Refactoring

Lecturer: Adel Vahdati

Refactoring: Definition

- A change made to the internal structure of software to make it
  - easier to understand, and
  - cheaper to modify.
- The observable behavior of the software should not be changed.

Refactoring: Why?

- Refactoring Improves the Design of Software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster

Refactoring: When?

- Refactor the third time you do something similar (The Rule of Three)
- Refactor When You Add Function
- Refactor When You Need to Fix a Bug
- Refactor As You Do a Code Review

Symptoms of Bad Code

- **1. Duplicated Code**
- **2. Long Method**
- **3. Large Class**
- **4. Long Parameter List**
- **5. Divergent Change:** When one class is commonly changed in different ways for different reasons.
- **6. Shotgun Surgery:** When every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.
- **7. Feature Envy:** A method that seems more interested in a class other than the one it actually is in.
- **8. Data Clumps:** Bunches of data that regularly appear together.

Symptoms of Bad Code (2)

- **9. Primitive Obsession:** Excessive use of primitives, due to reluctance to use small objects for small tasks.
- **10. Switch Statements**
- **11. Parallel Inheritance Hierarchies:** Where every time you make a subclass of one class, you also have to make a subclass of another.
- **12. Lazy Class:** A class that isn't doing enough to justify its maintenance.
- **13. Speculative Generality:** Classes and features have been added just because a need for them may arise someday.
- **14. Temporary Field:** An instance variable that is set only in certain circumstances.
- **15. Message Chains:** Transitive visibility chains.

Symptoms of Bad Code (3)

- **16. Middle Man:** Excessive delegation.
- **17. Inappropriate Intimacy:** Excessive interaction and coupling.
- **18. Alternative Classes with Different Interfaces:** Classes that do the same thing but have different interfaces for what they do.
- **19. Incomplete Library Class**
- **20. Data Class:** Classes that have fields, getting and setting methods for the fields, and nothing else.
- **21. Refused Bequest:** When subclasses do not fulfill the commitments of their superclasses.
- **22. Comments:** When comments are used to compensate for bad code.

Refactoring Patterns: Categories

- **Composing Methods:** Packaging code properly
- **Moving Features Between Objects:** Reassigning responsibilities
- **Organizing Data:** Making data easier to work with
- **Simplifying Conditional Expressions:** Making conditional logic less error-prone
- **Making Method Calls Simpler:** Making interfaces easy to understand and use
- **Dealing with Generalization:** Moving features around a hierarchy of inheritance
- **Big Refactorings:** Large-scale changes to code

Composing Methods: Extract Method

- You have a code fragment that can be grouped together.
- Turn the fragment into a method whose name explains the purpose of the method.

```
void printOwing() {
        printBanner();

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

```
void printOwing() {
        printBanner();
        printDetails(getOutstanding());
}

void printDetails (double outstanding) {
        System.out.println ("name:       " + _name);
        System.out.println ("amount     " + outstanding);
}
```

Composing Methods: Inline Method

- A method's body is just as clear as its name.
- Put the method's body into the body of its callers and remove the method.

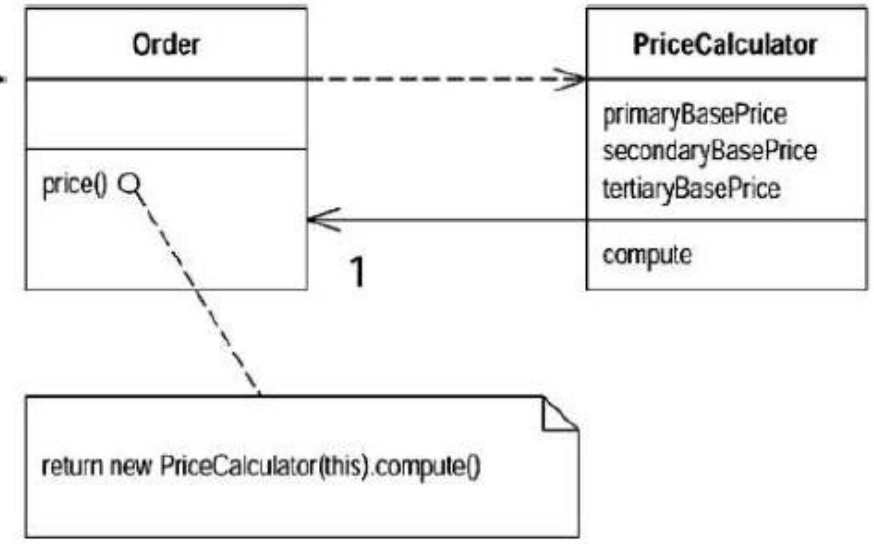
```
int getRating() {
        return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
        return _numberOfLateDeliveries > 5;
}
```

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

Composing Methods: Replace Method with Method Object

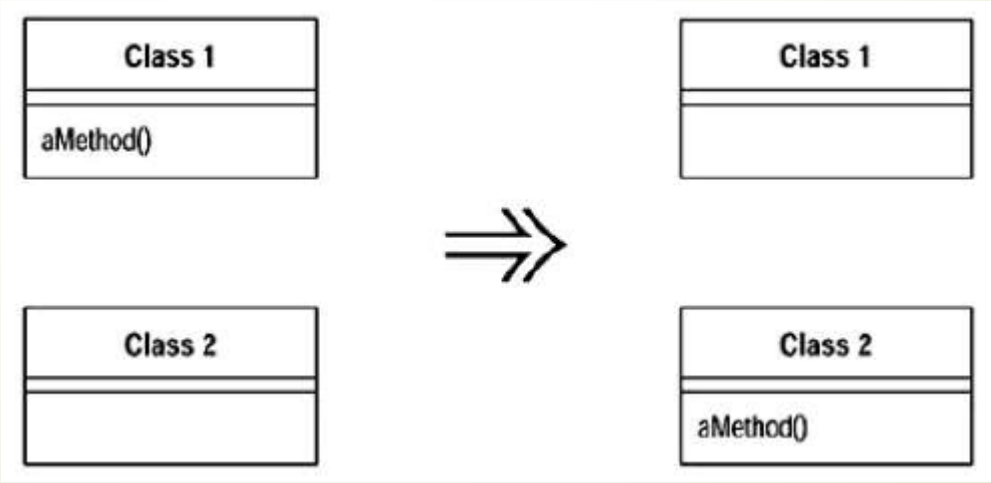
- You have a long method that uses local variables in such a way that you cannot apply Extract Method.
- Turn the method into an object so that all the local variables become fields on that object. It can then be decomposed into other methods on the same object.

```
class Order...
        double price() {
                double primaryBasePrice;
                double secondaryBasePrice;
                double tertiaryBasePrice;
                // long computation;
                ...
        }
```



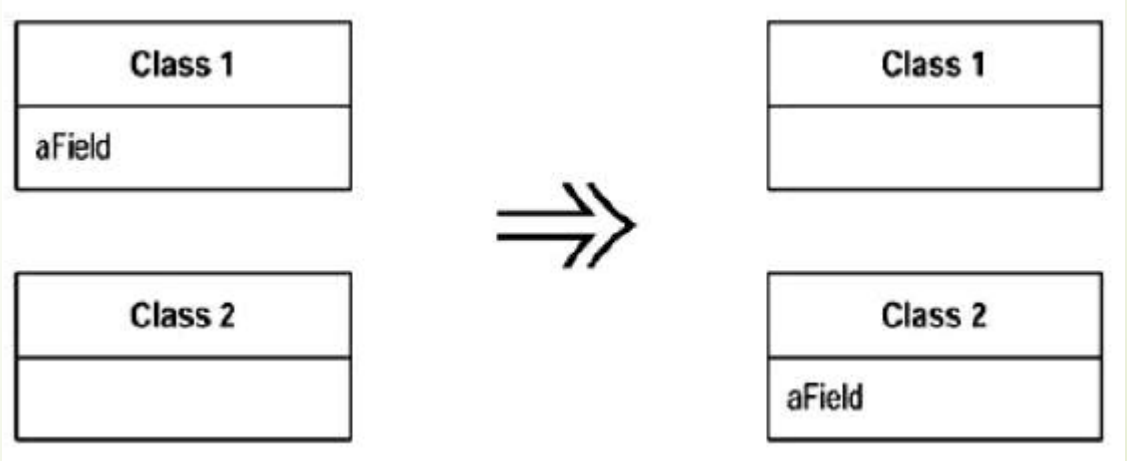
Moving Features Between Objects: Move Method

- A method is, or will be, using or used by more features 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.



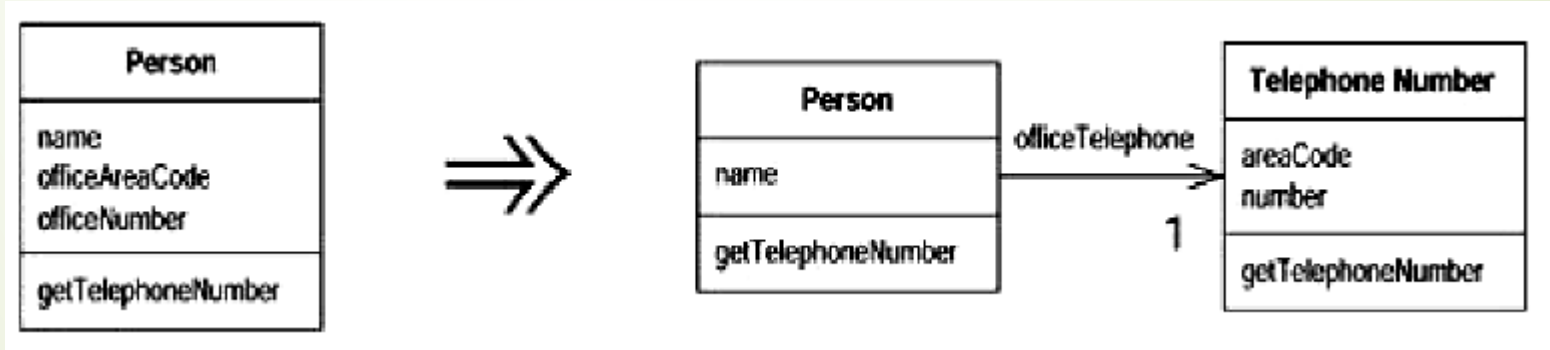
Moving Features Between Objects: Move Field

- 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.



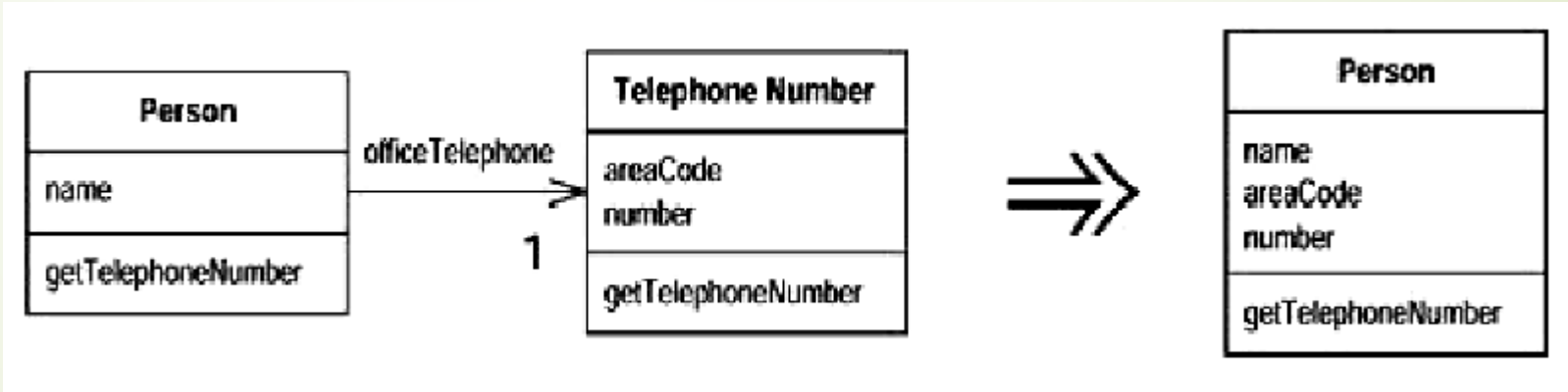
Moving Features Between Objects: Extract Class

- You have one class doing work that should be done by two.
- Create a new class and move the relevant fields and methods from the old class into the new class.



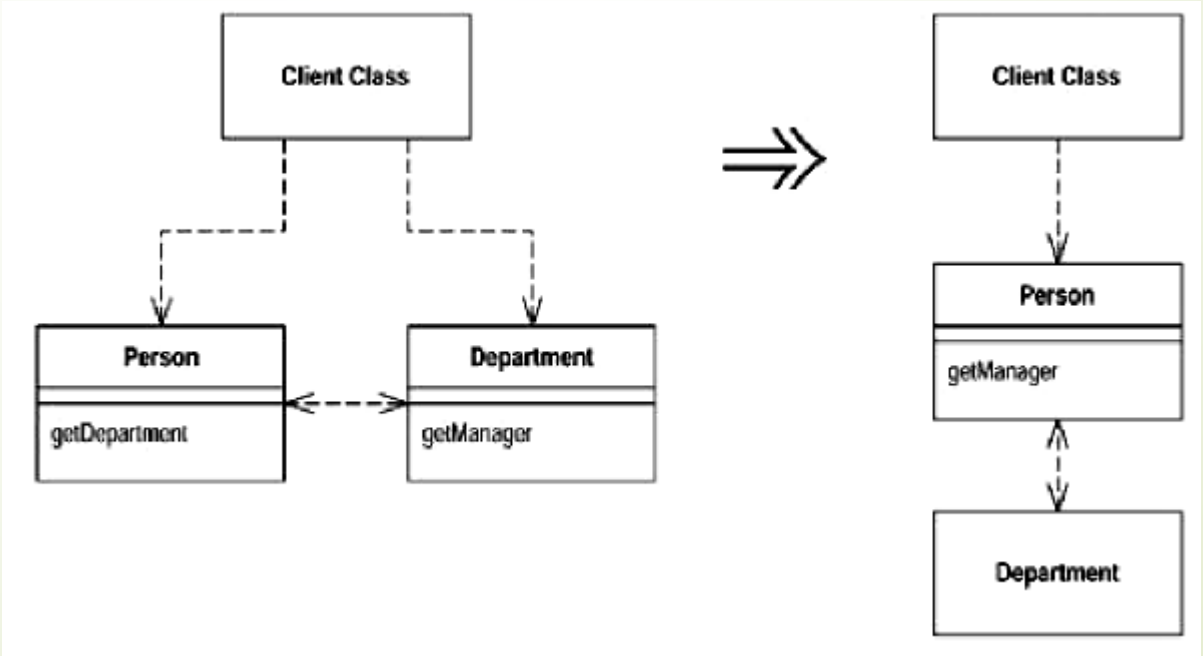
Moving Features Between Objects: Inline Class

- A class isn't doing very much.
- Move all its features into another class and delete it.



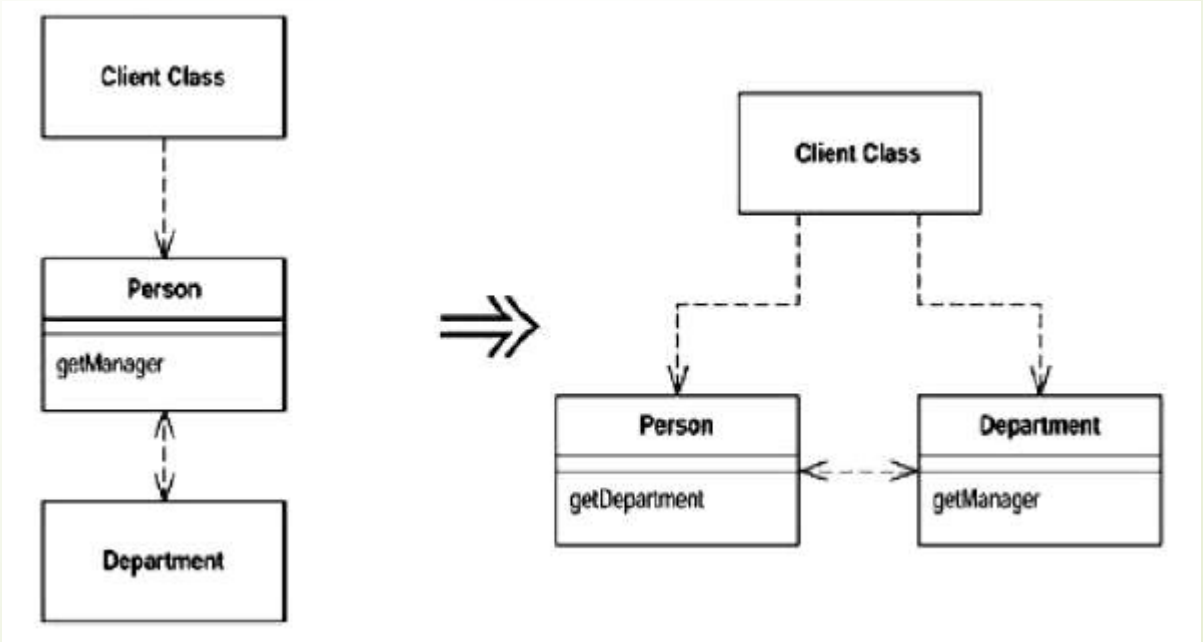
Moving Features Between Objects: Hide Delegate

- A client is calling a delegate class of an object.
- Create methods on the server to hide the delegate.



Moving Features Between Objects: Remove Middle Man

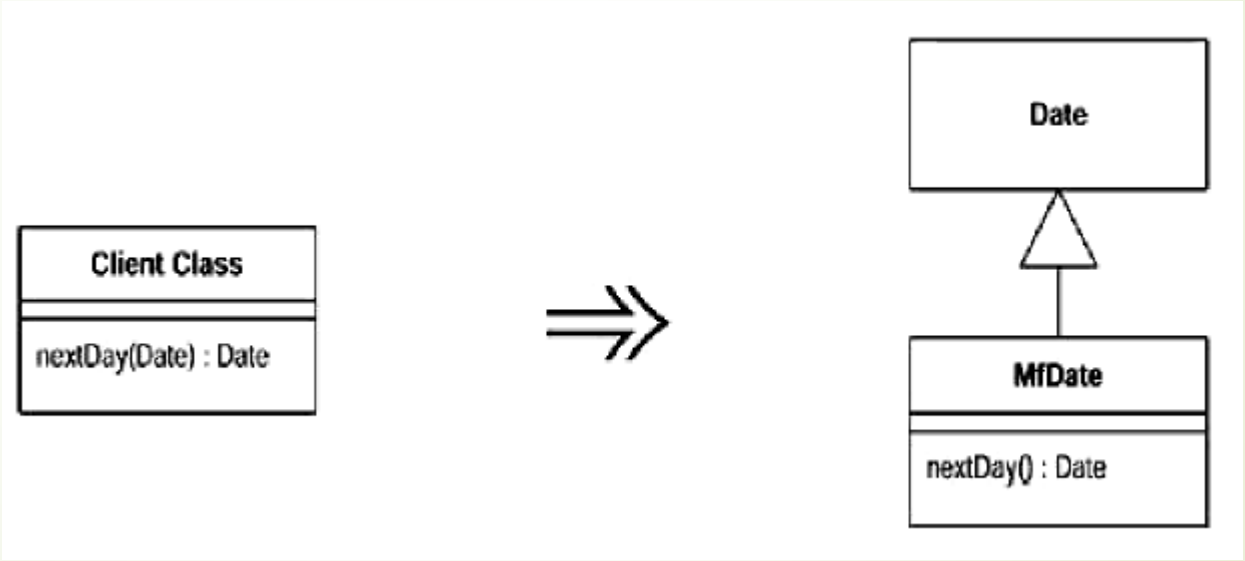
- A class is doing too much simple delegation.
- Get the client to call the delegate directly.



Moving Features Between Objects: Introduce Method/Class

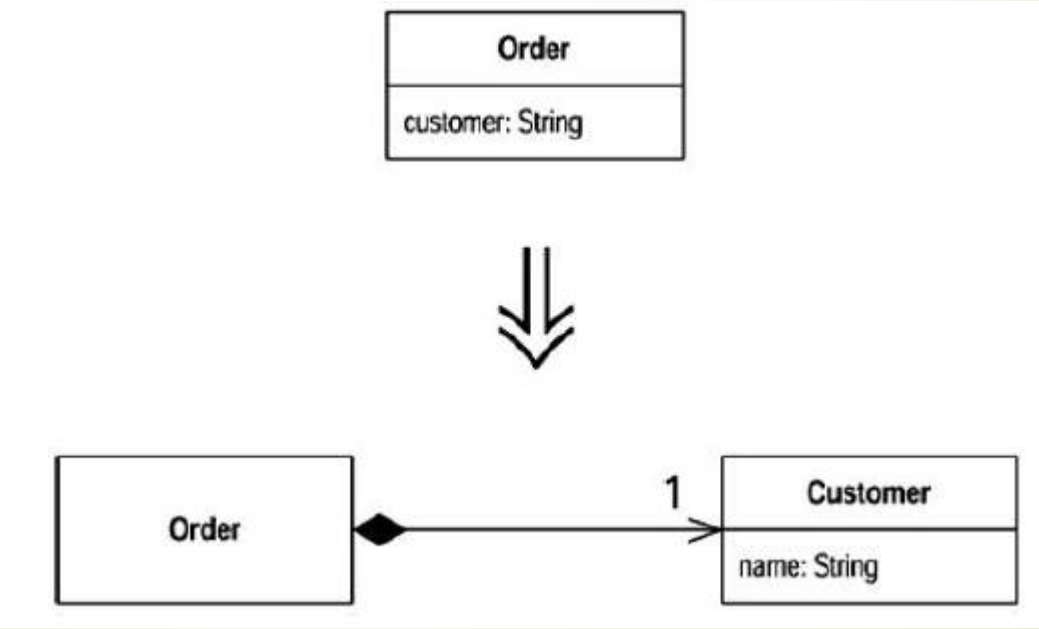
- **Introduce Foreign Method**
  - A server class you are using needs an additional method, but you can't modify the class.
  - Create a method in the client class with an instance of the server class as its first argument.
- **Introduce Local Extension**
  - A server class you are using needs several additional methods, but you can't modify the class.
  - Create a new class that contains these extra methods. Make this extension class a subclass or a wrapper of the original.

Moving Features Between Objects: Introduce Local Extension



Organizing Data: Replace Data Value with Object

- You have a data item that needs additional data or behavior.
- Turn the data item into an object.



Organizing Data: Replace Array with Object

- You have an array in which certain elements mean different things.
- Replace the array with an object that has a field for each element.

```
String[] row = new String[3];
row [0] = "Liverpool";
row [1] = "15";
```

```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

Organizing Data: Encapsulate Field

- There is a public field.
- Make it private and provide accessors.

```
public String _name
```

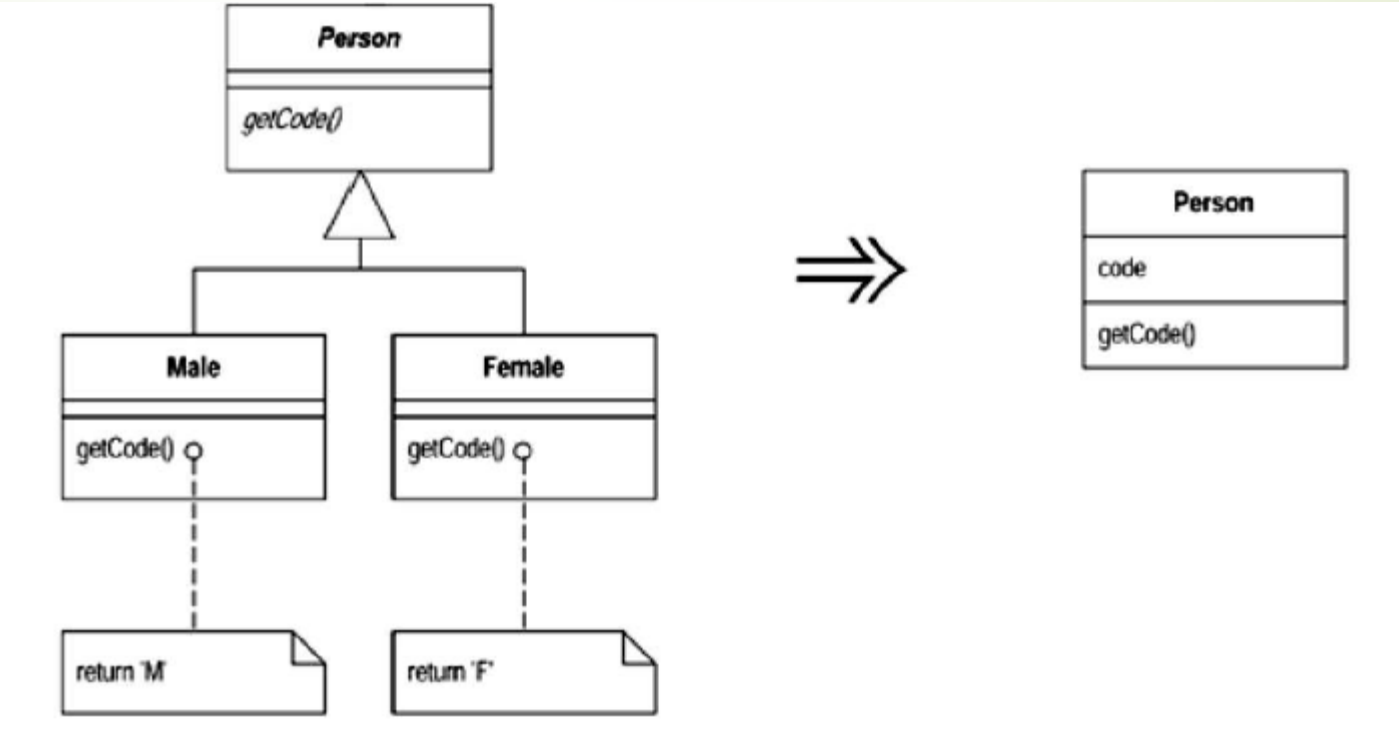
```
private String _name;
public String getName() {return _name;}
public void setName(String arg) {_name = arg;}
```

Organizing Data: Encapsulate Collection

- A method returns a collection.
- Make it return a read-only view and provide add/remove methods.

Organizing Data: Replace Subclass with Fields

- You have subclasses that vary only in methods that return constant data.
- Change the methods to superclass fields and eliminate the subclasses.



Simplifying Conditional Expressions: Decompose Conditional

- You have a complicated conditional (if-then-else) statement.
- Extract methods from the condition, then part, and else parts.

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

```
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge (quantity);
```

Simplifying Conditional Expressions: Consolidate Conditional Expression

- You have a sequence of conditional tests with the same result.
- Combine them into a single conditional expression and extract it.

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    // compute the disability amount
```

```
double disabilityAmount() {
    if (isNotEligableForDisability()) return 0;
    // compute the disability amount
```

Simplifying Conditional Expressions: Replace Nested Conditional with Guards Clauses

- A method has conditional behavior that does not make clear the normal path of execution.
- Use guard clauses for all the special cases.

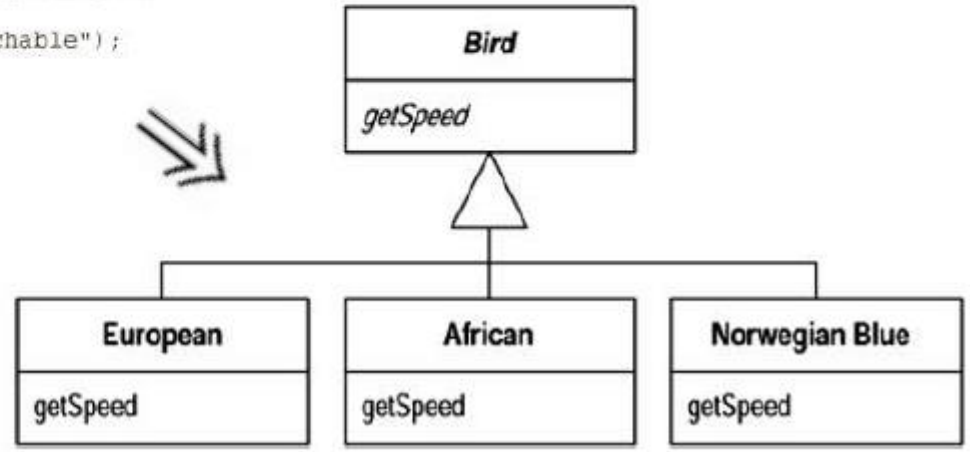
```
double getPayAmount() {
  double result;
  if (_isDead) result = deadAmount();
  else {
      if (_isSeparated) result = separatedAmount();
      else {
          if (_isRetired) result = retiredAmount();
          else result = normalPayAmount();
      };
  }
return result;
};
```

```
double getPayAmount() {
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  return normalPayAmount();
};
```

Simplifying Conditional Expressions: Replace Conditional with Polymorphism

- You have a conditional that chooses different behavior depending on the type of an object.
- Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.

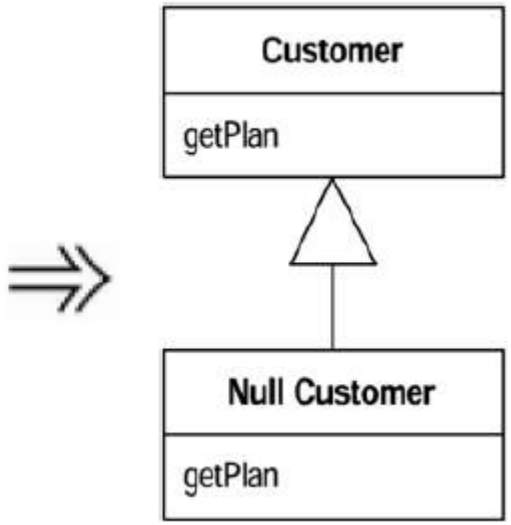
```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
           return getBaseSpeed();
        case AFRICAN:
           return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEGIAN_BLUE:
           return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw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



Simplifying Conditional Expressions: Introduce Null Object

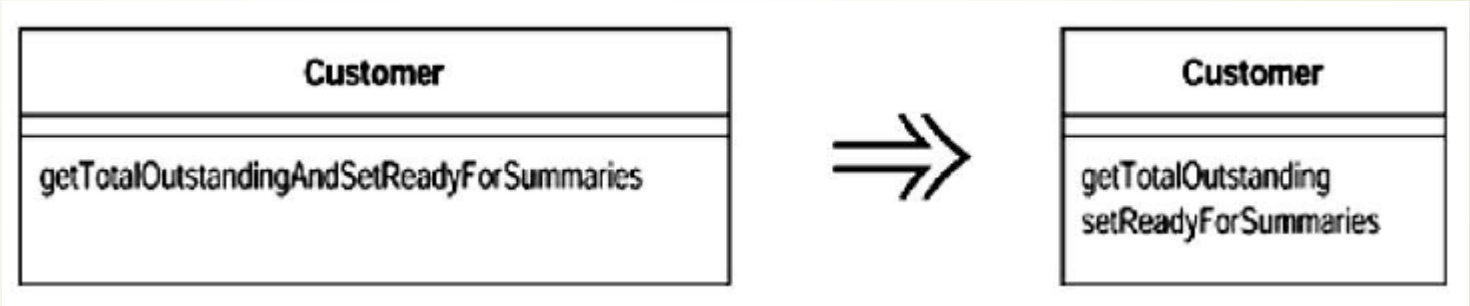
- You have repeated checks for a null value.
- Replace the null value with a null object.

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```



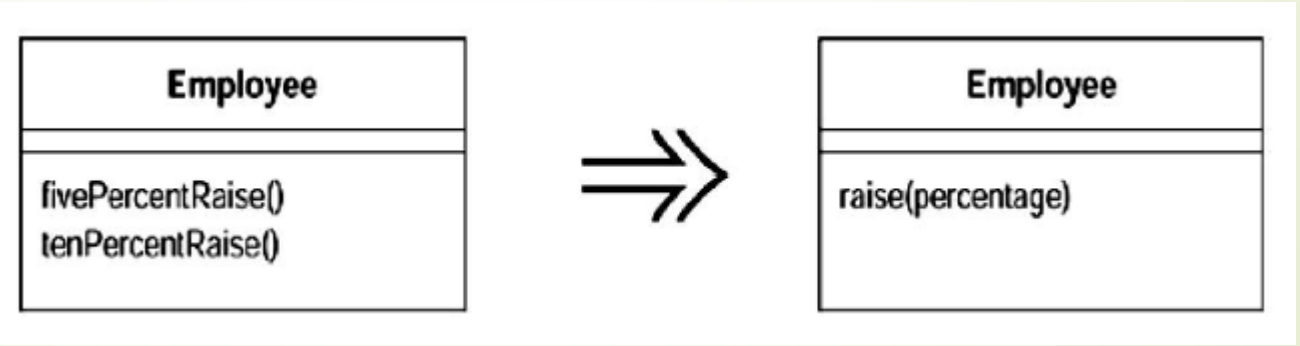
Making Method Calls Simpler: Separate Query from Modifier

- You have a method that returns a value but also changes the state of an object.
- Create two methods, one for the query and one for the modification.



Making Method Calls Simpler: Parameterize Method

- Several methods do similar things but with different values contained in the method body.
- Create one method that uses a parameter for the different values.



Making Method Calls Simpler: Replace Parameter with Explicit Methods

- You have a method that runs different code depending on the values of an enumerated parameter.
- Create a separate method for each value of the parameter.

```
void setValue (String name, int value) {
    if (name.equals("height"))
        _height = value;
    if (name.equals("width"))
        _width = value;
    Assert.shouldNeverReachHere();
}
```

```
void setHeight(int arg) {
    _height = arg;
}
void setWidth (int arg) {
    _width = arg;
}
```

Making Method Calls Simpler: Preserve Whole Object

- You are getting several values from an object and passing these values as parameters in a method call.
- Send the whole object instead.

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan = plan.withinRange(daysTempRange());
```

Making Method Calls Simpler: Replace Parameter with Method

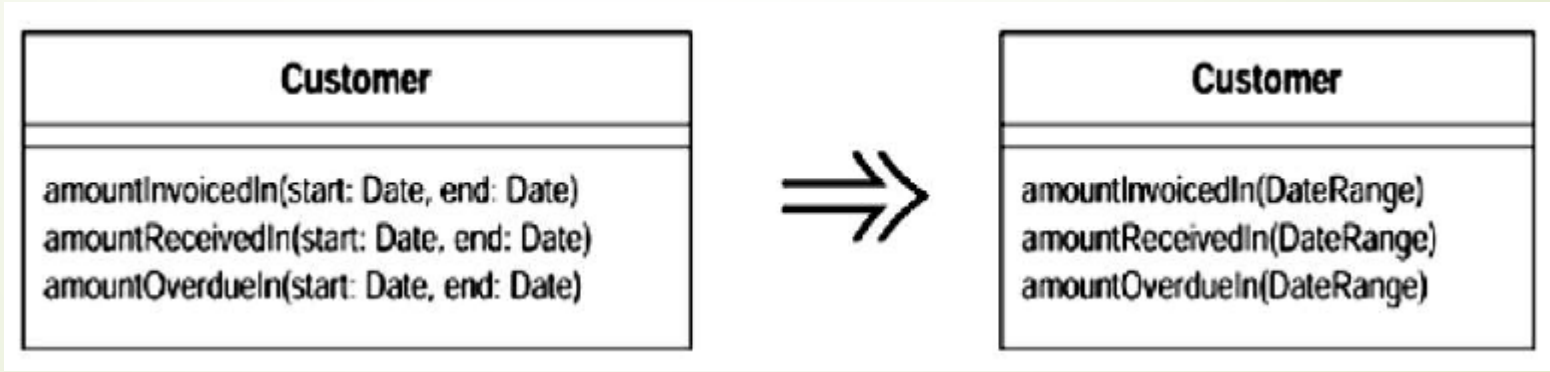
- An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.
- Remove the parameter and let the receiver invoke the method.

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

Making Method Calls Simpler: Introduce Parameter Object

- You have a group of parameters that naturally go together.
- Replace them with an object.



Dealing with Generalization: Pull Up Constructor Body

- You have constructors on subclasses with mostly identical bodies.
- Create a superclass constructor; call this from the subclass methods.

```
class Manager extends Employee...
  public Manager (String name, String id, int grade) {
      _name = name;
      _id = id;
      _grade = grade;
  }
```

```
  public Manager (String name, String id, int grade) {
      super (name, id);
      _grade = grade;
  }
```

Dealing with Generalization: Extract Subclass/Superclass

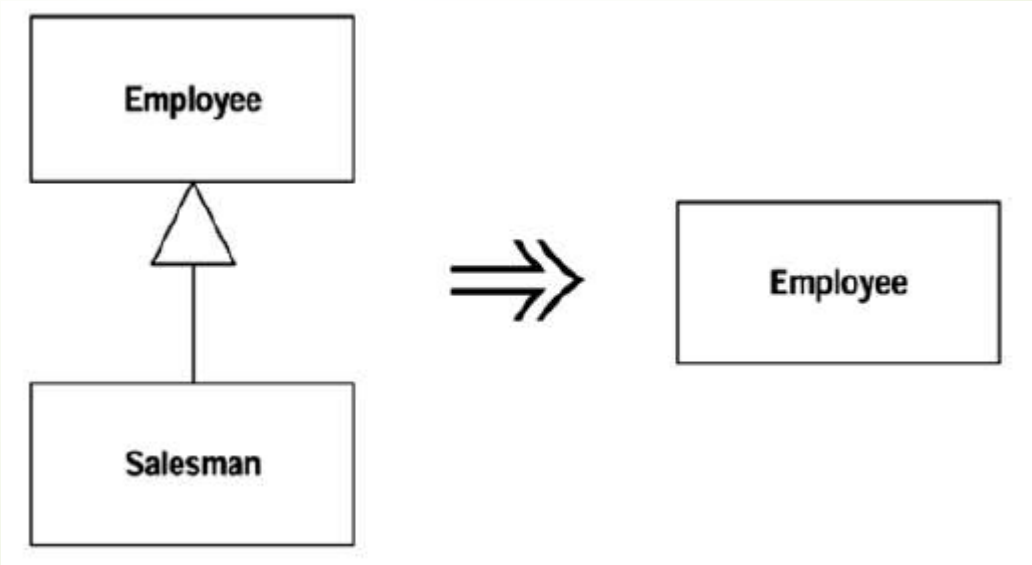
- **Extract Subclass**
  - A class has features that are used only in some instances.
  - Create a subclass for that subset of features.
- **Extract Superclass**
  - You have two classes with similar features.
  - Create a superclass and move the common features to the superclass.

Dealing with Generalization: Extract Interface

- Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.
- Extract the subset into an interface.

Dealing with Generalization: Collapse Hierarchy

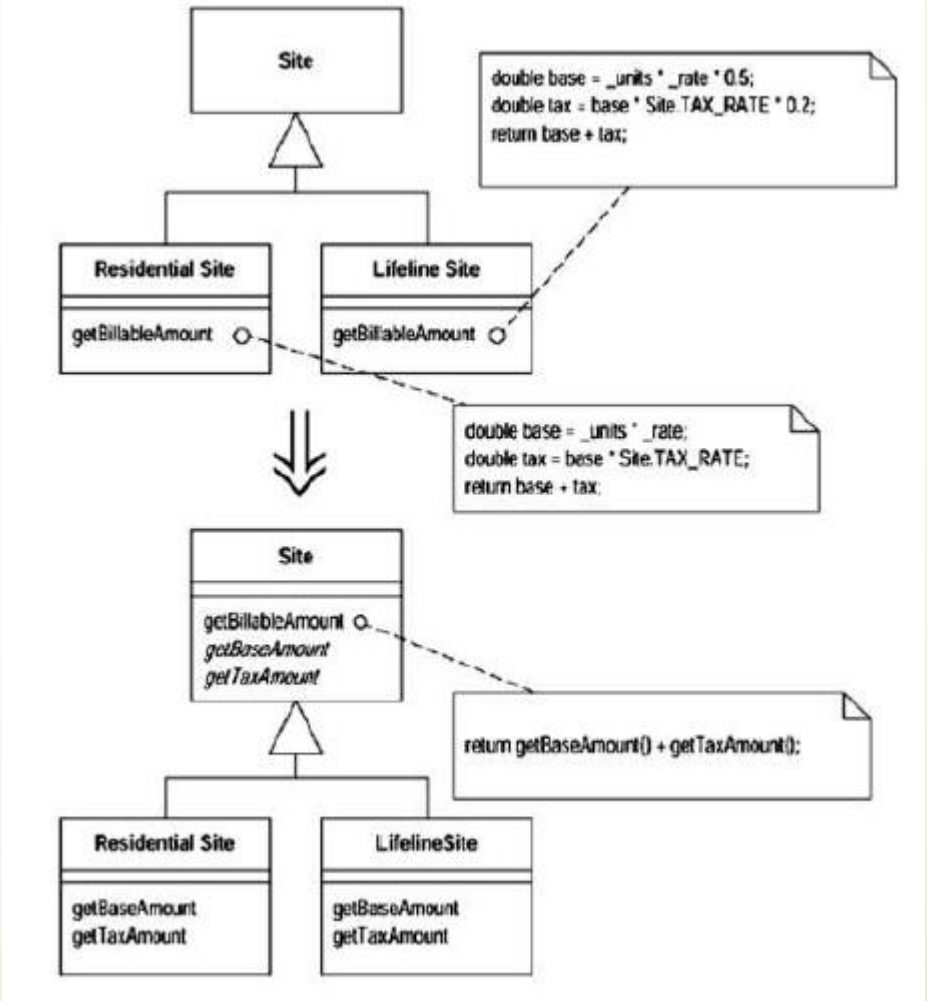
- A superclass and subclass are not very different.
- Merge them together.



Dealing with Generalization: Form Template Method

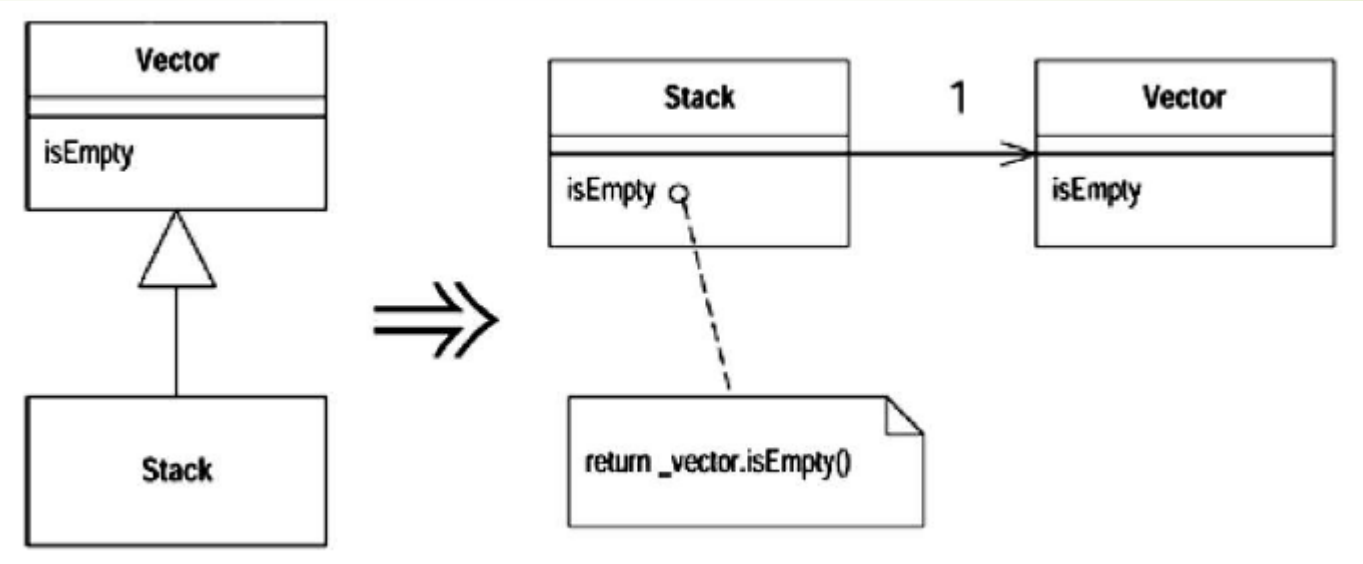
- You have two methods in subclasses that perform similar steps in the same order, yet the steps are different.
- Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.

Dealing with Generalization: Form Template Method



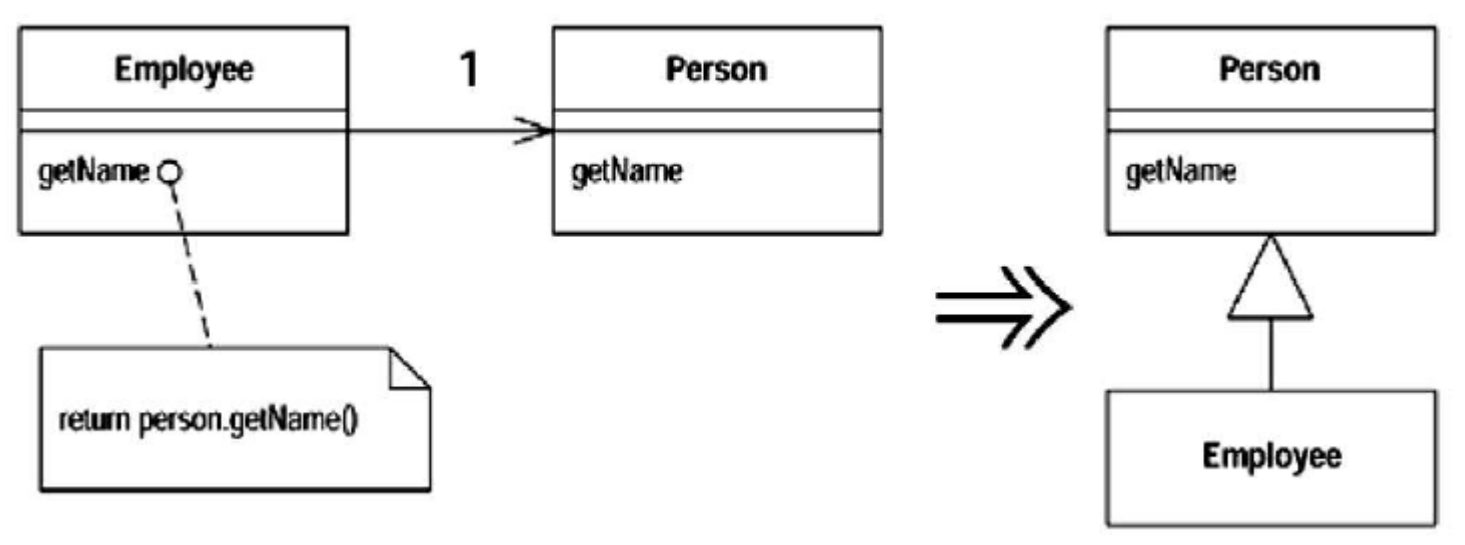
Dealing with Generalization: Replace Inheritance with Delegation

- A subclass uses only part of a superclass's interface or does not want to inherit data.
- Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.



Dealing with Generalization: Replace Delegation with Inheritance

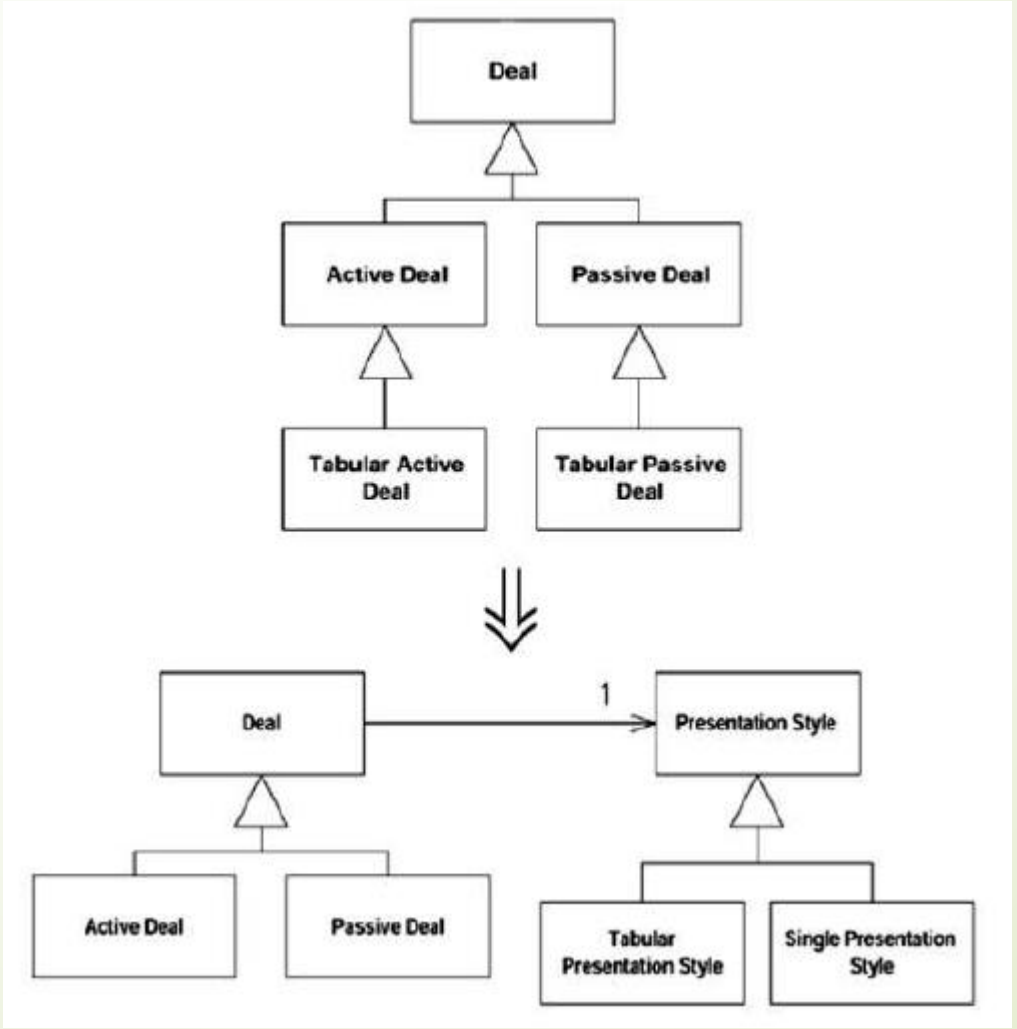
- You're using delegation and are often writing many simple delegations for the entire interface.
- Make the delegating class a subclass of the delegate.



Big Refactorings: Tease Apart Inheritance

- You have an inheritance hierarchy that is doing two jobs at once.
- Create two hierarchies and use delegation to invoke one from the other.

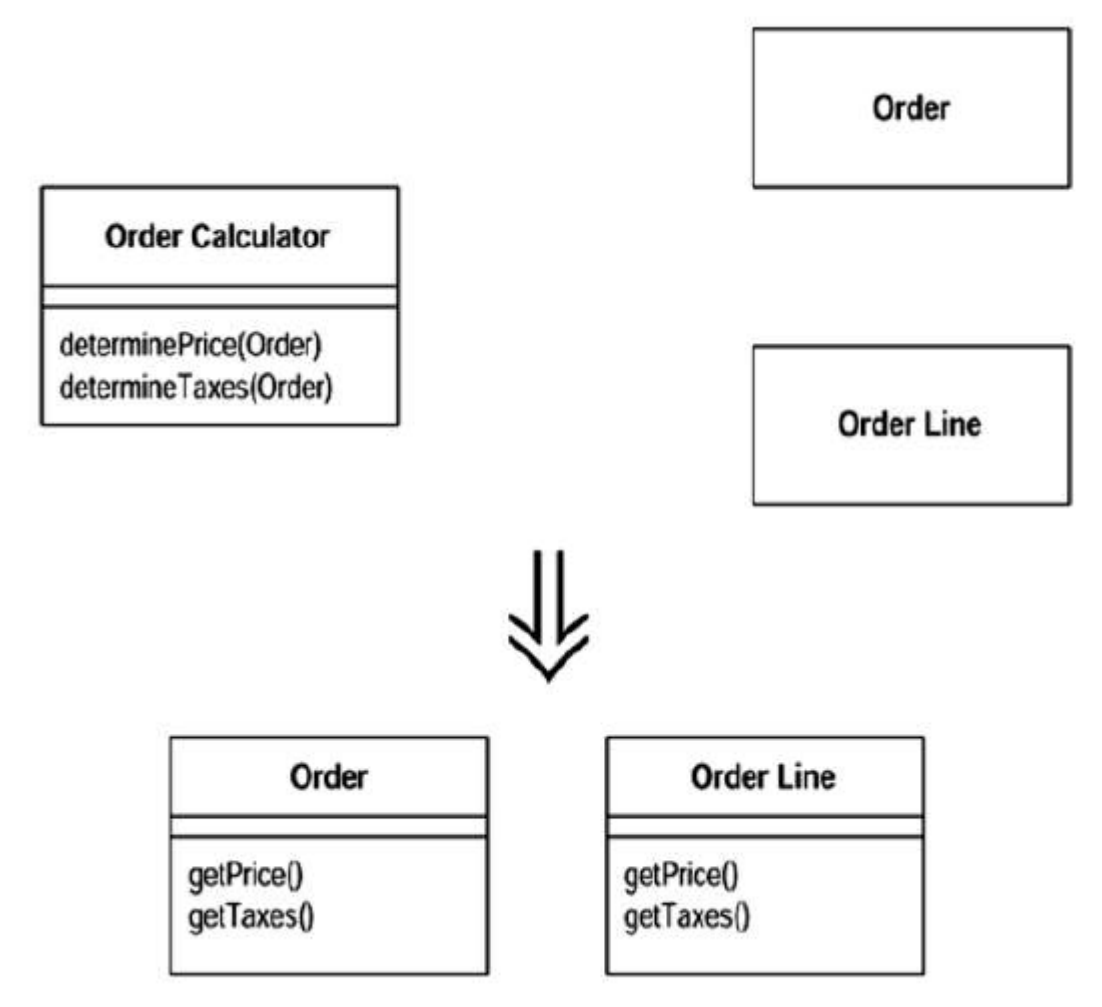
Big Refactorings: Tease Apart Inheritance



Big Refactorings: Convert Procedural Design to Objects

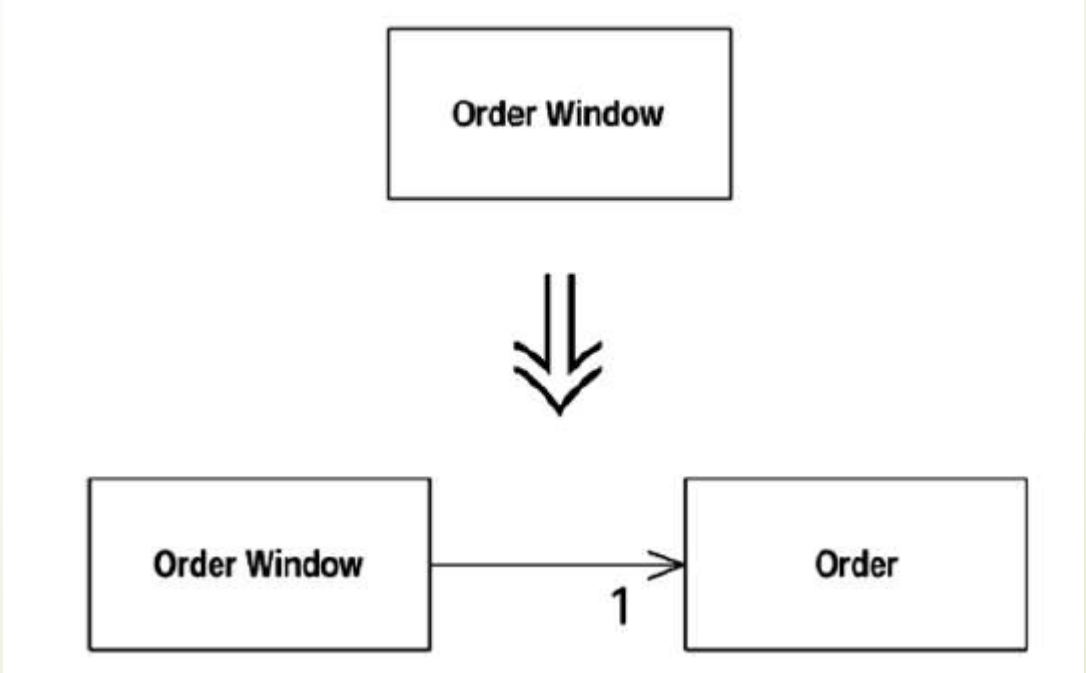
- You have code written in a procedural style.
- Turn the data records into objects, break up the behavior, and move the behavior to the objects.

Big Refactorings: Convert Procedural Design to Objects



Big Refactorings: Separate Domain from Presentation

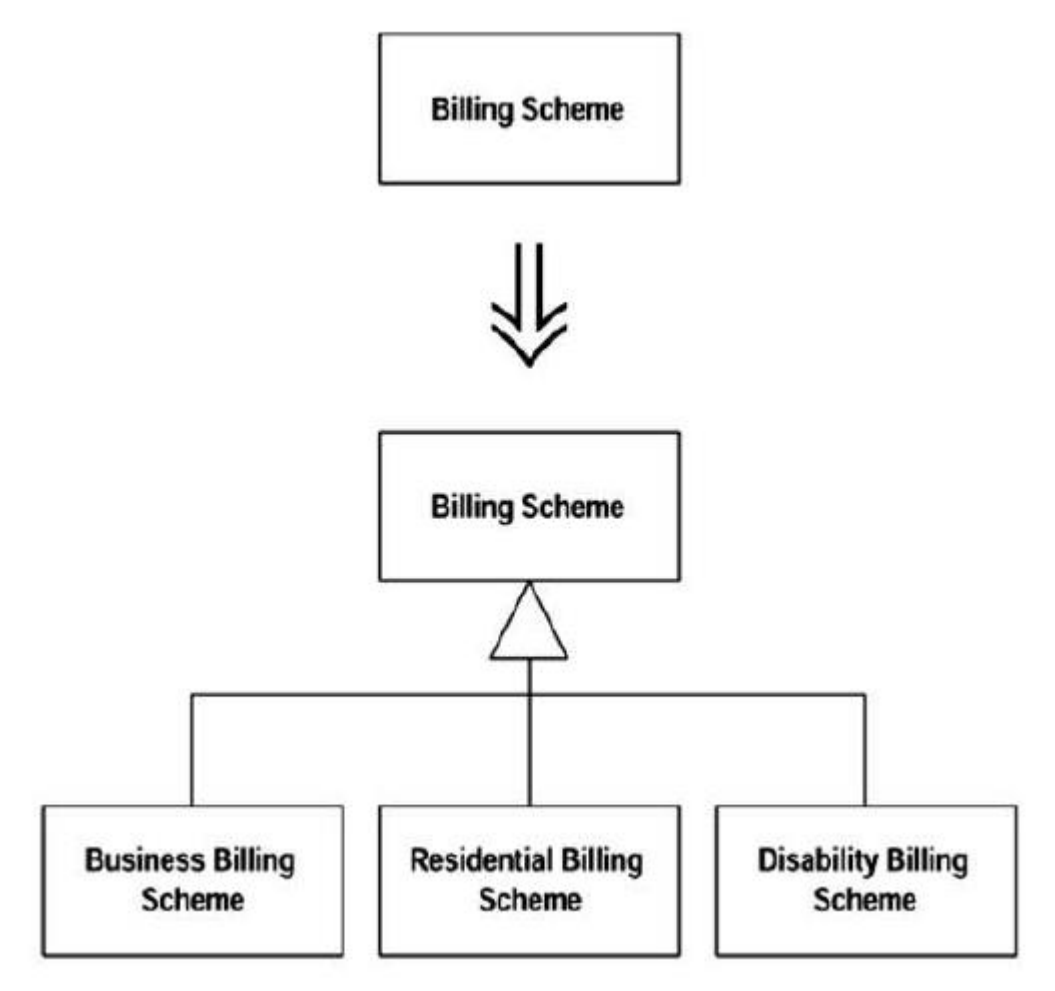
- You have GUI classes that contain domain logic.
- Separate the domain logic into separate domain classes.



Big Refactorings: Extract Hierarchy

- You have a class that is doing too much work, at least in part through many conditional statements.
- Create a hierarchy of classes in which each subclass represents a special case.

Big Refactorings: Extract Hierarchy



References

- Fowler, M., Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
- Fowler, M., Catalog of Refactorings, Published online at: http://refactoring.com/catalog/, December 2013 (last visited on: 1 December 2014).
- Ramsin, Raman. "Home." Department of Computer Science and Engineering, Sharif University of Technology. Accessed February 15, 2025. https://sharif.edu/~ramsin/index.htm.