Refactoring

http://sourcemaking.com/refactoring

Focus on the ones in Netbeans

Code Evolution

- Programs evolve and code is NOT STATIC
  - Code duplication
  - Outdated knowledge (now you know more)
    - Rethink earlier decisions and rework portions of the code
    - Customer changes
  - Performance
  - Clarifications for teammates

- Refactoring!

- In industry, it is common for refactoring not to be done due to time pressure
  - Fail to refactor now and there will be a far greater time investment to fix problems later on as size and dependencies increase
  - Code that needs refactoring can be viewed as a tumor or "growth"

Refactoring

- The process of rewriting a computer program
  - to improve its structure or readability
  - while explicitly preserving its external behavior
- A series of small behavior-preserving transformations
  - Each transformation (called a 'refactoring') does little
  - The system is also kept fully working after each refactoring
    - Reduces the chances that a system gets seriously broken during the restructuring
    - We can prove that after refactoring, behavior has not changed by rerunning our tests
- If not done regularly
  - Over time, as more and more code is written, system becomes harder to maintain and extend

Refactoring

- Refactoring does not fix bugs or add new functionality
  - Improves the understandability of the code
  - Changes code structure and design
    - e.g. eliminates duplication or optimize
  - Removes dead code

- Make it easier for human maintenance in the future
  - Adding new behavior to a program might be difficult with the program's current structure
  - Refactor it first to make it easy, and then add the new behavior

Refactoring

- Coined in analogy with the factorization of numbers and polynomials
  - $x2 - 1$ can be factored as $(x + 1)(x - 1)$
  - Revealing an internal structure that was previously not visible
    - such as the two roots at -1 and $+1$
  - Similarly, the change in visible code structure can often reveal the "hidden" internal structure of the original code
- Over 100 in total
  - 18 supported by eclipse (3.0)

Guidelines

- Make sure you have good tests before refactoring
  - Know quickly if your changes have broken system
- Don’t refactor and add/remove/change functionality at the same time
  - ***WHY?***
- *Refactor early and refactor often*

Refactoring

- Simple example:
  - Change a variable name into something more meaningful, such as from a single letter `i` to `interestRate`

- More complex examples
  - Eliminating duplicate code

Refactoring Recurring Code

- Eliminates duplicate code segments
  - Makes maintenance costly

- Consists of the following steps
  - Identifying recurring code segments
    - Same logic and often same exact code
    - **CAVEAT:** Not all code that looks alike is actually alike!
  - Capture this logic in a generic component defined ONCE
  - Restructure program so that every occurrence of the code segment is a reference to the generic component

- via
  - method invocation
  - inheritance
  - delegation

Refactoring *via Method Invocation*

- Class Computation
  - void method1(. . .) {
    - //…
    - computeStep1();
    - computeStep2();
    - computeStep3();
    - //..
    - }
  - void method2(. . .) {
    - //…
    - computeStep1();
    - computeStep2();
    - computeStep3();
    - //..
    - }
  - //..
  - }

- Class RefactoredComputation
  - void computeAll(. . .){
    - computeStep1();
    - computeStep2();
    - computeStep3();}
  - void method1(. . .) {
    - //…
    - computeAll();
    - //..
    - }
  - void method2(. . .) {
    - //…
    - computeStepAll();
    - //..
    - }
  - //..
  - }

via Method Invocation

- `Extract Method Refactoring`

- Effective only when
  - All methods that contain the recurring code segment belong to the same class
  - Each occurrence of the recurring code segment is contained within a single method

via Inheritance

- For recurring code segments in different classes
- class ComputationA{
  - void method1(…) {
    - //…
    - computeStep1();
    - computeStep2();
    - computeStep3();
    - //..}
  - //… }
- class ComputationB{
  - void method2(…) {
    - //…
    - computeStep1();
    - computeStep2();
    - computeStep3();
    - //..}
  - //… }

via Inheritance

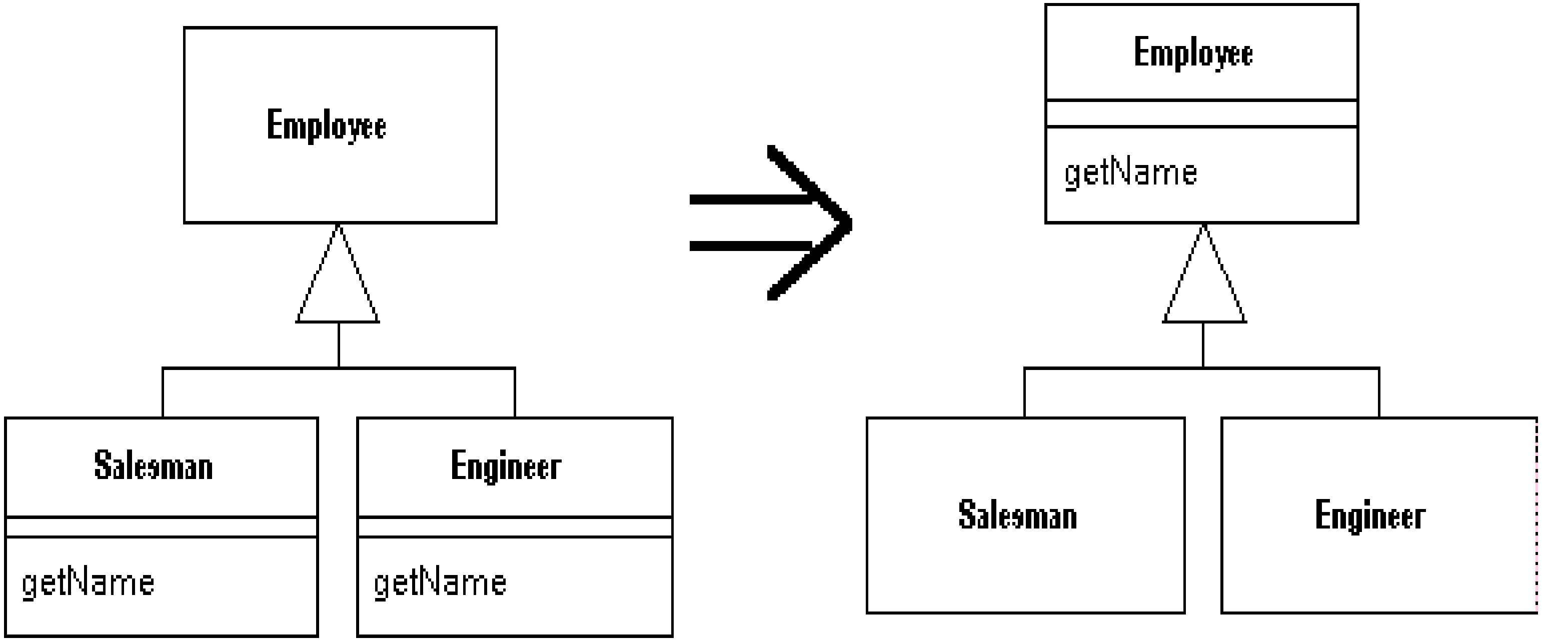
- Introduce a common superclass for `ComputationA` and `ComputationB`
- Place common code in a method in superclass
  - `class Common{`
    - `void computeAll( . . .) {`
      - `computeStep1();`
      - `computeStep2();`
      - `computeStep3();}`
    - `}`
- When extracting common code segments to a superclass, all fields involved in the computations must also be extracted and promoted
- ***Pull Up Method Refactoring***

via Inheritance

- class ComputationA extends Common{
  - void method1(…) {
    - //…
    - computeAll();
    - //..}
  - //… }

- class ComputationB extends Common{
  - void method2(…) {
    - //…
    - computeAll();
    - //..}
  - //… }

via Inheritance



via Delegation

- Done also for refactoring recurring code segments in different classes like *via inheritance*

- In cases where (at least one of) the involved classes already extend(s) other classes
  - Can’t extend any further

- Introduce a helper class

via Delegation

- class ComputationA extends SuperClass{
  - void method1(…) {
    - //…
    - computeStep1();
    - computeStep2();
    - computeStep3();
    - //..}
  - //… }

- class ComputationB{
  - void method2(…) {
    - //…
    - computeStep1();
    - computeStep2();
    - computeStep3();
    - //..}
  - //… }

via Delegation

- Place common code in a method in helper class
  - `class Helper{`
    - `void computeAll( . . .) {`
      - `computeStep1();`
      - `computeStep2();`
      - `computeStep3();}`
    - `}`
- Both classes need to contain references to the helper class

via Delegation

- class ComputationA extends SuperClass{
  - void method1(…) {
    - //…
    - **Helper helper = new Helper();**
    - **helper.computeAll();**
    - //..}
  - //… }

- class ComputationB {
  - void method2(…) {
    - //…
    - **Helper helper = new Helper();**
    - **helper.computeAll();**
    - //..}
  - //… }

Important Refactorings

- *Rename Method or Field*
- *Extract Method*
- *Pull Up Method or Field*
- *Push Down Method or Field*
- *Move Method or Field*
- *Encapsulate Field*
- *Decompose Conditional*
- *Replace Magic Number with Symbolic Constant*
- *100 or so more*

Bad Code Smells

- Duplicate Code
  - Number 1 enemy
  - Duplication in the same class or in different classes
- Long Methods
  - Methods should be short
  - Easier to understand and maintain
  - Do only what they are supposed to do
- Large Classes
  - A class trying to do too much
  - Too many instance variables (not very related to one another)
    - E.g. university person (student, faculty, staff, etc …)
- Long Parameter Lists
  - Pass enough to get everything you need
  - E.g. instead of passing all instance variables of an object, pass the object itself

Bad Code Smells

- Feature Envy
  - A method in a class seems more interested in a class other than the one it is actually in
  - Move Method to other class
- Comments
  - Don’t use them as deodorant
  - Thickly commented code implies that code is hard to understand and probably needs refactoring
- [http://www.cs.uu.nl/docs/vakken/mso/BadSmells.html]

Composing Methods

- Refactoring deals a lot with composing methods to package code properly
- Get rid of methods that are too long or do too much
  - A lot of their information gets buried by their complex logic
  - ***Extract Method***
  - *Replace Temp with Query*
  - *Remove Assignments to Parameters*

Extract Method

- You have a code fragment that can be grouped together
  - Reduce method size (Method is too long)
  - Clarity (Need comments to understand into purpose)
  - Eliminate redundancy (Code is duplicated in multiple methods)

- Turn the fragment into a method whose name explains the purpose of the method
  - shorter **well-named** methods
  - Can be used by other methods
  - Higher-level methods read more like a series of comments

```
void printOwing() {
    printBanner(); //print details
    System.out.println("name: " + _name);
    System.out.println("amount: " + getOutstanding());
}
```

Extract Method

- void printOwing() {
  - printBanner();
  - printDetails(getOutstanding());
- }
- void printDetails (double outstanding) {
  - System.out.println ("name: " + _name);
  - System.out.println ("amount " + outstanding);
- }

Extract Method

- Steps
  - Create a new method and name it after what it does
  - Copy extracted code from the source into the target
  - Scan the extracted method for references to any variables that are local in scope to the source method
    - ***These are local variables and parameters to the target method***
    - Temporary variables used only within the extracted code become temporary variables declared in target method
      - Remove from old code
    - Temporary variables that are read from the extracted code (used elsewhere) are passed into the target method as parameters
    - Check for local-scope variables modified by extracted code
      - One modified variable: treat extracted code as a query and assign the result to the variable concerned
      - More than one variable: can’t extract method as it stands
  - Replace extracted code in source method with a call to target method
  - Compile and test

Example: No Local Variables

```
void printOwing() {
    List listOfOrders = this.orders.elements();
    double outstanding = 0.0;
    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");
    // calculate outstanding
    for(Order order: listOfOrders)
        outstanding += order.getAmount();
    }
    //print details
    System.out.println ("name:" + this.name);
    System.out.println ("amount" + outstanding);
}
```

Example: No Local Variables

```
void printOwing() {
    List listOfOrders = this.orders.elements();
    double outstanding = 0.0;
    printBanner();
    // calculate outstanding
    for(Order order: listOfOrders)
        outstanding += order.getAmount();
    }
    //print details
    System.out.println ("name:" + this.name);
    System.out.println ("amount" + outstanding);
}
void printBanner() {
    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");
}
```

Example: Using Local Variables

```
void printOwing() {
    List listOfOrders = this.orders.elements();
    double outstanding = 0.0;
    printBanner();
    // calculate outstanding
    for(Order order: listOfOrders)
        outstanding += order.getAmount();
    }
    //print details
    printDetails(outstanding);
}
void printDetails (double outstanding) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Example: Reassigning a Local Variable

```
void printOwing() {
    List listOfOrders=this.orders.elements();
    double outstanding = 0.0;
    printBanner();
    // calculate outstanding
    for(Order order: listOfOrders)
        outstanding += order.getAmount();
    }
    //print details
    printDetails(outstanding);
}
```

```
double getOutstanding() {
    List listOfOrders =
    this.orders.elements();
    double outstanding = 0.0;
    for(Order order: listOfOrders)
        outstanding +=
        order.getAmount();
    return outstanding;
}
```

```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}
```

Example: Reassigning a Local Variable

- The `List` variable is used only in the extracted code
  - We can move it entirely within the new method
- The `outstanding` variable is used in both places
  - We need to return it from the extracted method
- The `outstanding` variable is initialized only to an obvious initial value
  - We can initialize it only within the extracted method
  - If something more involved happens to the variable, we have to pass in the previous value as a parameter

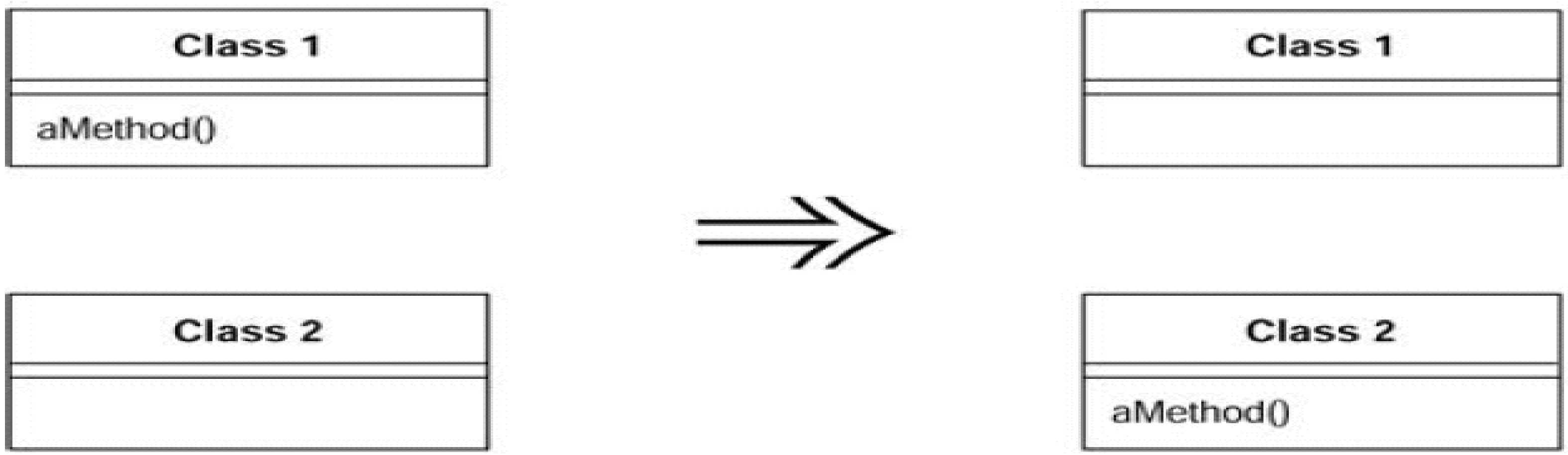
Moving Features Between Objects

- One of the most fundamental decision in object-oriented design is deciding where to put responsibilities
  - "*I've been working with objects for more than a decade, but I still never get it right the first time. That used to bother me, but now I realize that I can use refactoring to change my mind in these cases.*"
  - Martin Fowler

- ***Move Method***
- ***Move Field***
- *Extract Class*

Move Method

- A method is using more features or is used by more methods of another class than the class on which it is defined
  - *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*



Move Method

- Examine all class attributes used by the source method that are defined on the source class and consider whether they also should be moved
  - *If the attribute is used only by the method you are about to move, you might as well move it*
  - *If the attribute is used by other methods, consider moving them as well*
- Declare the method in the target class
  - *You may choose to use a different name, one that makes more sense in the target class*
- Copy the code from the source method to the target
  - Adjust the method to make it work in its new home
  - *If the method uses its source, you need to determine how to reference the source object from the target method*
    - *If there is no mechanism in the target class, pass the source object reference to the new method as a parameter*
- Compile the target class

Move Method

- Determine how to reference the correct target object from the source
  - *There may be an existing field or method that will give you the target*
  - *If not, see whether you can easily create a method that will do so*
  - *If not, you need to create a new field in the source that can store the target*
- Turn the source method into a delegating method
- Compile and test
- Decide whether to remove the source method or retain it as a delegating method
  - *Leaving the source as a delegating method is easier if you have many references*
- If you remove the source method, replace all the references with references to the target method
  - *You can compile and test after changing each reference, although it is usually easier to change all references with one search and replace*
- Compile and test

Move Method

```
class Account{
    private AccountType type;

    private int daysOverdrawn;

    double overdraftCharge() {
        if (type.isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7)
                result += (daysOverdrawn - 7) * 0.85;
            return result;}
        else
            return daysOverdrawn * 1.75;
    }

    double annualBankCharge() {
        double result = 25;
        if (daysOverdrawn > 0)
            result += overdraftCharge();
        return result;
    }

}
```

```
class AccountType{...}
```

Move Method

- Imagine
  - Several new account types
  - Each has its own rule for computing the overdraft charge

- Thus, we need to move the `overdraftCharge` method over to the `AccountType` class

- Start by looking at the features that the `overdraftCharge` method uses and consider whether to move a batch of methods together

- We need the `daysOverdrawn` field to remain on the account class
  - Will vary with individual accounts

- Copy the method body over to the account type and get it to fit

Move Method

- When we need to use a feature of the source class we can do one of the following:
  - (1) move this feature to the target class as well,
  - (2) pass the source object as a parameter to the method
  - (3) create or use a reference from the target class to the source

Move Method

- class AccountType...
  - double overdraftCharge(int daysOverdrawn){
    - if (**isPremium()**) {
      - double result = 10;
      - if (**daysOverdrawn**>7)
        - result+=(**daysOverdrawn**-7)*0.85;
      - return result;}
    - else
      - return daysOverdrawn * 1.75;
  - }

Move Method

```
class Account...
    private AccountType type;

    private int daysOverdrawn;

    double overdraftCharge() {
        return type.overdraftCharge(daysOverdrawn);}

    double annualBankCharge() {
        double result = 4.5;
        if (daysOverdrawn > 0)
            result += overdraftCharge();
        return result;
    }
```

- We can leave things like this, or we can remove the method in the source class
  - To remove the method I need to find all callers of the method and redirect them to call the method in account type:

Move Method

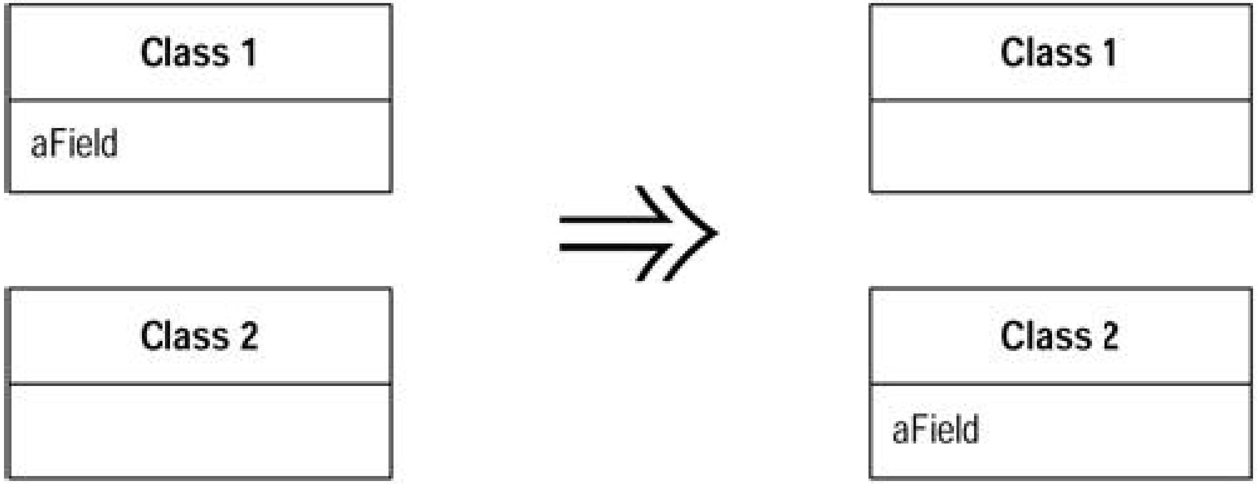
```
class Account...
    private AccountType type;
    private int daysOverdrawn;

    double annualBankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0)
            result += type.overdraftCharge(daysOverdrawn);
        return result;
    }
```

- Once we’ve replaced all the callers, we can remove the method declaration in account

Move Field

- A field is, or will be, used by another class more than the class on which it is defined
- *Create a new field in the target class, and change all its users*



Move Field

- As the system develops, we find the need for new classes and the need to shuffle responsibilities around
- A design decision that is reasonable and correct one week can become incorrect in another
- Consider moving a field if you see more methods on another class using the field than the class itself
    - This usage may be indirect, through getting and setting methods
    - We may choose to move the methods; this decision is based on interface
        - But if the methods seem sensible where they are, we move the field

Move Field

- If field is public, make it private and create a setter and a getter
- Compile and test
- Create a field in the target class with a getter and setter methods
- Compile the target class
- Determine how to reference the target object from the source
  - *An existing field or method may give you the target*
  - *If not, see whether you can easily create a method that will do so*
  - *If not, you may need to create a new field in the source that can store the target*

Move Field

- `class Account...`
  - `private AccountType type;`
  - `private double interestRate;`
  - `double interestForAmountDays(double amount, int days) {`
    - `return interestRate * amount * days / 365;`
  - `}`
- Move the interest rate field to the account type
- Assume there are several methods with that reference, of which `interestForAmountDays` is one example
- `class AccountType...`
  - `private double interestRate;`
  - `void setInterestRate (double arg) {`
    - `interestRate = arg;`
  - `}`
  - `double getInterestRate () {`
    - `return interestRate;`
  - `}`

Move Field

- Redirect the methods from the account class to use the account type and remove the interest rate field in the account
- `private double interestRate;`
  - `double interestForAmountDays (double amount, int days){`
    - `return` **`type.getInterestRate()`** `* amount * days / 365;`
  - `}`

Organizing Data

- Refactorings that make working with data easier
  - *Replace Array with Object*
  - *Change Unidirectional Association to Bidirectional*
  - *Replace Magic Number with Symbolic Constant*
  - ***Encapsulate Field***

Encapsulate Field

- There is a public field
  - *Make it private and provide accessors*
- `public String name`
- ➔
  - `private String name;`
  - `public String getName()`
    - `{return name;}`
  - `public void setName(String arg)`
    - `{name = arg;}`

Encapsulate Field

- One of the principal tenets of object orientation is encapsulation, or data hiding
  - Should never make your data public
  - When you make data public, other objects can change and access data values without the owning object's knowing about it
  - This separates data from behavior

Encapsulate Field

- Create getting and setting methods for the field

- Find all clients outside the class that reference the field
  - If the client uses the value, replace the reference with a call to the getting method
  - If the client changes the value, replace the reference with a call to the setting method

- Compile and test after each change

- Once all clients are changed, declare the field as private

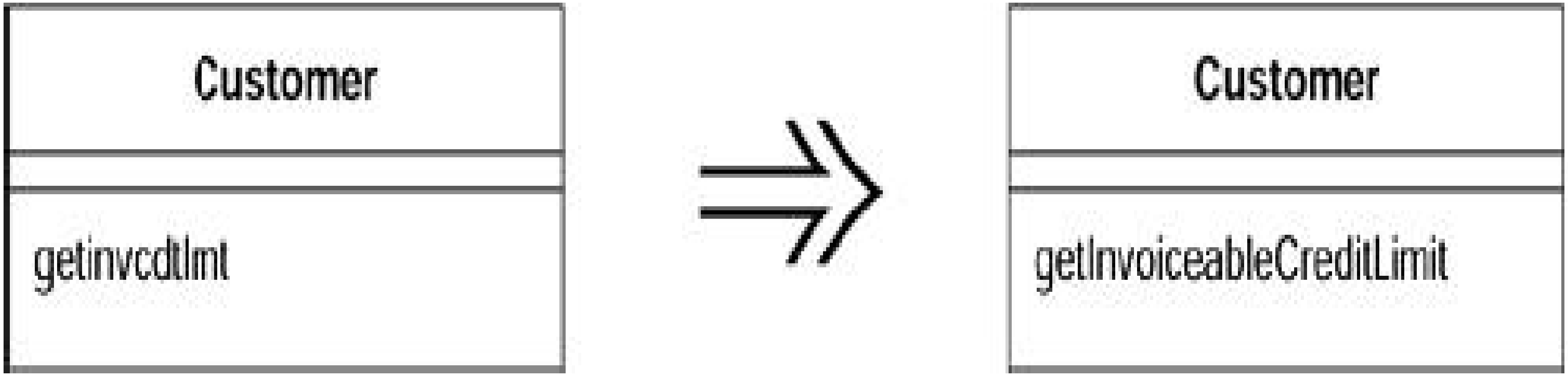
- Compile and test

Making Method Calls Simpler

- Objects are all about interfaces
- Coming up with interfaces that are easy to understand and use is a key skill in developing good object-oriented software
- We explore refactorings that make interfaces more straightforward
  - ***Rename Method***
  - *Add Parameter*
  - *Parameterize Method*
  - *Preserve Whole Object*
  - *Hide Method*
  - *Replace Error Code with Exception*

Rename Method

- The name of a method does not reveal its purpose
- *Change the name of the method*



Rename Method

- Methods should be named in a way that communicates their intention
- A good way to do this is to think what the comment for the method would be and turn that comment into the name of the method
- Sometimes you won't get your names right the first time
  - May well be tempted to leave it—after all it's only a name
- If you see a badly named method, it is imperative that you change it
  - Remember your code is for a human first and a computer second
- Good naming is a skill that requires practice; improving this skill is the key to being a truly skillful programmer

Rename Method

- Steps:
  - Check to see whether the method signature is implemented by a superclass or subclass
    - If it is, perform these steps for each implementation
  - Declare a new method with the new name
  - Copy the old body of code over to the new name and make any alterations to fit
  - Compile
  - Change the body of the old method so that it calls the new one
    - *If you only have a few references, you can reasonably skip this step*
  - Compile and test
  - Find all references to the old method name and change them to refer to the new one
  - Compile and test after each change
  - Remove the old method
    - *If the old method is part of the interface and you cannot remove it, leave it in place and mark it as deprecated*
  - Compile and test

Example

- `public String getTelephoneNumber() {`
  - `return ("(" + officeAreaCode + ") " + officeNumber);`
- `}`
- Rename the method to `getOfficeTelephoneNumber`
- `class Person...`
  - `public String getTelephoneNumber(){`
    - `return getOfficeTelephoneNumber();`
  - `}`
  - `public String getOfficeTelephoneNumber() {`
    - `return ("(" + officeAreaCode + ") " + officeNumber);`
  - `}`
- Find the callers of the old method, and switch them to call the new one

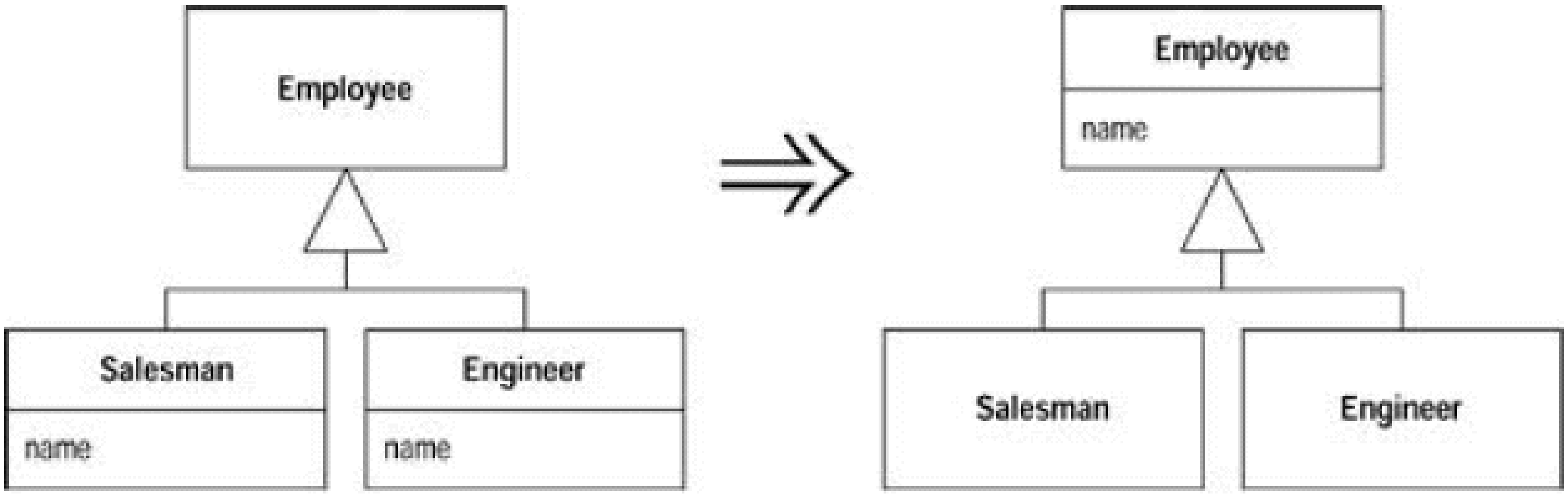
Dealing with Generalization

- Mostly dealing with moving methods around a hierarchy of inheritance

- *`Pull Up Field`*
- *`Pull Up Method`*
- *`Push Down Method`*
- *`Push Down Field`*

Pull Up Field

- Two subclasses have the same field
- *Move the field to the superclass*

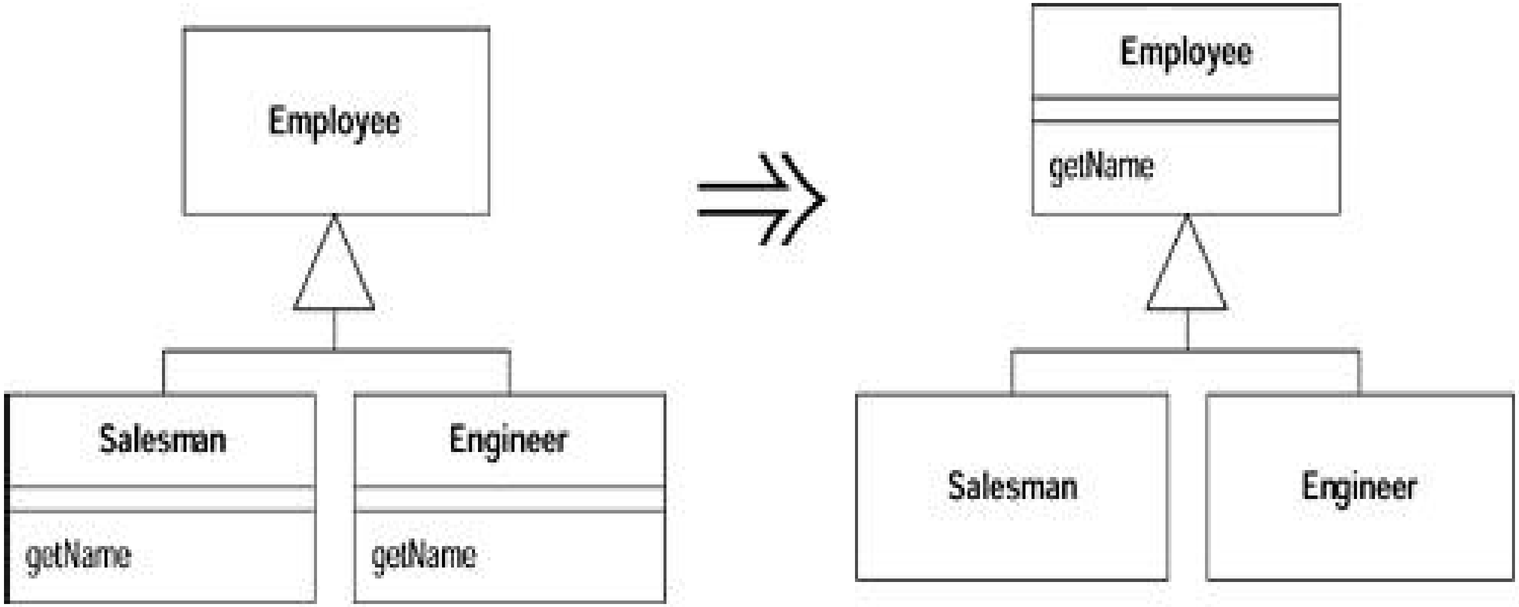


Pull Up Field

- Steps:
  - Inspect all uses of the candidate fields to ensure they are used in the same way
  - If fields do not have same name, rename the fields so that they have the name you want to use for the superclass field
  - Compile and test
  - Create a new field in the superclass
    - *If the fields are private, you will need to* `protect` *the superclass field so that the subclasses can refer to it*
  - Delete the subclass fields
  - Compile and test
  - Consider using encapsulation the new field

Pull Up Method

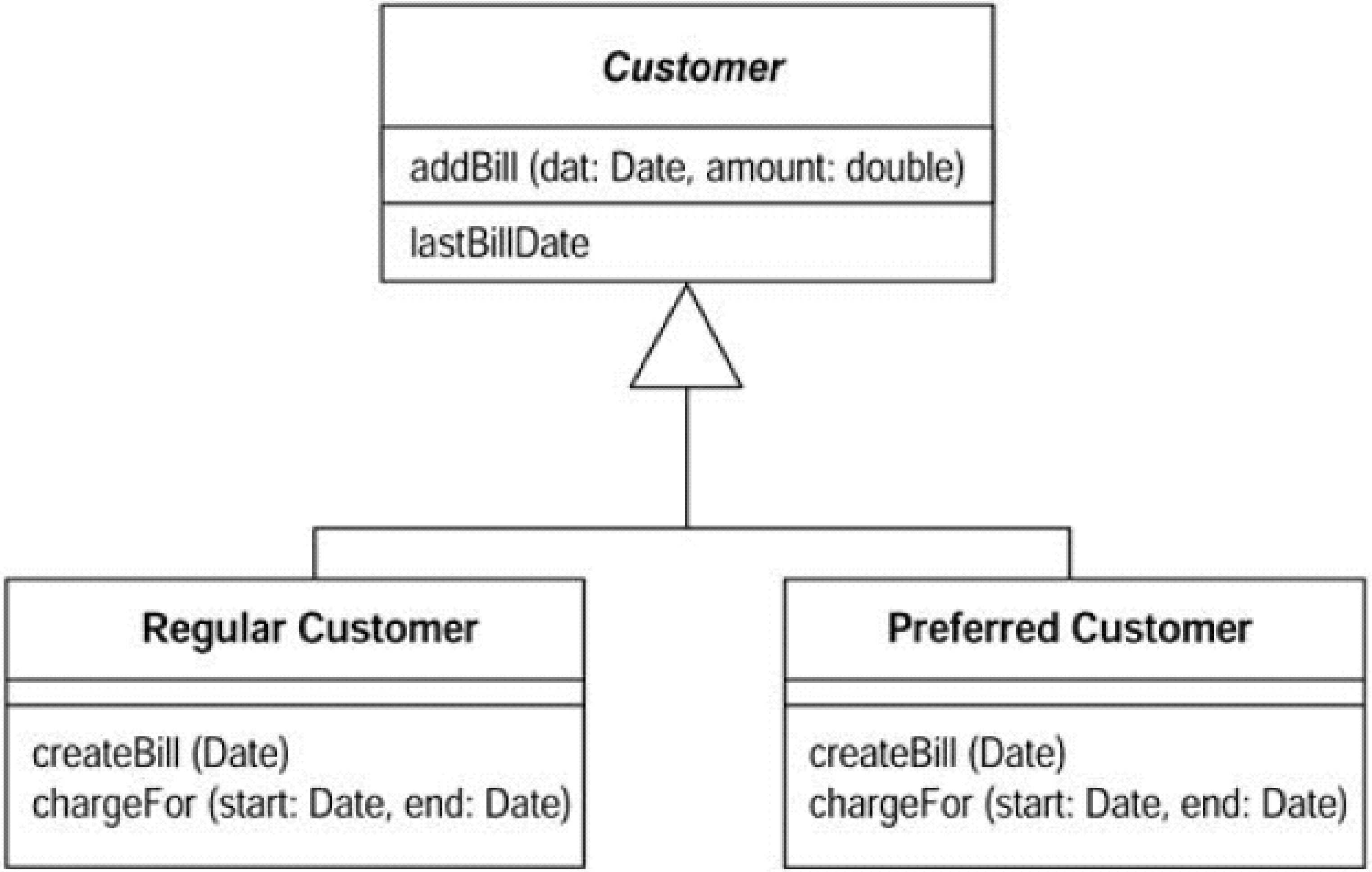
- You have methods with identical results on subclasses
- *Move them to the superclass*



Pull Up Method

- Steps:
  - Inspect the methods to ensure they are identical
  - If the methods have different signatures, change the signatures to the one you want to use in the superclass
  - Create a new method in the superclass, copy the body of one of the methods to it, adjust and compile
    - *If the method calls another method that is present on both subclasses but not the superclass, declare an abstract method on the superclass*
    - *If the method uses a subclass field, use Pull Up Field*
  - Delete one subclass method
  - Compile and test
  - Keep deleting subclass methods and testing until only the superclass method remains
  - Take a look at the callers of this method to see whether you can change a required type to the superclass

Example



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

Example

- The createBill method is identical for each class:

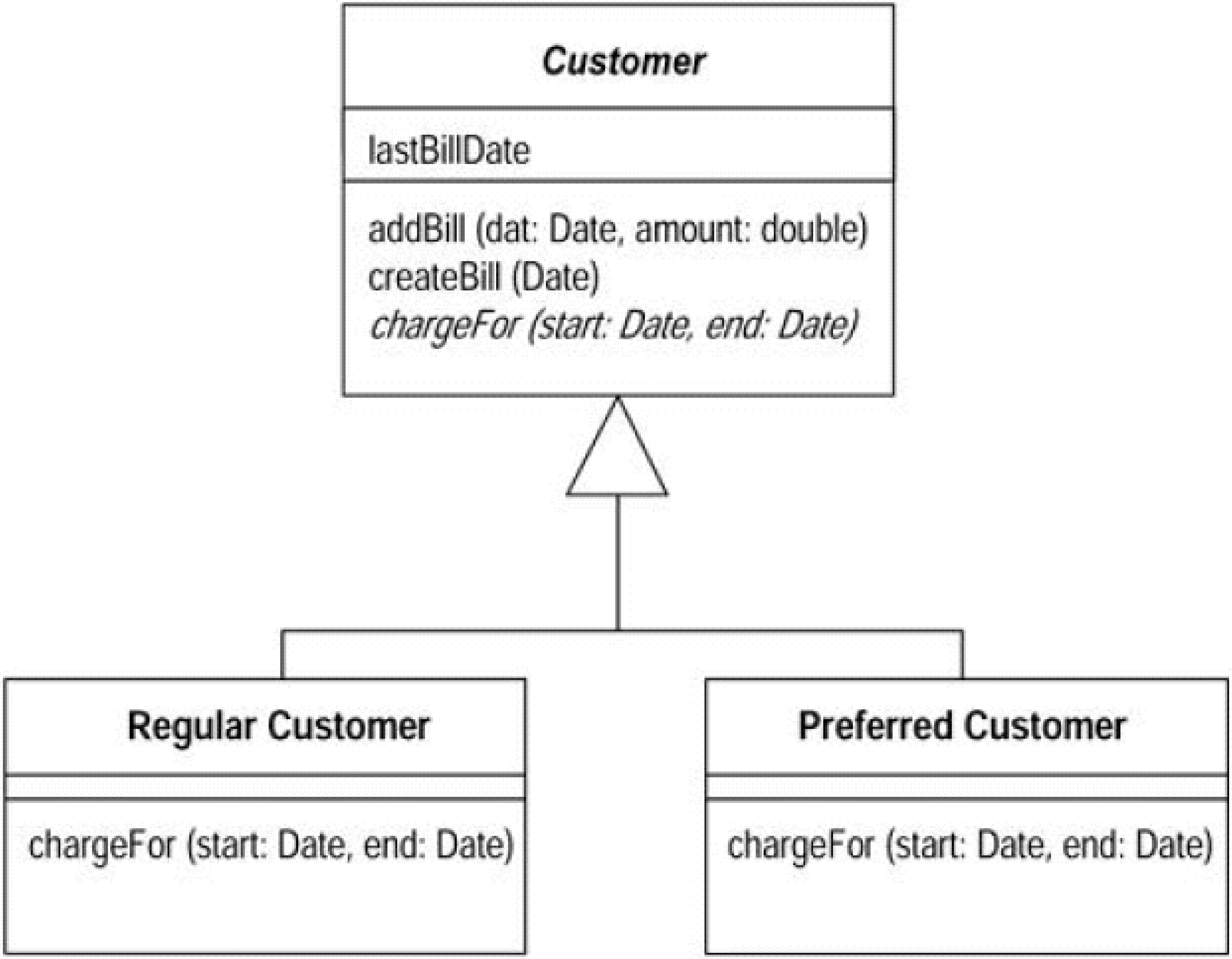
```
void createBill (date Date){
    double chargeAmount = chargeFor (lastBillDate, date);
    addBill (date, charge);
}
```

- Assume we can't move the method up into the superclass, because chargeFor is different on each subclass
- First declare it on the superclass as abstract:

```
class Customer...
    abstract double chargeFor(date start,date end)
```

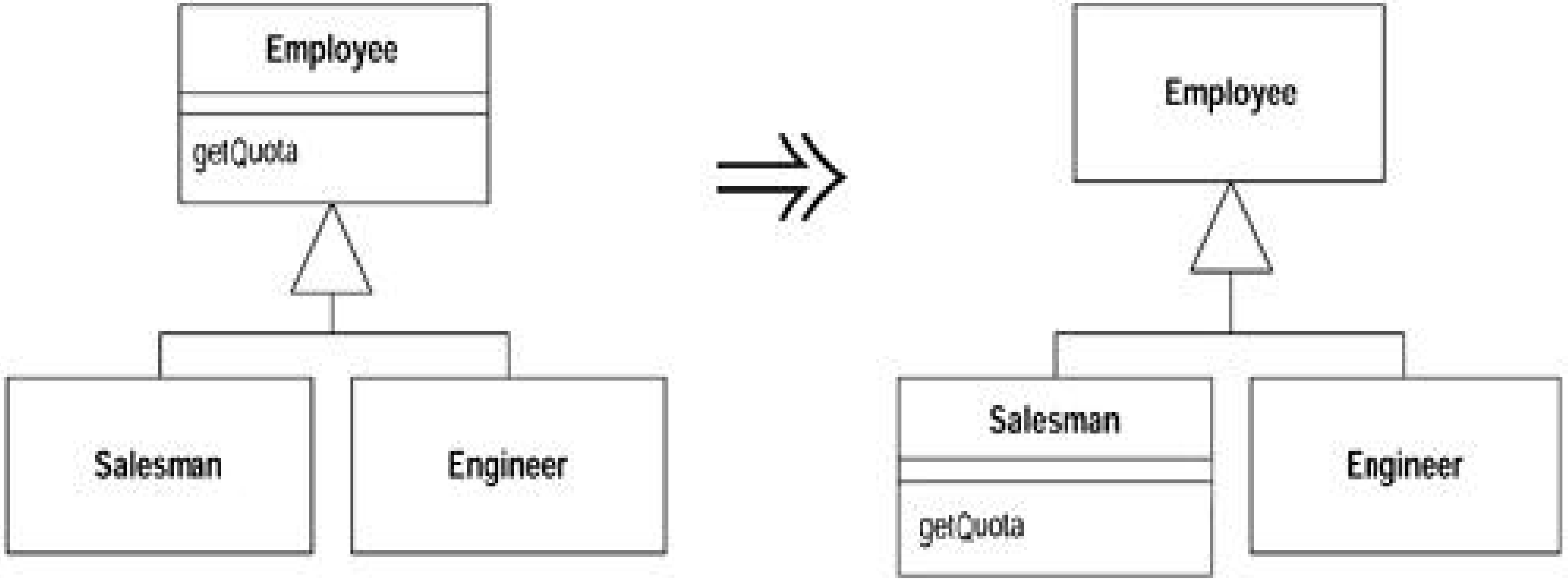
Example

- Copy `createBill` from one of the subclasses
- Compile with that in place and then remove the `createBill` method from one of the subclasses, compile, and test
- Then remove it from the other, compile, and test


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

- Behavior on a superclass is relevant only for some of its subclasses
- *Move it to those subclasses*

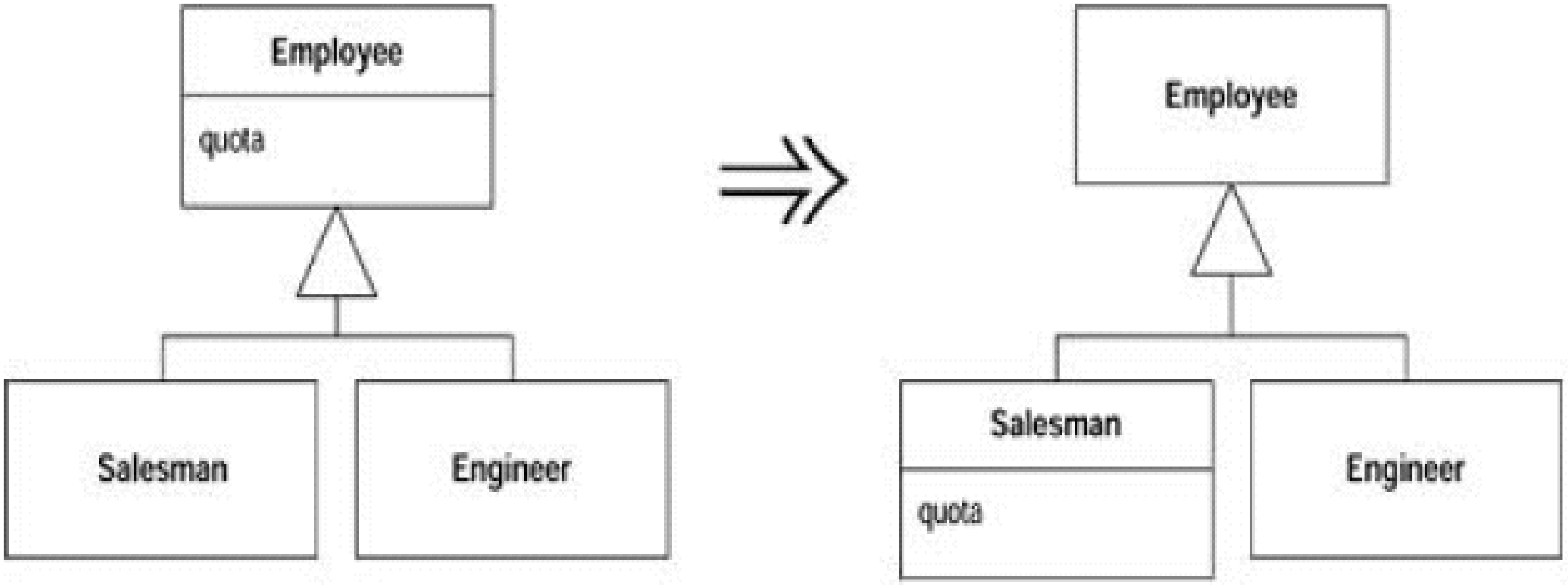


Push Down Method

- Steps:
  - Declare a method in all subclasses and copy the body into each subclass
    - *You may need to declare fields as protected for the method to access them*
    - *Usually you do this if you intend to push down the field later*
    - *Otherwise use an accessor on the superclass*
    - *If this accessor is not public, you need to declare it as protected.*
  - Remove method from superclass
  - Compile and test
  - Remove the method from each subclass that does't need it
  - Compile and test

Push Down Method

- A field is used only by some subclasses
- *Move the field to those subclasses*



Push Down Field

- The opposite of *Pull Up Field*
- Steps
  - Declare the field in all subclasses
  - Remove the field from the superclass
  - Compile and test
  - Remove the field from all subclasses that don't need it
  - Compile and test