Patterns in Software Engineering

Lecturer: Raman Ramsin

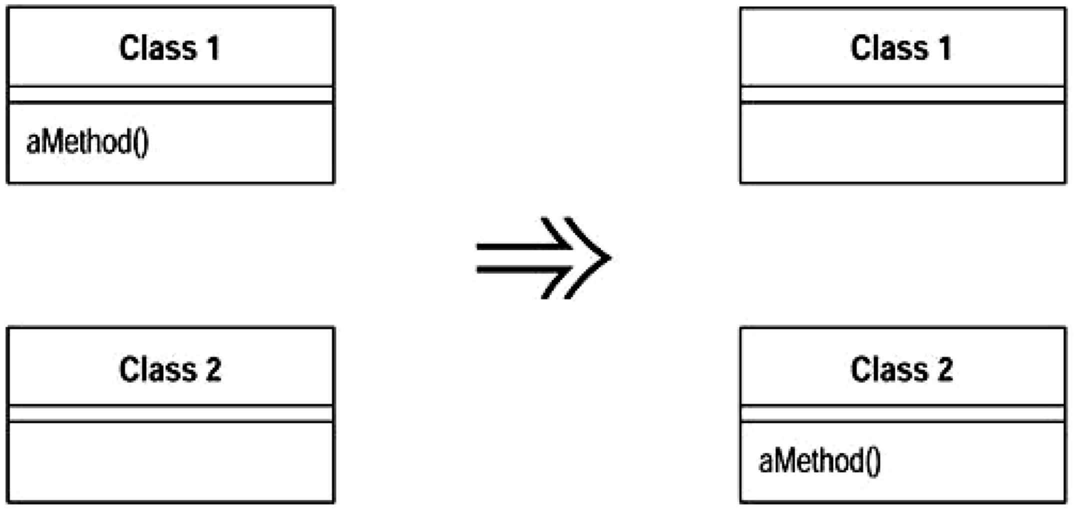
Lecture 11

Refactoring Patterns

Part 2

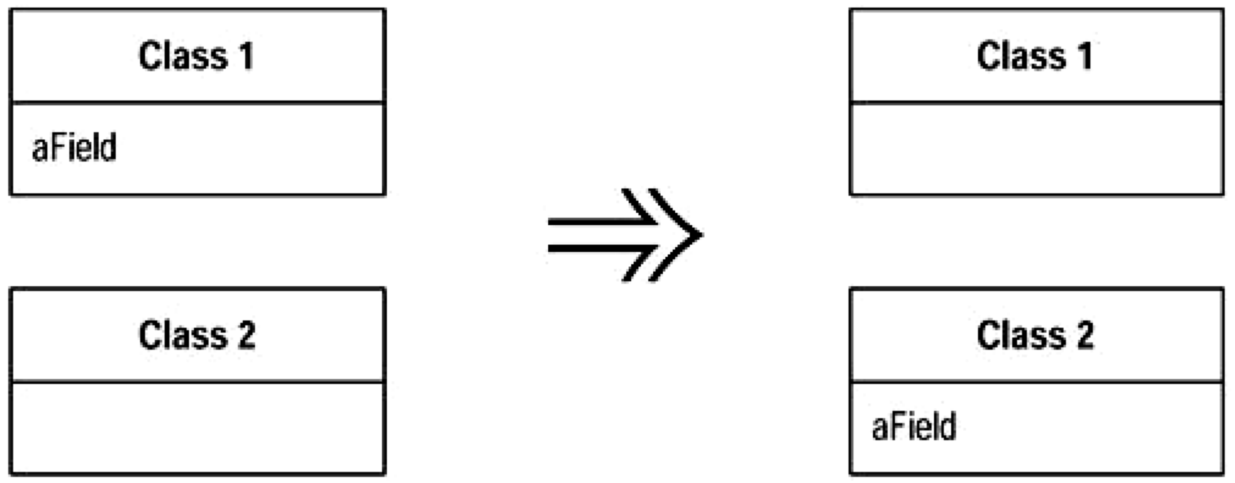
Moving Features: *Move Function*

- **Move Function**
  - A function is, or will be, using or used in another context than the context in which it currently resides.
  - *Create a new function with a similar body in the new context. Either turn the old function into a simple delegation, or remove it altogether.*



Moving Features: *Move Field*

- **Move Field**
  - A field is, or will be, used by another context more than the context in which it already resides.
  - *Create a new field in the target context, and change all its users.*



Moving Features: *Slide Statements*

- **Slide Statements**
  - Several lines of code access the same data structure, but they are intermingled with code accessing other data structures.
  - *Move them together.*

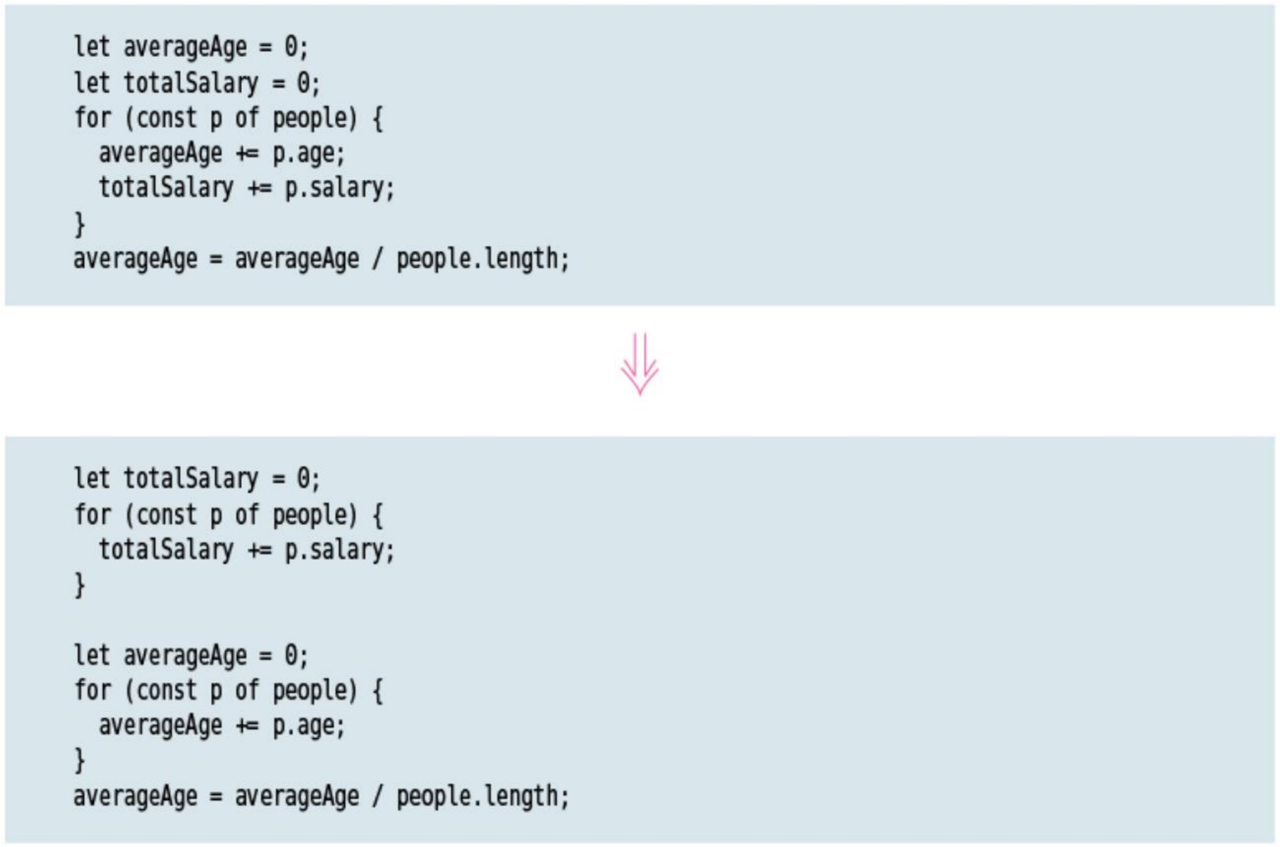
```
const pricingPlan = retrievePricingPlan();
const order = retreiveOrder();
let charge;
const chargePerUnit = pricingPlan.unit;
```

$\Downarrow$

```
const pricingPlan = retrievePricingPlan();
const chargePerUnit = pricingPlan.unit;
const order = retreiveOrder();
let charge;
```

Moving Features: *Split Loop*

- **Split Loop**
  - You’re doing two different things in the same loop, and whenever you need to modify the loop you have to understand both things.
  - *Split the loop into two independent ones.*



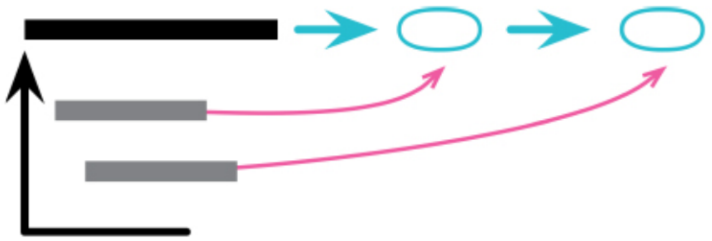
```
let averageAge = 0;
let totalSalary = 0;
for (const p of people) {
  averageAge += p.age;
  totalSalary += p.salary;
}
averageAge = averageAge / people.length;
```

```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}

let averageAge = 0;
for (const p of people) {
  averageAge += p.age;
}
averageAge = averageAge / people.length;
```

Moving Features: *Replace Loop with Pipeline*

- **Replace Loop with Pipeline**
  - You are using loops to iterate over a collection of objects.
  - *Use Collection Pipelines instead, which describe the processing as a series of operations, each consuming and emitting a collection.*



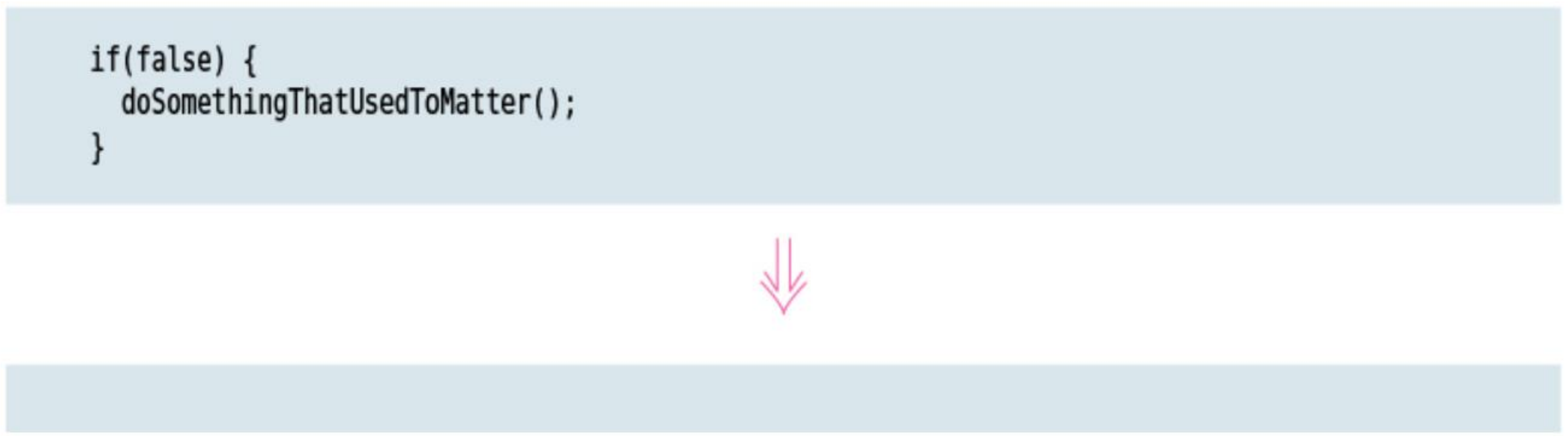
```
const names = [];
for (const i of input) {
  if (i.job === "programmer")
    names.push(i.name);
}
```

⇓

```
const names = input
  .filter(i => i.job === "programmer")
  .map(i => i.name)
;
```

Moving Features: *Remove Dead Code*

- **Remove Dead Code**
  - Unused code is becoming a significant burden when trying to understand how the software works.
  - *Remove it mercilessly.*



```
if(false) {
  doSomethingThatUsedToMatter();
}
```

$\Downarrow$

Organizing Data: *Split Variable*

- **Split Variable**
  - A variable has more than one responsibility within the method.
  - *It should be replaced with multiple variables, one for each responsibility.*

```
let temp = 2 * (height + width);
console.log(temp);
temp = height * width;
console.log(temp);
```

⇓

```
const perimeter = 2 * (height + width);
console.log(perimeter);
const area = height * width;
console.log(area);
```

Organizing Data: *Change Reference to Value*

- **Change Reference to Value**
  - There is a changeable object, or data structure, nested within another.
  - *Provide immutable copies of it (such as Value Objects) to pass around.*

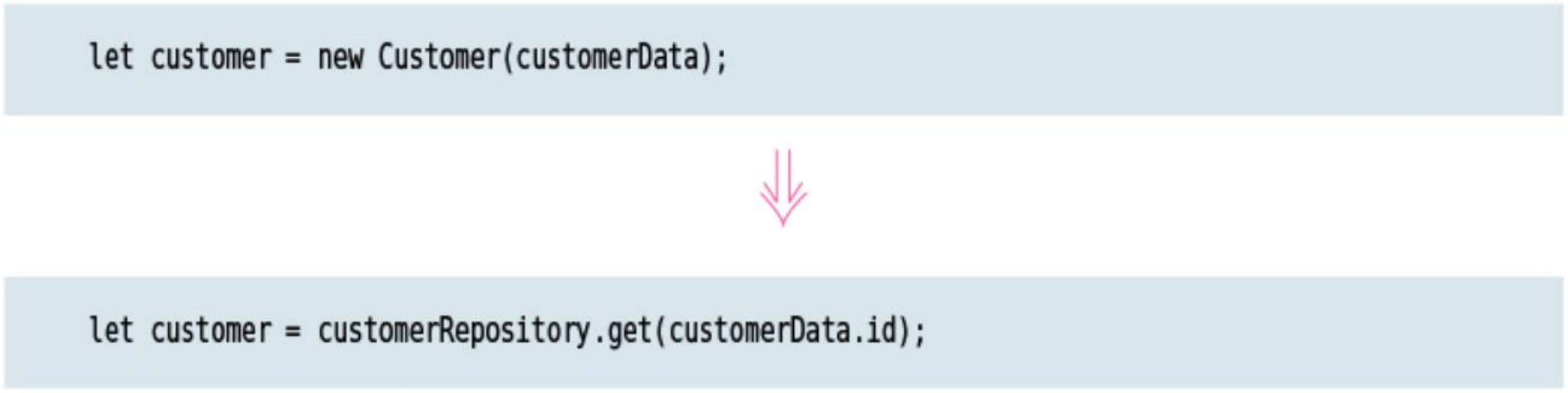
```
class Product {
  applyDiscount(arg) {this._price.amount -= arg;}
```

⇓

```
class Product {
  applyDiscount(arg) {
    this._price = new Money(this._price.amount - arg, this._price.currency);
  }
```

Organizing Data: *Change Value to Reference*

- **Change Value to Reference**
  - Immutable copies of an object or data structure are passed around, but they need to be updated based on changes made to the original.
  - *Change the copied data into a single reference.*



```
let customer = new Customer(customerData);
```

```
let customer = customerRepository.get(customerData.id);
```

Simplifying Conditional Logic: *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;
```

$$\Downarrow$$

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

Simplifying Conditional Logic: *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
```

$$\Downarrow$$

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

Simplifying Conditional Logic: *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.*

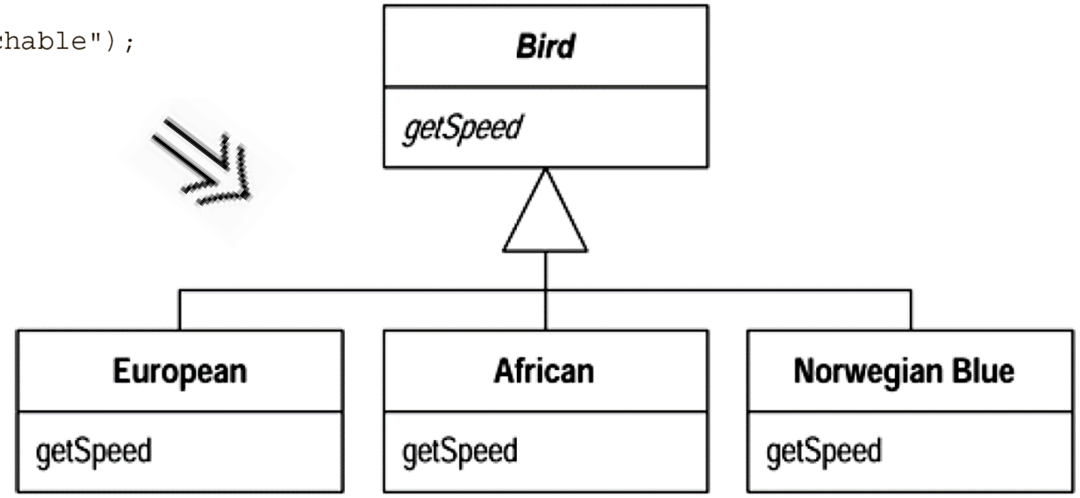
```
double getPayAmount() {
  double result;
  if (_isDead) result = deadAmount();
  else {
      if (_isSeparated) result = separatedAmount();
      else {
          if (_isRetired) result = retiredAmount();
          else result = normalPayAmount();
      };
  }
return result;
};
```

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

Simplifying Conditional Logic: *Replace Conditional with Polymorphism*

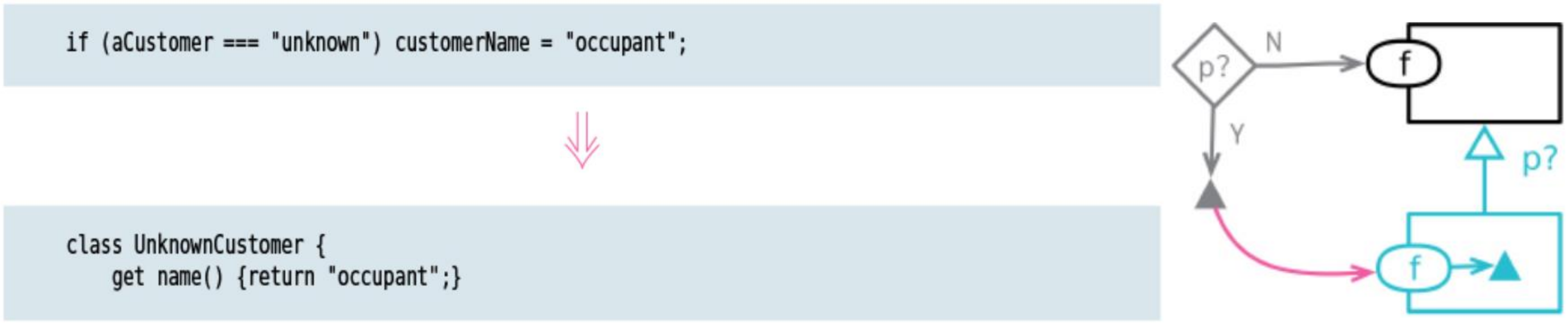
- **Replace Conditional with Polymorphism**
  - You have a conditional that chooses different behavior depending on the type of an object.
  - *Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.*

```
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 Logic: *Introduce Special Case*

- **Introduce Special Case**
  - Many users of a data structure check a specific value, and then do the same thing.
  - *Use the Special Case pattern to create a special-case element that captures all the common behavior.*



```
if (aCustomer === "unknown") customerName = "occupant";
```

⇓

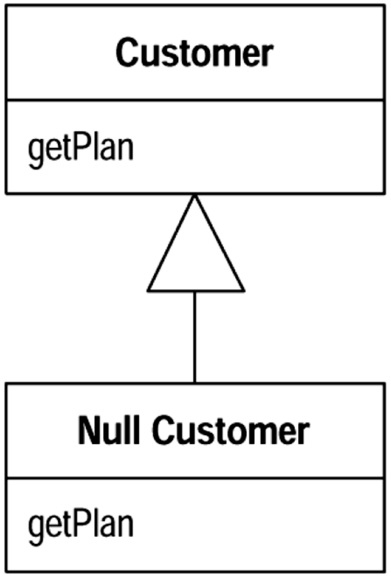
```
class UnknownCustomer {
    get name() {return "occupant";}
```

Simplifying Conditional Logic: *Introduce Special Case: Null Object*

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

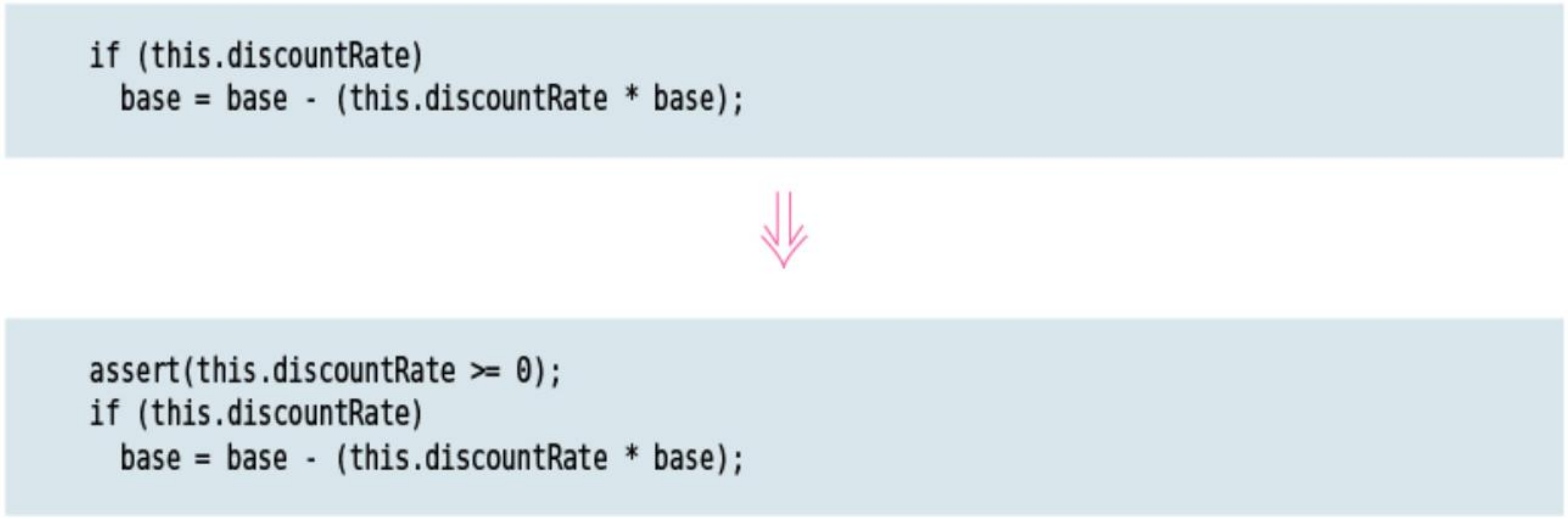
```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```

$\Rightarrow$



Simplifying Conditional Logic: *Introduce Assertion*

- **Introduce Assertion**
  - Sections of code work only if certain conditions are true. Such assumptions are not stated and can only be deduced by looking through the algorithm.
  - *Use assertions to state the conditions explicitly; failure of an assertion indicates a programmer error.*



```
if (this.discountRate)
  base = base - (this.discountRate * base);
```

```
assert(this.discountRate >= 0);
if (this.discountRate)
  base = base - (this.discountRate * base);
```

Reference

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