Issues
  - Producing negative lines of code, what an idea!
  - "We pay you to add new features, not to improve the code!"
- If it ain't broke, don't fix it
  - "We do not have a problem, this is our software!"

# Obstacles to Refactoring

- Performance
  - Refactoring may slow down the execution
  - The secret to writing fast software
    - "Write tunable software first then tune it"
    - Typically only 10% of your system consumes 90% of the resources so just focus on 10 %
  - Refactorings help to localize the part that need change
  - Refactorings help to concentrate the optimizations
- Development is always under time pressure
  - Refactoring takes time
  - Refactoring better right after a software release

# Conclusion: Know-when & Know-how

- "Know when" is as important as "know-how"
- Use "code smells" as symptoms
- Rule of the thumb
  - "Once and Only Once" (Kent Beck)
    - A thing stated more than once implies refactoring

# Research Issues

- Tool Support
  - Visualization, automated refactoring
  - How to make more open?
- Scalability of composable refactorings (conflict resolution)
- Analysis of dependencies among refactorings
- Formally, what is behavior and how can we prove that it is preserved by a refactoring?
  - Real-time, embedded, safety critical behavior?

# Research Issues

- How to specify refactoring in a language-independent manner?
- How to apply refactoring at higher levels of abstraction (e.g., UML models)?
- Combining clone detection and analysis with clone refactoring
- Can we get some objective measurements of how refactoring affects software quality?

# Clone Detection, Analysis, and Refactoring