Patterns in Software Engineering

Lecturer: Raman Ramsin

Lecture 11

Refactoring Patterns

Part 2

Organizing Data: *Self Encapsulate Field*

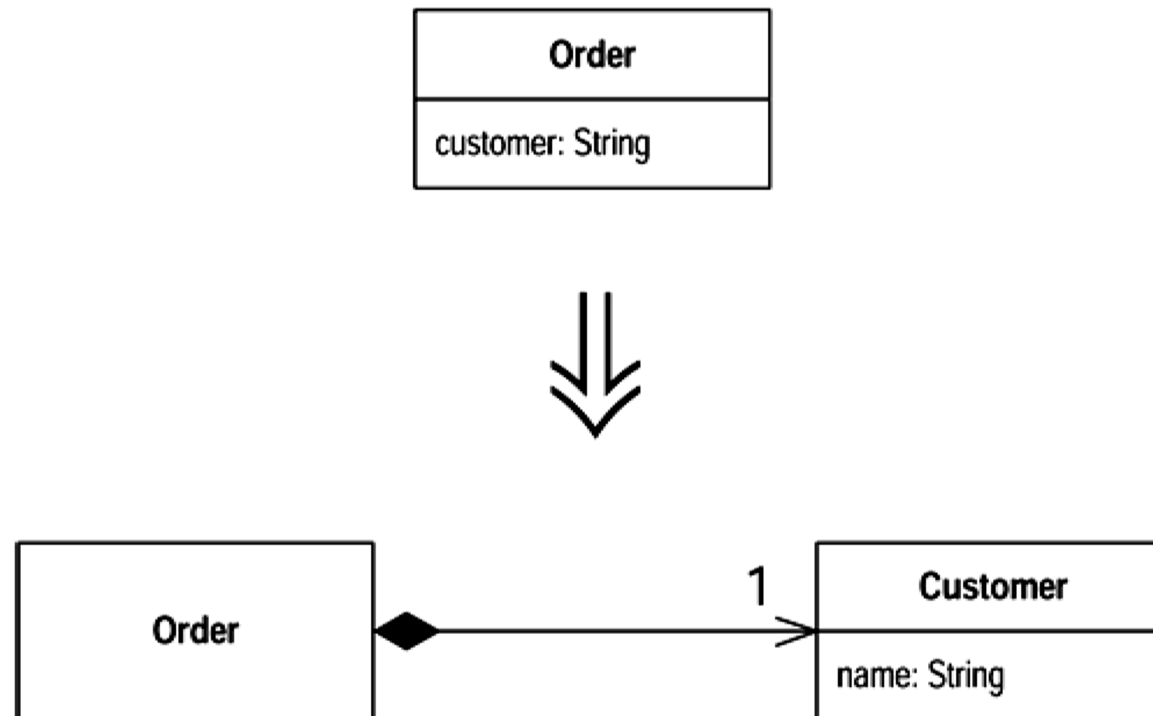
- **Self Encapsulate Field**
  - You are accessing a field directly, but the coupling to the field is becoming awkward.
  - *Create getting and setting methods for the field and use only those to access the field.*

```
private int _low, _high;
boolean includes (int arg) {
    return arg >= _low && arg <= _high;
}
```

```
private int _low, _high;
boolean includes (int arg) {
    return arg >= getLow() && arg <= getHigh();
}
int getLow() {return _low;}
int getHigh() {return _high;}
```

Organizing Data: *Replace Data Value with Object*

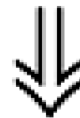
- **Replace Data Value with Object**
  - You have a data item that needs additional data or behavior.
  - *Turn the data item into an object.*



Organizing Data: *Replace Array with Object*

- **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.*

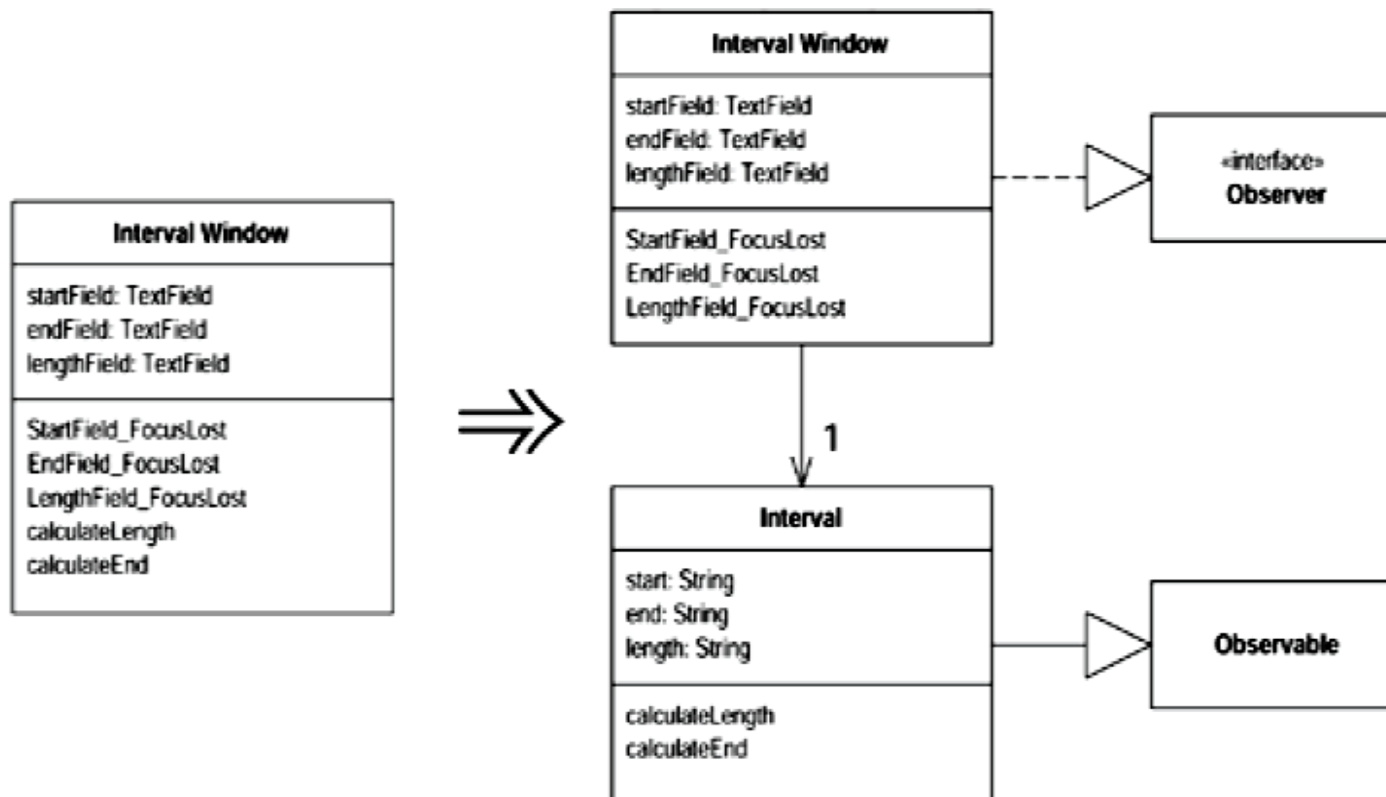
```
String[] row = new String[3];
row [0] = "Liverpool";
row [1] = "15";
```



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

Organizing Data: *Duplicate Observed Data*

- **Duplicate Observed Data**
  - You have domain data available only in a GUI control, and domain methods need access.
  - *Copy the data to a domain object. Set up an observer to synchronize the two pieces of data.*



Organizing Data: *Encapsulate Field*

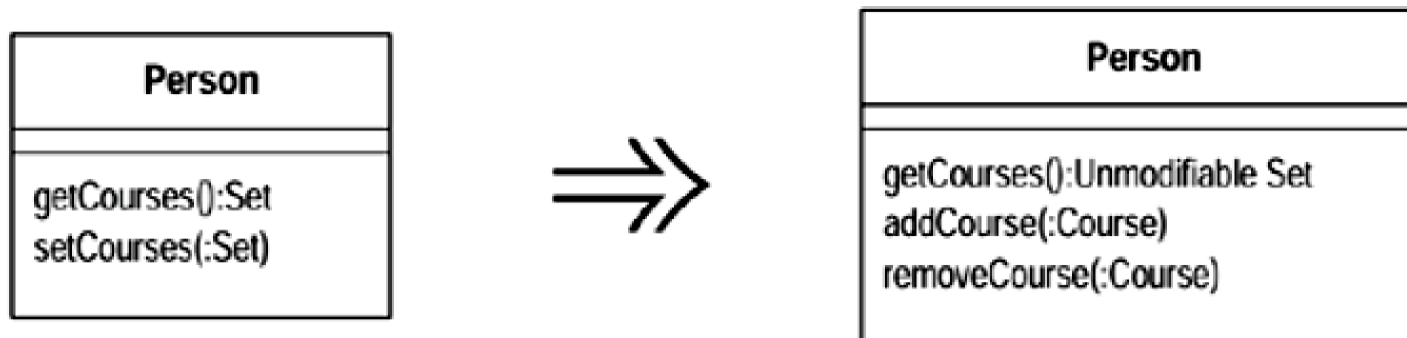
- **Encapsulate Field**
  - There is a public field.
  - *Make it private and provide accessors.*

```
public String _name
```

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

Organizing Data: *Encapsulate Collection*

- **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*

- **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.*

Person
getCode()
Male
getCode()
return 'M'
Female
getCode()
return 'F'

⇛

Person
code
getCode()

Simplifying Conditional Expressions: *Decompose Conditional*

- **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*

- **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*

- **Replace Nested Conditional with Guard Clauses**
  - A method has conditional behavior that does not make clear the normal path of execution.
  - *Use guard clauses for all the special cases.*

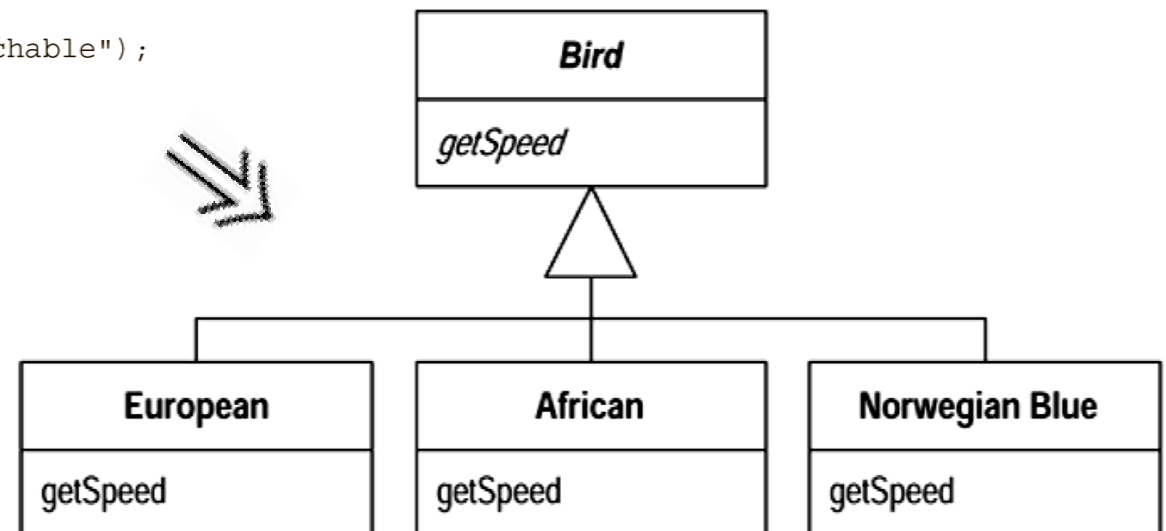
```
double getPayAmount() {
  double result;
  if (_isDead) result = deadAmount();
  else {
      if (_isSeparated) result = separatedAmount();
      else {
          if (_isRetired) result = retiredAmount();
          else result = normalPayAmount();
      };
  }
return result;
};
```

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

Simplifying Conditional Expressions: *Replace Conditional with Polymorphism*

- **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.*

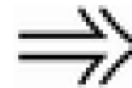
```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
           return getBaseSpeed();
        case AFRICAN:
           return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
        case NORWEGIAN_BLUE:
           return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw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

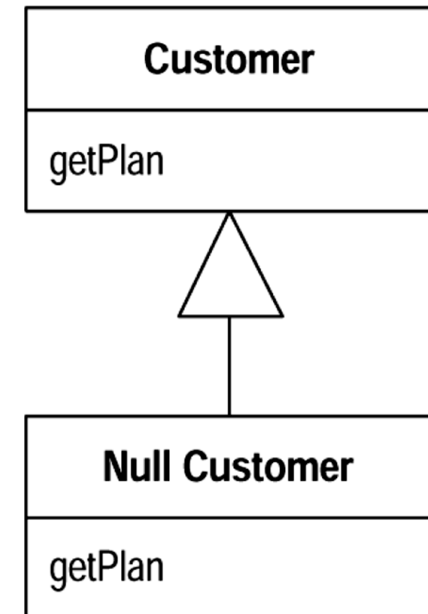


Simplifying Conditional Expressions: *Introduce Null Object*

- **Introduce Null Object**
  - You have repeated checks for a null value.
  - *Replace the null value with a null object.*

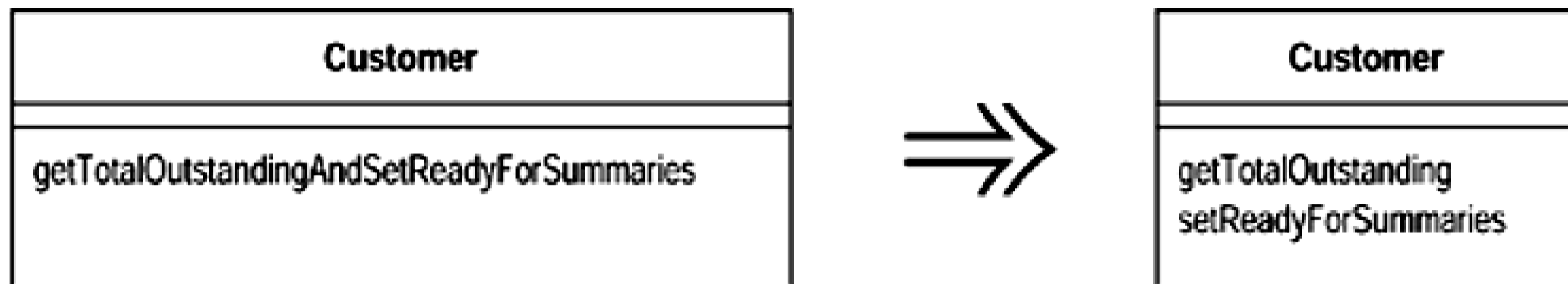
```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```





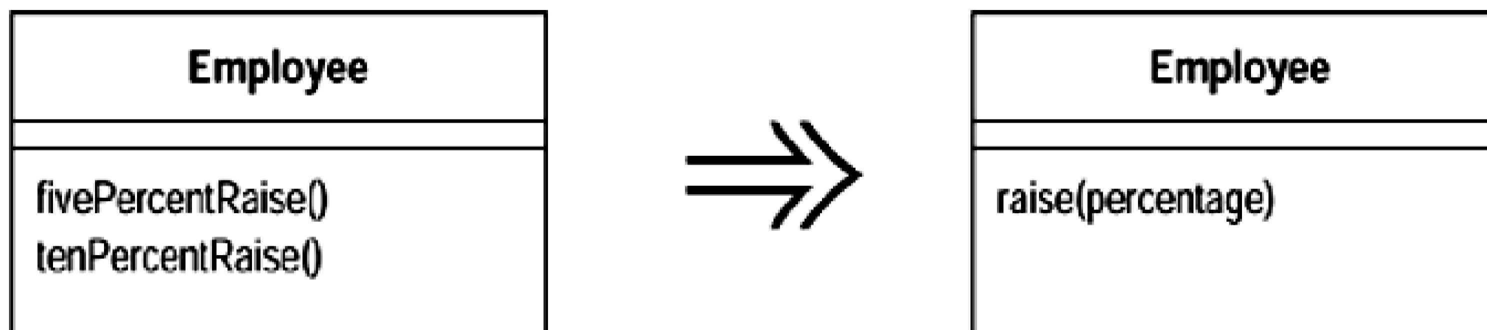
Making Method Calls Simpler: *Separate Query from Modifier*

- **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*

- **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*

- **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*

- **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*

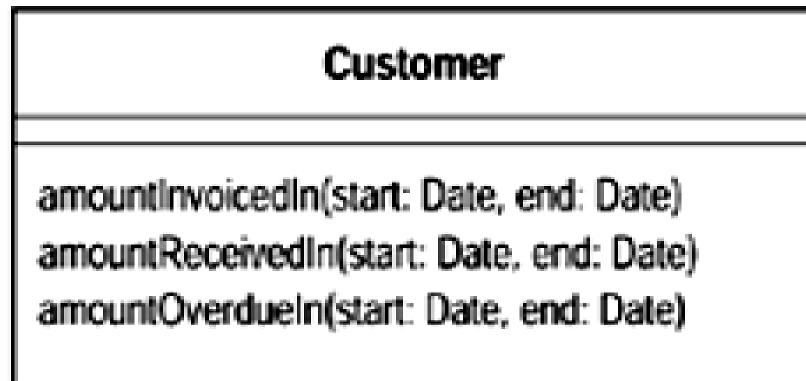
- **Replace Parameter with Method**
  - An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.
  - *Remove the parameter and let the receiver invoke the method.*

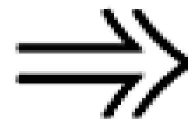
```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

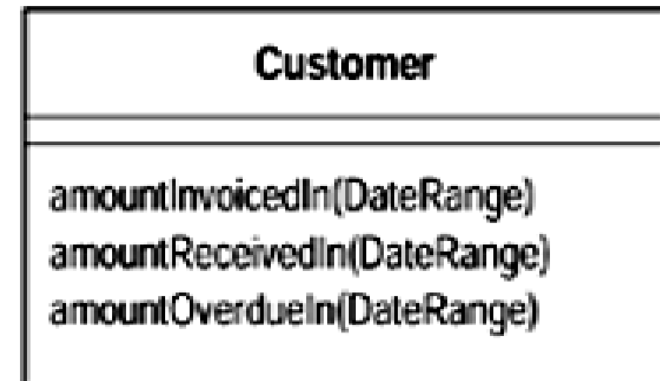
```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

Making Method Calls Simpler: *Introduce Parameter Object*

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







Reference

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