

# Лекция 11: Рефакторинг

Юрий Литвинов  
[y.litvinov@spbu.ru](mailto:y.litvinov@spbu.ru)

12.05.2026

# Рефакторинг

- ▶ Изменение во внутренней структуре программного обеспечения, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения
  - ▶ Приведение кода в порядок
  - ▶ Изменение внешнего окружения
  - ▶ Борьба с деградацией архитектуры
  - ▶ Не связан с оптимизацией работы кода
- ▶ Декомпозиция на большое количество элементарных действий
- ▶ Альтернатива проектированию архитектуры?

# Зачем нужно делать рефакторинг

- ▶ Улучшение структуры программного обеспечения
- ▶ Облегчение понимания кода
- ▶ Помощь в поиске ошибок
- ▶ Ускорение разработки нового кода
- ▶ Изменение культуры кодирования

# Дублирование кода

- ▶ Копипаст суть ересь
- ▶ “Выделение метода”
- ▶ “Подъём метода”
- ▶ “Выделение класса”

# Длинный метод

- ▶ Короткие методы: понятность, переиспользуемость
- ▶ Усложняет чтение — требуется переключение контекстов
  - ▶ Давайте понятные имена
- ▶ Семантическое расстояние между что делает код и как
- ▶ “Выделение метода”
- ▶ Комментарии внутри тела метода — повод задуматься

# Большой класс

- ▶ Слишком много атрибутов
- ▶ Слишком много кода
- ▶ “Выделение класса” и “Выделение подкласса”

# Длинный список параметров

- ▶ Сложно
- ▶ Глобальные переменные нельзя, “временные поля” тоже нельзя
- ▶ “Выделение объекта-параметра”

# Слишком много ответственостей класса

- ▶ Single Responsibility Principle
- ▶ Несколько изменений затрагивают один класс
- ▶ Разделить класс на два (три, пять, и ещё один, чтобы связать их)

# «Стрельба дробью»

- ▶ Предыдущий “запах” наоборот — одно изменение затрагивает несколько классов
  - ▶ Легко пропустить важное изменение
- ▶ “Перемещение метода” и “Перемещение поля”
- ▶ “Встраивание класса”

# «Завистливые функции»

- ▶ Метод обращается к чужому классу чаще, чем к своему
- ▶ “Перемещение метода”
  - ▶ Плюс “Выделение метода”
- ▶ Исключение: паттерны “Стратегия” и “Посетитель”

# Группы данных

- ▶ Набор значений, которые используются вместе
- ▶ “Выделение класса”
  - ▶ Можно вынести туда содержательную функциональность
    - ▶ Найдя “Завистливые функции”

# Операторы типа switch

- ▶ Один и тот же switch в нескольких местах программы
  - ▶ Легко забыть поменять кого-то из них
- ▶ Коды типов
- ▶ Заменить на полиморфизм
  - ▶ “Выделение метода”, “Перемещение метода”
  - ▶ “Замена кода типа подклассами”
  - ▶ “Заменой кода типа состоянием/стратегией”
  - ▶ “Замена условного оператора полиморфизмом”
  - ▶ “Введение Null-объекта”

## «Ленивый класс»

- ▶ Ненужный класс усложняет сопровождение
- ▶ Результат рефакторинга, либо забота о будущем
- ▶ “Встраивание класса”

# Временное поле

- ▶ Атрибут, нужный только во время работы метода/передачи параметров
- ▶ “Выделение класса”, “Введение Null-объекта”
  - ▶ Улучшит инкапсуляцию

# Цепочки сообщений

- ▶ `object.getX().getY().getZ();`
- ▶ “Скрытие делегирования”

# «Неуместная близость»

- ▶ Чрезмерный доступ к состоянию/private по смыслу поведению другого класса
- ▶ Наследование
- ▶ “Перемещение метода”, “Перемещение поля”
- ▶ “Выделение класса”

# Классы данных

- ▶ Только поля, геттеры и сеттеры
- ▶ Не всегда плохо
- ▶ “Инкапсуляция поля”, “Инкапсуляция коллекции”
- ▶ “Перемещение метода”, “Выделение метода”, “Скрытие метода”

# Комментарии

- ▶ Код должен быть понятным и без комментариев
- ▶ Комментарии могут играть роль “дезодоранта”
- ▶ Комментарии нужны
  - ▶ Но должны пояснить, почему, а не как
- ▶ “Выделение метода”
- ▶ assert-ы

## Выделение метода (Extract Method)

```
void printOwing(double amount) {  
    printBanner();  
    // вывод деталей  
    System.out.println ("name: " + _name);  
    System.out.println ("amount " + amount);  
}
```



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

# Встраивание метода (Inline Method)

```
int getRating() {  
    return moreThanFiveLateDeliveries() ? 2 : 1;  
}
```

```
boolean moreThanFiveLateDeliveries() {  
    return _numberOfLateDeliveries > 5;  
}
```



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

## Введение поясняющей переменной

```
if ((platform.toUpperCase().indexOf("MAC") > -1)
    && (browser.toUpperCase().indexOf("IE") > -1)
    && wasInitialized() && resize > 0) {

    // do something
}
```



```
final boolean isMacOS = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
final boolean isResized = resize > 0;
```

```
if (isMacOS && isIEBrowser && wasInitialized() && isResized) {
    // do something
}
```

# Декомпозиция условного оператора (Decompose Conditional)

```
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);
```

# Расщепление временной переменной (Split Temporary Variable)

```
double temp = 2 * (_height + _width);
```

```
System.out.println(temp);
```

```
temp = _height * _width;
```

```
System.out.println(temp);
```



```
final double perimeter = 2 * (_height + _width);
```

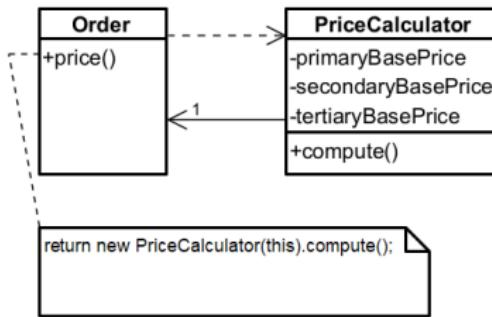
```
System.out.println (perimeter);
```

```
final double area = _height * _width;
```

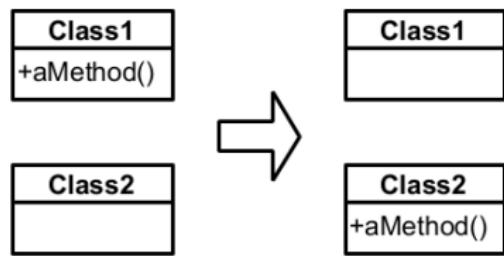
```
System.out.println (area);
```

# Замена метода объектом (Replace Method with Method Object)

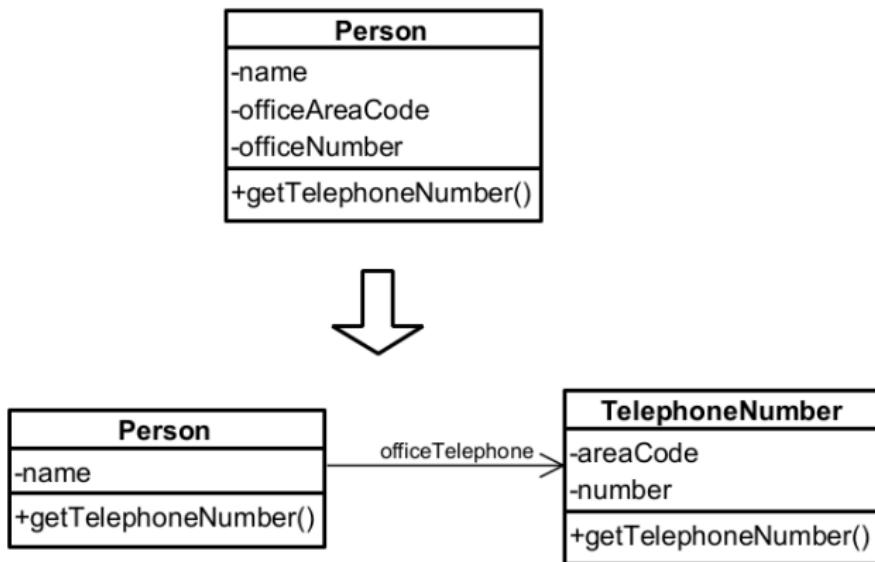
```
class Order {
    double price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // ДЛИННЫЕ ВЫЧИСЛЕНИЯ;
    }
}
```



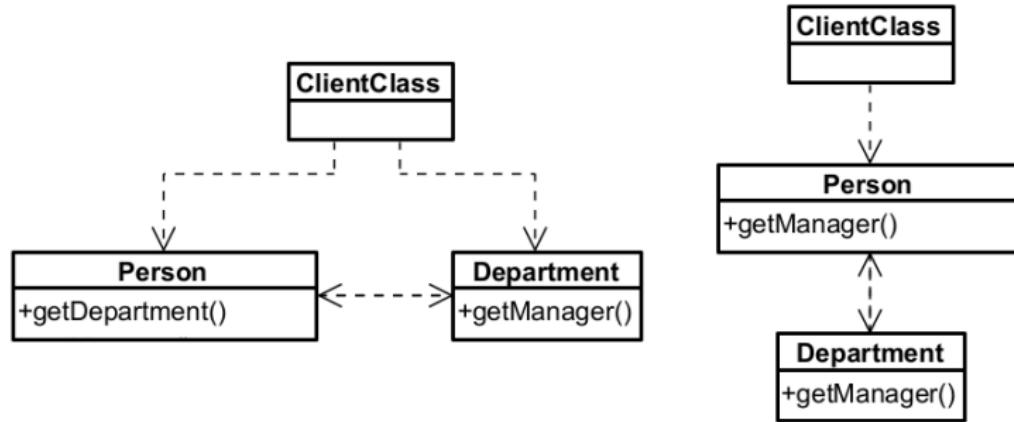
# Перемещение метода (Move Method)



# Выделение класса (Extract Class)



# Скрытие делегирования (Hide Delegate)



# Введение внешнего метода (Introduce Foreign Method)

```
Date newStart = new Date(previousEnd.getYear(),  
    previousEnd.getMonth(), previousEnd.getDate() + 1);
```



```
Date newStart = nextDay(previousEnd);  
static Date nextDay(Date arg) {  
    return new Date (arg.getYear(), arg.getMonth(), arg.getDate() + 1);  
}
```

## Самоинкапсуляция поля (Self Encapsulate Field)

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



```
private int _low, _high;  
  
int getLow() { return _low; }  
int getHigh() { return _high; }
```

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

## Замена магического числа именованной константой

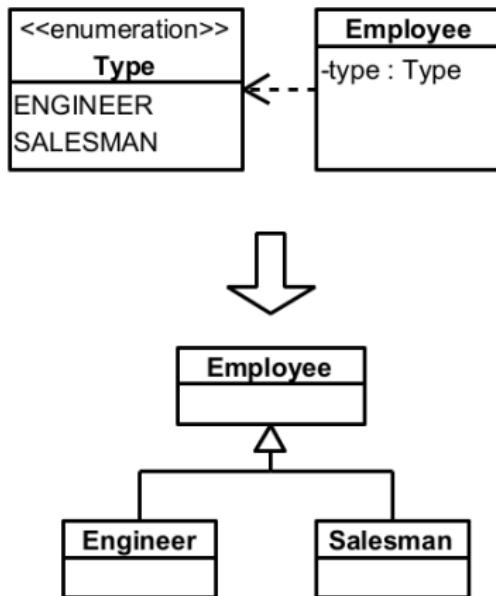
```
double potentialEnergy(double mass, double height) {  
    return mass * 9.81 * height;  
}
```



```
double potentialEnergy(double mass, double height) {  
    return mass * GRAVITATIONAL_CONSTANT * height;  
}
```

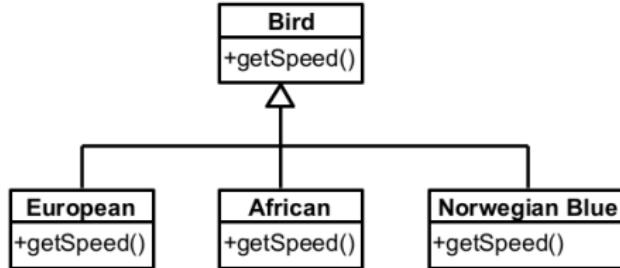
```
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

# Замена кода типа подклассами (Replace Type Code with Subclasses)



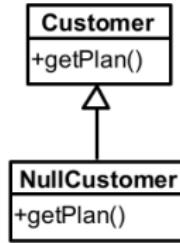
# Замена условного оператора полиморфизмом (Replace Conditional with Polymorphism)

```
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");
}
```

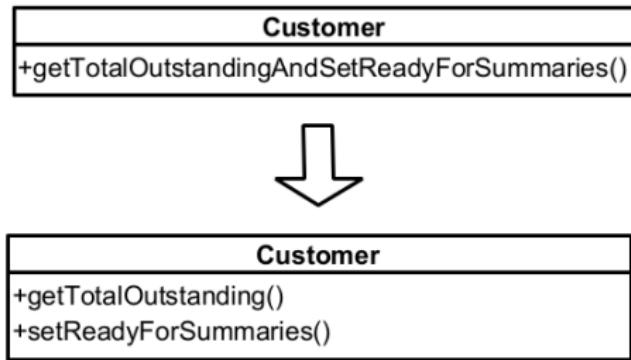


# Введение Null-объекта (Introduce Null Object)

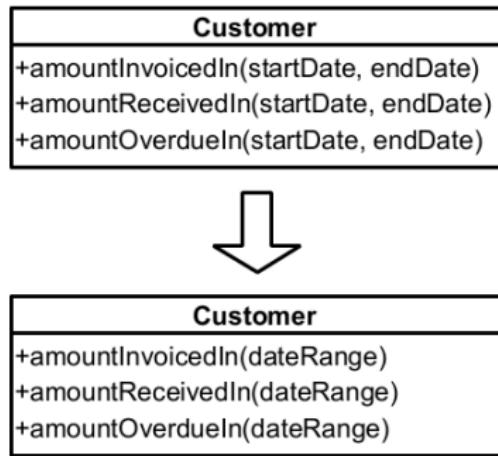
```
if (customer == null)  
    plan = BillingPlan.basic();  
else  
    plan = customer.getPlan();
```



# Разделение запроса и модификатора (Separate Query from Modifier)



# Введение объекта-параметра (Introduce Parameter Object)



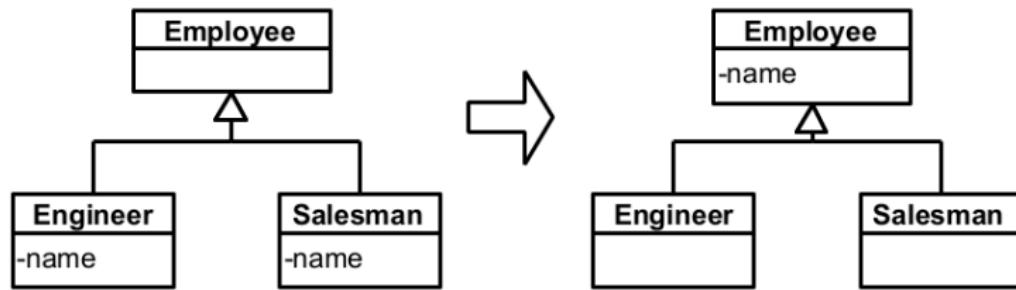
# Замена конструктора фабричным методом (Replace Constructor with Factory Method)

```
Employee(int type) {  
    _type = type;  
}
```

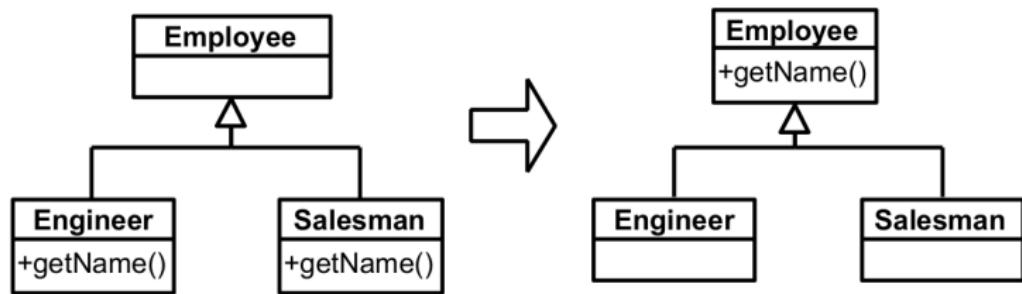


```
static Employee create(int type) {  
    return new Employee(type);  
}
```

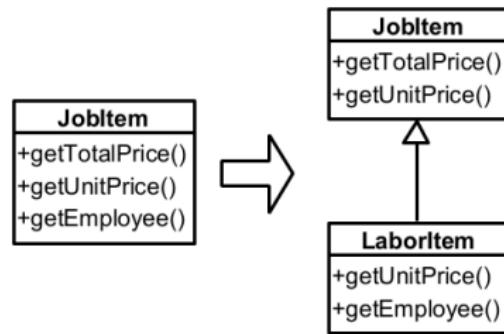
# Подъем поля (Pull Up Field)



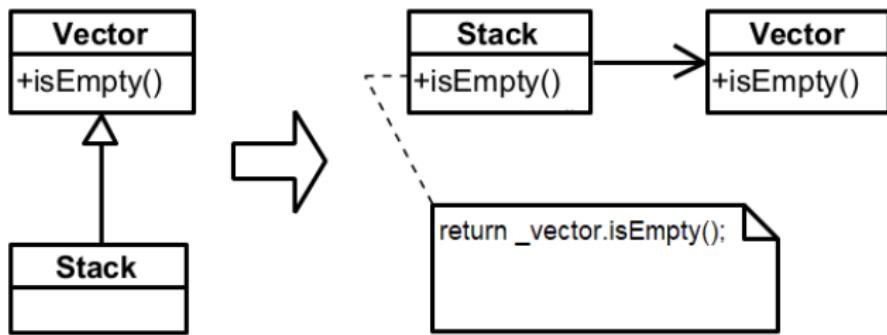
# Подъем метода (Pull Up Method)



# Выделение подкласса (Extract Subclass)



# Замена наследования делегированием (Replace Inheritance with Delegation)



# Когда имеет смысл делать рефакторинг

- ▶ Отдельное планирование
- ▶ “Правило трёх ударов”
- ▶ При добавлении новой функциональности
- ▶ При исправлении ошибок
- ▶ При изучении и ревью кода
- ▶ При устранении технического долга

# Проблемы при проведении рефакторинга

- ▶ Работа с данными
- ▶ Изменение интерфейсов сущностей
  - ▶ Сохранение старого интерфейса
  - ▶ Методы-обёртки
  - ▶ Работа с исключениями
- ▶ Глобальные изменения архитектуры
- ▶ Рефакторинг и оптимизация

# Когда рефакторинг делать точно не стоит

- ▶ Код проще переписать с нуля
- ▶ Близость дедлайнов
- ▶ Нет юнит-тестов

# Что почитать

