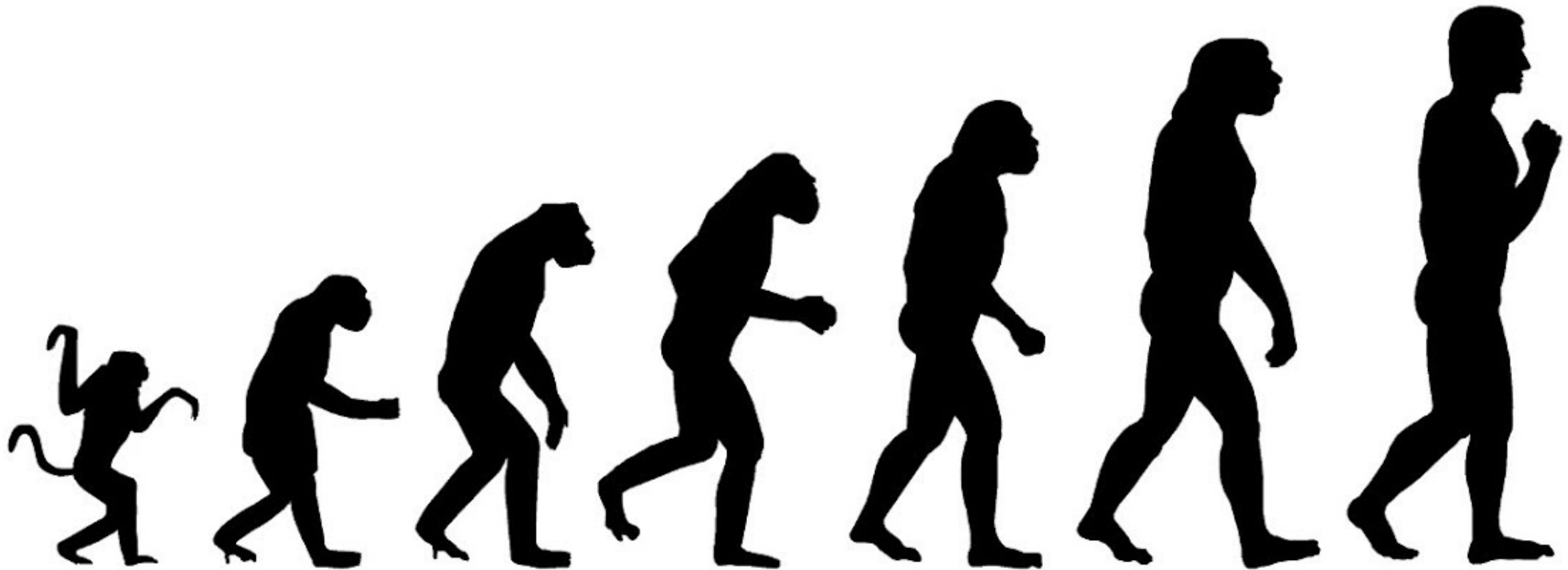


ОБЪЕКТНО- ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ



РЕФАКТОРИНГ



Refactoring

Improving the Design of Existing Code

ЧТО ТАКОЕ РЕФАКТОРИНГ?

Рефакторинг — это контролируемый процесс улучшения кода, без написания новой функциональности. Результат рефакторинга — это чистый код и простой дизайн.

Чистый код — это код, который просто читать, понимать и поддерживать. Чистый код улучшает предсказуемость разработки и повышает качество продукта.

ПРОЦЕСС РЕФАКТОРИНГА



Пошаговые изменения, сопровождаемые частыми запусками тестов — это то, что делает процесс рефакторинга эффективным и безопасным.

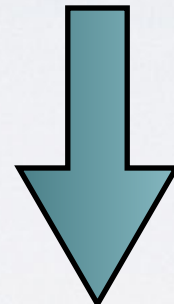
ПРИЕМЫ РЕФАКТОРИНГА

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

СОСТАВЛЕНИЕ МЕТОДОВ

ВЫДЕЛЕНИЕ МЕТОДА (EXTRACT METHOD)

Есть фрагмент кода, который можно сгруппировать.



Преобразуйте фрагмент кода в метод, название которого объясняет его назначение.

ВЫДЕЛЕНИЕ МЕТОДА (EXTRACT METHOD)

```
void printOwing(const std::string& name, const double amount){  
    printBanner();  
    // вывод деталей  
    std::cout << "name: " << name << std::endl;  
    std::cout << "amount: " << amount << std::endl;  
}
```



```
void printOwing(const std::string& name, const double amount){  
    printBanner();  
    printDetails(name, amount);  
}  
  
void printDetails(const std::string& name, const double amount){  
    std::cout << "name: " << name << std::endl;  
    std::cout << "amount: " << amount << std::endl;  
}
```


ПРИЧИНЫ РЕФАКТОРИНГА

Если метод кажется слишком длинным, или код требует комментариев, объясняющих его назначение, тогда преобразуйте данный фрагмент кода в отдельный метод.

Чем больше строк кода в методе, тем сложнее разобраться в том, что он делает.

ТЕХНИКА

1. Создайте новый метод и назовите его соответственно назначению метода (тому, **что, а не как** он делает).
2. Замените фрагмент кода на вызов нового метода.
3. Если переменные объявлены перед выделенным фрагментом, значит, их следует передать в параметры нового метода. Иногда от таких переменных проще избавиться с помощью **замены переменных вызовом метода**.
4. Если локальная переменная изменяется в выделенном коде и используется дальше в основном методе, то значение этой переменной следует вернуть в основной метод, чтобы ничего не сломать.

ТЕХНИКА



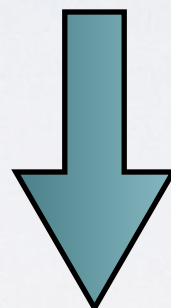
Если локальные переменные и параметры препятствуют выделению метода, можно применить замену временной переменной вызовом метода, замену параметров объектом и передачу всего объекта.

ДОСТОИНСТВА

1. Улучшает читабельность кода (при условии, что программист **ответственно** подходит к наименованию методов и функций).
2. Убирает дублирование кода.
3. Изолирует независимые части кода, уменьшая вероятность ошибок (например, по вине переопределения не той переменной).

ВСТРАИВАНИЕ МЕТОДА (INLINE METHOD)

Есть тело метода очевиднее самого метода.



**Замените вызовы метода его содержимым и
удалите сам метод.**

ВСТРАИВАНИЕ МЕТОДА (INLINE METHOD)

```
class PizzaDelivery
{
    int GetRating(){
        return MoreThanFiveLateDeliveries() ? 2 : 1;
    }

    bool MoreThanFiveLateDeliveries(){
        return numberOfLateDeliveries > 5;
    }
};
```



```
class PizzaDelivery
{
    int GetRating(){
        return numberOfLateDeliveries > 5 ? 2 : 1;
    }
};
```


ПРИЧИНЫ РЕФАКТОРИНГА

Если метод просто выполняет делегирование другому методу, и во всем этом делегировании можно просто заблудиться.

Зачастую методы не бывают слишком короткими изначально, а становятся такими в результате изменений в программе. Поэтому не стоит бояться избавляться от ставших ненужными методов.

ТЕХНИКА

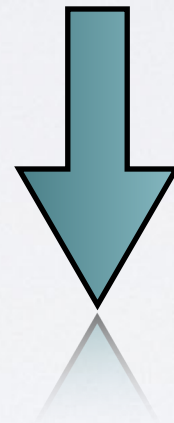
1. Убедитесь, что метод не является полиморфным.
2. Избегайте встраивания, если есть подклассы, перегружающие метод; они не смогут перегрузить отсутствующий метод.
3. Найдите все вызовы метода. Замените эти вызовы содержимым метода.
4. Удалите метод.

ДОСТОИНСТВА

Минимизируя количество бесполезных методов, мы уменьшаем общую сложность кода.

ВСТРАИВАНИЕ ПЕРЕМЕННОЙ (INLINE TEMP)

Есть временная переменная, которой присваивается результат простого выражения (и больше ничего).



Замените обращения к переменной этим выражением.

ВСТРАИВАНИЕ ПЕРЕМЕННОЙ (INLINE TEMP)

```
bool HasDiscount(Order order)
{
    double basePrice = order.BasePrice();
    return basePrice > 1000;
}
```



```
bool HasDiscount(Order order)
{
    return order.BasePrice() > 1000;
}
```

ПРИЧИНЫ РЕФАКТОРИНГА

Встраивание локальной переменной почти всегда используется как часть замены переменной вызовом метода или для облегчения выделения метода.

ТЕХНИКА

1. Добавьте к переменной модификатором `const`, если этого еще не сделано, и скомпилируйте код. Так вы убедитесь, что значение этой переменной присваивается действительно один раз.
2. Найдите все места, где используется переменная, и замените их выражением, которое ей присваивалось.
3. Удалите объявление переменной и строку присваивания ей значения.

ДОСТОИНСТВА

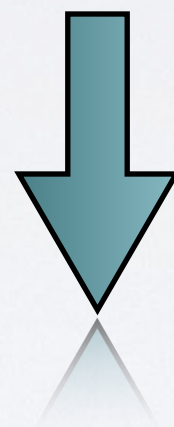
Сам по себе данный рефакторинг не несёт почти никакой пользы. Тем не менее, если переменной присваивается результат выполнения какого-то метода, у вас есть возможность немного улучшить читабельность программы, избавившись от лишней переменной.

НЕДОСТАТКИ

Иногда временные переменные служат для кеширования, то есть сохранения результата какой-то дорогостоящей операции, который будет в ходе работы использован несколько раз повторно. Перед тем как осуществлять рефакторинг, убедитесь, что в вашем случае это не так.

РАСЩЕПЛЕНИЕ ПЕРЕМЕННОЙ (SPLIT TEMPORARY VARIABLE)

Есть локальная переменная, которая используется для хранения разных промежуточных значений внутри метода (за исключением переменных циклов).



Используйте разные переменные для разных значений. Каждая переменная должна отвечать только за одну определённую вещь.

РАСЩЕПЛЕНИЕ ПЕРЕМЕННОЙ (SPLIT TEMPORARY VARIABLE)

```
double temp = 2 * (height + width);  
std::cout << temp << std::endl;  
temp = height * width;  
std::cout << temp << std::endl;
```



```
const double perimeter = 2 * (height + width);  
std::cout << perimeter << std::endl;  
const double area = height * width;  
std::cout << area << std::endl;
```

ПРИЧИНЫ РЕФАКТОРИНГА

Если вы «экономите» переменные внутри функции, повторно используете их для различных несвязанных целей, у вас обязательно начнутся проблемы в тот момент, когда потребуются внести какие-то изменения в код, содержащий эти переменные.

ТЕХНИКА

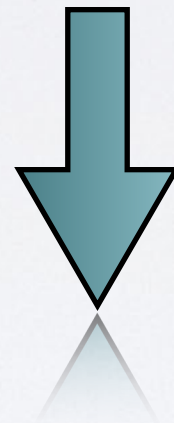
1. Найдите место в коде, где переменная в первый раз заполняется каким-то значением. В этом месте переименуйте ее, причем новое название должно соответствовать присваиваемому значению.
2. Подставьте её новое название вместо старого в тех местах, где использовалось это значение переменной.
3. Повторите операцию для случаев, где переменной присваивается новое значение.

ДОСТОИНСТВА

- Каждый элемент программы должен отвечать только за одну вещь.
- Улучшается читабельность кода.
- Помогает в дальнейшем выделить повторяющиеся участки кода в отдельные методы.

ЗАМЕНА ПЕРЕМЕННОЙ ВЫЗОВОМ МЕТОДА (REPLACE TEMP WITH QUERY)

Если помещаете результат какого-то выражения в локальную переменную, чтобы использовать её далее в коде.



Выделите все выражение в отдельный метод и возвращайте результат из него. Замените использование вашей переменной вызовом метода.

ЗАМЕНА ПЕРЕМЕННОЙ ВЫЗОВОМ МЕТОДА (REPLACE TEMP WITH QUERY)

```
double calculateTotal()  
{  
    double basePrice = quantity * itemPrice;  
    if(basePrice > 1000){  
        return basePrice * 0.95;  
    }  
    else{  
        return basePrice * 0.98;  
    }  
}
```



```
double basePrice()  
{  
    return quantity * itemPrice;  
}
```

```
double calculateTotal()  
{  
    if(basePrice() > 1000){  
        return basePrice() * 0.95;  
    }  
    else{  
        return basePrice() * 0.98;  
    }  
}
```

ПРИЧИНЫ РЕФАКТОРИНГА

Как подготовительным этап для применения выделения метода для какой-то части очень длинного метода.

Иногда можно найти это же выражение и в других методах, что заставляет задуматься о создании общего метода для его получения.

ТЕХНИКА

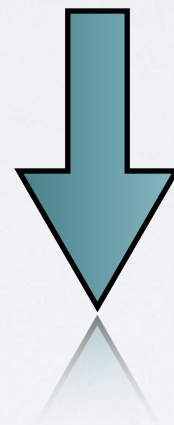
1. Убедитесь, что переменной в пределах метода присваивается значение только один раз. Если это не так, используйте расщепление переменной.
2. Добавьте к переменной модификатором `const`.
3. Используйте извлечение метода для того, чтобы переместить интересующее выражение в новый метод.
4. Убедитесь, что этот метод только возвращает значение и не меняет состояние объекта. Если он как-то влияет на видимое состояние объекта, используйте разделение запроса и модификатора.
5. Замените использование переменной вызовом вашего нового метода.

ДОСТОИНСТВА

- Улучшает читабельность кода.
- Помогает убрать дублирование кода, если заменяемая строка используется более чем в одном методе.

УДАЛЕНИЕ ПРИСВАИВАНИЙ ПАРАМЕТРАМ (REMOVE ASSIGNMENTS TO PARAMETERS)

Код выполняет присваивание параметру.




**Воспользуйтесь вместо этого локальной
переменной.**

УДАЛЕНИЕ ПРИСВАИВАНИЙ ПАРАМЕТРАМ (REMOVE ASSIGNMENTS TO PARAMETERS)

```
int discount(int inputVal, int quantity)
{
    if(inputVal > 50){
        inputVal -= 2;
    }
    //...
}
```

```
int discount(int inputVal, int quantity)
{
    int result = inputVal;

    if(result > 50){
        result -= 2;
    }
    //...
}
```



ПРИЧИНЫ РЕФАКТОРИНГА

Причины проведения этого рефакторинга такие же, как и при расщеплении переменной.

- Если параметр передаётся по ссылке, то можно случайно все сломать.
- Множественные присваивания усложняют понимание того, какие именно данные должны находиться в параметре в определенный момент времени.

ТЕХНИКА

1. Создайте локальную переменную и присвойте ей начальное значение рассматриваемого параметра.
2. Во всем коде метода после этой строки замените использование параметра новой локальной переменной.

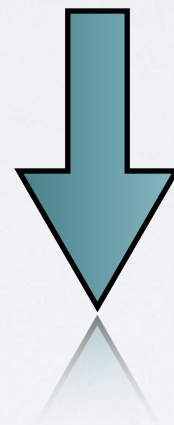
ДОСТОИНСТВА

- Каждый элемент программы должен отвечать только за одну вещь.
- Улучшается читабельность кода.
- Помогает в дальнейшем выделить повторяющиеся участки кода в отдельные методы.

УПРОЩЕНИЕ ВЫЗОВОВ МЕТОДОВ

ПЕРЕИМЕНОВАНИЕ МЕТОДА (RENAME METHOD)

Название метода не раскрывает суть того, что он делает.



Измените название метода.

ПЕРЕИМЕНОВАНИЕ МЕТОДА (RENAME METHOD)

```
std::string getsnm()  
{  
    //...  
}
```

```
std::string getSecondName()  
{  
    //...  
}
```



ПРИЧИНЫ РЕФАКТОРИНГА

- Неудачное название с самого начала.
- Удачное название, но ввиду расширения функциональности, имя метода стало не актуальным.

ТЕХНИКА

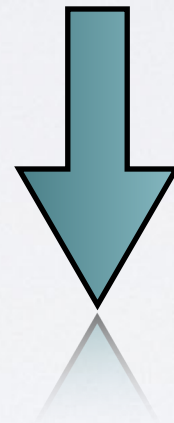
1. Проверьте, не определён ли метод в родительском классе или подклассе. Если так, нужно будет повторить все шаги и в этих классах.
2. Создайте новый метод с новыми именем. Скопируйте туда код старого метода. Удалите весь код в старом методе и вставьте вместо него вызов нового метода.
3. Найдите все обращения к старому методу и замените их обращениями к новому.
4. Удалите старый метод.

ДОСТОИНСТВА

Улучшает читабельность кода.

ДОБАВЛЕНИЕ ПАРАМЕТРА (ADD PARAMETER)

Методу не хватает данных для осуществления каких-то действий.



Создайте новый параметр (не поле класса), чтобы передать эти данные.

ДОБАВЛЕНИЕ ПАРАМЕТРА (ADD PARAMETER)

```
std::string Customer::getContact()  
{  
    //...  
}
```

```
std::string Customer::getContact(const Date& date)  
{  
    //...  
}
```



ПРИЧИНЫ РЕФАКТОРИНГА

Необходимо внести какие-то изменения в метод. Эти изменения требуют дополнительной информации или данных, которые ранее в метод не подавались.

ТЕХНИКА

1. Проверьте, не определён ли метод в родительском классе или подклассе. Если так, нужно будет повторить все шаги и в этих классах.
2. Создайте новый метод с новыми именем. Скопируйте туда код старого метода. Удалите весь код в старом методе и вставьте вместо него вызов нового метода.
3. Найдите все обращения к старому методу и замените их обращениями к новому.
4. Удалите старый метод.

ДОСТОИНСТВА

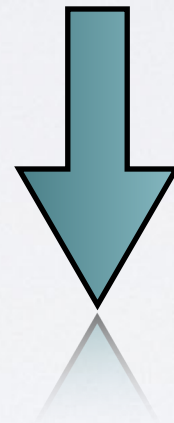
Параметр лучше добавлять тогда, когда требуются какие-то эпизодические или часто изменяющиеся данные, которые нет смысла держать в объекте все время.

НЕДОСТАТКИ

- Рост списка параметров.
- Возможно класс не содержит необходимых данных.

УДАЛЕНИЕ ПАРАМЕТРА (REMOVE PARAMETER)

Параметр не используется в теле метода.



Удалите неиспользуемый параметр.

УДАЛЕНИЕ ПАРАМЕТРА (REMOVE PARAMETER)

```
std::string Customer::getContact(const Date& date)
{
    //...
}
```

```
std::string Customer::getContact()
{
    //...
}
```



ПРИЧИНЫ РЕФАКТОРИНГА

Программисты часто добавляют параметры и неохотно их удаляют.

Тот, кто вызывает ваш метод, должен озаботиться передачей правильных значений.

ТЕХНИКА

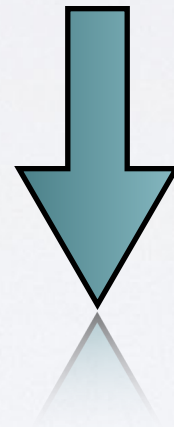
1. Проверьте, не определён ли метод в родительском классе или подклассе. Если так, используется ли там параметр? Если в какой-то из реализаций параметр используется, воздержитесь от рефакторинга.
2. Создайте новый метод с новыми именем. Скопируйте туда код старого метода. Удалите весь код в старом методе и вставьте вместо него вызов нового метода.
3. Найдите все обращения к старому методу и замените их обращениями к новому.
4. Удалите старый метод.

ДОСТОИНСТВА

Метод должен содержать только действительно необходимые параметры.

РАЗДЕЛЕНИЕ ЗАПРОСА И МОДИФИКАТОРА (SEPARATE QUERY FROM MODIFIER)

Есть метод, который возвращает какое-то значение, но при этом в процессе работы он изменяет что-то внутри объекта.



Разделите метод на два разных метода. Один из них возвращает значение, а второй модифицирует объект.

РАЗДЕЛЕНИЕ ЗАПРОСА И МОДИФИКАТОРА (SEPARATE QUERY FROM MODIFIER)

```
int Customer::getOutstandingAndSetReadyForSummaries()  
{  
    //...  
}
```

```
int Customer::getOutstanding()  
{  
    //...  
}  
  
void Customer::setReadyForSummaries()  
{  
    //...  
}
```



ПРИЧИНЫ РЕФАКТОРИНГА

Отделяйте код получения каких-то данных от кода, который изменяет что-то внутри объекта.

Код получения данных называют запросами, а код изменения видимого состояния объекта — модификаторами.

ТЕХНИКА

1. Создайте новый метод-запрос, который бы возвращал то, что возвращал оригинальный метод.
2. Сделайте так, чтобы оригинальный метод возвращал только результат вызова нового метода-запроса.
3. Замените все обращения к оригинальному методу вызовом метода-запроса. Непосредственно перед этой строкой нужно вставить вызов метода-модификатора.
4. Избавьтесь от кода возврата значения в оригинальном методе. После этого он станет правильным методом-модификатором.

ДОСТОИНСТВА

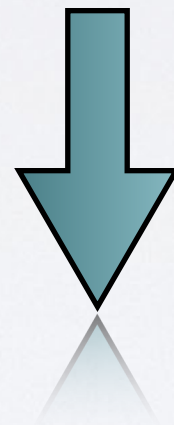
Если у вас есть запрос, который не меняет состояние программы, вы можете вызывать его сколько угодно раз, не опасаясь того, что результат изменится от самого факта вызова метода.

НЕДОСТАТКИ

В некоторых случаях, удобно возвращать какие-то данные после осуществления команды. Например, удаляя что-то из базы данных, вы хотите узнать, сколько при этом строк было удалено.

ПЕРЕДАЧА ВСЕГО ОБЪЕКТА (PRESERVE WHOLE OBJECT)

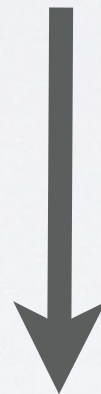
Вы получаете несколько значений из объекта, а затем передаёте их в метод как параметры.



Передавайте весь объект.

ПЕРЕДАЧА ВСЕГО ОБЪЕКТА (PRESERVE WHOLE OBJECT)

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



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

ПРИЧИНЫ РЕФАКТОРИНГА

Перед вызовом метода необходимо каждый раз вызывать методы получения параметров.

Любые изменения сигнатуры методов приводят к ряду правок в разных частях программы.

ТЕХНИКА

1. Создайте параметр в методе для объекта, из которого можно получить нужные значения.
2. Удаляйте по одному старые параметры из метода, заменяя их в коде вызовами соответствующих методов объекта-параметра. Тестируйте программу после каждой замены параметра.
3. Удалите код получения значений из объекта-параметра, который стоял перед вызовом метода.

ДОСТОИНСТВА

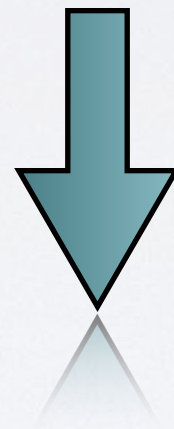
- Вместо пачки разнообразных параметров используется один объект с понятным названием.
- Если методу понадобятся ещё какие-то данные из объекта, не нужно будет переписывать все места, где вызывается этот метод, а только лишь внутренности самого метода.

НЕДОСТАТКИ

Можно потерять универсальность метода.

ЗАМЕНА ПАРАМЕТРОВ ОБЪЕКТОМ (INTRODUCE PARAMETER OBJECT)

В методах встречается повторяющаяся группа параметров.



Замените эти параметры объектом.

ЗАМЕНА ПАРАМЕТРОВ ОБЪЕКТОМ (INTRODUCE PARAMETER OBJECT)

```
int Customer::amountInvoicedIn(const Date& start, const Date& end);  
int Customer::amountReceivedIn(const Date& start, const Date& end);  
int Customer::amountOverdueIn(const Date& start, const Date& end);
```



```
int Customer::amountInvoicedIn(const DateRange& date);  
int Customer::amountReceivedIn(const DateRange& date);  
int Customer::amountOverdueIn(const DateRange& date);
```

ПРИЧИНЫ РЕФАКТОРИНГА

Одинаковые группы параметров зачастую встречаются не в единственном методе. Это приводит к дублированию кода, как самих параметров, так и частых операций над ними.

ТЕХНИКА

1. Создайте новый класс, который будет представлять вашу группу параметров. Сделайте так, чтобы данные объектов этого класса нельзя было изменить после создания.
2. В методе, к которому применяем рефакторинг, **добавьте новый параметр**, в котором будет передаваться ваш объект-параметр. Во всех вызовах метода передавайте в этот параметр объект, создаваемый из старых параметров метода.
3. Теперь начинайте по одному удалять старые параметры из метода, заменяя их в коде полями объекта-параметра. Тестируйте программу после каждой замены параметра.
4. По окончании оцените, есть ли возможность и смысл перенести какую-то часть метода (а иногда и весь метод) в класс объекта-параметра.

ДОСТОИНСТВА

- Улучшает читабельность кода.
- Позволяет избежать дублирование кода.

КОНЕЦ ПЕРВОЙ ЧАСТИ

Рефакторинг



ДО



ПОСЛЕ