Agile Software Development

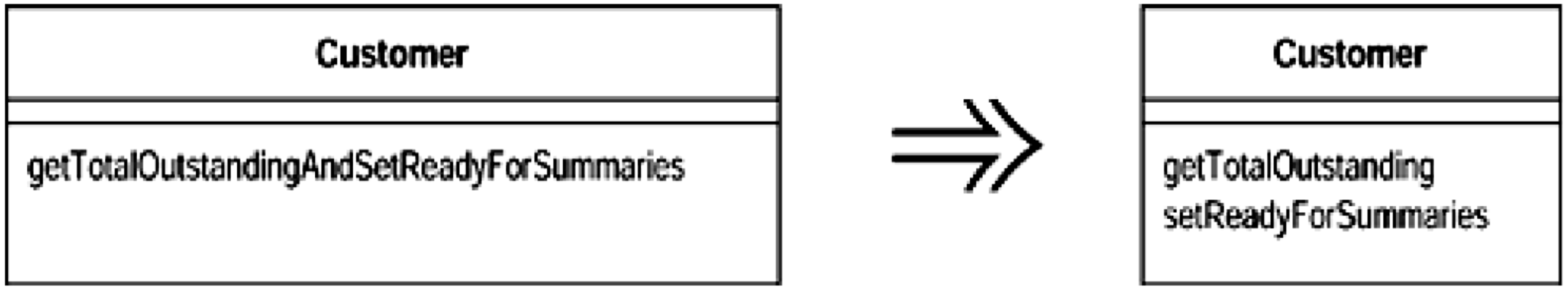
Lecturer: **Raman Ramsin**

Lecture 9

Refactoring – Part 3

Refactoring APIs: *Separate Query from Modifier*

- **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.*



Refactoring APIs: *Parameterize Function*

- **Parameterize Function**
  - Several functions do similar things but with different values contained in the function body.
  - *Create one function that uses a parameter for the different values.*

```
function tenPercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.1);
}
function fivePercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.05);
}
```

$\Downarrow$

```
function raise(aPerson, factor) {
  aPerson.salary = aPerson.salary.multiply(1 + factor);
}
```

Refactoring APIs: *Remove Flag Argument*

- **Remove Flag Argument**
  - You have a Function that runs different code depending on the values of an enumerated parameter.
  - *Create a separate function for each value of the parameter.*

```
function setDimension(name, value) {
  if (name === "height") {
    this._height = value;
    return;
  }
  if (name === "width") {
    this._width = value;
    return;
  }
}
```

⇓

```
function setHeight(value) {this._height = value;}
function setWidth (value) {this._width = value;}
```

Refactoring APIs: *Preserve Whole Object*

- **Preserve Whole Object**
  - You are getting several values from an object and passing these values as parameters in a function call.
  - *Send the whole object instead.*

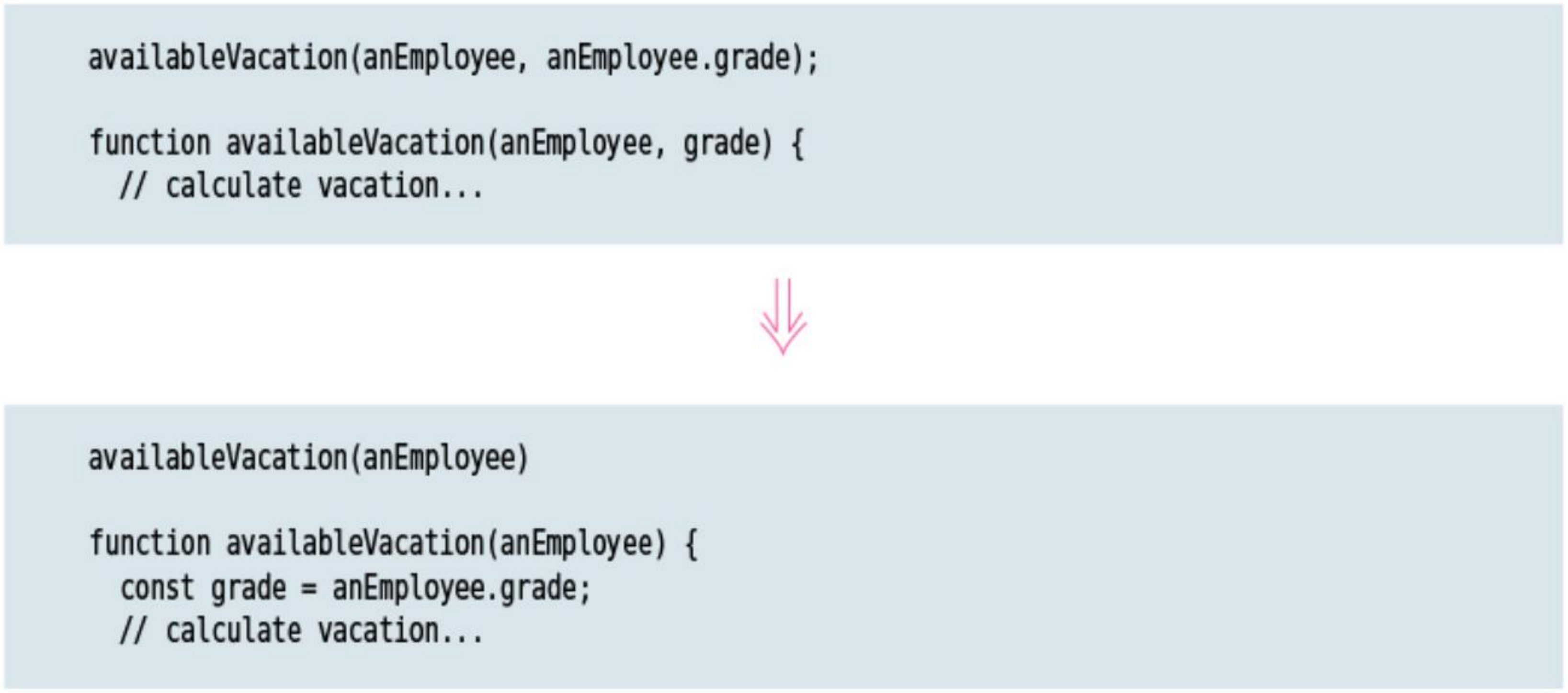
```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

$\Downarrow$

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

Refactoring APIs: *Replace Parameter with Query*

- **Replace Parameter with Query**
  - A function call passes in a value that the function can just as easily determine for itself.
  - *Remove the parameter and let the receiver determine the value.*



```
availableVacation(anEmployee, anEmployee.grade);

function availableVacation(anEmployee, grade) {
  // calculate vacation...
```

```
availableVacation(anEmployee)

function availableVacation(anEmployee) {
  const grade = anEmployee.grade;
  // calculate vacation...
```

Dealing with Inheritance: *Pull-Up/Push-Down Method/Field*

- **Pull Up Method/Field**
  - A method/field is present in all the subclasses.
  - *Move the method/field to the superclass.*
- **Push Down Method/Field**
  - A method/field of the superclass is not relevant to all the subclasses.
  - *Move the method/field to the relevant subclasses.*

Dealing with Inheritance: *Pull Up Constructor Body*

- **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;
  }
```

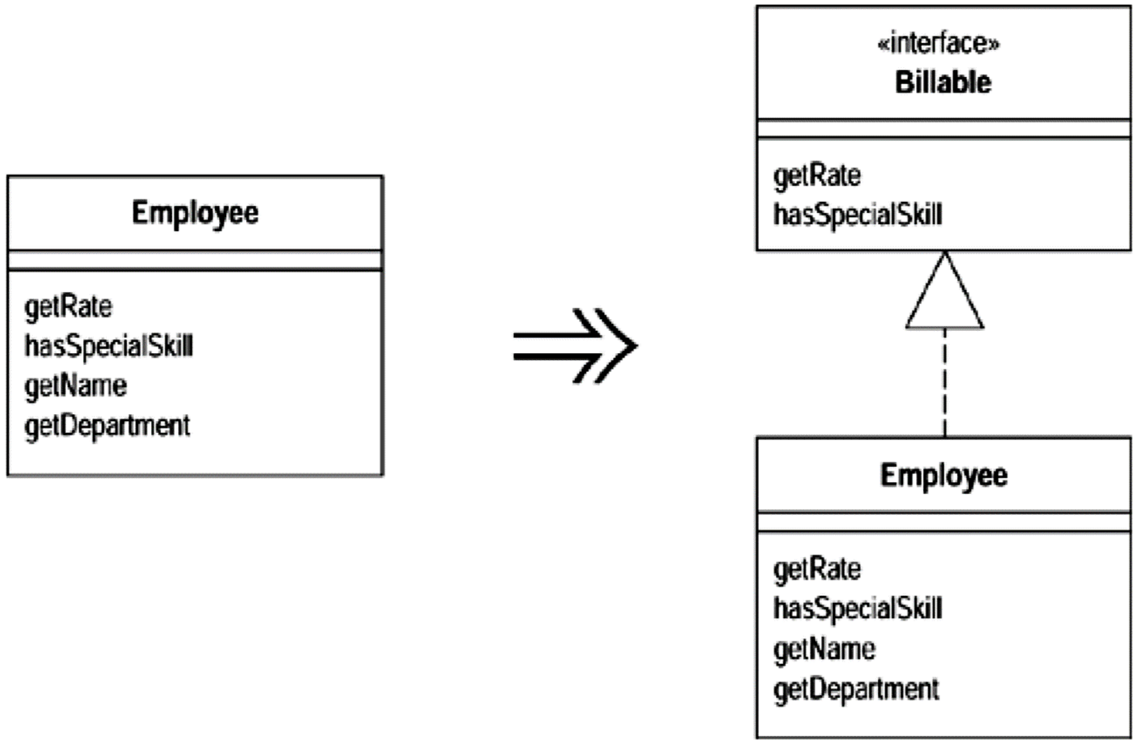
Dealing with Inheritance: *Extract Subclass/Superclass*

- **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.*

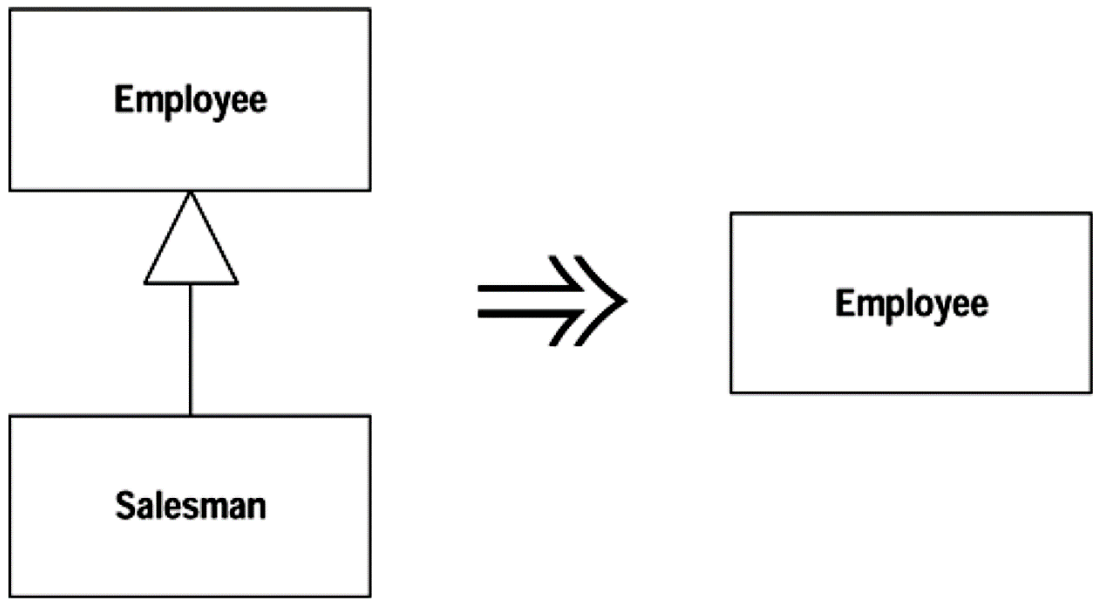
Dealing with Inheritance: *Extract Interface*

- **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 Inheritance: *Collapse Hierarchy*

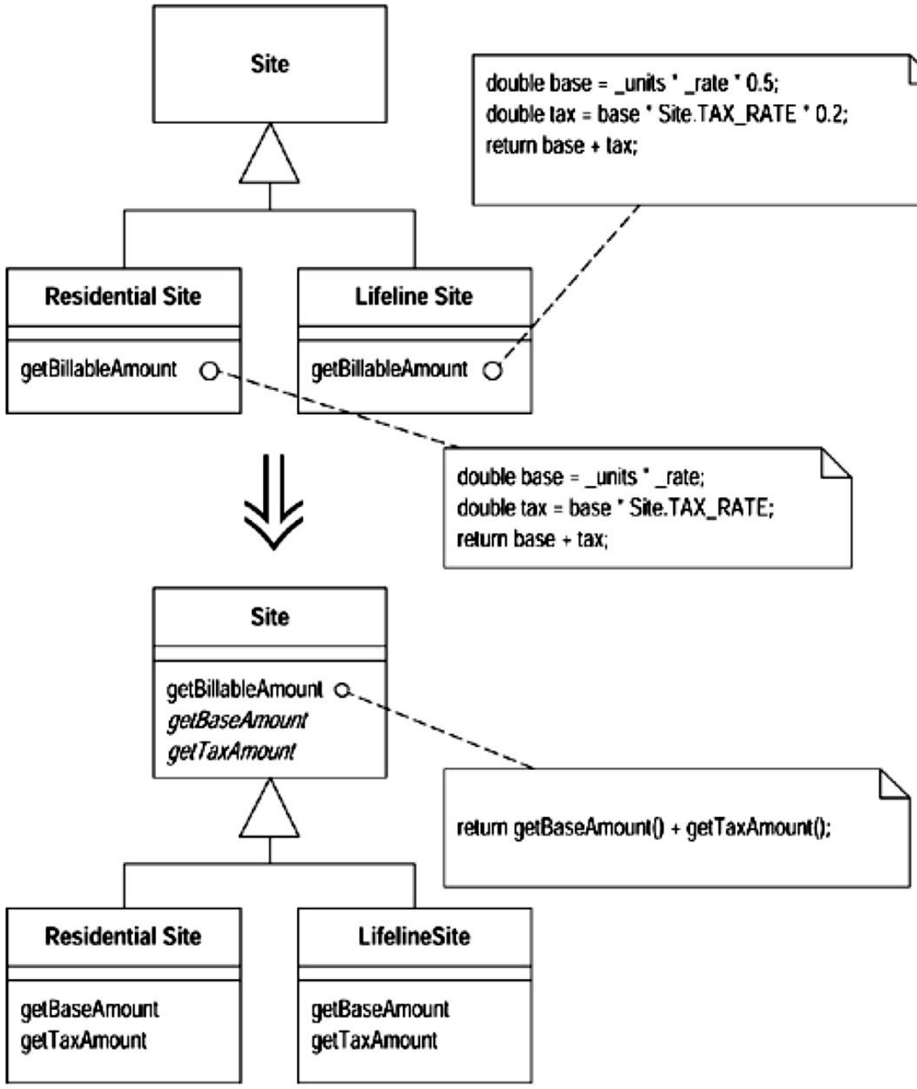
- **Collapse Hierarchy**
  - A superclass and subclass are not very different.
  - *Merge them together.*



Dealing with Inheritance: *Form Template Method*

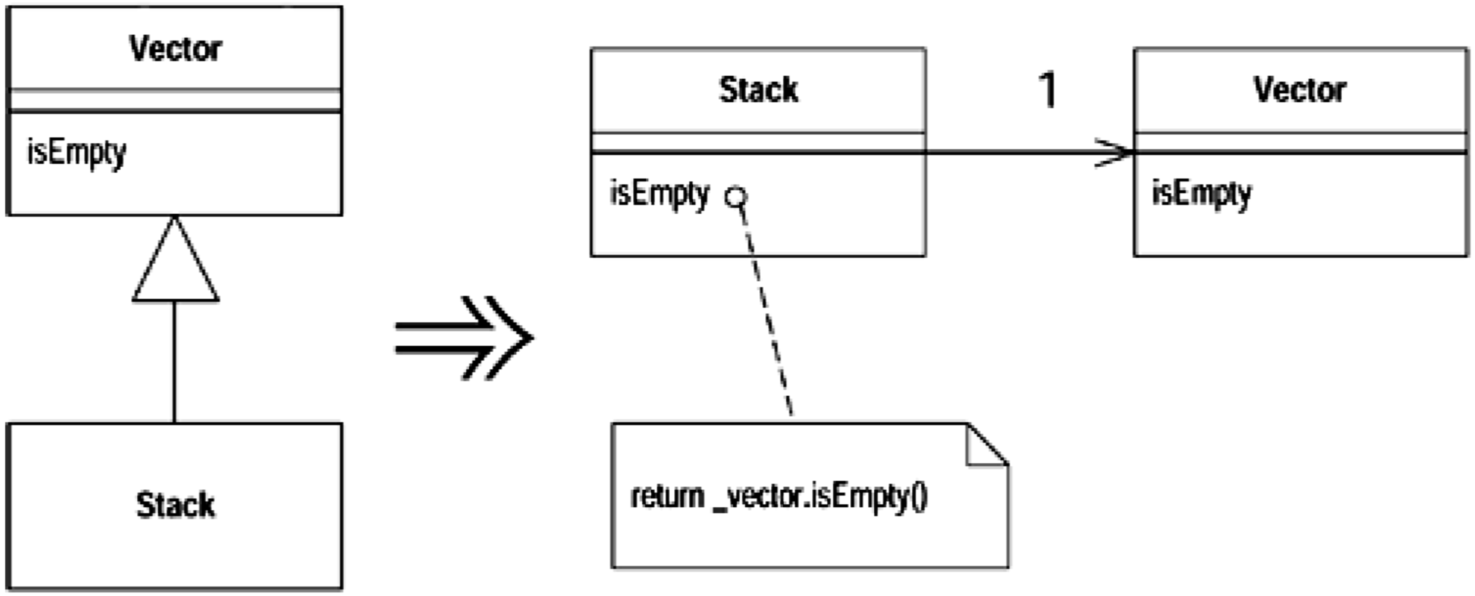
- **Form Template Method**
  - You have two methods in subclasses that perform similar steps in the same order, yet the steps are different.
  - *Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.*

Dealing with Inheritance: *Form Template Method*



Dealing with Inheritance: *Replace Superclass with Delegate*

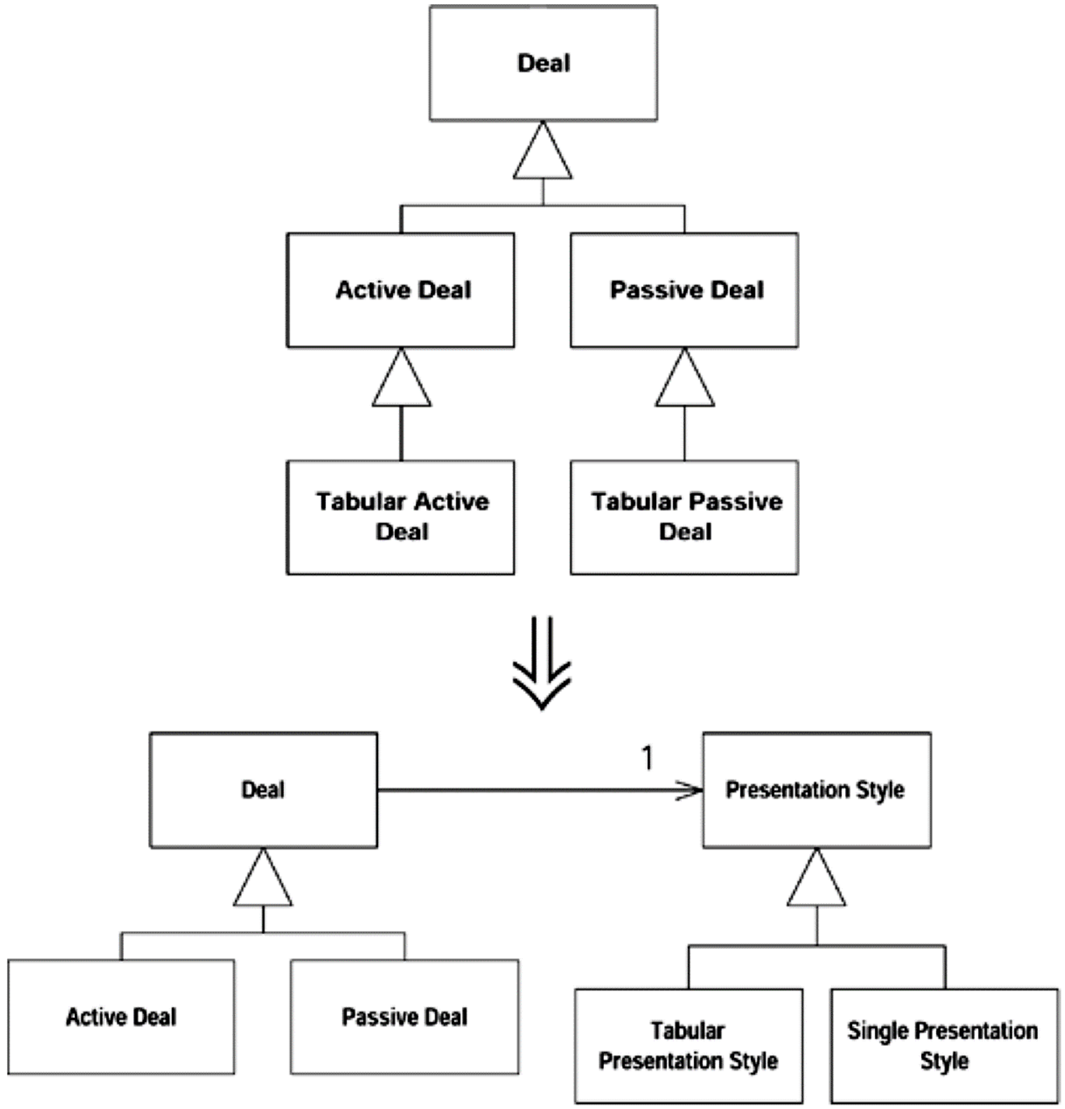
- **Replace Superclass with Delegate**
  - A subclass uses only part of a superclass's interface or does not want to inherit data.
  - *Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.*



Big Refactorings: *Tease Apart Inheritance*

- **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.*

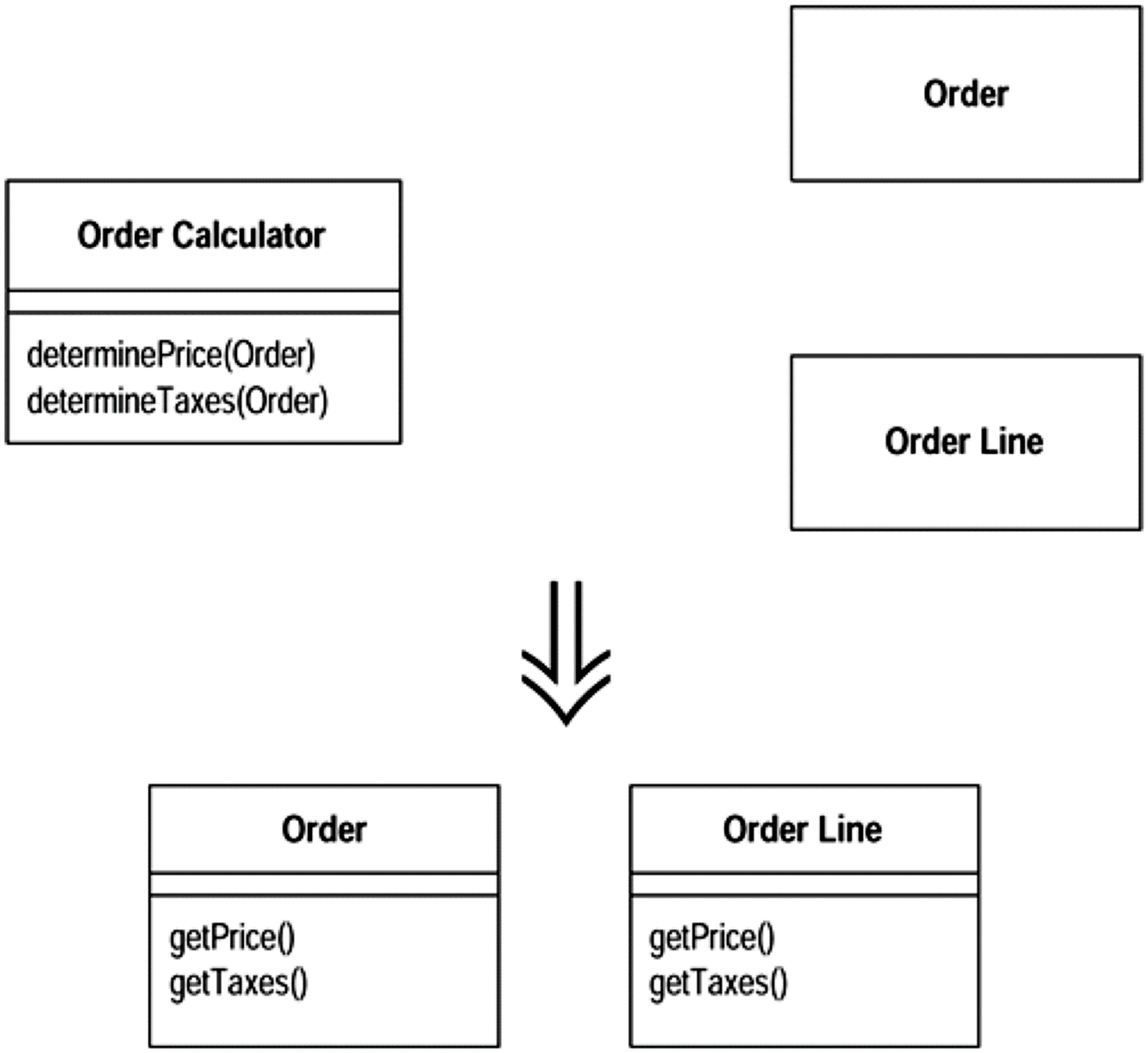
Big Refactorings: *Tease Apart Inheritance*



Big Refactorings: *Convert Procedural Design to Objects*

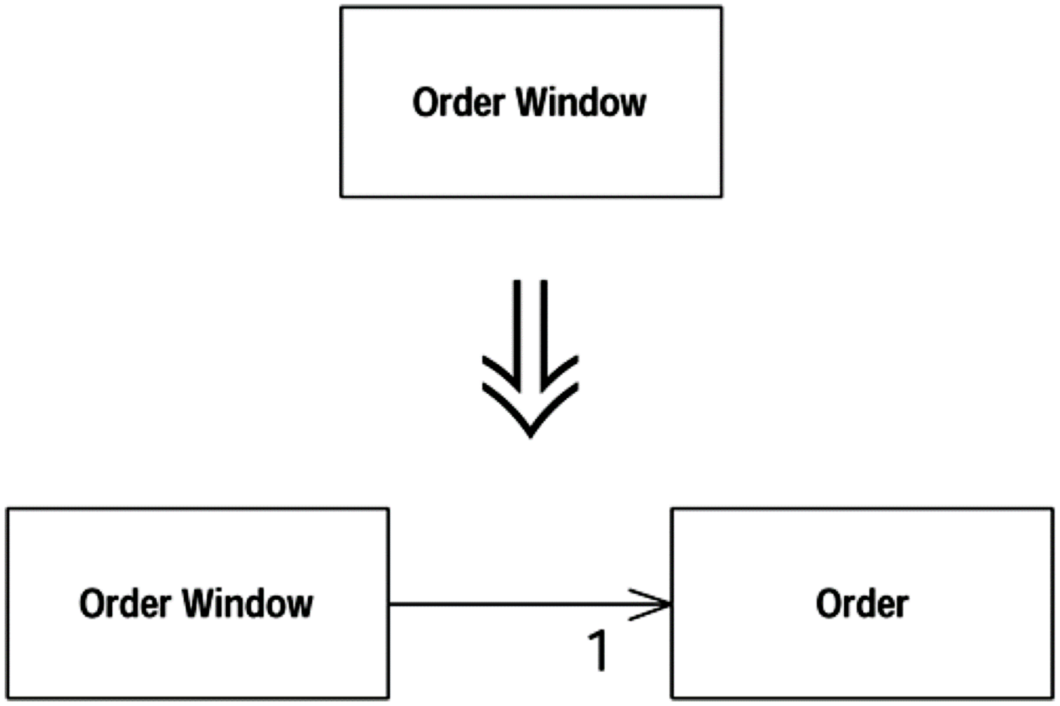
- **Convert Procedural Design to Objects**
  - You have code written in a procedural style.
  - *Turn the data records into objects, break up the behavior, and move the behavior to the objects.*

Big Refactorings: *Convert Procedural Design to Objects*



Big Refactorings: *Separate Domain from Presentation*

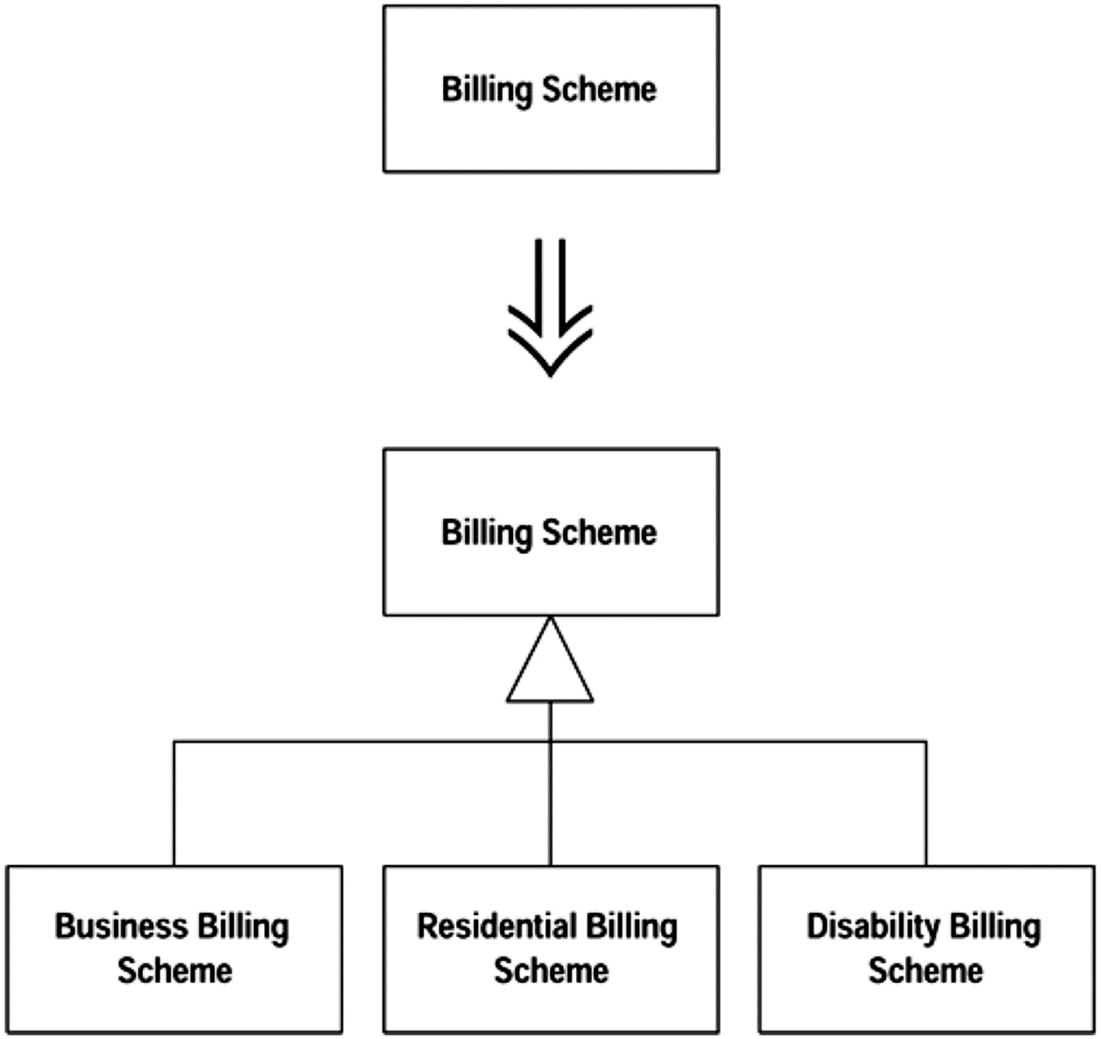
- **Separate Domain from Presentation**
  - You have GUI classes that contain domain logic.
  - *Separate the domain logic into separate domain classes.*



Big Refactorings: *Extract Hierarchy*

- **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*



Reference

- Fowler, M., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999.
- Fowler, M., *Refactoring: Improving the Design of Existing Code,* 2nd Edition, Addison-Wesley, 2019.