CODE SMELLS
&
REFACTORING

Structure

- This section is structured as follows.
- We discuss **refactoring in general**, what it is, where to use it and various issues around its application.
- We then examine a **number of specific refactorings**.
- Finally we explore **code smells**, hints that refactoring may be required.

(Based on material by Martin Fowler, http://sourcemaking.com, and the Eclipse tutorial material at www.ibm.com. Supplement slides with extra reading.)

What is Refactoring?

- Refactoring is the process of change the internal structure of software to make it easier to understand and modify, without changing its observable behaviour.

- “without changing observable behaviour” -- the program may behave differently internally, but its *observable functionality* remains identical.

Simple Refactoring Examples

```
class Customer {
   ...
   Date d;
}
```

```
class Customer {
   ...
   Date dateOfLastPayment;
}
```

```
float average = 0.0;
while (!list.empty()){
  current = list.next();
  average += current.value();
}
average = average/list.size();
```

```
float total = 0.0;
while (!list.empty()){
  current = list.next();
  total += current.value();
}
float average = total/list.size();
```

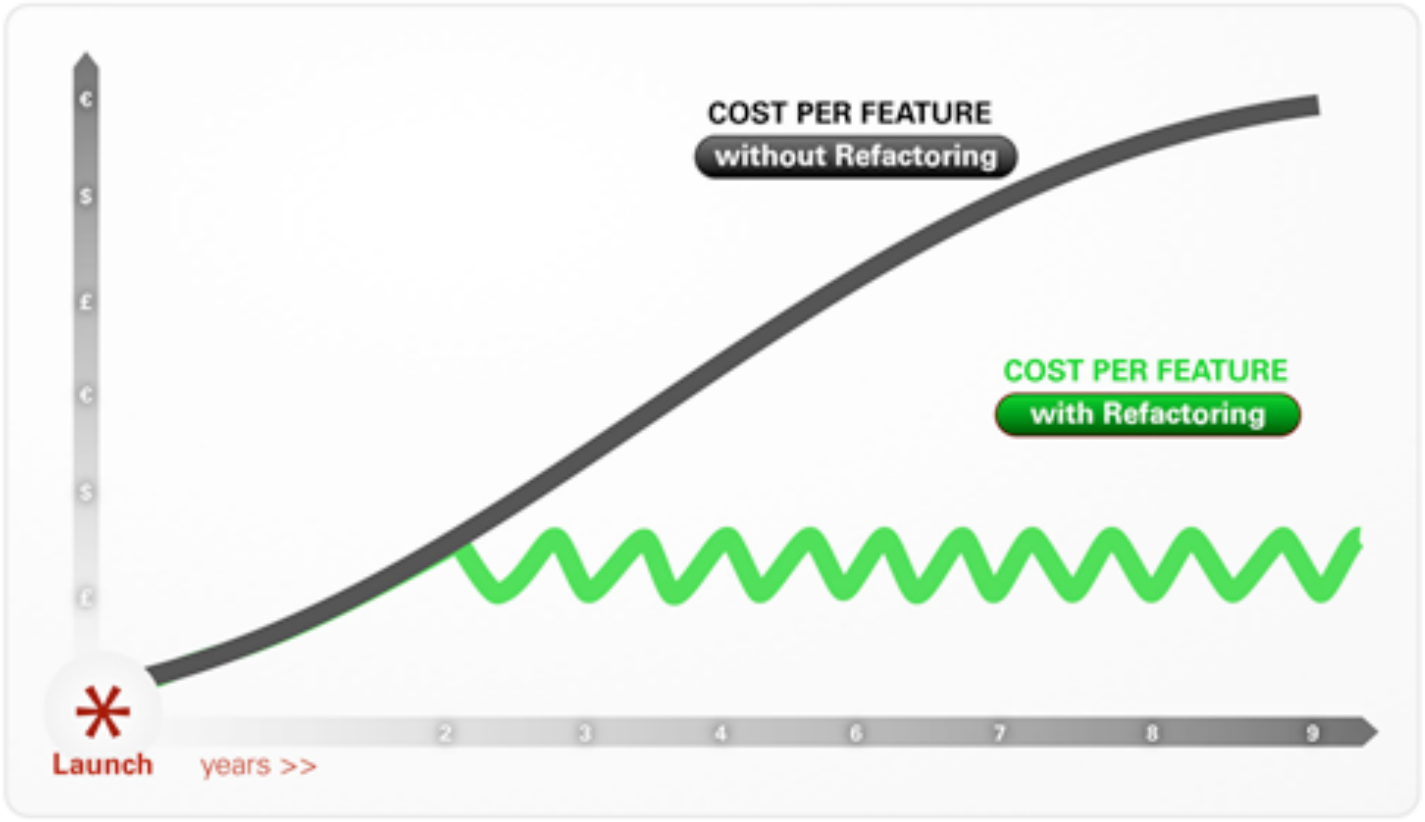
"I have rewritten—often several times—every word I have ever published. My pencils outlast their erasers."

— VLADIMIR NABOKOV, SPEAK, MEMORY, 1966

Why Refactor?

- Refactoring improves the design of software
  - Without refactoring program design will **decay** over time
  - Poorly designed code is time-consuming to update, and updating it is more likely to introduce bugs
- **Decay?** Really?
  - When software is updated in an undisciplined way, its design will match its functionality less and less — it appears to **decay**.
- Refactoring makes software easier to understand
  - somebody else will have to read your code, or you will have to return to it in the future
  - Makes fixing bugs easier
  - Enables you to program faster

Refactoring aims to reduce maintenance costs



With thanks to Sibylle Peter & Sven Ehrke of Canoo Engineering

When to Refactor I

- Refactor whenever you add functionality
  - Frequently the existing design does not allow you to easily add the new feature
  - Also helps you to understand the code you are modifying
  - Recall the "red, green, refactor" cycle of Test-Driven Development.

- The "rule of three" heuristic
  - You *might* duplicate code *once*, or hack a single exceptional case *once*. However if it happens again, refactor.

When to Refactor II

- Refactor when you need to fix a bug
    - A bug may be a sign that the code was not clear enough for the developer to spot the bug in the first place

- Refactor as you do a code review
    - Code reviews help spread knowledge through the development team
    - (XP pair programming is active code review taken to its limit)

Refactoring and Design

- Agile software development advocates that you implement the first approach that comes to you, get it working, and then refactor it to improve its design

- Refactoring changes the role of software design
    - You are no longer looking for the "perfect" solution when doing design but a reasonable one
    - Plan to refactor to a more flexible design only when required to

- Refactoring can lead to a simpler, more flexible software design

The Role of Test Cases

- If you use a refactoring tool to refactor, you can be certain that program behaviour will not change
  - Not 100% certain -- refactoring tools can have bugs too!
- Test cases mean you can refactor radically in confidence
  - Running the test cases assures us that the refactoring has not introduced any new bugs
- The synergy between automated testing and refactoring is a vital one in current software development practice.

Preconditions

- In order for a refactoring to preserve program behaviour, certain **preconditions** must hold
    - e.g., if you rename a field from `alpha` to `beta`, there must not already be a field called `beta` in the class

- Using a refactoring tool, you don't have to check preconditions
    - The tool will do this for you

- For each refactoring we look at, you should have an idea what the key elements of the preconditions are.

HAPPY HOLLOW
COFFEE
REFACTORING

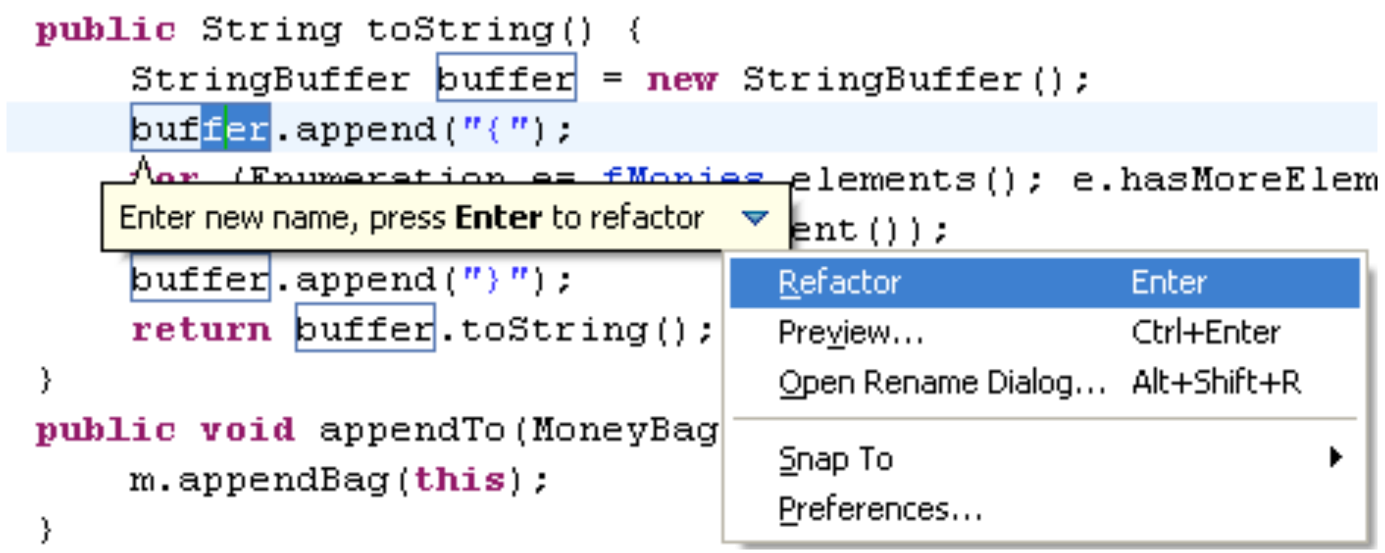
Individual Refactorings (supported by e.g. Eclipse)

- Rename method/field/class
- Move class/method/field
- Extract local variable/constant/method
- Pull up method/field
- Push down method/field
- Inline method/class
- Extract superclass/interface
- + many more

- Note that refactorings can be applied in any language, but from here on we'll assume we're using **Java**.

Rename <anything>

- Allows you to rename variables, classes, methods, packages, folders, and almost any Java identifier



```
public String toString() {
    StringBuffer buffer = new StringBuffer();
    buffer.append("(");
    for (Enumeration e= fMonies.elements(); e.hasMoreElem
                                           ent());
    buffer.append(")");
    return buffer.toString();
}
public void appendTo(MoneyBag
    m.appendBag(this);
}
```

Enter new name, press **Enter** to refactor

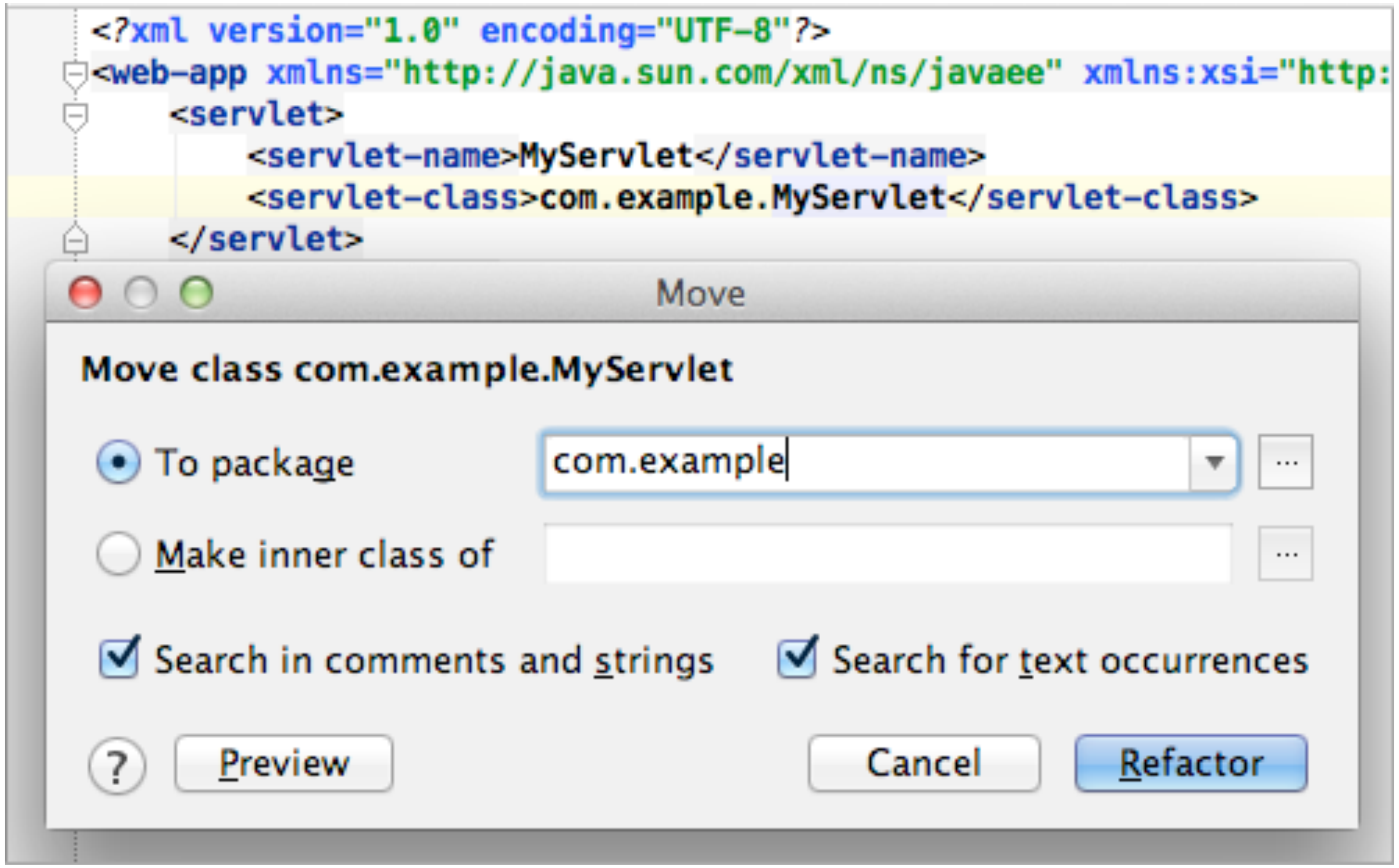
Comments on Rename

- Rename is by far the most commonly used refactoring!
- Useful for making code clearer
  - E.g., don’t comment field declarations; instead Rename the field to a better name that needs no comment
- We’ll see that some other refactorings create new program entities (methods, field etc.)
  - Rename can then be used to give them proper names

(Oddly enough, the common implementation of this refactoring contains subtle bugs due to the complexity of Java)

Move class/method/field

- Move a class to another package or a method/field to another class.
- For moving a method/field, the source class must have a reference to the target class.



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http:
    <servlet>
        <servlet-name>MyServlet</servlet-name>
        <servlet-class>com.example.MyServlet</servlet-class>
    </servlet>
```

Move

Move class com.example.MyServlet

[✓ To package] com.example

Make inner class of

[✓] Search in comments and strings

[✓] Search for text occurrences

Move Method Example

- A project has a number of participants. The **`participate`** method checks if the receiving person is in the given project.

```
class Project {
   public Person[] getParticipants(){return participants;}
   private Person[] participants;
}

class Person {
  boolean participate(Project p) {
    for(int i=0; i<p.getParticipants().length; i++) {
       if (p.getParticipants[i].getId() == id) return(true);
    }
    return(false);
  }
   public int getId(){return id;}
   private int id;
}
```

(Thanks for Marian Vitek for the example)

Issues with this code

- The **`participate`** method uses the data of the **`Project`** class rather than the class that it is in. This is an example of the **feature envy** code smell that we'll see later.

- The **`participate`** method violates the Law of Demeter, or flaunts with it at least.

- These issues are fixed nicely by moving the **`participate`** method to its natural home, the **`Project`** class.

Applying Move Method

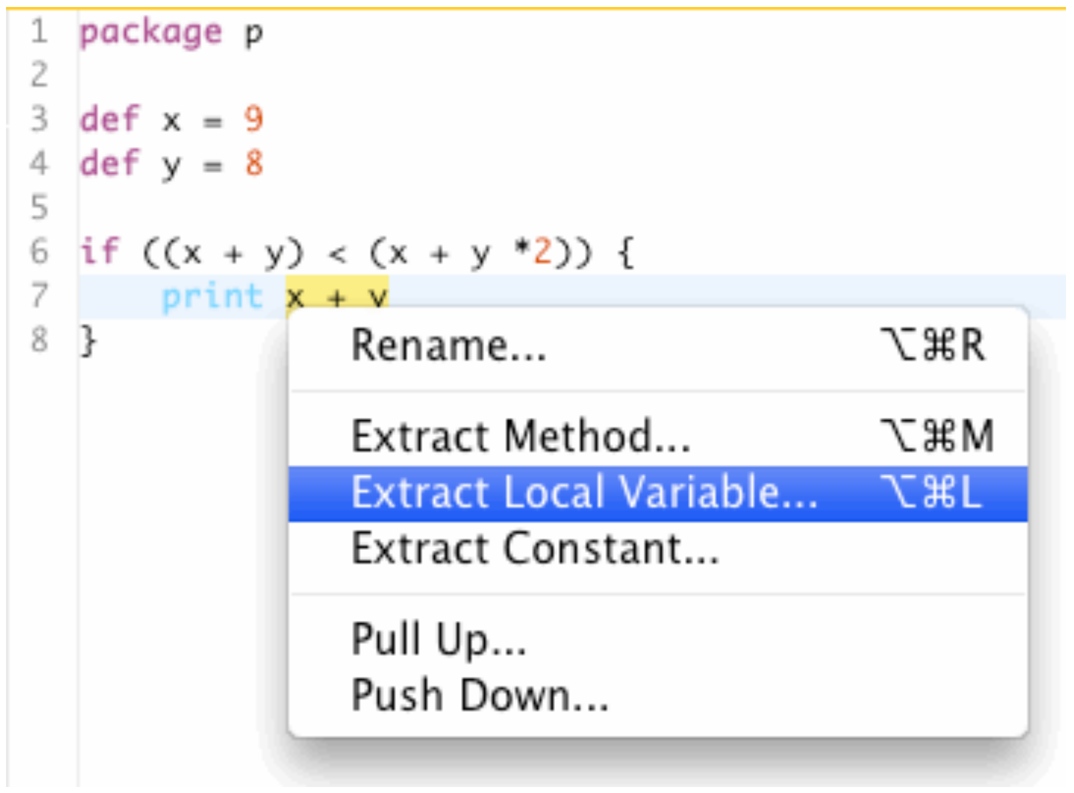
- After moving the **`participate`** method the code appears as below. Think about how it has been improved.

```
class Project {

  boolean participate(Person p) {
    for(int i=0; i<participants.length; i++) {
      if (participants[i].id == p.getId()) return(true);
    }
    return(false);
  }

  // getParticipants method removed
  private Person[] participants;
}

class Person {
  public int getId(){return id;}
  private int id;
}
```

Will clients of the Person class have to be changed as well?

Extract local variable

- Assign the result of an expression to a new local variable.
- Use it to simplify a complex expression by dividing it into multiple lines
- Useful also if the same expression is evaluated more than once

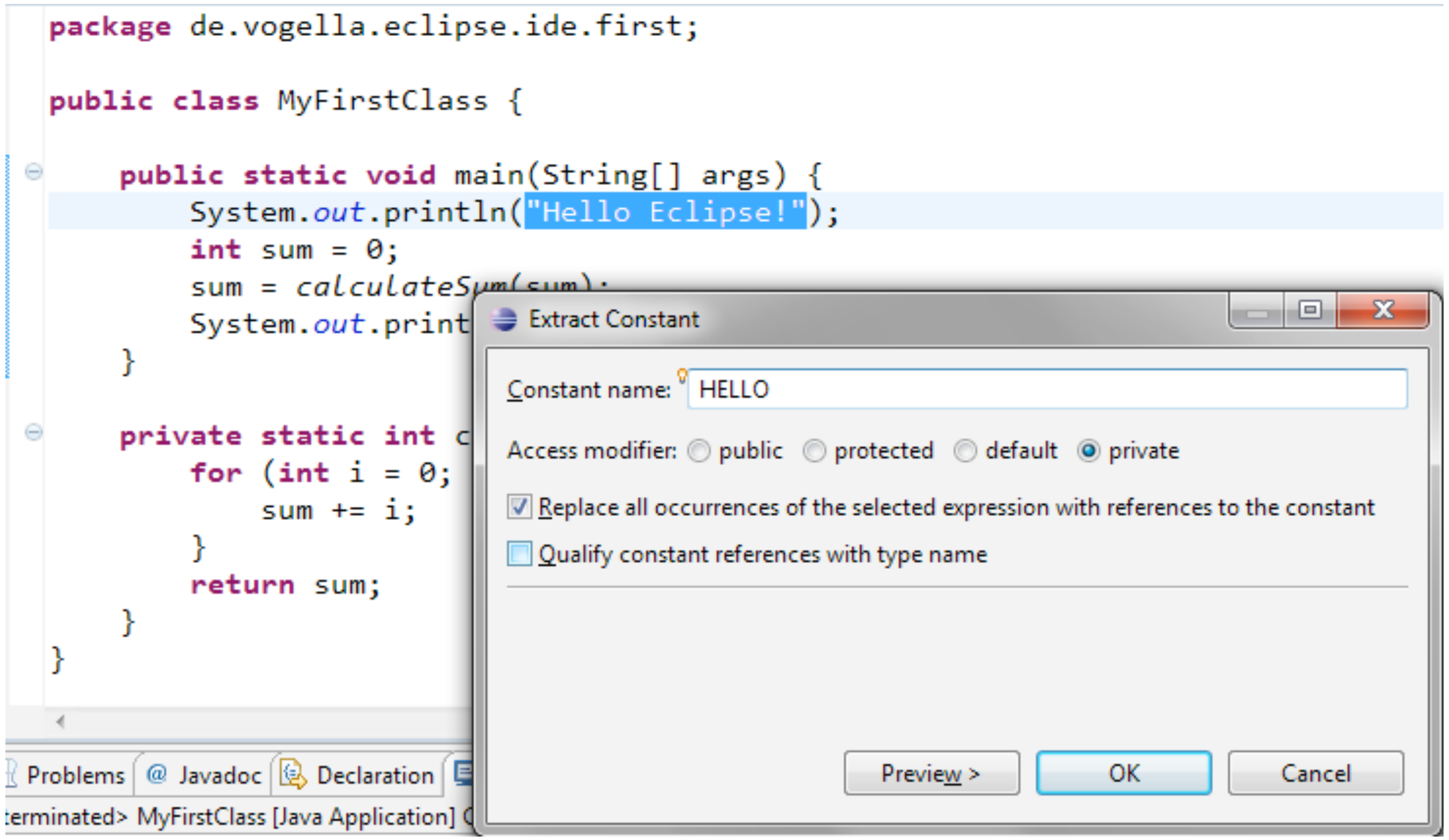


```
package p

def x = 9
def y = 8

if ((x + y) < (x + y *2)) {
    print x + y
}
```

```
n = x + y;
if (n < (x + y*2)){
  print n;
}
```

Extract constant

- Converts any number or string literal to a static final field
    - All uses of that number or string literal in the class are updated to use that field, instead of the number or string literal itself
- After applying this refactoring, you can modify the constant in just one place, instead of doing a global search and replace



```
package de.vogella.eclipse.ide.first;

public class MyFirstClass {

    public static void main(String[] args) {
        System.out.println("Hello Eclipse!");
        int sum = 0;
        sum = calculateSum(sum);
        System.out.print
    }

    private static int c
        for (int i = 0;
            sum += i;
        }
        return sum;
    }
}
```

Extract Constant

Constant name: HELLO

Access modifier: public protected default [✓ private]

[✓] Replace all occurrences of the selected expression with references to the constant

[ ] Qualify constant references with type name

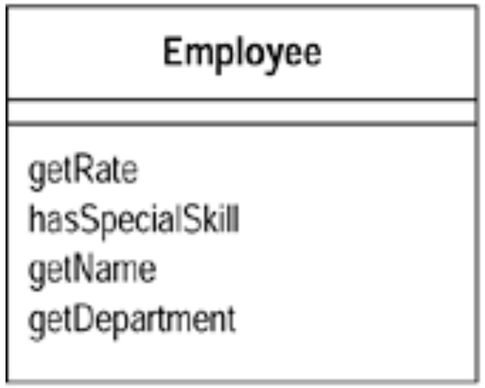
Convert local variable to field

- Takes a local variable and converts (promotes it) it to a private field of the class.
- After this refactoring, all references to the local variable now refer to the field.

- Note that where possible, always prefer to use a local variable rather than a field:
    - A local variable only adds to the complexity of the block it belongs to; a field adds complexity to its entire class.

Extract interface

- Creates a new interface out of (some of) the methods defined in a class.
  - The class can be updated to **`implement`** this interface
  - References to the class may be updated to references to the interface



Very useful when refactoring to observe ISP and related principles

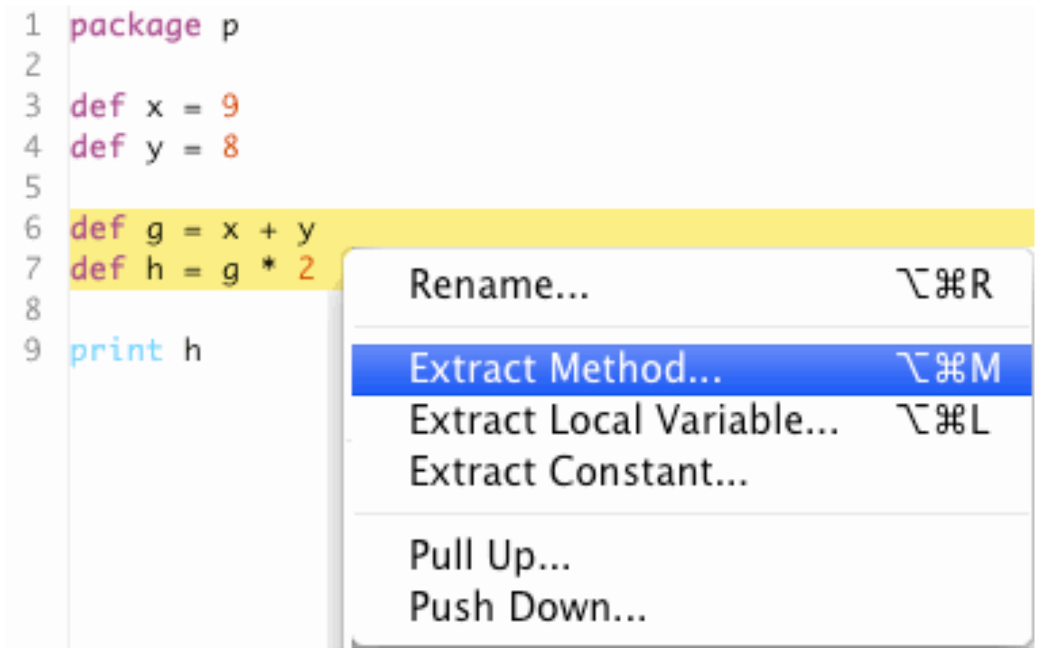
Extract method

- Convert a selected block of code into a method.

- Useful when:
  - a method is simply too long
  - a piece of code is duplicated across several methods.
  - A portion of a method does a coherent task and it's clearer to split this off into its own method.

- The block of code is put into its own method and the original block replaced with a call to the newly-created method.

Extract method example



```
package p

def x = 9
def y = 8

def g = x + y
def h = g * 2

print h
```

```
...

h = foo(x, y)

print h

def foo(i, j){
  return (i+j)*2;
}
```

Inline method

- Sometimes a method's body is just as clear as its name
  - only true for a very short method!
- This refactoring puts the method's body into the body of its callers.
  - Then the method can be removed
- This refactoring is the reverse of Extract Method

```
int getRating() {
  return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
  return numberOfLateDeliveries > 5;
}
```

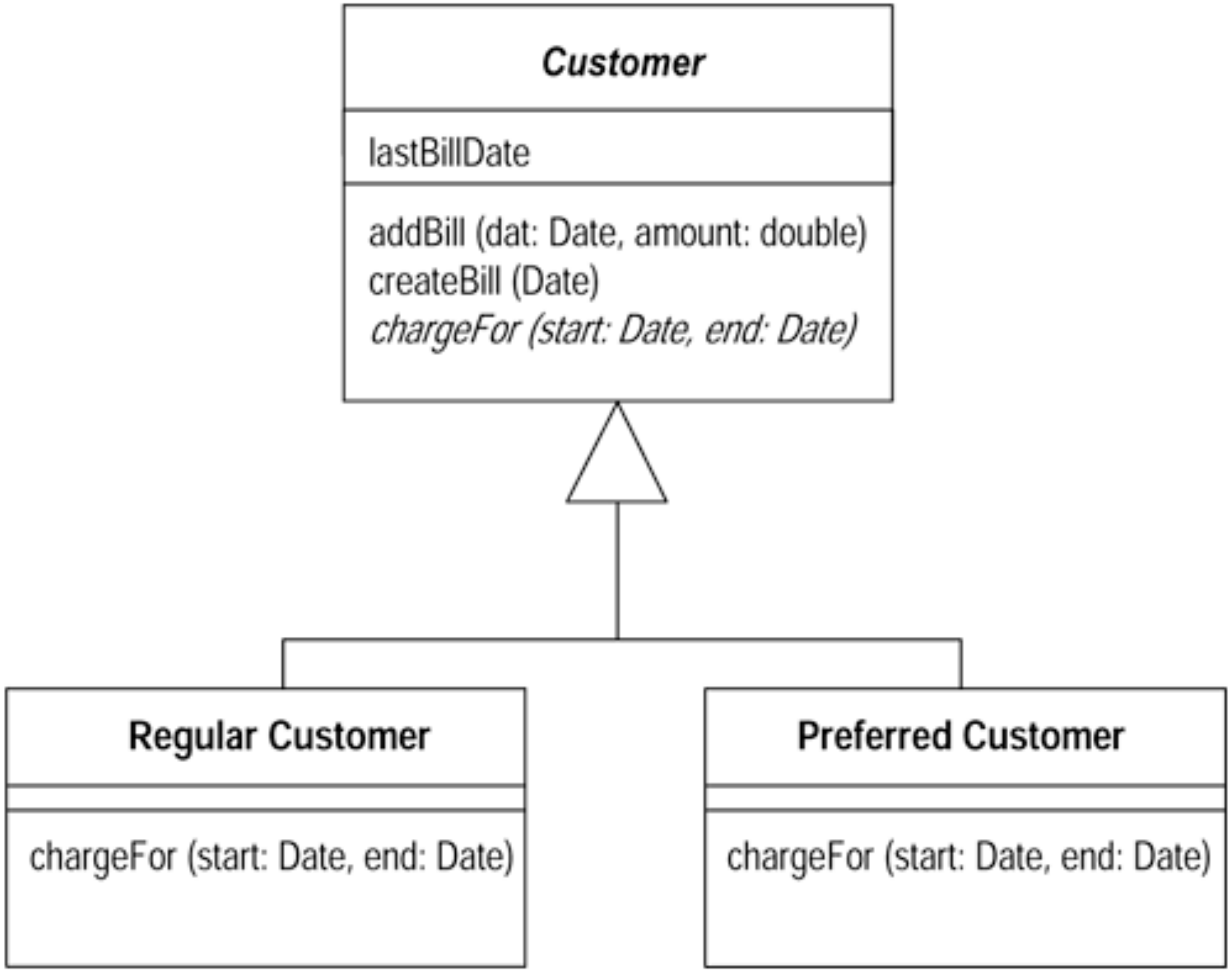
⇨

```
int getRating() {
  return (numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Pull up method/field

- Moves a method or a field from a class (or set of sibling classes) to a shared superclass.
- Avoids the duplication of the method/field in the subclasses
- Sometimes the methods in the sibling classes may be slightly different and need some adjusting prior to being moved.
- It may be useful to pull up a method from one class so that it becomes available in the sibling classes, or simply because it makes more sense in the parent class.
- This refactoring gets tricky when the method refers to features available in the subclasses, but not in the parent class.

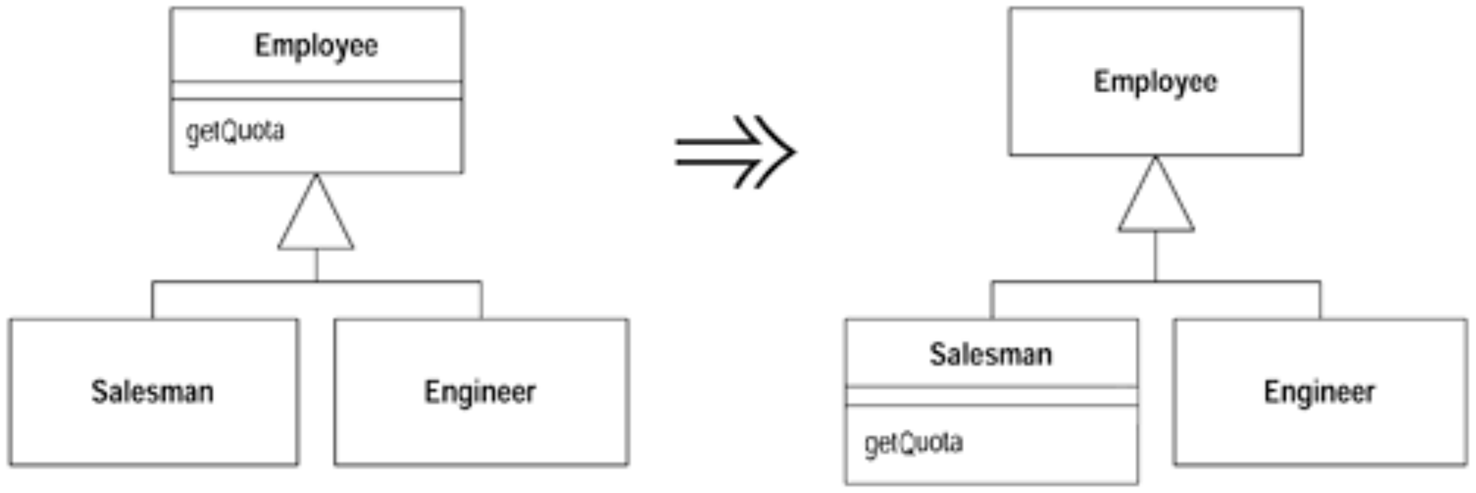
Pull Up Method Example



Customer
lastBillDate
addBill (dat: Date, amount: double)
createBill (Date)
chargeFor (start: Date, end: Date)
Regular Customer
chargeFor (start: Date, end: Date)
Preferred Customer
chargeFor (start: Date, end: Date)

Push down method/field

- Moves a method or a field from a class to one or more immediate subclasses.
- Reverse of Pull Up Method/Field.
- Use this refactoring when a method or field is used only in a number of subclasses and it's clearer to move it there.
- If this refactoring leads to much code duplication, don't apply it.



Code Smells

Code Smells

- A code smell is a hint that there's something wrong in your code that maybe should be fixed by refactoring.
- Simple example: Long Method.
  - What refactoring comes to mind?
- There's no need to fix all smells (this would be excessive)
  - E.g., some "smells" aren't bothersome
  - Smelly inert code may not be of concern
- Judicious refactoring can be used to resolve the problem underlying the smell.
- Here we look at a number of well-known smells...

Bad Smells in Code

Dozens of code smells have been proposed in the literature.

Here we look at a selection of them.

- Duplicated Code
- Long Method
- God Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Primitive Obsession
- Speculative Generality
- Data Class
- Refused Bequest

A Really Big Caveat

- Never read smells&refactoring like a medicine bottle
  - "At temperatures over 38.5 degrees, take two spoonfuls every 4 hours."
- The smell is a **hint**. They may or may not be anything actually wrong with the code.
- There may be something wrong, but it may not be worthwhile fixing it
  - e.g. a small piece of code duplication that would require a system overhaul to resolve
- Across a software base, code smell analysis can help in identifying problem areas and in deciding what to do.

Duplicated Code

- If the same code structure occurs in more than one place, it may be useful refactor to remove the duplication

- The simplest duplicated code problem is when you have the same piece of code in two methods of the same class
    - Consider performing Extract Method and invoke the new method from both places

- Another common duplication problem is having the same expression in two related subclasses
    - Consider performing Extract Method in both classes, then Pull Up Method to the shared superclass

- If code is duplicated in two unrelated classes, consider applying Extract Class, and then use the new component in the other class.
    - But ask yourself is it *really* duplicated code?

Long Method

- The longer a method is the more difficult it is to understand
- What's long?
  - *A dozen lines* -- Martin Fowler
  - *200 lines* -- Steve McConnell
- There is little value in seeking a specific threshold
  - A Long Method is one that is too long and should be split
- The Extract Method refactoring usually helps here

God Class

- A "God Class" is one that knows too much about the rest of the system and/or has too much responsibility
- Often shows up as a class with too many fields and methods.
- A class with too much code may also a source of code duplication
- Extract Class and Extract Subclass may help

Switch Statements

- When you see a switch statement that affects method behaviour you should consider polymorphism
- Use Extract Method to extract the switch statement and then Move Method to get it into the class where the polymorphism is needed
- If you only have a few case that effect a single method then polymorphism is overkill. In this case Replace Parameter with Explicit Methods is a good option
- If one of your conditional cases is null, try Introduce Null Object

Long Parameter List

- Long parameter lists are hard to understand and to maintain
- A method needn't be passed everything it needs in the parameter list
  - Pass in enough so the method can get to everything it needs
  - keeping the Law of Demeter in mind!
- Consider using Replace Parameter with Method when you can get the data in one parameter by making a request of an object you already know about, e.g.

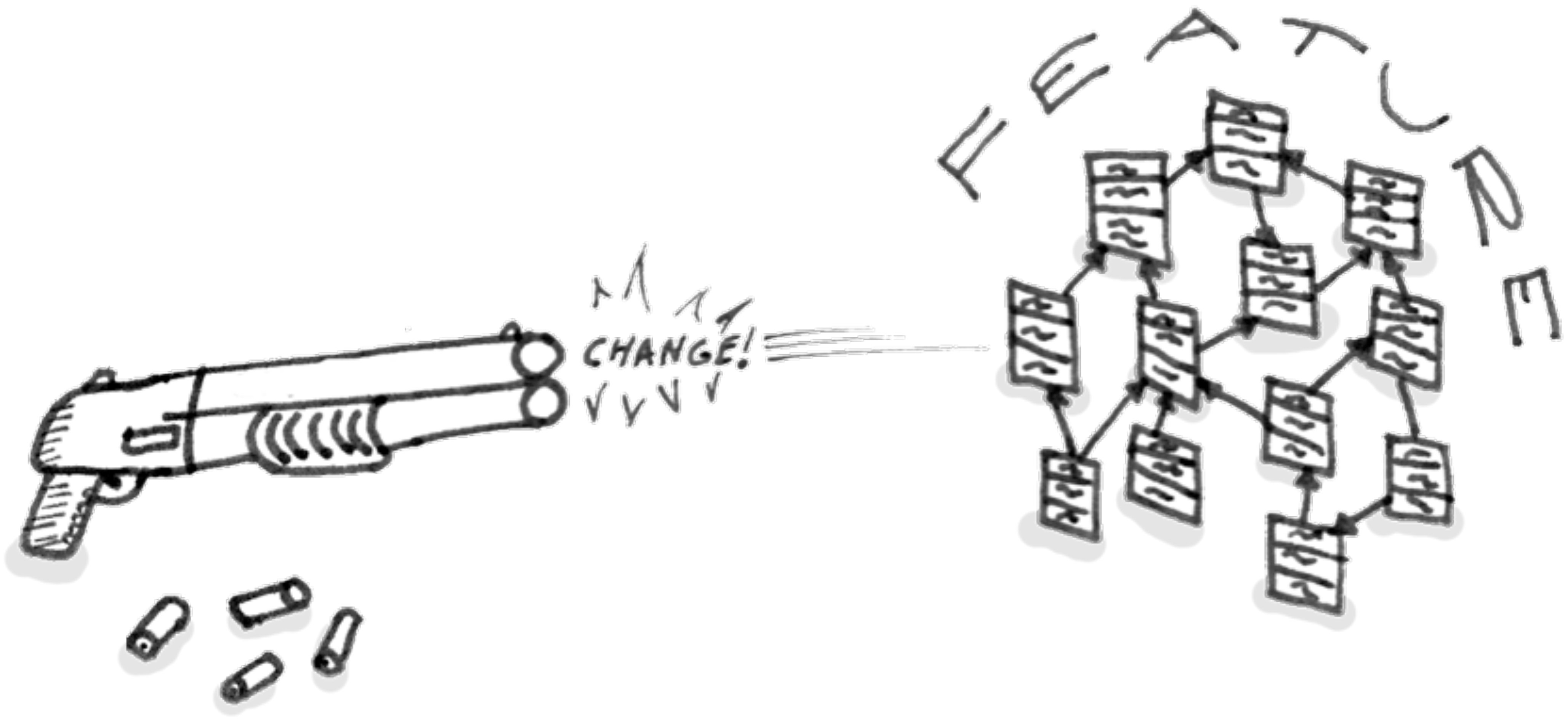
```
int basePrice = quantity * itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice(basePrice, discountLevel);
```

```
int basePrice = quantity * itemPrice;
double finalPrice = discountedPrice(basePrice);
```

Divergent Change

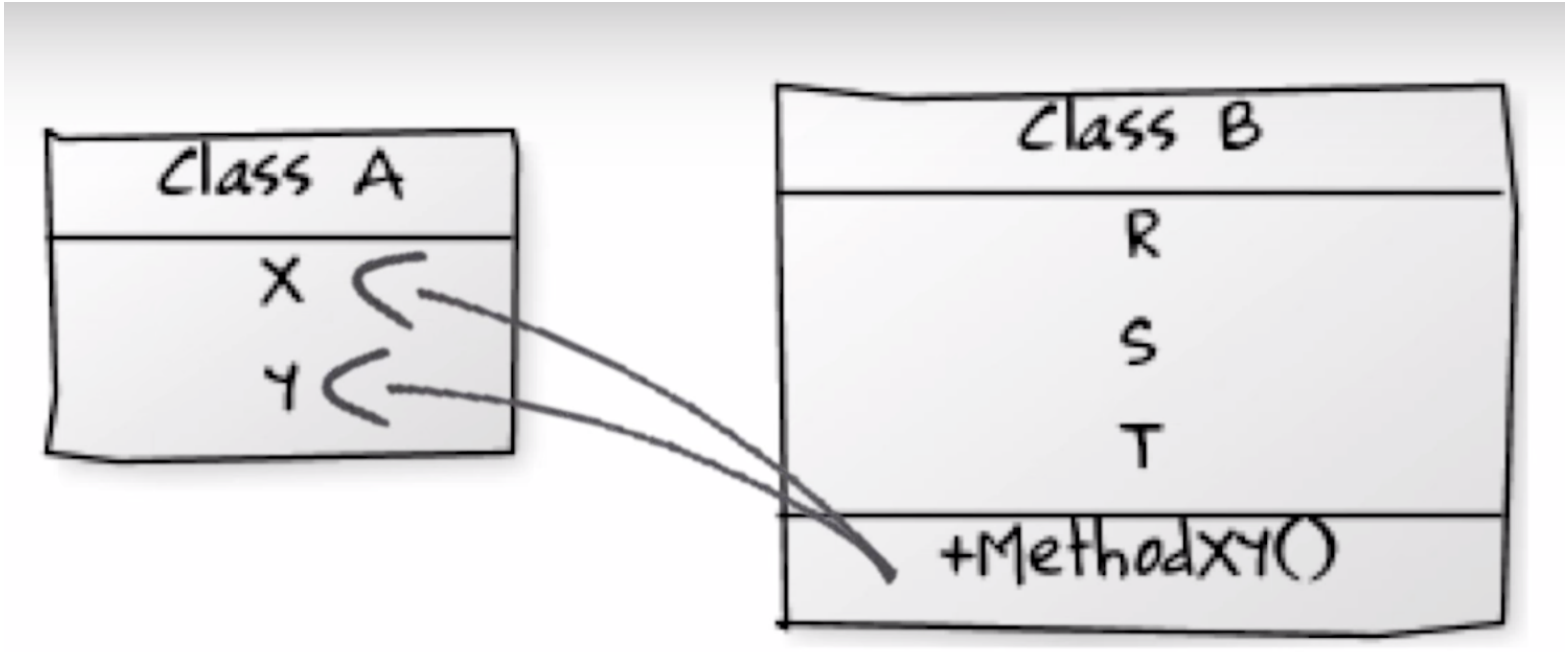
- Divergent change occurs when one class is commonly changed in different ways for different reasons
- Say in a class you have to change four methods every time a timetable changes, and three other methods whenever a new user is added...
  - Suggests that the Single Responsibility Principle is being violated
  - Probably better modelled as two classes
- To clean this up, identify everything that changes for a particular reason and use Extract Class to put them all together
  - This may or may not be easy

Shotgun Surgery



Shotgun Surgery

- Whenever you make a certain kind of change, you have to make many little changes to many different classes **=> shotgun surgery**
  - it's easy to miss an important change, which makes maintenance very challenging.
  - Opposite of Divergent Change in a sense, but also suggests that the SRP is being violated
- Consider using Move Method and Move Field to put all the change-prone elements in a single class
  - Recall 'separate the stable and the unstable' guideline
- If no current class is suitable then it may be best to create a new class to bring the relevant behaviour together


Class A
X
Y
Class B
R
S
T
+MethodXY()

Feature Envy

- Feature Envy occurs when method is more interested in a class other in the one that it is in
- Usual solution is to use Move Method to move it to its appropriate class
- Sometimes only part of the method suffers from envy so in that case you can use Extract Method on the jealous part and Move Method to move it to its rightful home.
- There are several **design patterns** that appear envious by design, e.g. **Strategy**
  - this is an example of where a code smell does not indicate any underlying problem — quite the opposite.

Primitive Obsession

- Java has primitives for integers, floats, boolean etc.
    - (Strings and dates are modelled as classes in Java, but are primitives in some other languages)

- Don't be reluctant to use "small" classes to model simple abstractions, e.g.
    - an amount in a currency,
    - ranges with an upper and lower bound,
    - special strings such as a telephone numbers.
- Such classes can make your code much clearer

- This smell occurs when primitive types are used where a class should have been preferred.
    - e.g. using a String for a phone number, instead of creating a proper PhoneNumber class.

Speculative Generality

- This smell occurs when a developer implements functionality that they speculate may be needed someday, but isn't actually required now.
- One clear sign of Speculative Generality is when the only users of a class or method are the test cases
- If you have abstract classes that are not doing enough then use Collapse Hierarchy
- Unnecessary delegation can be removed with Inline Class
- Methods with unused parameters can be fixed with Remove Parameter
- Methods named with odd abstract names can be repaired with Rename Method

Data Class

- This is a class that has fields, getting and setting methods, and nothing much else
    - Its methods are dumb data holders and are manipulated in too much detail by other classes
- Use Remove Setting Method on any field that should not be changed
- Look for where these getters and setters are used by other classes and try to use Move Method to move behaviour into the data class
- If you can't move a whole method, use Extract Method to create a method that can be moved

Refused Bequest

- A **Refused Bequest** is a method a class inherits from its superclass but doesn't use it

- If this happens a lot, the class hierarchy has been badly designed and needs to be dismantled and redesigned.

- Rejecting a **public** method is normally a serious matter
  - Polymorphic replacement is no longer possible. Also violates the Liskov principle.

- Push Down Method/Field may help here, or Replace Inheritance with Delegation.

Some Challenges when Refactoring

- Databases
  - An application may be tightly coupled its database schema.
  - The code and the database are often managed by different people/ teams, so refactoring enterprise database applications is a challenge.
- Changing interfaces
  - If colleagues are using the classes you've developed, you cannot refactor the interfaces, unless your colleagues' code is refactored as well.
- Test cases
  - If interfaces are changed during refactoring, test cases will need to be refactored as well.
- Radical refactoring is hard
  - A large code overhaul is challenging.

Refactoring and Performance

- A common concern with refactoring is its effect on performance.
- Refactoring will typically make software slower but it also makes the software more amenable to performance tuning
  - In all but extreme real-time contexts, write tuneable software first and then tune it for sufficient performance
- Profiling ensures that you focus your tuning efforts in the right places
  - e.g. minimising database access is usually far more important than optimising in-memory processing

Summary

- Refactoring is the process of improving the design of a program, without changing its behaviour.
  - Vital part of the software development process

- Individual refactorings have been named and automated versions are available in most IDE
  - Rename, Move, etc.

- Code smells are a hint that something may be wrong in your code.
  - Many smells have been identified and named. Tools exist to detect them (e.g. Findbugs).
  - Refactoring can be used to resolve code smells, but don't do this blindly