Refactoring

Comp 47480: Contemporary Software Development


CODE SMELLS
&
REFACTORING

What is Refactoring?

Refactoring is the process of changing 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.

Roadmap of this section

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 beneficial.

Refactoring is central to this module — why?

Agile relies on **emergent design** -- the design of the software evolving over time. This design evolution is enabled, and driven, by refactoring.

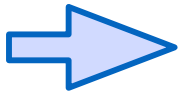
Testing: (1) Unit tests can be used to check that a refactoring hasn't changed behaviour. (2) Refactoring is a core step in Test-Driven Development.

OO Principles: The principles we looked at typically inform the refactoring process — often you refactor to improve adherence to these principles

Design Patterns: The goal of a refactoring sequence may be to introduce a design pattern to a program.

Simple Refactoring Examples

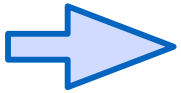
```
class Customer {
   ...
   Date date;
}
```



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

Rename

```
class A extends B {
   ...
}
```

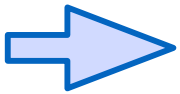


```
class A {
   public A() {
      myB = new B();
   }
   ...
   private B myB;
}
```

Replace Inheritance With Delegation

Another Refactoring Example

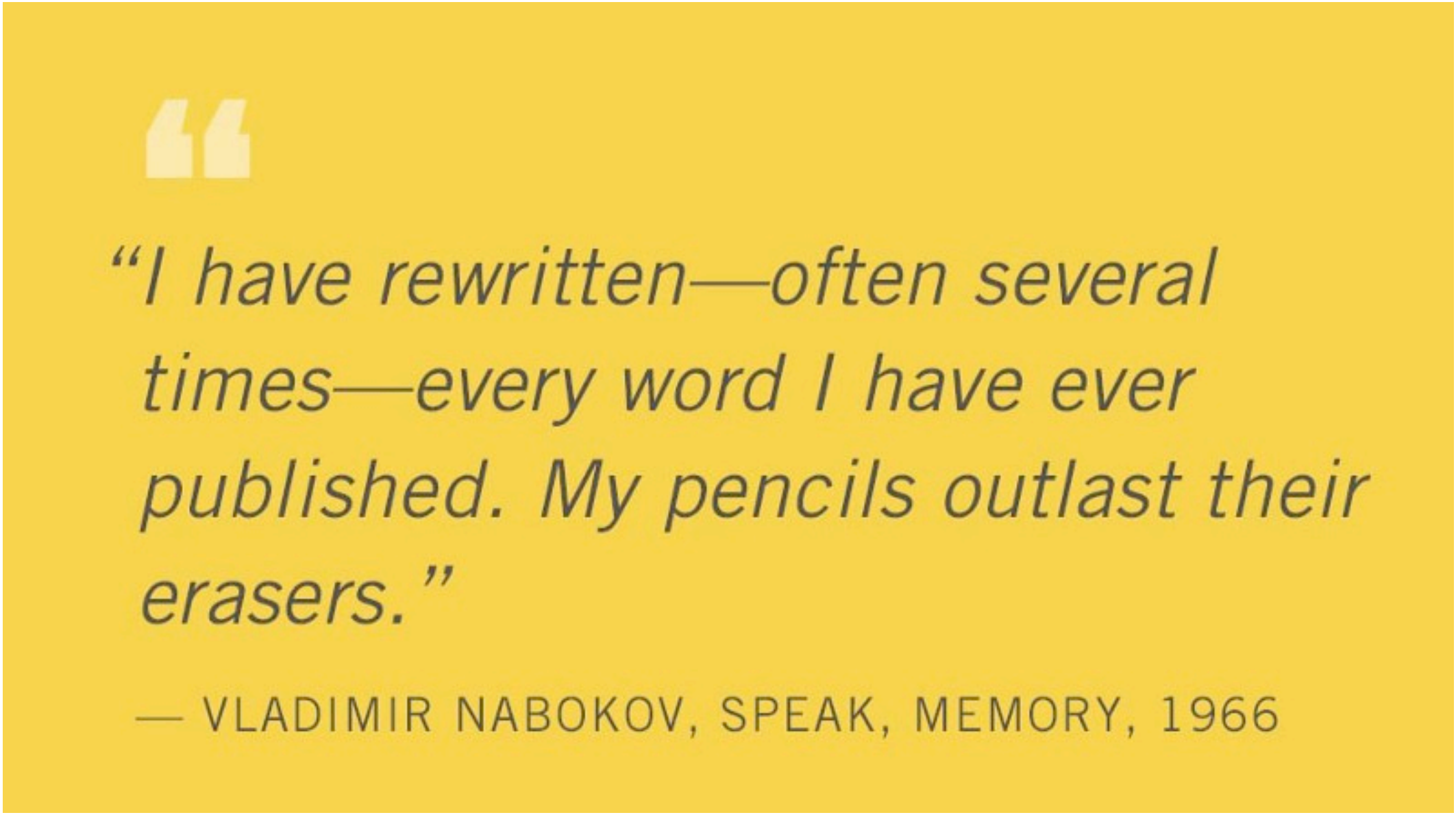
```
List<Double> numbers = ...;
double average = 0.0;
for (double num : numbers){
  average += num;
}
average = average / numbers.size();
```



```
List<Double> numbers = ...;
double runningTotal = 0.0;
for (double num : numbers){
  runningTotal += num;
}
double average = runningTotal / numbers.size();
```

Many refactorings we apply don't have formal names.

Refactoring transcends software



Why Refactor?

Without refactoring program design will **decay** over time.

Decay? Really?

- When software is updated in an undisciplined way, its design will match its functionality less and less — it appears to **decay**.

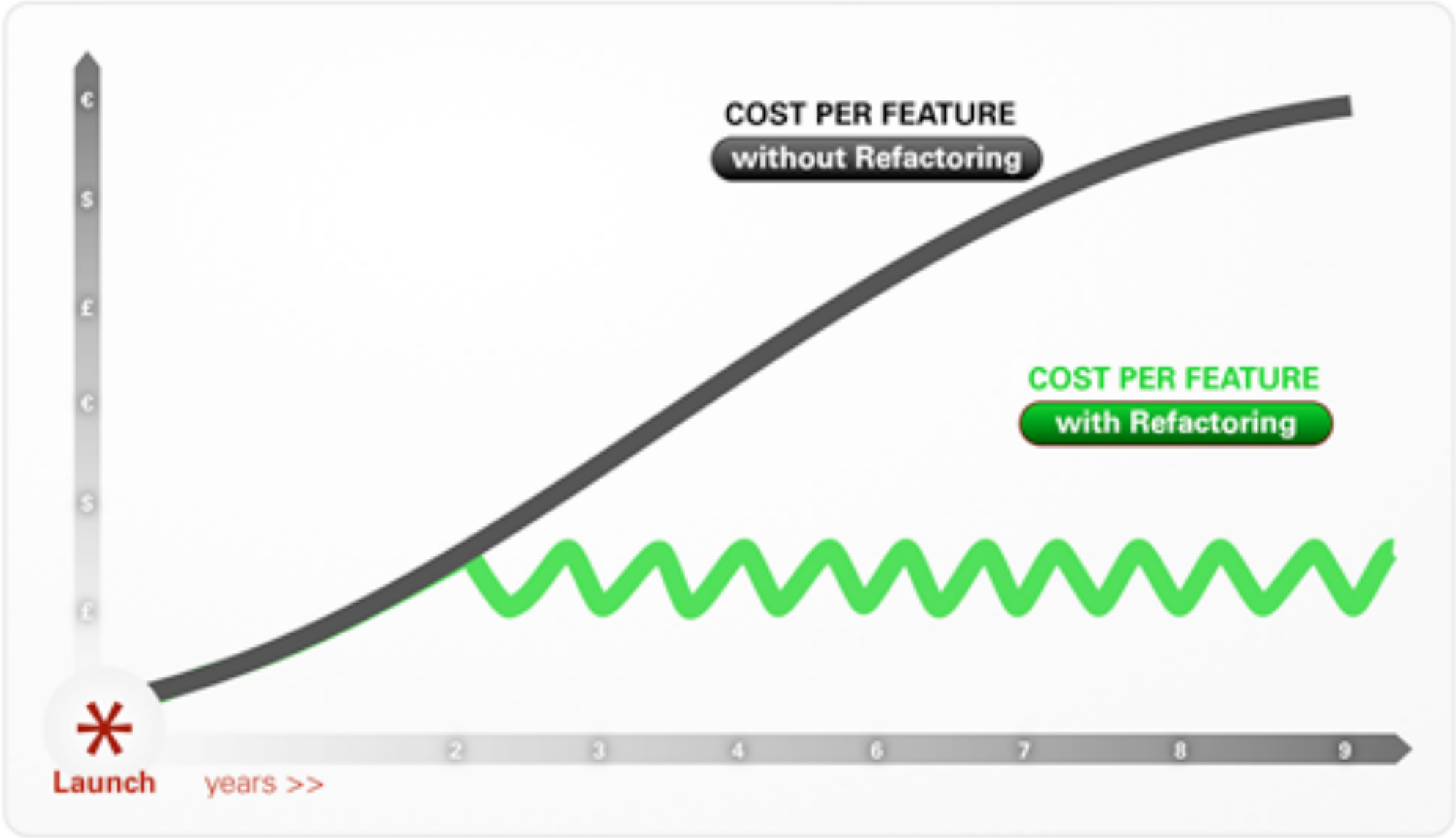
Refactoring makes software easier to understand

- Somebody else will have to read your code, or you will have to return to it in the future

- Makes fixing bugs easier, less chance of creating bugs

- Enables you to program faster

Refactoring aims to reduce maintenance costs



When to Refactor

Refactor whenever you add functionality. If the existing design makes it hard to add the new feature, refactor it.

Refactor as part of TDD. Recall the "red, green, refactor" cycle of Test-Driven Development.

Refactor when you fix a bug. A bug may be a sign that the code was not clear enough in the first place.

Refactor after a code review. A code review is where you learn the team development practices.

Note the so-called "rule of three". You *might* duplicate code *once*, or hack a single exceptional case *once*. However if it happens again, refactor to solve the design problem properly.

The Role of Test Cases

- If you use a refactoring tool to refactor, you can be confident that program behaviour will not change
  - Not 100% certain -- refactoring tools 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

- A refactoring tool will check preconditions for you

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


COFFEE
REFACTORING

Individual Refactorings

In this section we look at a number of individual refactorings.

There are many other refactorings, but once you understand a number of them the rest seem natural.

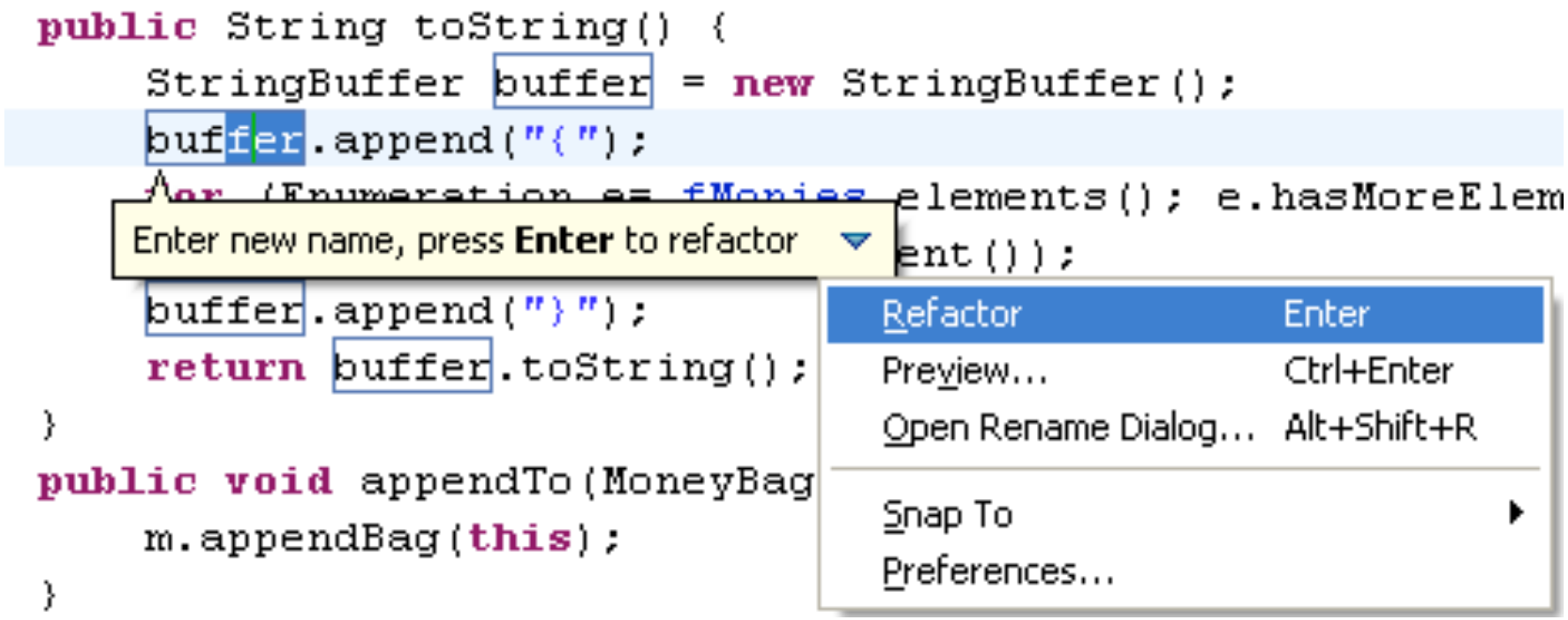
Refactorings are mostly language-independent, but we'll assume we're using a statically typed language like Java or C#.

Individual Refactorings

Rename method/field/class	Improve naming
Move class/method/field, **Pull Up** method/field, **Push Down** method/field	Moving entities to better locations
Extract local variable/constant, **Convert** Local Variable to Field	Improving code within a class
Extract Method **Inline** Method	Splitting and merging methods
Extract/Collapse superclass **Extract** interface	Improving inheritance hierarchy and interfacing

Rename <anything>

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

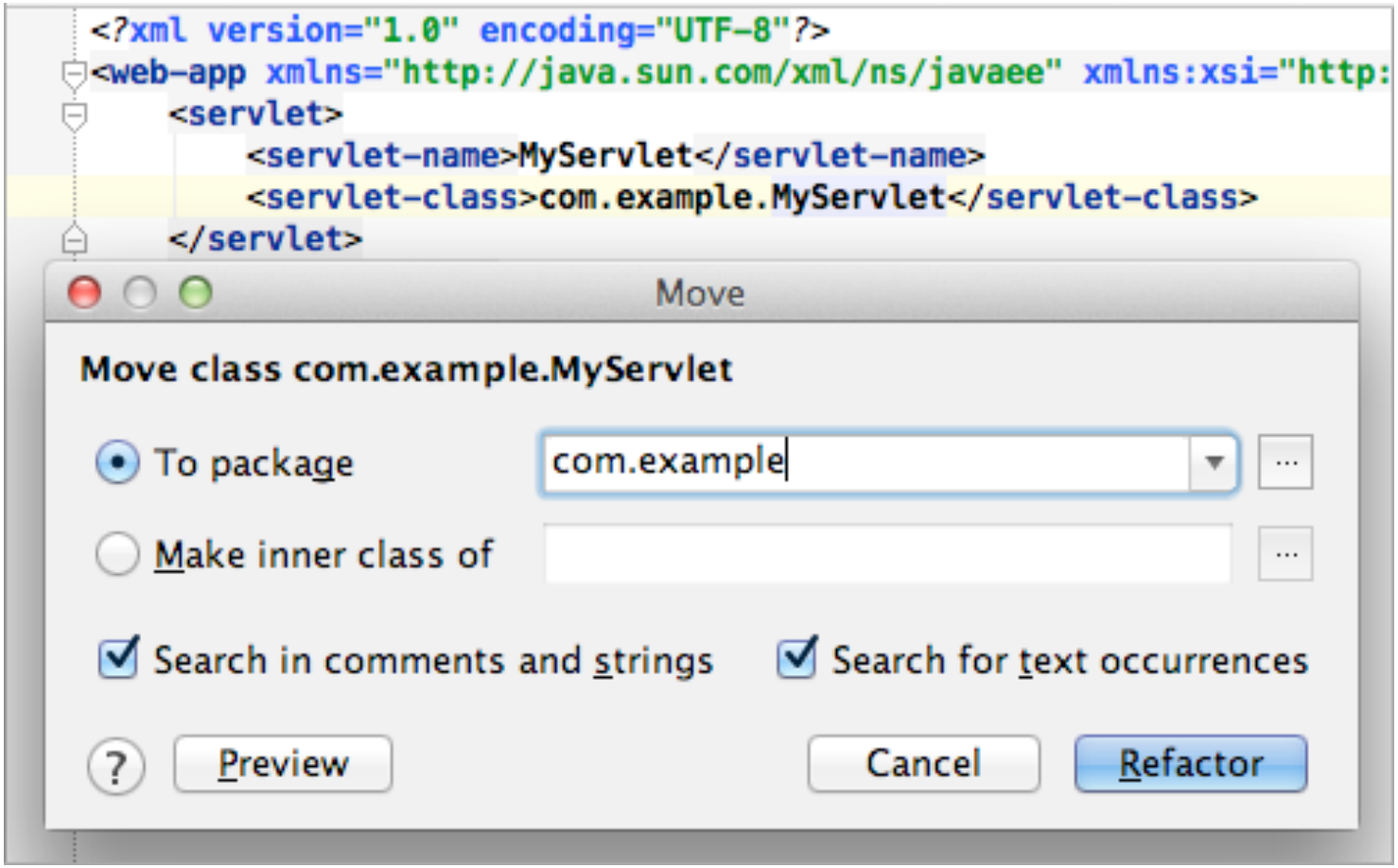


Comments on Rename

- Rename is by far the most commonly used refactoring!
  - although it only clarifies code, not changes its structure
- 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

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.



Move Method Example

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

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

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

Issues with this code

- The **`participatesIn`** method uses data of the **`Project`** class rather than its own. This violates the SRP and is an example of the **feature envy** code smell that we'll see later.

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

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

Applying Move Method

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

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

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

getParticipants method removed

participatesIn method removed

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

Pull Up Method/Field

Moves a method or a field from a class (or set of sibling classes) to a (common) superclass, thus avoiding the duplication of the method/field in the subclasses.

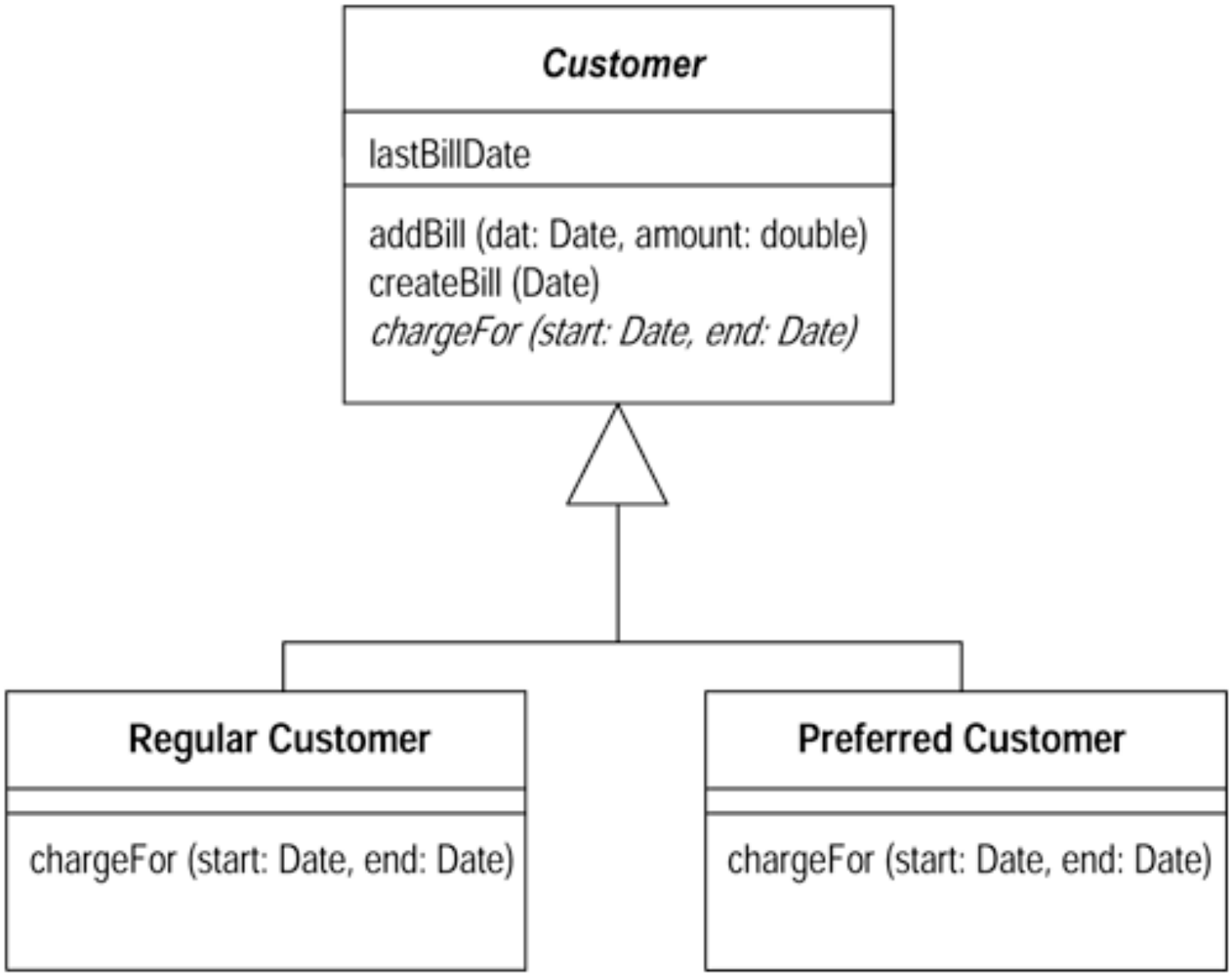
The methods in the sibling classes may be slightly different and need some adjusting prior to being moved.

Pull Up Method may be applied to make a method available in the sibling classes, or simply because it makes more sense in the superclass.

This refactoring is tricky when the method refers to features available in the subclass, but not in the superclass.

How to handle this?

Pull Up Method Example

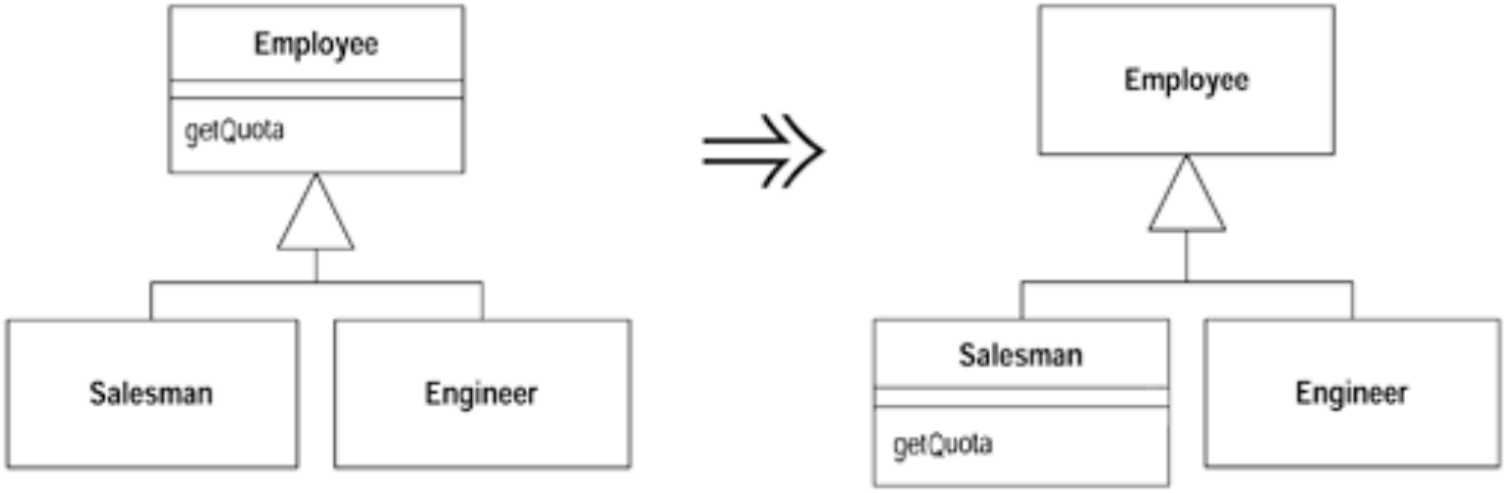


Push Down Method/Field

Moves a method or a field from a class to one or more immediate subclasses.

Reverse of Pull Up Method/Field.

Apply this refactoring when a method or field is used only in a number of subclasses and it's clearer to move it there.



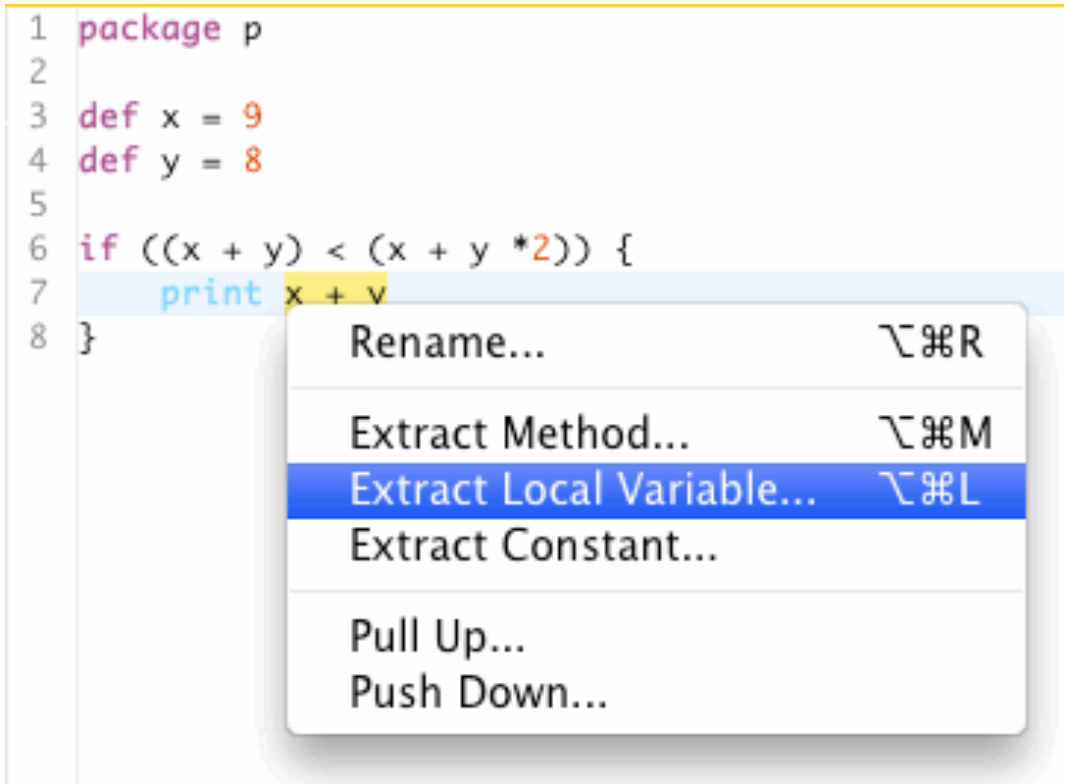
What principle is being violated?

Extract Local Variable

Assign the result of an expression to a new local variable.

Simplifies a complex expression by dividing it into multiple lines

Useful also if the same expression is evaluated more than once




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

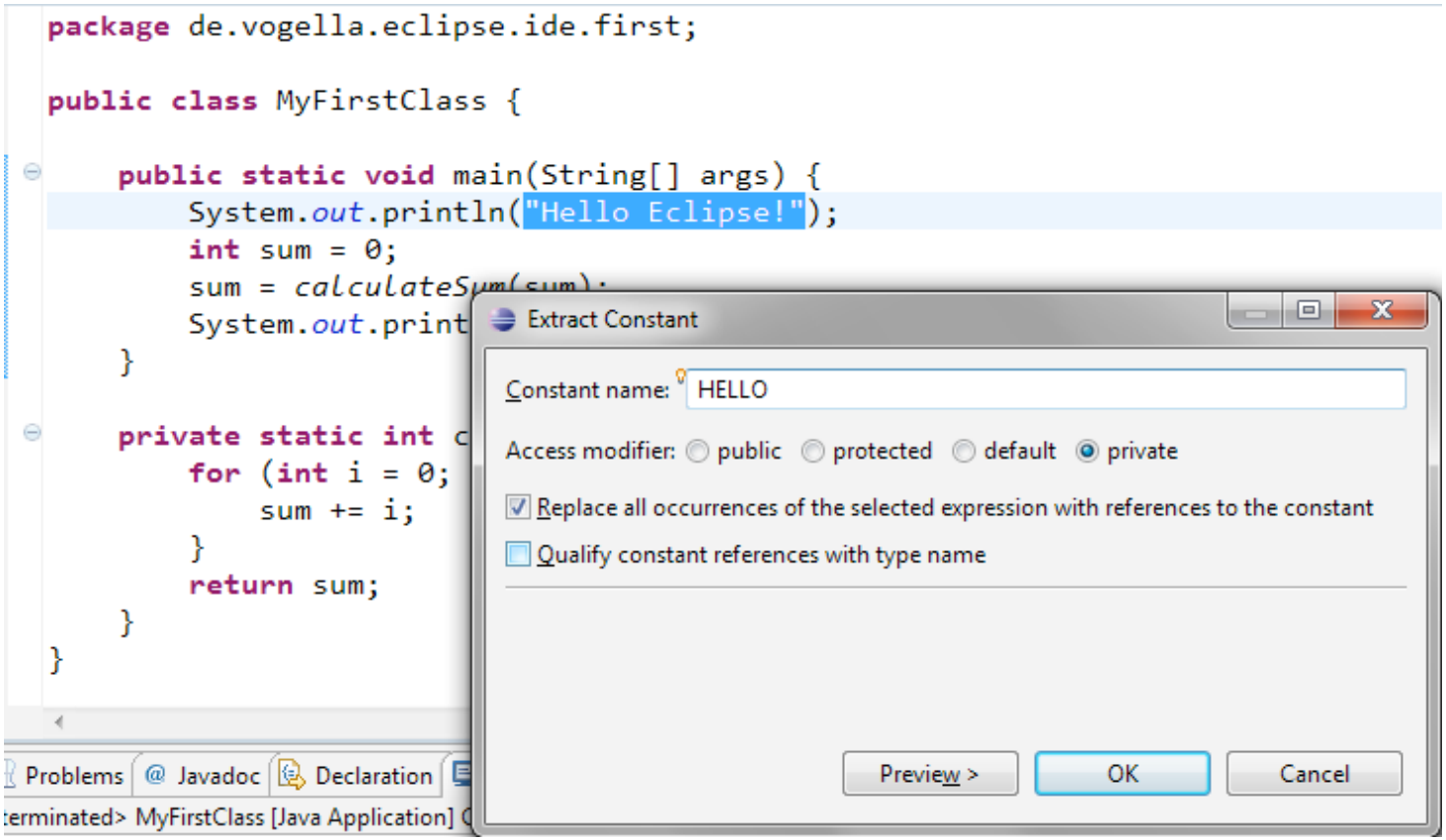
```
sum = x + y;
if (sum < (x + y*2)){
  print sum;
}
```

Extract Constant

Converts a number or string literal to a static final field (Java).

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.



Convert Local Variable to Field

Converts (promotes) a local variable it to a private field.

(Only use a field where necessary: a field adds complexity to its entire class; a local variable only adds to the complexity of the block it belongs to.)

Apply this refactoring when the need arises for a local variable within a method to be accessed in another method as well.

The inverse, Convert Field to Local Variable, is arguably a more useful refactoring.

Extract Method

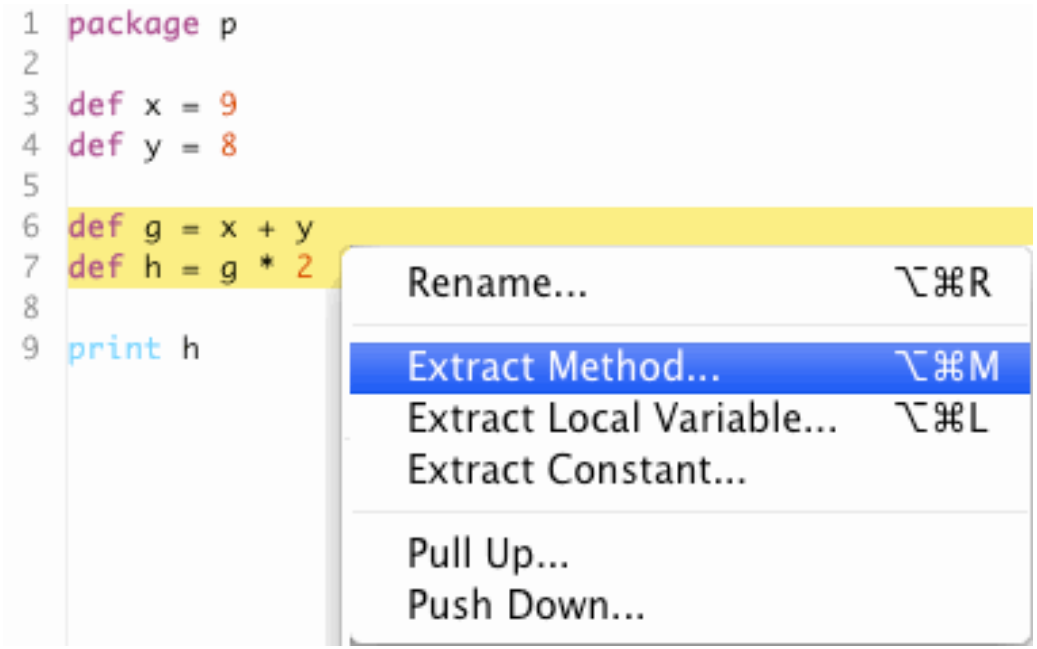
Convert a selected block of code into a method.

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

Extract method example




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

h = foo(x, y)
print h
```

Rename would now be used to give `foo` a better name

Inline Method

Occasionally a method's body is just as clear as its name

- only ever the case for a very short method!

This refactoring replaces calls to the method with the body of the method itself.

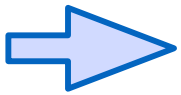
Then the method can be removed.

This refactoring is the reverse of Extract Method

Inline Method Example

```
int getRating() {
  return (moreThanFiveLateDeliveries()) ? 2 : 1;
}

boolean moreThanFiveLateDeliveries() {
  return numberOfLateDeliveries > 5;
}
```

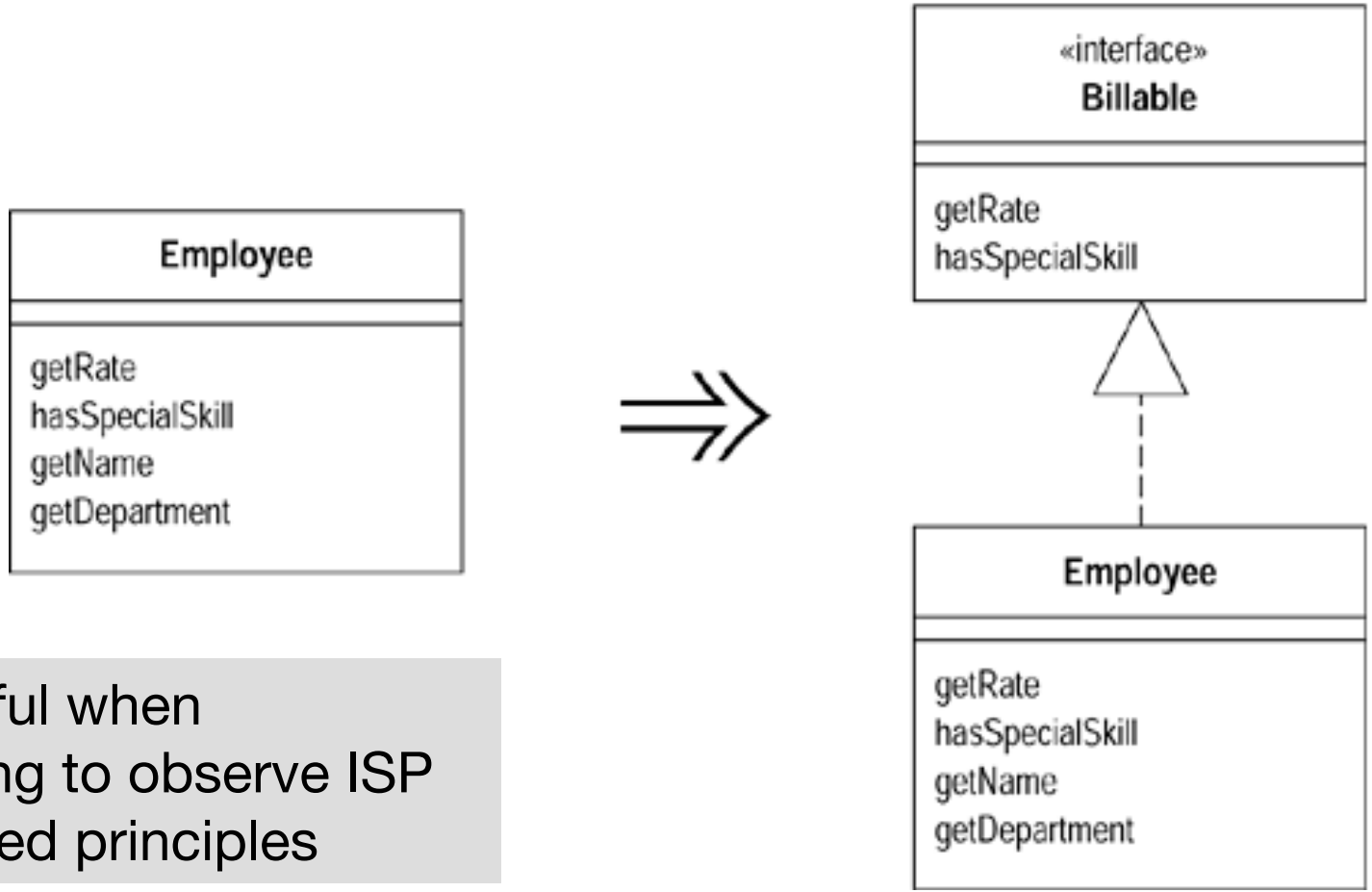


```
int getRating() {
  return (numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Inlining **`moreThanFiveLateDeliveries`** leads to clearer code.

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


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?

No need to fix all smells (this would be crazy)

* Some "smells" don't indicate problems
* Some problems aren't bothersome, or are not of concern
* Badly designed *stable* code is probably not a concern

Judicious refactoring may be used to resolve the design 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 only a hint**. They may or may not be anything actually wrong with the code.

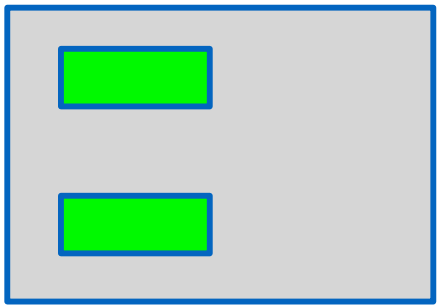
- It may not be worthwhile fixing the problem:
  - e.g. a small, inconsequential piece of code duplication that would require a system overhaul to resolve

- Code smell detection can help in identifying problem areas and in deciding what to do.

Duplicated Code I

If the same code fragment 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.

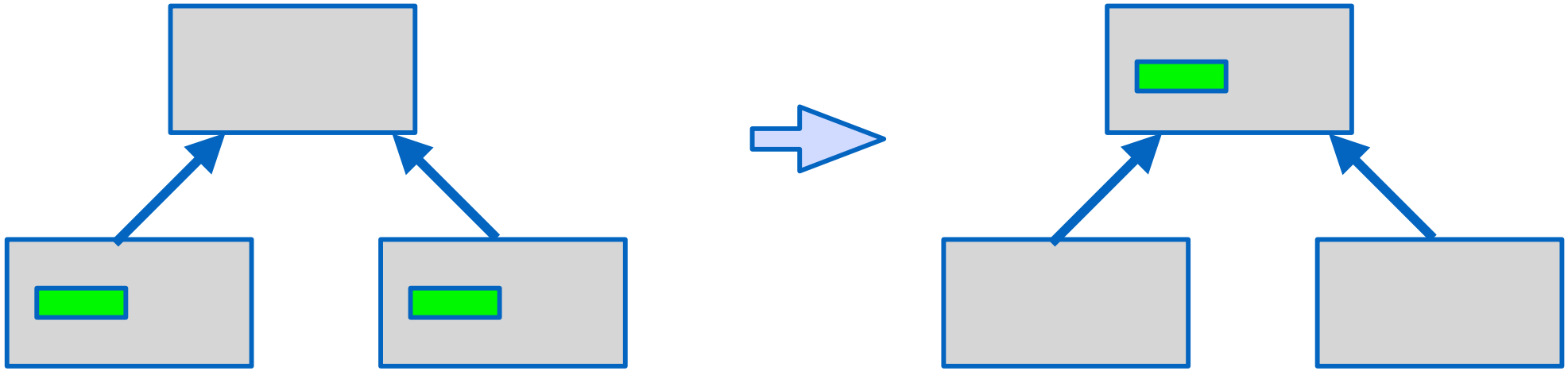


Consider performing **Extract Method** and invoke the new (private) method from both places.

Duplicated Code II

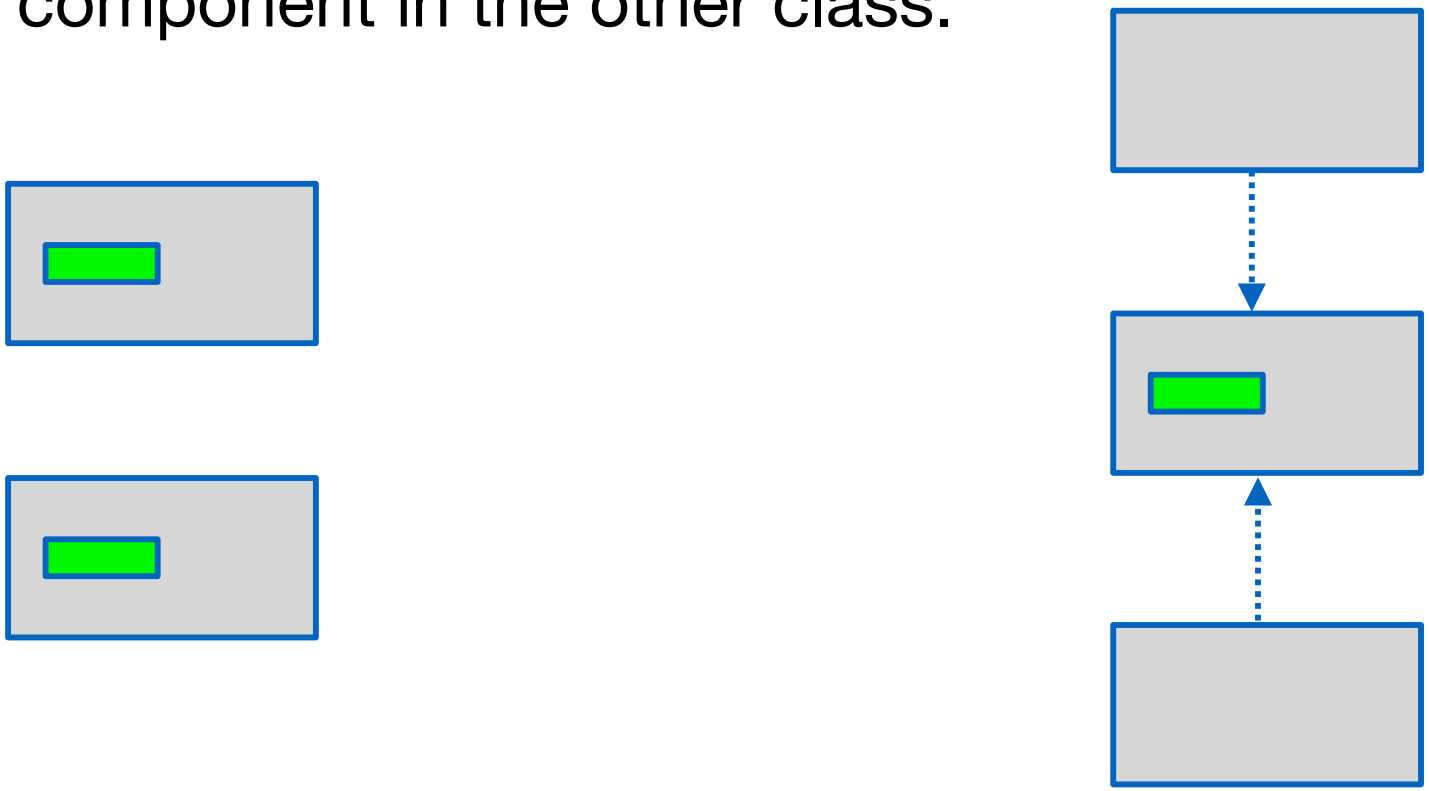
Another common duplication problem is having the same code fragment in two related subclasses.

Consider performing **Extract Method** in both classes, then **Pull Up Method** to the shared superclass



Duplicated Code II

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? And is it worth fixing?

Don’t overdo it!

One developer’s complaint about being over-zealous about removing duplicated code:

“I have seen duplicate code checkers complain about JUnit test cases being too similar. And yes, you probably could have extracted two lines of code into a private method so the three test cases with those lines could share it, but then to describe those two trivial lines of code you would need to spend a month of Sundays trying to come up with a sensible name, and this is somewhat missing the point of refactoring.”

Long Method

- The longer a method is, the more difficult it is likely to be to understand.

- What's long? Opinions vary! Thresholds from two authors
  - *A dozen lines* -- Martin Fowler
  - *200 lines* -- Steve McConnell

- There is little value in seeking a specific threshold
  - A Long Method is simply one that is too long and should be split

- **Extract Method** is usually key to resolving this smell.

God Class

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

Switch Statements I

A switch or decision statement that affects method behaviour should probably be replaced with polymorphism

```
enum StudentType {UNDERGRAD,POSTGRAD};

class Student {
   ...
   if (studentType == UNDERGRAD){
      ...
   }
   else {
      ...
   }
   ...
   private StudentType studentType;
}
```

A better design is probably two **`Student`** subclasses called **`Undergraduate`** and **`Postgraduate`**.

Switch Statements II

Use Extract Method to extract the relevant code and then Move Method to get code to the appropriate class.

In the example, all code related to how undergraduates are handled should be moved to the **`Undergraduate`** class.

If you only have a few cases in a single method then polymorphism is probably overkill.

If one of your conditional cases is null, consider applying the **Null Object** Design Pattern.

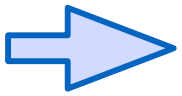
Long Parameter List

Long parameter lists are hard to understand and maintain.

A method needn't be passed everything it needs in the parameter list
* Pass in enough so it can get everything it needs
* (keeping the Law of Demeter in mind)

Consider using **Replace Parameter with Method Call**, where you replace one parameter with a request to an object you already know about, e.g.

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



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

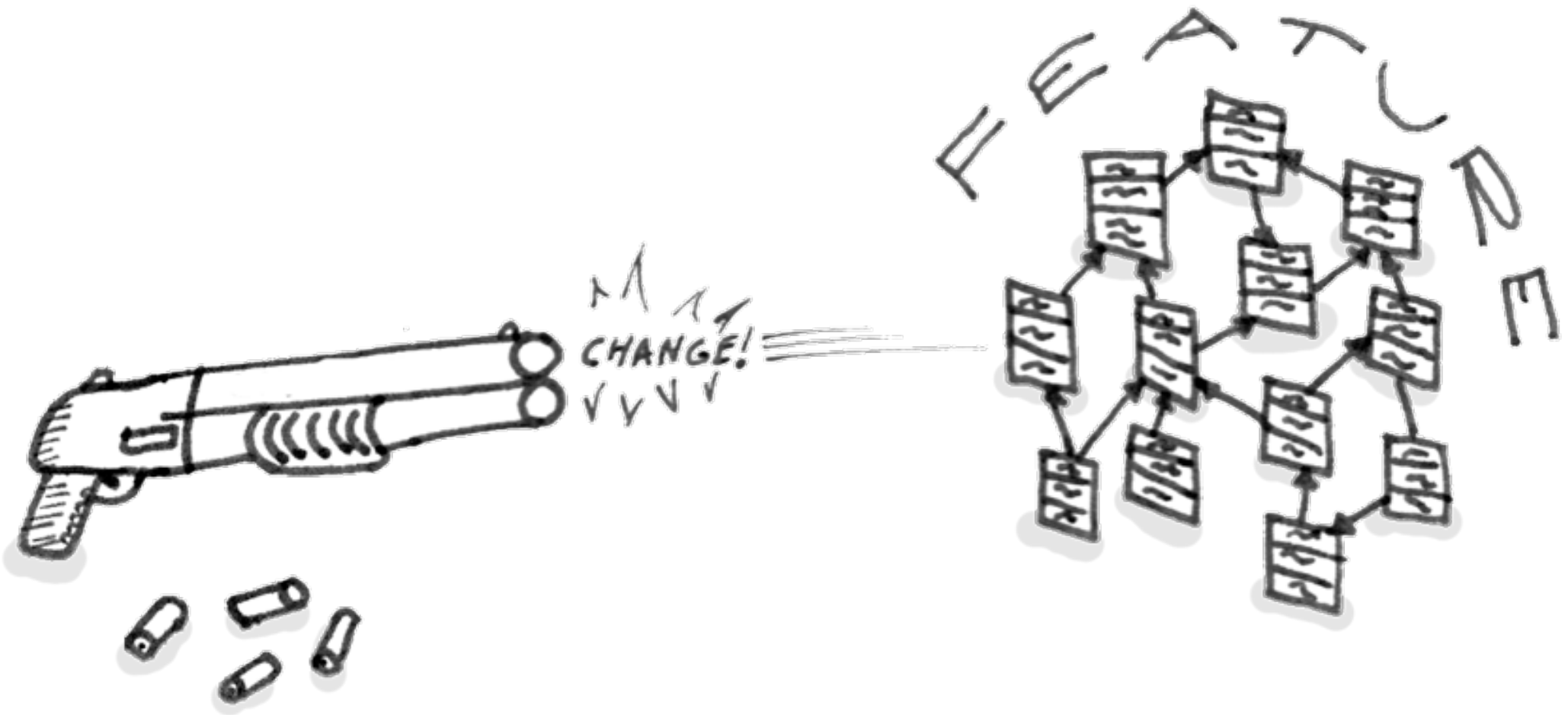
Divergent Change

- Divergent change occurs when one class is commonly changed in different ways for different reasons

- E.g. in a class in an invoicing application you have to change one set of methods every time the invoice format changes, and another set of methods whenever a new user is added...
  - Suggests a violation of the SRP
  - This class is 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!

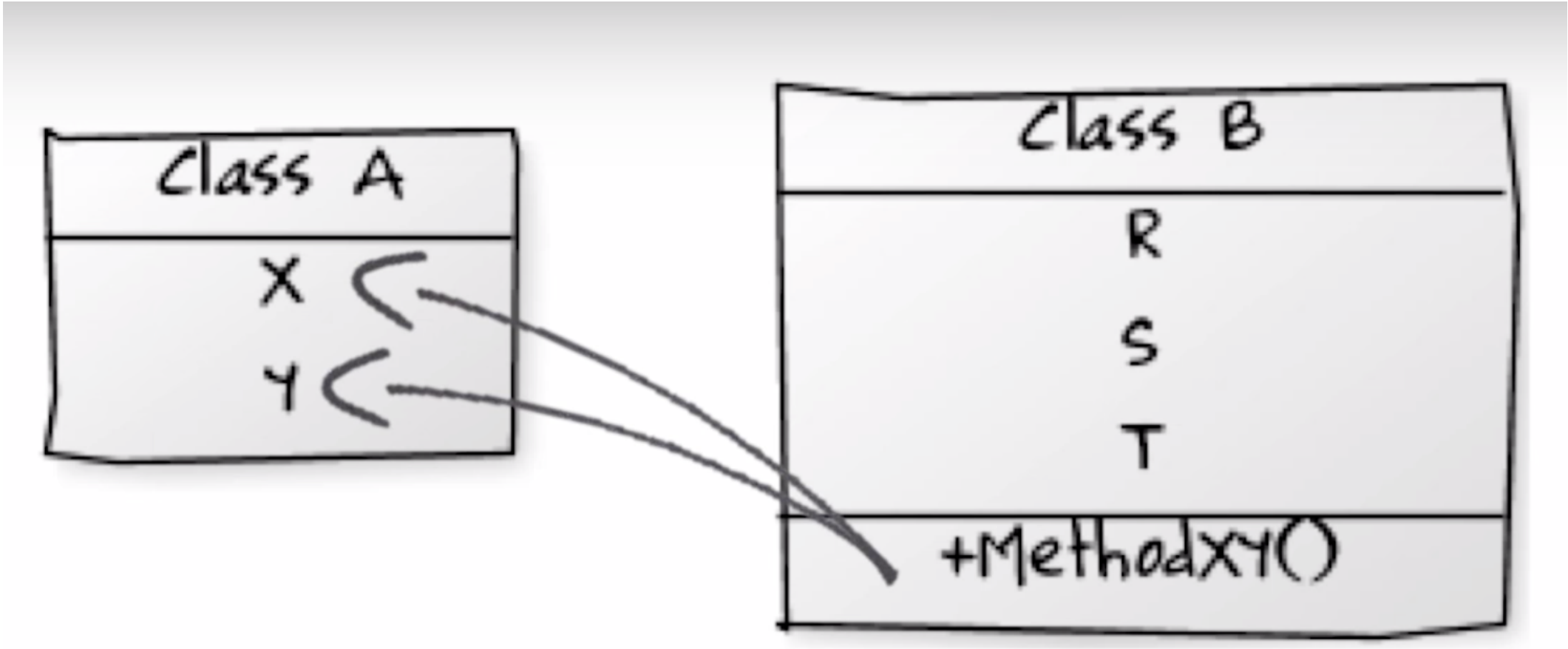
Shotgun Surgery



Shotgun Surgery

- If, whenever you change a certain feature, you make changes to 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

Feature Envy



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
- If only part of the method suffers from envy then consider using **Extract Method** on the envious 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 where a code smell does not indicate any underlying problem — quite the opposite.

Primitive Obsession

- Java has primitives for integers, floats, boolean etc.
- Primitive Obsession occurs when a primitive type is used where a class should have been preferred.
  - e.g. using a String for a phone number, instead of creating a proper PhoneNumber class.
- 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 student number.
- Such classes can make your code much clearer

Speculative Generality

- This is when a developer implements functionality that they speculate may be needed someday, but isn't currently required.
- One clear sign of Speculative Generality is when the only users of a class or method are the test cases. Here are some of the refactorings you might apply:
  - 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

- A data class has fields, getters and setters, and little or nothing else
  - Any interesting functionality it has is implemented by other classes, and often duplicated

- Use **Remove Setting Method** on any field that should not be changed

- Look 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**.

Refactoring Challenges I

- **Databases**
  - An application may be tightly coupled its database schema.
  - Code refactoring may require that the database schema be changed as well, so refactoring enterprise database applications is a challenge.

- **Changing Interfaces**
  - If refactoring involves changing class interfaces, then other classes that use these interfaces must be changed as well.

Refactoring Challenges II

- **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, but sometimes necessary

Refactoring and Performance

- How does refactoring affect performance?
- Refactoring will typically make software run more slowly
  - This rarely matters!
  - Refactored software is more amenable to performance tuning
- Profiling ensures that you focus your tuning efforts in the right places
  - e.g. minimising database access is usually far more significant than optimising in-memory processing

Summary

- **Refactoring** is the process of improving the design of a program, without changing its behaviour. It's a vital part of the software development process

- Individual refactorings have been named and automated versions are available in most IDEs
  - 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. JDeodorant, Findbugs).
  - Refactoring can be used to resolve code smells, but don't do this blindly