

Software Refactoring

Comp 30160: Object Oriented Design

(Based on material by Martin Fowler, <http://sourcemaking.com>, and the Eclipse tutorial material at www.ibm.com. Slides are quite bare in places, supplement with extra reading.)

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.

What is Refactoring?

- Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.
- Refactor (verb): to restructure software by applying a series of refactorings without changing its observable behaviour.
- “without changing observable behaviour” -- the program may behave differently internally, but its functionality remains identical.

Why Refactor?

- Refactoring improves the design of software
 - Without refactoring the design of the program will **decay** over time
 - Poorly designed code usually takes more code to do the same things, often because the code does the same thing in different places
- Refactoring makes software easier to understand
 - In most software development environments, somebody else will eventually have to read your code
- Refactoring helps you find bugs
- Refactoring helps you program faster

When to Refactor I

- The “rule of three” heuristic
 - You *might* duplicate code *once*, or hack a single exceptional case *once*. However if it happens again, refactor.
- Refactor whenever you add functionality
 - Helps you to understand the code you are modifying
 - Sometimes the existing design does not allow you to easily add the feature
- Recall the “red, green, refactor” mantra of Test-Driven Development.

When to Refactor II

- Refactor when you need to fix a bug
 - A bug report is a sign the code needs refactoring because the code was not clear enough for you to see 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)

Some Challenges when Refactoring

- Databases
 - Many applications are tightly coupled to the database schema that supports them
 - 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 may well need to be refactored as well.
- Big design changes can be difficult to refactor
 - A large code overhaul is challenging.

Refactoring and Design

- XP advocates that you implement the first approach that comes to you, get it working, and then refactor it into shape
- The point is that refactoring changes the role of upfront design
- You are no longer looking for the perfect solution when doing design but a reasonable one
- Refactoring can lead to simpler designs while improving flexibility

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 to 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

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

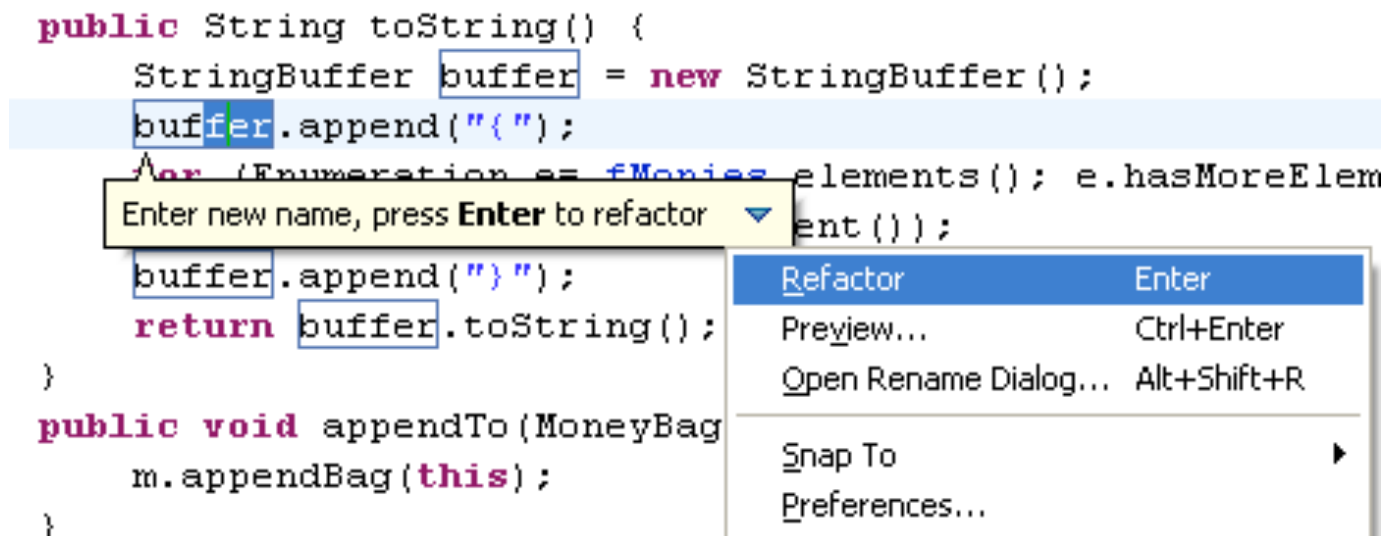
- In order for a refactoring to preserve program behaviour, certain preconditions must hold
 - If you rename a field from `alpha` to `beta`, there must not already be a field called `beta` in the class
- Using a refactoring tool, you don't have to check preconditions
 - The tool will do this for you
- For each refactoring we look at, you should have an idea what the key elements of the preconditions are.

Individual Refactorings (supported by Eclipse)

- Rename method/field/class
 - Move method/field
 - Extract method
 - Pull up method/field
 - Push down method/field
 - Inline ...
 - Extract Superclass
 - Extract Interface
 - + many more
-
- Note that refactoring can be applied in any language, but from here on we'll assume we're using **Java**.

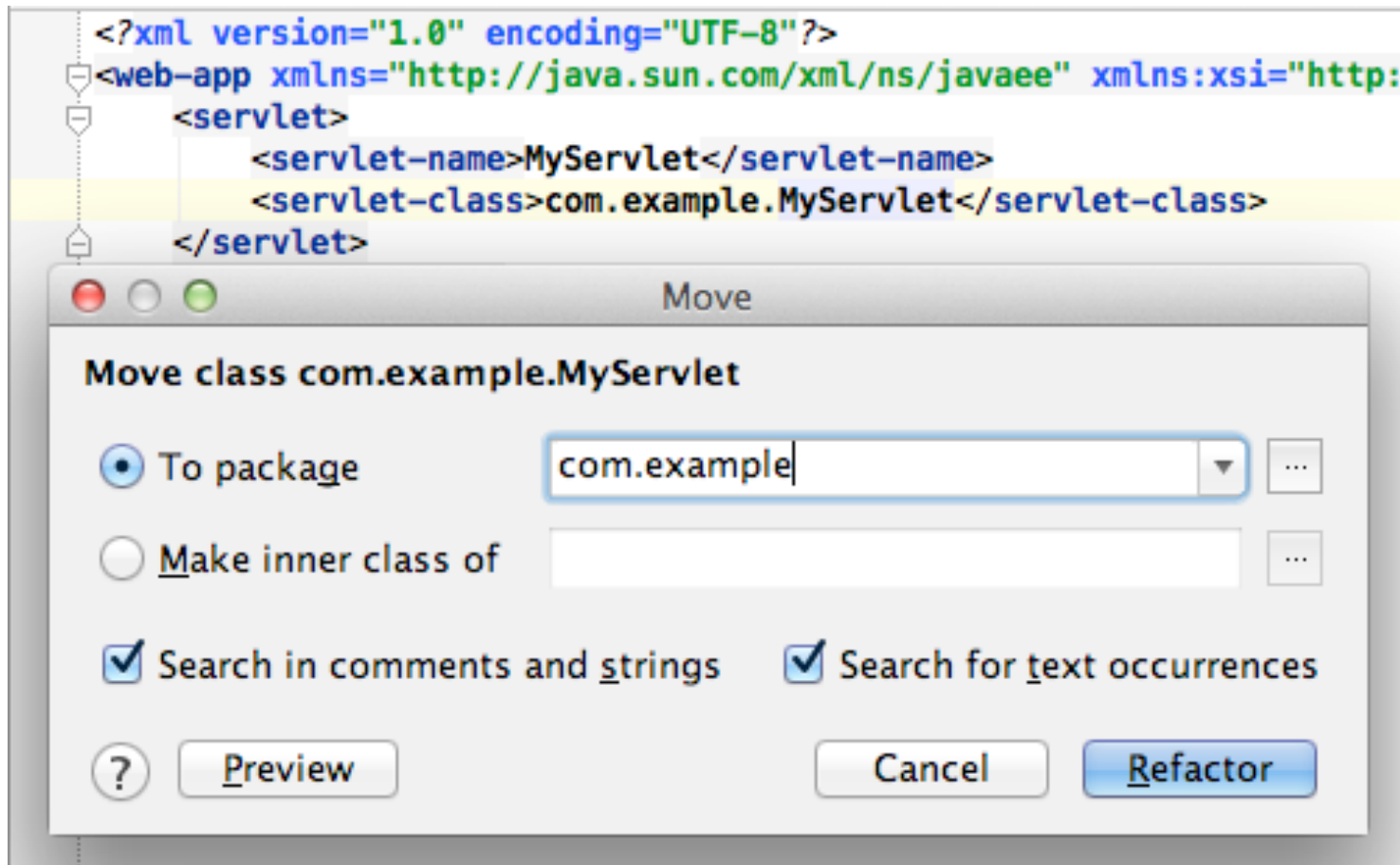
Rename <anything>

- Allows you to rename variables, classes, methods, packages, folders, and almost any Java identifiers
- Useful for making code clearer
- By far the most commonly used refactoring!
- (Oddly enough, the common implementation of this refactoring contains subtle bugs due to the baroque complexity of the Java Language Specification!)



Move class/method/field

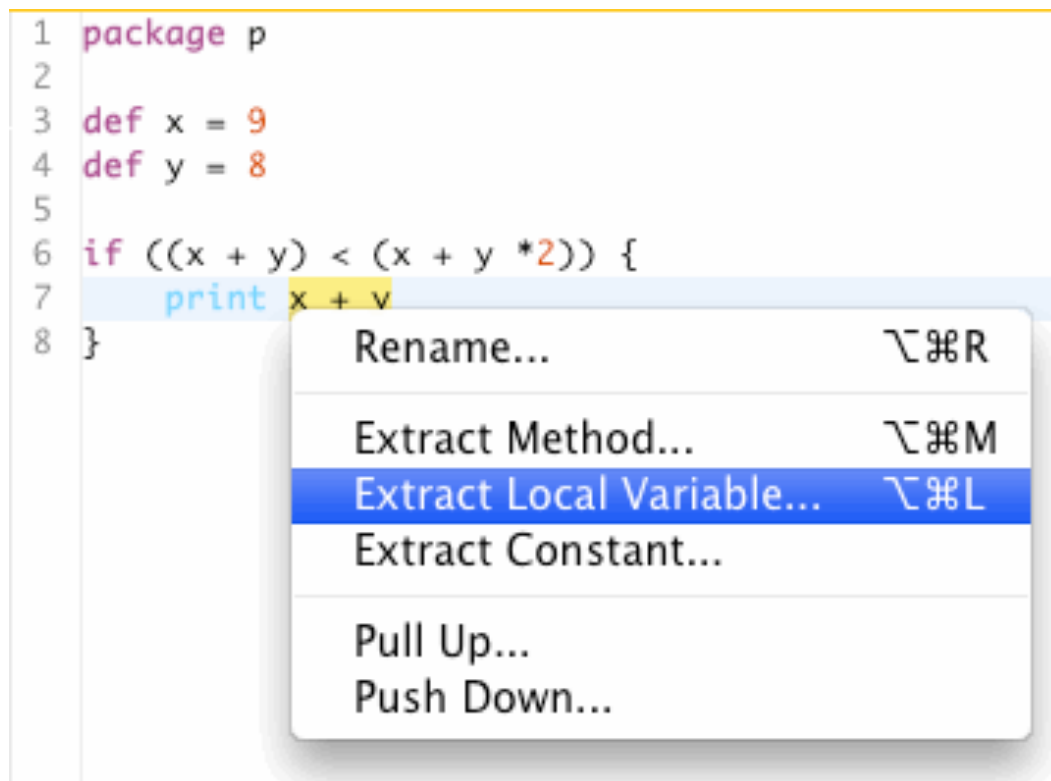
- Allows you to move a class to another package or a method/field to another class.
- For moving a method/field, the source class must have a reference to the target class.



Extract local variable

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

```
1 package p
2
3 def x = 9
4 def y = 8
5
6 if ((x + y) < (x + y * 2)) {
7   print x + y
8 }
```



The screenshot shows a code editor with the following code:

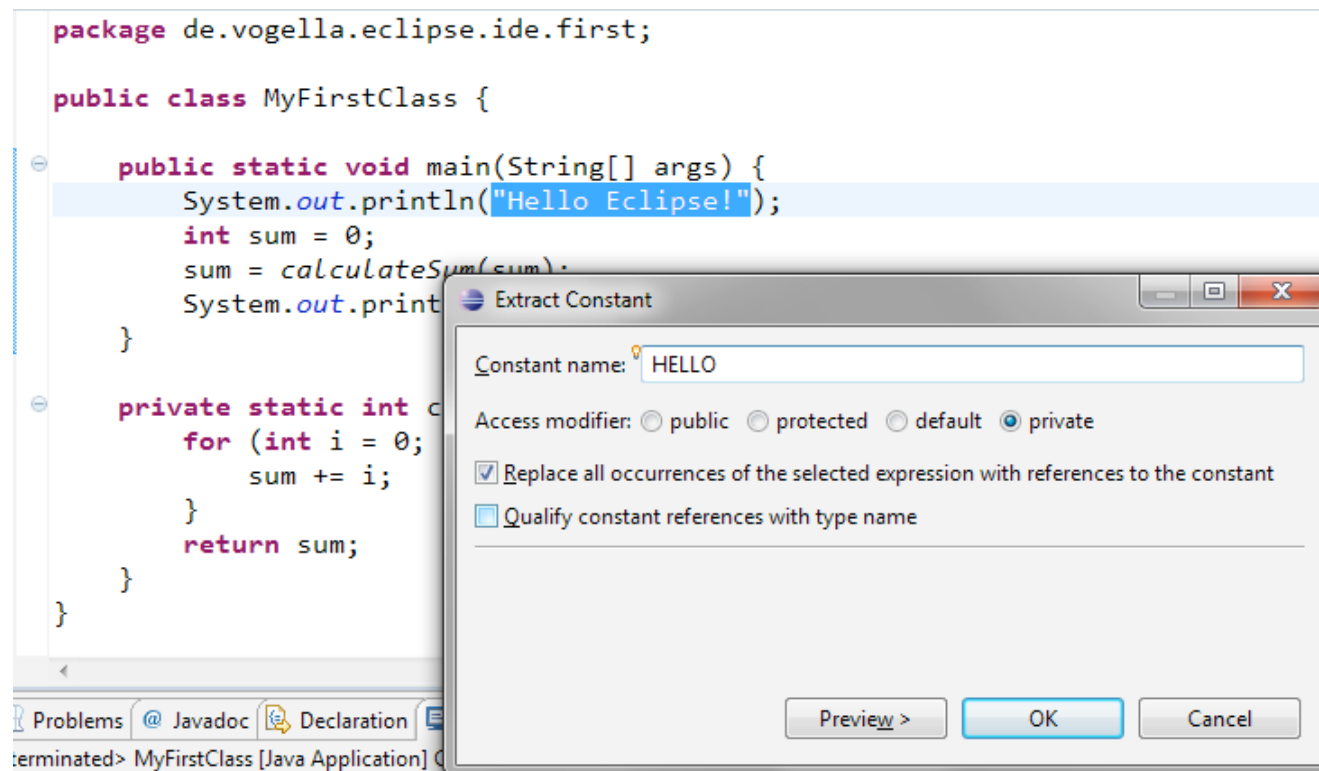
```
1 package p
2
3 def x = 9
4 def y = 8
5
6 if ((x + y) < (x + y * 2)) {
7   print x + y
8 }
```

The line `print x + y` is highlighted. A context menu is open over this line, showing the following options:

- Rename... $\backslash\text{⌘}R$
- Extract Method... $\backslash\text{⌘}M$
- Extract Local Variable... $\backslash\text{⌘}L$**
- Extract Constant...
- Pull Up...
- Push Down...

Extract constant

- Converts any number or string literal in your code 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 number or string literal in just one place (the value of the field), instead of doing a search and replace throughout your code

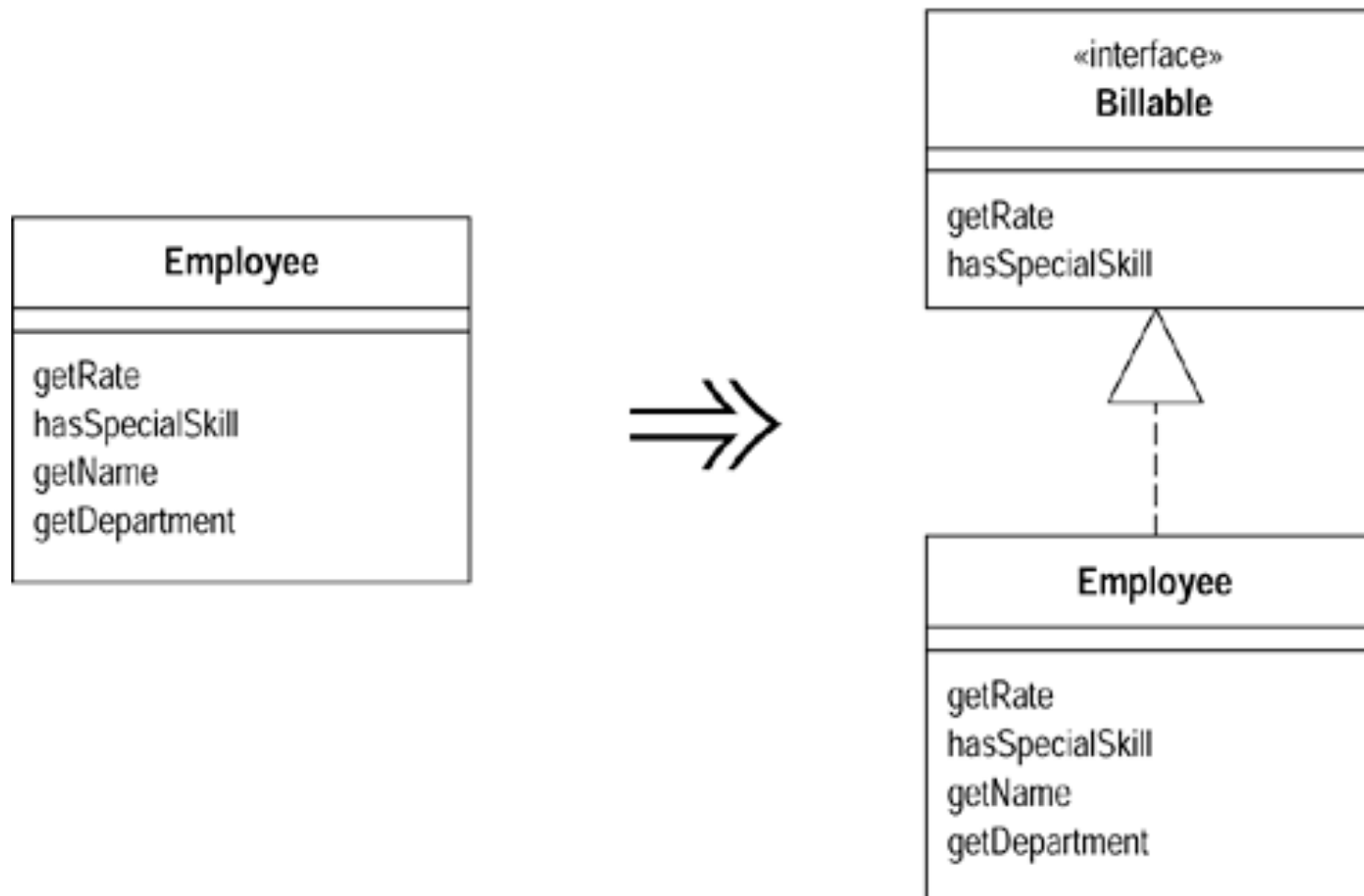


Convert local variable to field

- 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.

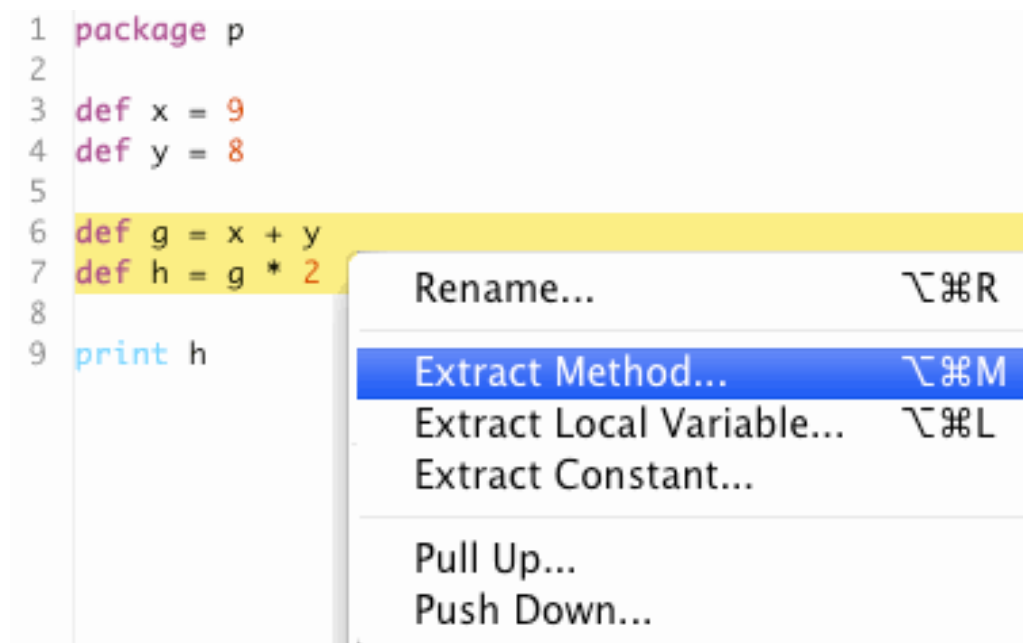
Extract interface

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



Extract method

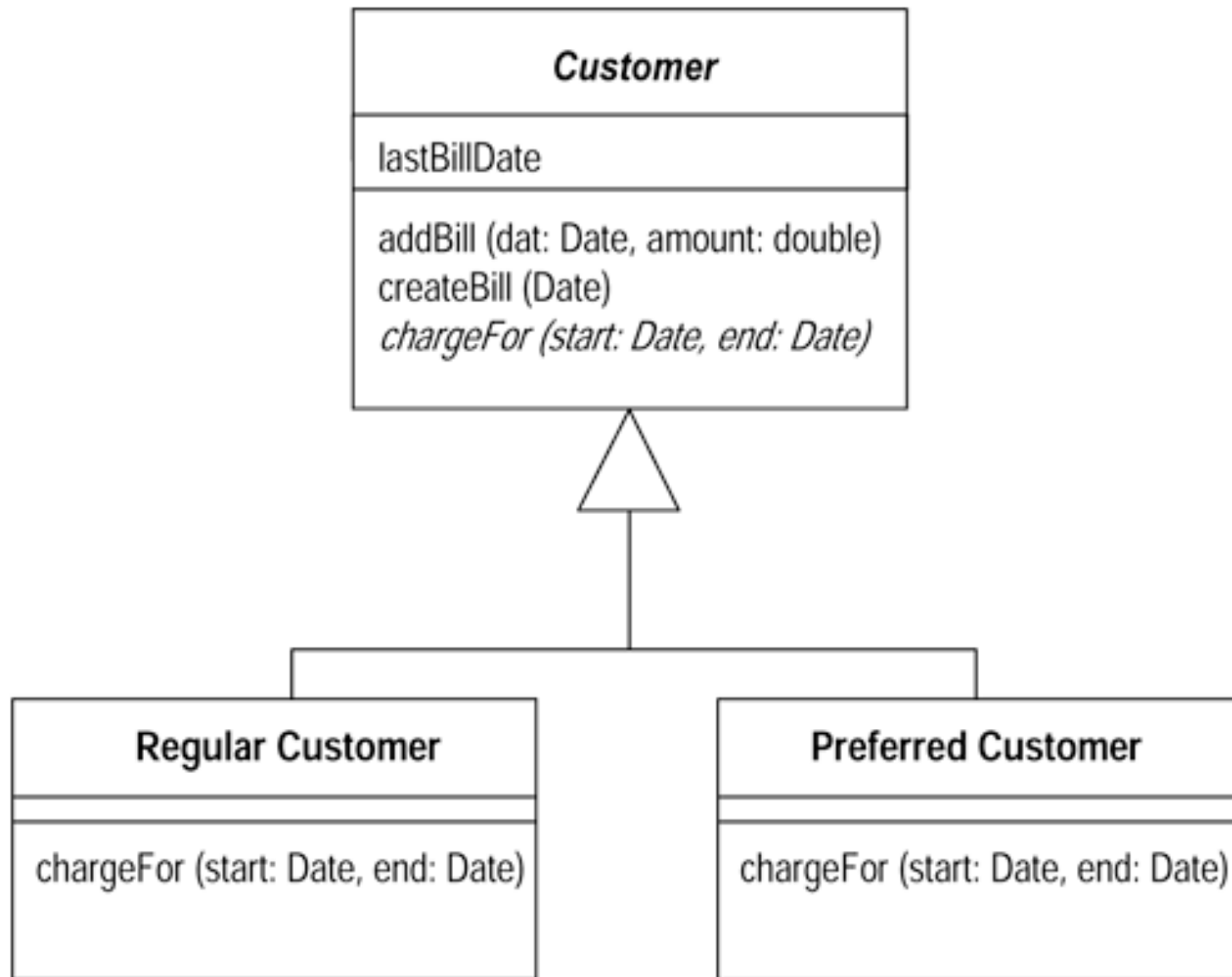
- Select a block of code and convert it to a method. It's useful when:
 - a method is too long and you want to subdivide blocks of it into different methods
 - a piece of code is duplicated across many methods.
 - A piece of a method does a coherent task and it's clearer to split this off into its own method.
- The block of code is put into its own method and the original block replaced with a call to the newly-created method.



Pull up method/field

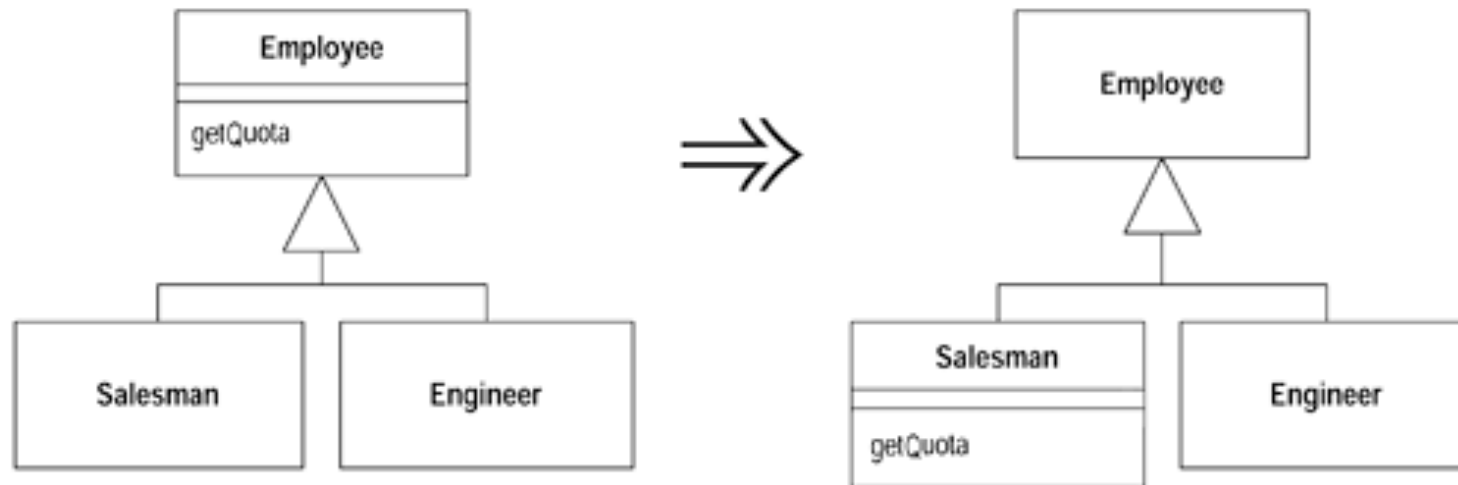
- Moves a method or a field from a class (or set of sibling classes) to the immediate superclass.
- Avoids the duplication of the method/field in the subclasses
- Sometimes the methods in the sibling classes may be slightly different and need some adjusting prior to being moved.
- It may be useful to 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



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

- 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.
- There's no need to fix all smells (this would be excessive)
- Judicious refactoring can be used to resolve the problem underlying the smell.
- Here's we look at a number of well-known smells...

Bad Smells in Code

This is a list of code smells you should become familiar with.

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

A Really Big Caveat

- Never read refactoring&smells like a medicine bottle
 - “At temperatures over 38.5 degrees, take two spoons 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, the program may be improved if you find a way to unify the duplication
- The simplest duplicated code problem is when you have the same expression in two methods of the same class
 - Consider performing Extract Method and invoke the code from both places
- Another common duplication problem is having the same expression in two sibling subclasses
 - Consider performing Extract Method in both classes then Pull Up Method
- If you have duplicated code in two unrelated classes, consider using Extract Class in one class and then use the new component in the other class.

Long Method

- The longer a method is the more difficult it is to understand
 - What's long?
 - *A dozen lines* -- Fowler
 - *200 lines* -- McConnell
- Extract Method is usually part of the solution

God Class

- A "God Class" is an object that controls way too many other objects in the system.
- When a class is trying to do too much, it often shows up as too many fields and methods.
- A class with too much code is also a breeding ground for duplication
- In both cases Extract Class and Extract Subclass may help

Long Parameter List

- 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
- Long parameter lists are hard to understand and difficult to maintain
- 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

Divergent Change

- Divergent change occurs when one class is commonly changed in different ways for different reasons
 - Say you have to change 4 different methods every time the database changes, and 3 other methods whenever a new user is added...
 - Suggests that two classes are better than one
- To clean this up you 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

- This situation occurs when every time you make a certain kind of change, you have to make a lot of little changes to a lot of different classes
- When the required changes are scattered they are hard to find, it's easy to miss an important change, and maintenance is challenging.
- Opposite of Divergent Change in a sense
- Consider using Move Method and Move Field to put all the change-prone elements in a single class
- If no current class looks like a good candidate then it may be best to create a new class to bring the relevant behaviour together

Feature Envy

- Feature Envy involves a method that seems more interested in a class other in the one that it is in
- The method clearly wants to be elsewhere, so use Move Method to get it there
- 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 get it home.
- There are several design patterns that appear envious by design, e.g., **Strategy**
 - this is an example of where smells may not be bad

Primitive Obsession

- Java has primitives for integers/floats, but strings and dates, which are primitives in many environments, are modelled as classes
- 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.

Lazy Class

- Each class you create costs money and time to maintain and understand
- A class that is not carrying its weight should be eliminated
- If you have subclasses that are not doing enough try to use Collapse Hierarchy. Nearly useless components should be subjected to Inline Class

Speculative Generality

- This smell comes from thinking "I think we need the ability to do this someday" and you need all sorts of hooks and special cases to handle things that are not required
- One clear sign of Speculative Generality is when the only users of a class or method are 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 should be repaired with Rename Method

Data Class

- These 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 is ungrateful and doesn't use it
- If this happens a lot, the class hierarchy has been badly designed and needs to be dismantled and rebuilt.
- A bit of rejection here and there may not be serious
- Rejecting a **public** method is normally a serious matter
 - Polymorphic replacement is no longer possible. Also violates the Liskov principle.
- Push Down Method/Field can help here, or Replace Inheritance with Delegation.

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