

ОБЪЕКТНО- ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ



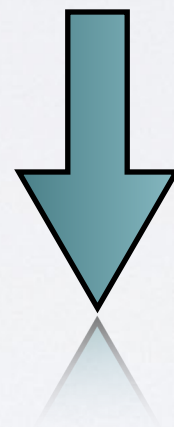
ПРИЕМЫ РЕФАКТОРИНГА

- Составление методов.
- Перемещение функций между объектами.
- Организация данных.
- Упрощение условных выражений.
- Упрощение вызовов методов.
- Решение задач обобщения.

ПЕРЕМЕЩЕНИЕ ФУНКЦИЙ МЕЖДУ ОБЪЕКТАМИ

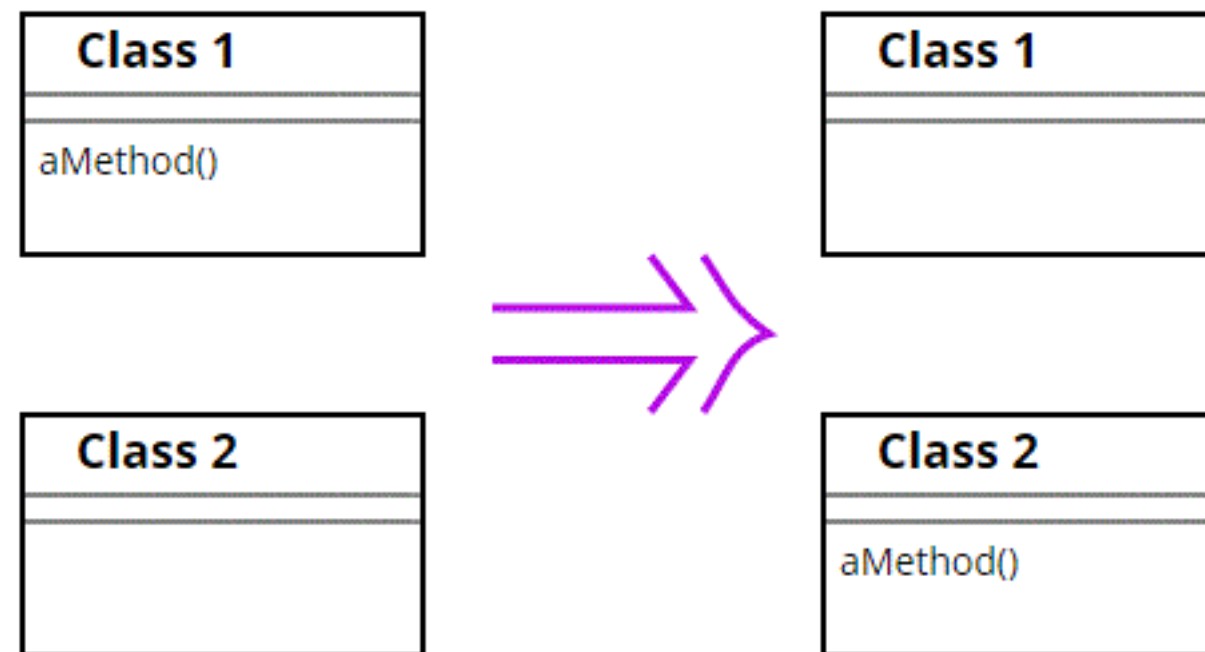
ПЕРЕМЕЩЕНИЕ МЕТОДА (MOVE METHOD)

Метод используется в другом классе больше, чем в собственном.



Перенесите метод в другой класс. Оригинальный метод либо удаляется, либо является обращением к новому методу.

ПЕРЕМЕЩЕНИЕ МЕТОДА (MOVE METHOD)

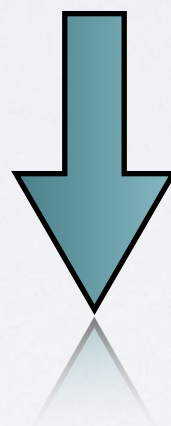


ПРИЧИНЫ РЕФАКТОРИНГА

- **Переместить метод в класс, который содержит данные, с которыми работает этот метод.**
- **Убрать или уменьшить зависимость классов.**

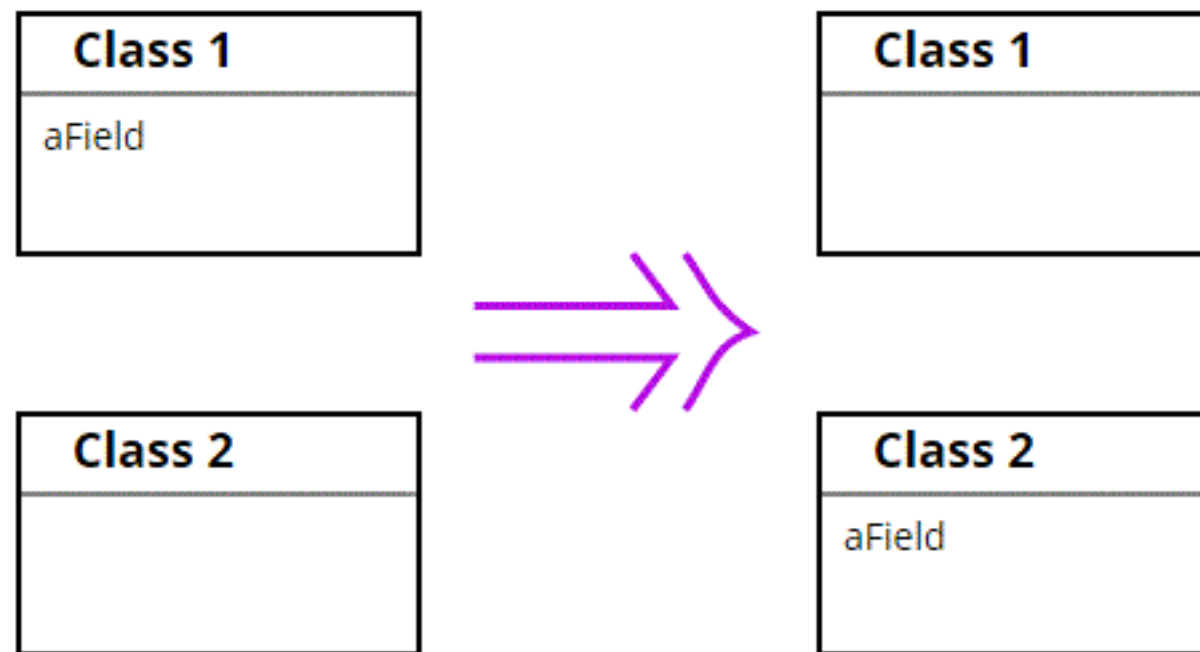
ПЕРЕМЕЩЕНИЕ ПОЛЯ (MOVE FIELD)

Поле используется в другом классе больше, чем в собственном.



Создайте поле в новом классе и перенаправьте к нему всех пользователей старого поля.

ПЕРЕМЕЩЕНИЕ ПОЛЯ (MOVE FIELD)

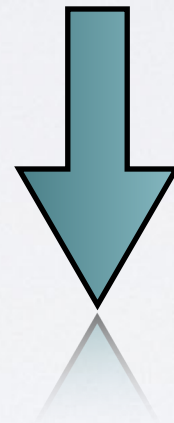


ПРИЧИНЫ РЕФАКТОРИНГА

- Как часть рефакторинга извлечение класса.
- Поле должно быть там, где находятся методы, которые его используют (либо, где их больше).

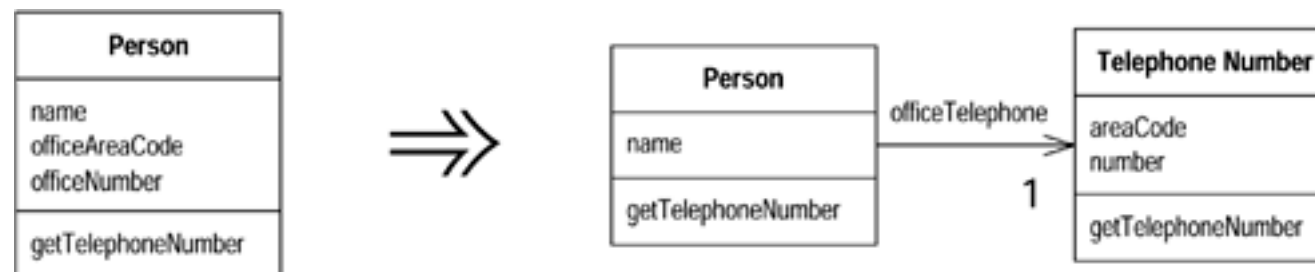
ИЗВЛЕЧЕНИЕ КЛАССА (EXTRACT CLASS)

Один класс работает за двоих.



Создайте новый класс, переместите в него поля и методы, отвечающие за определённую функциональность.

ИЗВЛЕЧЕНИЕ КЛАССА (EXTRACT CLASS)



ПРИЧИНЫ РЕФАКТОРИНГА

**Не соответствие принципу единственной
ответственности.**

ДОСТОИНСТВА

Соблюдение принципа единственной ответственности:

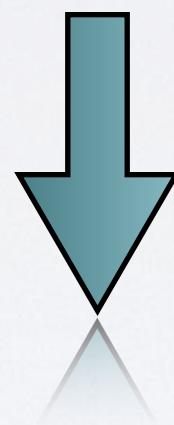
- Класс более очевиден и понятен.
- Класс более надежен и устойчив к изменениям.

НЕДОСТАТКИ

Если при проведении этого рефакторинга вы перестараетесь, придётся прибегнуть к встраиванию класса.

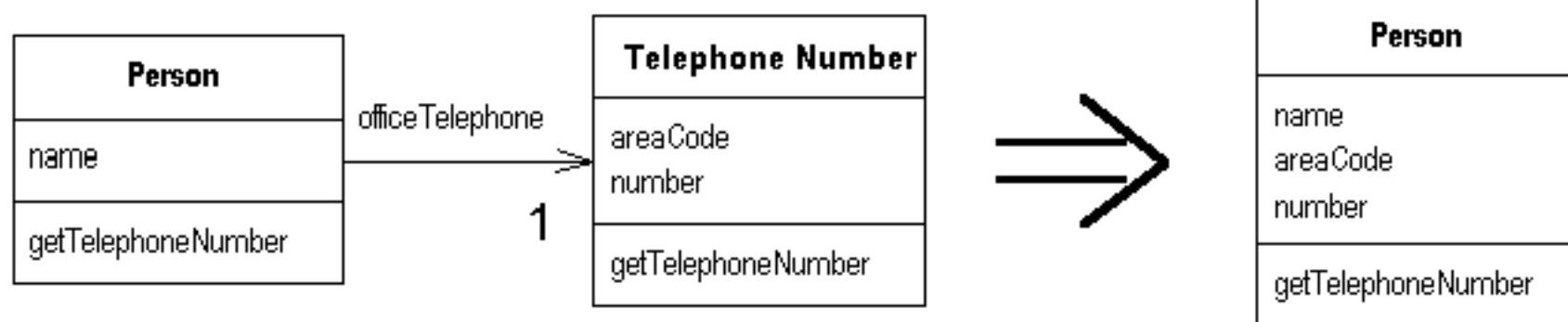
ВСТРАИВАНИЕ КЛАССА (INLINE CLASS)

Класс почти ничего не делает, ни за что не отвечает, и никакой ответственности для этого класса не планируется.



Переместите все фичи из описанного класса в другой.

ВСТРАИВАНИЕ КЛАССА (INLINE CLASS)



ПРИЧИНЫ РЕФАКТОРИНГА

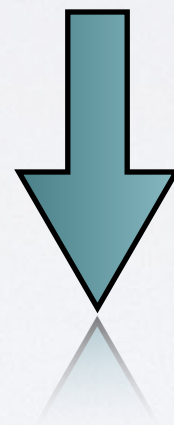
Часто этот рефакторинг оказывается следствием недавнего «переселения» части фич класса в другие, после чего от исходного класса мало что осталось.

ДОСТОИНСТВА

Меньше бесполезных классов — больше свободной оперативной памяти, в том числе, и у вас в голове.

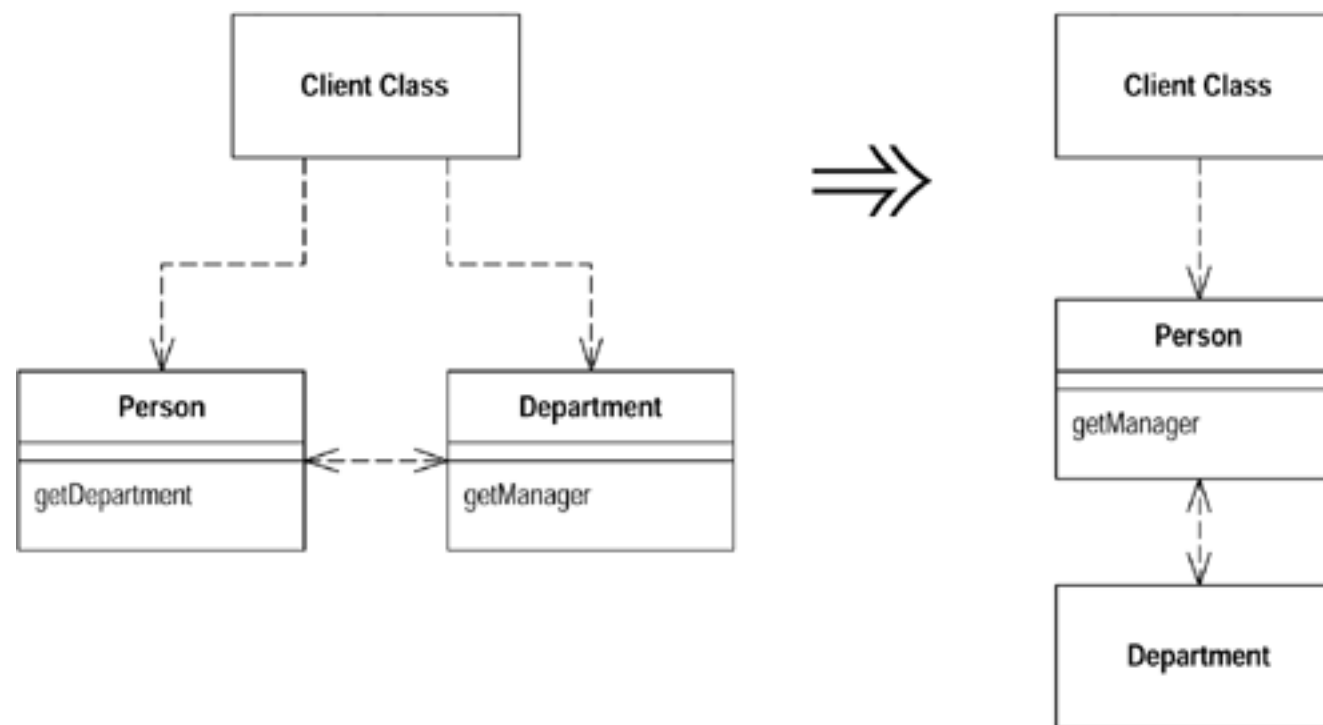
СОКРЫТИЕ ДЕЛЕГИРОВАНИЯ (HIDE DELEGATE)

Клиент получает объект В из поля или метода объекта А. Затем клиент вызывает какой-то метод объекта В.



Создайте новый метод в классе А, который бы делегировал вызов объекту В.

СОКРЫТИЕ ДЕЛЕГИРОВАНИЯ (HIDE DELEGATE)



ПРИЧИНЫ РЕФАКТОРИНГА

Клиент связан с навигацией по структуре классов. Любые изменения промежуточных связей означают необходимость модификации клиента.

ДОСТОИНСТВА

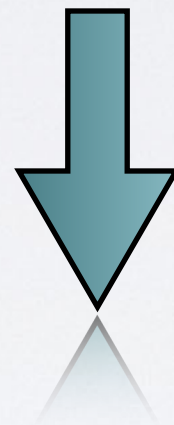
Скрывает делегирование от клиента. Чем меньше клиентский код знает подробностей о связях между объектами, тем проще будет впоследствии вносить изменения в программу.

НЕДОСТАТКИ

Если требуется создать слишком много делегирующих методов, класс-сервер (класс, к которому клиент имеет непосредственный доступ) рискует превратиться в лишнее промежуточное звено.

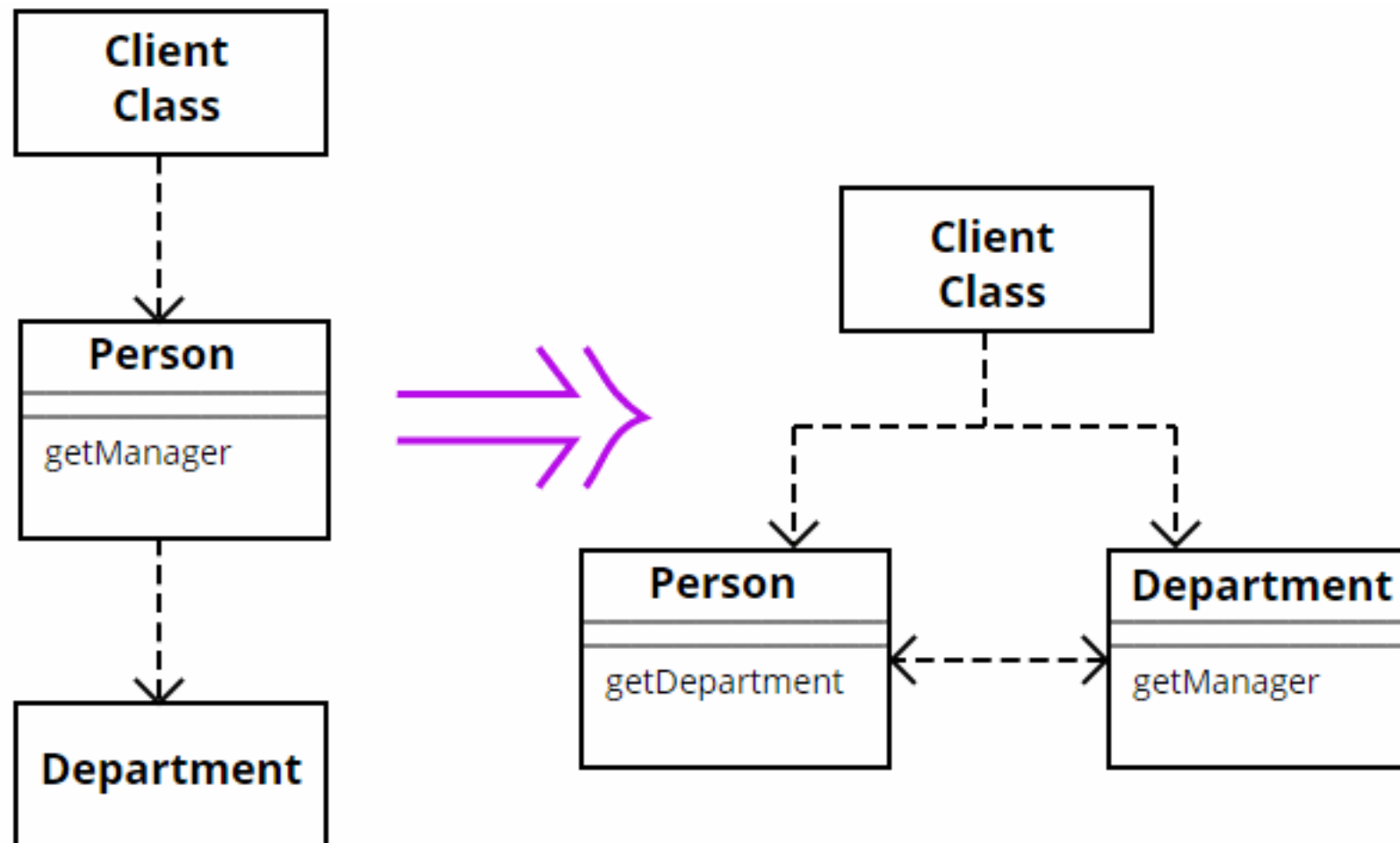
УДАЛЕНИЕ ПОСРЕДНИКА (REMOVE MIDDLE MAN)

Класс имеет слишком много методов, которые просто делегируют работу другим объектам.



Удалите эти методы и заставьте клиента вызывать конечные методы напрямую.

УДАЛЕНИЕ ПОСРЕДНИКА (REMOVE MIDDLE MAN)



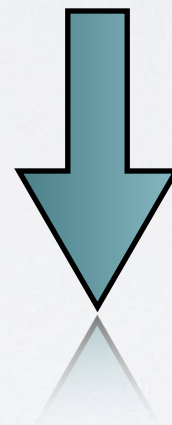
ПРИЧИНЫ РЕФАКТОРИНГА

- Класс-сервер ничего не делает сам по себе, создавая бесполезную сложность. Нужен ли этот класс вообще?
- Новая фича в делегате приводит к необходимости создавать новый делегирующий метод в классе-сервере. Накладно при большом количестве изменений.

ОРГАНИЗАЦИЯ ДАННЫХ

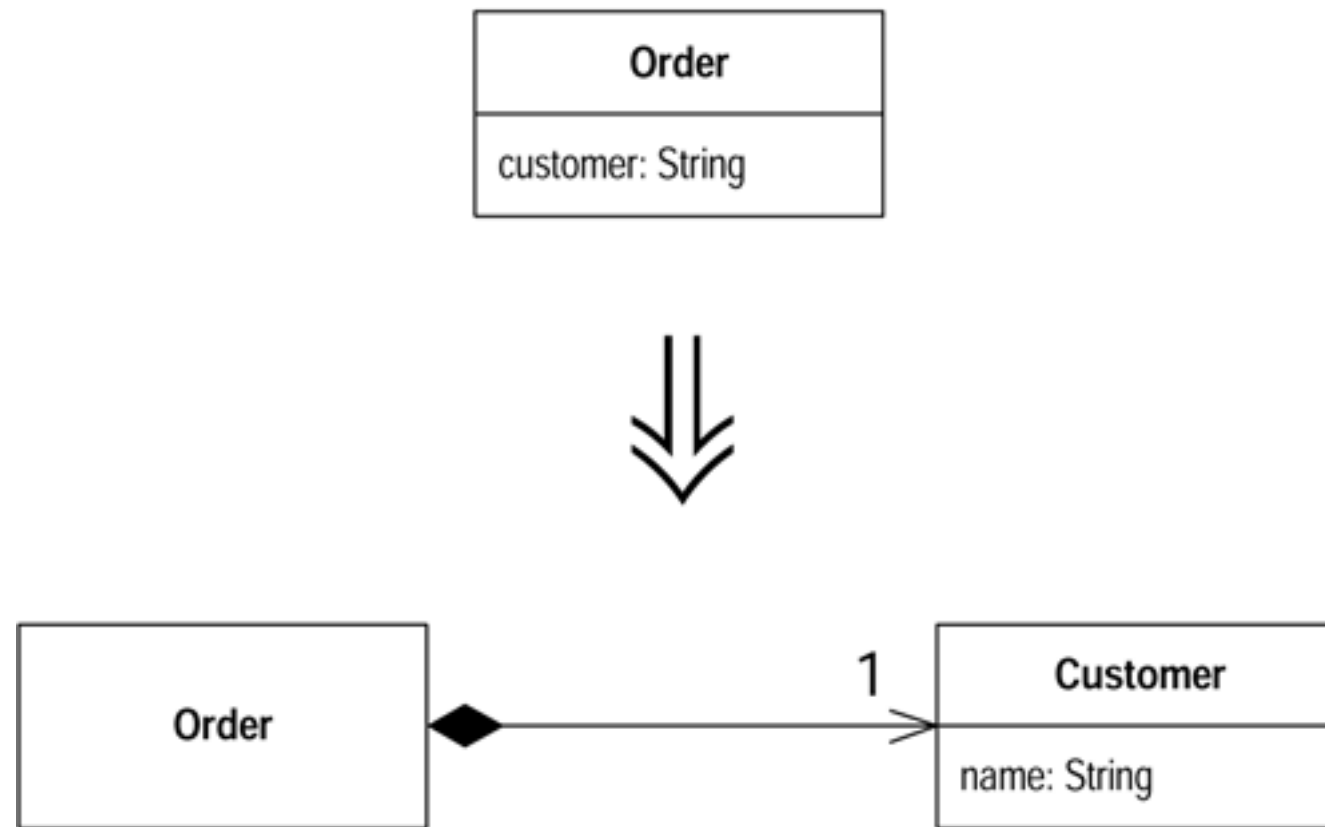
ЗАМЕНА ПРОСТОГО ПОЛЯ ОБЪЕКТОМ (REPLACE DATA VALUE WITH OBJECT)

В классе есть поле простого типа, у которого имеется своё поведение и связанные данные.



Создайте новый класс, поместите в него старое поле и его поведения, храните объект этого класса в исходном классе.

ЗАМЕНА ПРОСТОГО ПОЛЯ ОБЪЕКТОМ (REPLACE DATA VALUE WITH OBJECT)



ПРИЧИНЫ РЕФАКТОРИНГА

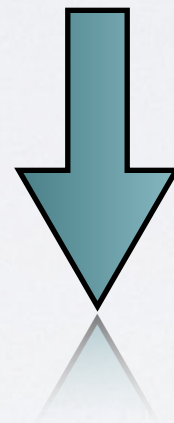
Поле класса примитивного типа перестало быть простым и обзавелось связанными данными и поведением.

ДОСТОИНСТВА

Улучшает связность внутри классов. Данные и поведения этих данных находятся в одном классе.

ЗАМЕНА ПОЛЯ-МАССИВА ОБЪЕКТОМ (REPLACE ARRAY WITH OBJECT)


Есть массив, в котором хранятся разнотипные данные.



**Замените массив объектом, который будет иметь
отдельные поля для каждого элемента.**

ЗАМЕНА ПОЛЯ-МАССИВА ОБЪЕКТОМ (REPLACE ARRAY WITH OBJECT)

```
std::string arr[2] = { "Liverpool", "15" };
```



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

ПРИЧИНЫ РЕФАКТОРИНГА

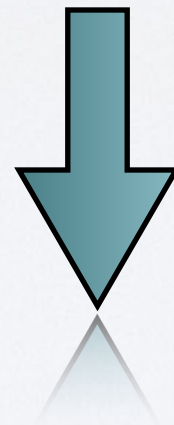
Массив подходит для хранения однотипных данных. В случае разнотипных данных можно ошибиться номером ячейки.

ДОСТОИНСТВА

- В образовавшийся класс можно переместить все связанные поведения.
- Поля класса гораздо проще документировать, чем ячейки массива.

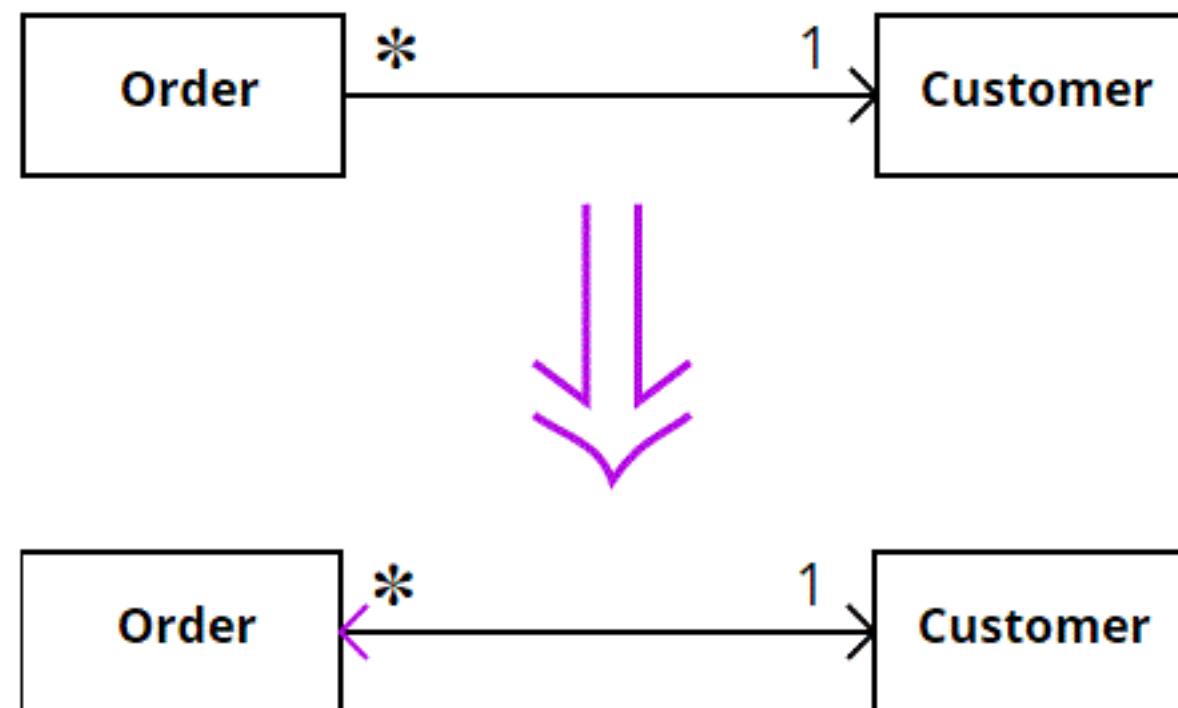
ЗАМЕНА ОДНОНАПРАВЛЕННОЙ СВЯЗИ ДВУНАПРАВЛЕННОЙ (CHANGE UNIDIRECTIONAL ASSOCIATION TO BIDIRECTIONAL)

Есть два класса, которым нужно использовать фичи друг друга, но между ними существует только односторонняя связь.



Добавьте недостающую связь в класс, в котором она отсутствует.

ЗАМЕНА ОДНОНАПРАВЛЕННОЙ СВЯЗИ ДВУНАПРАВЛЕННОЙ (CHANGE UNIDIRECTIONAL ASSOCIATION TO BIDIRECTIONAL)



ПРИЧИНЫ РЕФАКТОРИНГА

Между классами изначально была односторонняя связь.
Однако с течением времени клиентскому коду
потребовался доступ в обе стороны этой связи.

ДОСТОИНСТВА

Если в классе возникает необходимость в обратной связи, её можно попросту вычислить.

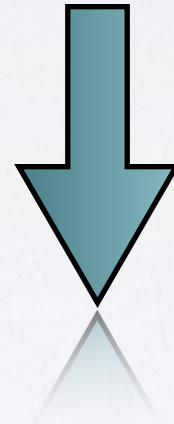
Однако, если такие вычисления оказываются довольно сложными, лучше хранить обратную связь.

НЕДОСТАТКИ

- Двусторонние связи гораздо сложнее в реализации и поддержке, чем односторонние.
- Двусторонние связи делают классы зависимыми друг от друга. При односторонней связи один из них можно было использовать отдельно от другого.

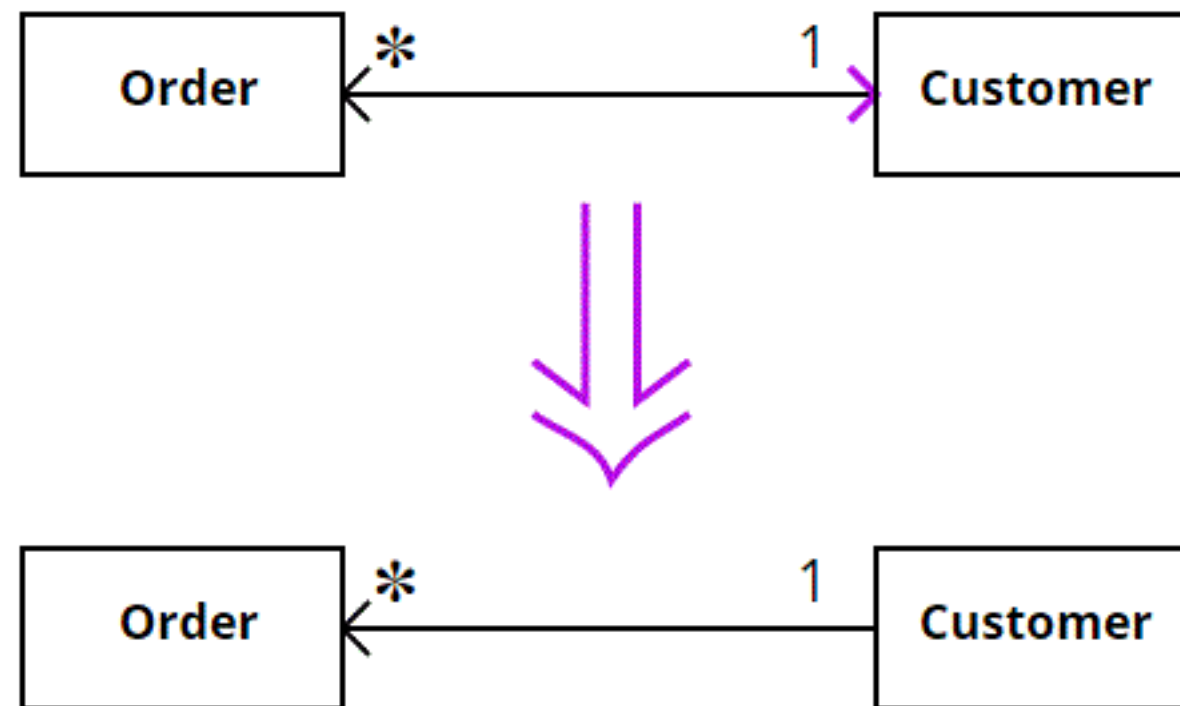
ЗАМЕНА ДВУХНАПРАВЛЕННОЙ СВЯЗИ ОДНОНАПРАВЛЕННОЙ (CHANGE BIDIRECTIONAL ASSOCIATION TO UNIDIRECTIONAL)

Есть двухсторонняя связь между классами, но один из классов больше не использует фичи другого.



Уберите неиспользуемую связь.

ЗАМЕНА ДВУХНАПРАВЛЕННОЙ СВЯЗИ ОДНОНАПРАВЛЕННОЙ (CHANGE BIDIRECTIONAL ASSOCIATION TO UNIDIRECTIONAL)



ПРИЧИНЫ РЕФАКТОРИНГА

Двустороннюю связь, как правило, сложнее поддерживать, чем одностороннюю.

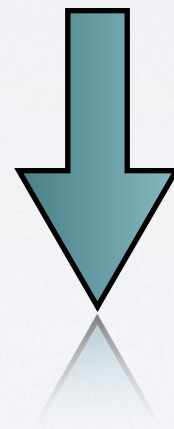
Существует проблема тесной зависимости между классами.

ДОСТОИНСТВА

- Упрощает код класса, которому не нужна связь.
Меньше кода — проще поддержка.
- Уменьшает зависимость между классами.

ЗАМЕНА МАГИЧЕСКОГО ЧИСЛА СИМВОЛЬНОЙ КОНСТАНТОЙ (REPLACE MAGIC NUMBER WITH SYMBOLIC CONSTANT)

В коде используется число, которое несёт какой-то определённый смысл.



**Замените это число константой с человеко-
читаемым названием, объясняющим смысл этого
числа.**

ПРИЧИНЫ РЕФАКТОРИНГА

Магические числа затрудняют понимание программы и усложняют её рефакторинг.

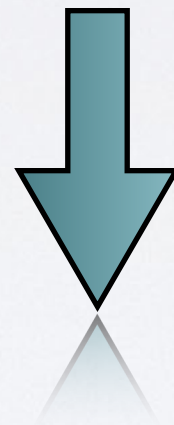
Дополнительные сложности возникают, когда нужно поменять определённое магическое число.

ДОСТОИНСТВА

- Символьная константа может служить живой документацией смысла значения, которое в ней хранится.
- Значение константы намного проще заменить.
- Убирает дублирование использования числа или строки по всему коду.

ИНКАПСУЛЯЦИЯ ПОЛЯ (ENCAPSULATE FIELD)

Есть публичное поле.



Сделайте поле приватным и создайте для него
методы доступа.

ПРИЧИНЫ РЕФАКТОРИНГА

- Соккрытие данных.
- При публичном доступе данные отделяются от поведений, связанных с этими данными.

ДОСТОИНСТВА

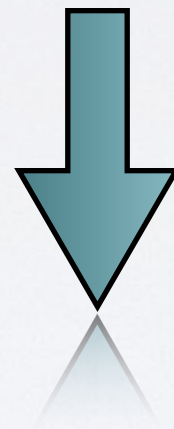
- Данные и поведение находятся в одном месте, что проще поддерживать и развивать.
- Можно производить какие-то сложные операции, связанные с доступом к полям объекта.

КОГДА НЕЛЬЗЯ ПРИМЕНЯТЬ

Когда инкапсуляция полей нежелательна из соображений, связанных с повышением производительности. Например, `class Point`.

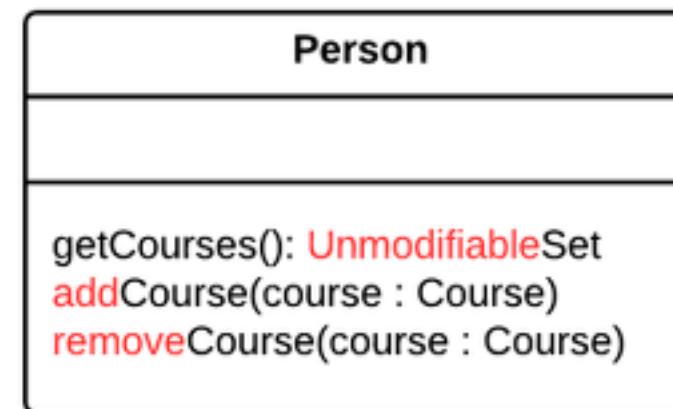
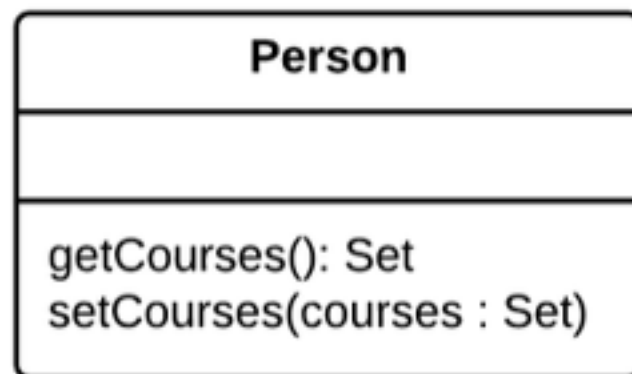
ИНКАПСУЛЯЦИЯ КОЛЛЕКЦИИ (ENCAPSULATE COLLECTION)

Класс содержит поле-коллекцию и простой геттер и сеттер для работы с этой коллекцией.



Сделайте возвращаемое геттером значение доступным только для чтения и создайте методы добавления/удаления элементов этой коллекции.

ИНКАПСУЛЯЦИЯ КОЛЛЕКЦИИ (ENCAPSULATE COLLECTION)



ПРИЧИНЫ РЕФАКТОРИНГА

Метод получения не должен возвращать сам объект коллекции, потому что это позволило бы клиентам изменять содержимое коллекции без ведома владеющего ею класса.

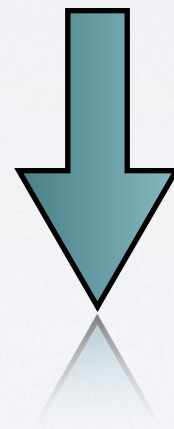
ДОСТОИНСТВА

- Поле коллекции инкапсулировано внутри класса.
- Можно предоставить более удобный интерфейс взаимодействия с коллекцией.
- Можно ограничить доступ к нежелательным стандартным методам коллекции.

УПРОЩЕНИЕ УСЛОВНЫХ ВЫРАЖЕНИЙ

РАЗБИЕНИЕ УСЛОВНОГО ОПЕРАТОРА (DECOMPOSE CONDITIONAL)

Есть сложный условный оператор if/else или switch.




**Выделите в отдельные методы все сложные части
оператора.**

РАЗБИЕНИЕ УСЛОВНОГО ОПЕРАТОРА (DECOMPOSE CONDITIONAL)

```
if (date.before(SUMMER_START) || date.after(SUMMER_END))  
{  
    charge = quantity * winterRate + winterServiceCharge;  
}  
else {  
    charge = quantity * summerRate;  
}
```

```
if (isSummer(date)) {  
    charge = summerCharge(quantity);  
}  
else {  
    charge = winterCharge(quantity);  
}
```



ПРИЧИНЫ РЕФАКТОРИНГА

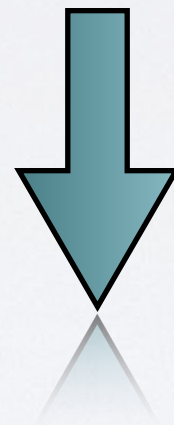
Чем длиннее кусок кода, тем сложнее понять, что он делает.

ДОСТОИНСТВА

Извлекая код условного оператора в методы с понятным названием, улучшается читабельность.

ОБЪЕДИНЕНИЕ УСЛОВНЫХ ОПЕРАТОРОВ (CONSOLIDATE CONDITIONAL EXPRESSION)

Есть несколько условных операторов, ведущих к одинаковому результату или действию.




Объедините все условия в одном условном операторе.

ОБЪЕДИНЕНИЕ УСЛОВНЫХ ОПЕРАТОРОВ (CONSOLIDATE CONDITIONAL EXPRESSION)

```
double disabilityAmount() {  
    if (seniority < 2) {  
        return 0;  
    }  
    if (monthsDisabled > 12) {  
        return 0;  
    }  
    if (isPartTime) {  
        return 0;  
    }  
    //...  
}
```

```
double disabilityAmount() {  
    if (isNotEligableForDisability()) {  
        return 0;  
    }  
    //...  
}
```



ПРИЧИНЫ РЕФАКТОРИНГА

Код содержит множество чередующихся операторов, которые выполняют одинаковые действия.

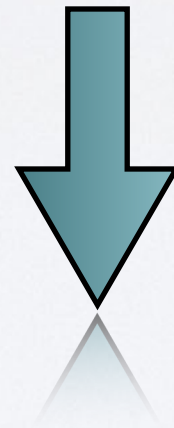
Главная цель объединения операторов — извлечь условие оператора в отдельный метод, упростив его понимание.

ДОСТОИНСТВА

- Убирает дублирование управляющего кода.
- Объединив все операторы в одном, вы позволяете выделить это сложное условие в новый метод с названием, отражающим суть этого выражения.

ОБЪЕДИНЕНИЕ ДУБЛИРУЮЩИХСЯ ФРАГМЕНТОВ В УСЛОВНЫХ ОПЕРАТОРАХ (CONSOLIDATE DUPLICATE CONDITIONAL FRAGMENTS)

**Одинаковый фрагмент кода находится во всех
ветках условного оператора.**



Вынесите его за рамки оператора.

ОБЪЕДИНЕНИЕ ДУБЛИРУЮЩИХСЯ ФРАГМЕНТОВ В УСЛОВНЫХ ОПЕРАТОРАХ (CONSOLIDATE DUPLICATE CONDITIONAL FRAGMENTS)

```
if (isSpecialDeal()) {  
    total = price * 0.95;  
    send();  
}  
else {  
    total = price * 0.98;  
    send();  
}
```

```
if (isSpecialDeal()) {  
    total = price * 0.95;  
}  
else {  
    total = price * 0.98;  
}  
send();
```



ПРИЧИНЫ РЕФАКТОРИНГА

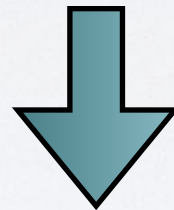
**Дублирующий код находится внутри всех веток
условного оператора, по причине результатом эволюции
кода внутри веток оператора.**

ДОСТОИНСТВА

Убирает дублирование кода.

ЗАМЕНА ВЛОЖЕННЫХ УСЛОВНЫХ ОПЕРАТОРОВ ГРАНИЧНЫМ ОПЕРАТОРОМ (REPLACE NESTED CONDITIONAL WITH GUARD CLAUSES)

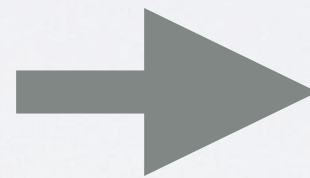
**Есть группа вложенных условных операторов,
среди которых сложно выделить нормальный ход
выполнения кода.**



**Выделите все проверки специальных или граничных
случаев выполнения в отдельные условия и
поместите их перед основными проверками.**

ЗАМЕНА ВЛОЖЕННЫХ УСЛОВНЫХ ОПЕРАТОРОВ ГРАНИЧНЫМ ОПЕРАТОРОМ (REPLACE NESTED CONDITIONAL WITH GUARD CLAUSES)

```
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();  
}
```


ПРИЧИНЫ РЕФАКТОРИНГА

```
if () {  
    if () {  
        do {  
            if () {  
                if () {  
                    if () {  
                        ...  
                    }  
                }  
            }  
        }  
        ...  
    }  
    while ();  
    ...  
}  
else {  
    ...  
}  
}
```

«Условный оператор из ада»



КОНЕЦ ВТОРОЙ ЧАСТИ

```
// Ох, лучше бы вам не трогать
// этот кусок архитектуры
int DoIt() {
    int x1 = DoSomeMagic1();
    int x2 = DoSomeMagic2();
    int x3 = DoSomeMagic3();
    int x4 = DoSomeMagic4();
    int x5 = DoSomeMagic5();
    return x1 + x2 + x3 + x4 + x5;
}
```