**Запахи кода**

1. **«Стрельба дробью»**

**«Стрельба дробью»** похожа на расходящуюся модификацию, но является ее противоположностью. Учуять ее можно, когда при выполнении любых модификаций приходится вносить множество мелких изменений в большое число классов. Когда изменения разбросаны повсюду, их трудно находить и можно пропустить важное изменение.

В такой ситуации следует использовать **«Перемещение метода»** (*Создайте новый метод в классе, который использует его больше других, и перенесите туда код из старого метода. Код оригинального метода превратите в обращение к новому методу в другом классе либо уберите его вообще.)* и **«Перемещение поля»** (*Создайте поле в новом классе и перенаправьте к нему всех пользователей старого поля.*), чтобы свести все изменения в один класс. Если среди имеющихся классов подходящего кандидата нет, создайте новый класс. Часто можно воспользоваться **«Встраиванием класса»** (*Переместите все вещи из описанного класса в другой.* *Меньше бесполезных классов — больше свободной оперативной памяти.*), чтобы поместить целую связку методов в один класс. Возникнет какое-то число расходящихся модификаций, но с этим можно справиться.

1. **Завистливые функции**

Метод обращается к данным другого объекта чаще, чем к собственным данным. Этот запах может появиться после перемещения каких-то полей в класс данных. В этом случае операции с данными, возможно, также следует переместить в этот класс.

Следует придерживаться такого правила: то, что изменяется одновременно, нужно хранить в одном месте. Обычно данные и функции, использующие эти данные, также изменяются вместе (хотя бывают исключения).

* Если метод явно следует перенести в другое место, примените «**Перемещение метода**» (*Создайте новый метод в классе, который использует его больше других, и перенесите туда код из старого метода. Код оригинального метода превратите в обращение к новому методу в другом классе либо уберите его вообще.)*.
* Если только часть метода обращается к данным другого объекта, примените «**Извлечение метода**» (*Выделите участок кода в новый метод (или функцию) и вызовите этот метод вместо старого кода.*)к этой части.

1. **Одержимость элементарными типами**

Как и большинство других запахов, этот начинается с маленькой слабости. Программисту понадобилось поле для хранения каких-то данных. Он подумал, что создать поле элементарного типа куда проще, чем заводить новый класс. Это и было сделано. Потом понадобилось другое поле, и оно было добавлено схожим образом. Не успели оглянуться, как класс уже разросся до грандиозных размеров.

Примитивные типы нередко используются для «симуляции» типов. Это когда вместо отдельного типа данных вы имеете набор чисел или строк, который составляет список допустимых значений для какой-то сущности. Зачастую этим конкретным числам и строкам даются понятные имена с помощью констант, что и является причиной их широкого распространения.

Ещё одним плохим способом использования примитивных типов является «симуляция» полей. При этом класс содержит большой массив разнообразных данных, а в роли индексов массива для получения этих данных используются строковые константы, заданные в классе.

Решение:

* Если вы имеете множество разнообразных полей примитивных типов, возможно, некоторые из них можно логически сгруппировать и перенести в свой собственный класс. Ещё лучше, если в этот класс вы сможете перенести и поведения, связанные с этими данными. Справиться с этой проблемой поможет **замена значения данных объектом** (*Создайте новый класс, поместите в него старое поле и его поведения, храните объект этого класса в исходном классе*).
* Если значения этих примитивных полей используются в параметрах методов, используйте **замену параметров объектом** (*если в ваших методах встречается повторяющаяся группа параметров, то замените эти параметры объектом*).
* В случаях, когда в переменных закодированы какие-то сложные данные, используйте **замену кодирования типа классом** (*Создайте новый класс и применяйте его объекты вместо значений закодированного типа*), **замену кодирования типа подклассами** (*Для каждого значения закодированного типа, создайте подклассы. А затем, вынесите соответствующие поведения из исходного класса в эти подклассы. Управляющий код замените полиморфизмом*) или замену кодирования типа состоянием/стратегией (*Замените кодирование типа объектом-состоянием. При необходимости заменить значение поля с кодированием типа, в него подставляется другой объект-состояние.*).
* Если среди переменных есть массивы, используйте **замену массива объектом** (*Замените массив объектом, который будет иметь отдельные поля для каждого элемента*).

1. **Параллельные иерархии наследования**

Параллельные иерархии наследования в действительности являются особым случаем **«стрельбы дробью»**. В данном случае всякий раз при порождении подкласса одного из классов приходится создавать подкласс другого класса. Признаком этого служит совпадение префиксов имен классов в двух иерархиях классов.

Общая стратегия устранения дублирования состоит в том, чтобы заставить экземпляры одной иерархии ссылаться на экземпляры другой. С помощью **«Перемещения метода** (*Создайте новый метод в классе, который использует его больше других, и перенесите туда код из старого метода. Код оригинального метода превратите в обращение к новому методу в другом классе либо уберите его вообще.)* и **«Перемещение поля»** (*Создайте поле в новом классе и перенаправьте к нему всех пользователей старого поля.*) иерархия в ссылающемся классе исчезает.

1. **Ленивый класс**

На понимание и поддержку классов всегда требуются затраты времени и денег. А потому, если класс не делает достаточно много, чтобы уделять ему достаточно внимания, он должен быть уничтожен.

Это может произойти, если класс был задуман как полнофункциональный, но в результате рефакторинга ужался до неприличных размеров.

Либо класс создавался в расчёте на некие будущие разработки, до которых руки так и не дошли.

Решение:

* Почти бесполезные компоненты должны быть подвергнуты **встраиванию класса** (*Переместите все вещи из описанного класса в другой.* *Меньше бесполезных классов — больше свободной оперативной памяти.*).
* При наличии подклассов с недостаточными функциями попробуйте **свёртывание иерархии** (*Слейте подкласс и суперкласс воедино*).

1. **Временное поле**

Временные поля — это поля, которые нужны объекту только при определённых обстоятельствах. Только тогда они заполняются какими-то значениями, оставаясь пустыми в остальное время.

Зачастую временные поля создаются для использования в алгоритме, который требует большого числа входных данных. Так, вместо создания большого числа параметров в таком методе, программист решает создать для этих данных поля в классе. Эти поля используются только в данном алгоритме, а в остальное время простаивают.

Такой код очень трудно понять. Вы ожидаете увидеть данные в полях объекта, а они почему-то пустуют почти все время.

Решение:

* Временные поля и весь код, работающий с ними, можно поместить в свой собственный класс с помощью **извлечения класса** (*Создайте новый класс, переместите в него поля и методы, отвечающие за определённую функциональность*). По сути, вы создадите **объект-метод** (*Преобразуйте метод в отдельный класс так, чтобы локальные переменные стали полями этого класса. После этого можно без труда разделить метод на части*).
* **Введите Null-объект** и встройте его вместо кода проверок на наличие значений во временных полях.

1. **Посредник**

Если класс выполняет одно действие — делегирует работу другому классу — стоит задуматься, зачем он вообще существует.

Данный запах может быть результатом фанатичной борьбы с **цепочками вызовов** (*Вы видите в коде цепочки вызовов вроде такой $a->b()->c()->d()*).

Кроме того, бывает так, что вся полезная нагрузка класса постепенно перемещается в другие классы, в результате кроме делегирующих методов в нем ничего не остается.

Решение: если большую часть методов класс делегирует другому классу, нужно воспользоваться **удалением посредника** (*Класс имеет слишком много методов, которые просто делегируют работу другим объектам. Удалите эти методы и заставьте клиента вызывать конечные методы напрямую*).

1. **Неуместная близость**

Один класс использует служебные поля и методы другого класса.

Смотрите внимательно за классами, которые проводят слишком много времени вместе. Хорошие классы должны знать друг о друге как можно меньше. Такие классы легче поддерживать и повторно использовать.

Самый простой выход — при помощи [**перемещения метода**](https://refactoring.guru/ru/move-method) (*Создайте новый метод в классе, который использует его больше других, и перенесите туда код из старого метода. Код оригинального метода превратите в обращение к новому методу в другом классе либо уберите его вообще.)* и [**перемещения поля**](https://refactoring.guru/ru/move-field)(*Создайте поле в новом классе и перенаправьте к нему всех пользователей старого поля.*) перенести части одного класса в другой (в тот, где они используются). Однако это может сработать только в том случае, если оригинальный класс не использует перемещаемые поля и методы.

1. **Отказ от наследства**

Если подкласс использует лишь малую часть унаследованных методов и свойств суперкласса, это является признаком неправильной иерархии. При этом ненужные методы могут просто не использоваться либо быть переопределёнными и выбрасывать исключения.

Кто-то создал наследование между классами только из побуждений повторного использования кода, находящегося в суперклассе. При этом суперкласс и подкласс могут являться совершенно различными сущностями.

Решение:

* Если наследование не имеет смысла, и подкласс в действительности не является представителем суперкласса, следует избавиться от отношения наследования между этими классами, применив [**замену наследования делегированием**](https://refactoring.guru/ru/replace-inheritance-with-delegation) (*Создайте поле и поместите в него объект суперкласса, делегируйте выполнение методов объекту-суперклассу, уберите наследование*).
* Если наследование имеет смысл, нужно избавиться от лишних полей и методов в подклассе. Для этого необходимо извлечь из родительского класса все поля и методы, которые нужны подклассу, в новый суперкласс, и сделать оба класса его наследниками ([**извлечение суперкласса**](https://refactoring.guru/ru/extract-superclass) (*Создайте для них общий суперкласс и перенесите туда одинаковые поля и методы*)).

1. **Комментарии**

Метод содержит множество поясняющих комментариев.

Зачастую комментарии создаются с хорошими намерениями, когда автор и сам понимает, что его код недостаточно очевиден и понятен. В таких случаях комментарии играют роль «дезодоранта», т.е. пытаются заглушить «дурной запах» недостаточно проработанного кода.

Самый лучший комментарий — это хорошее название метода или класса.

Если вы чувствуете, что фрагмент кода будет непонятным без комментария, попробуйте изменить структуру кода так, чтобы любые комментарии стали излишними.

Решение:

* Если комментарий предназначен для того, чтобы объяснить сложное выражение, возможно, это выражение лучше разбить на понятные подвыражения с помощью [**извлечения переменной**](https://refactoring.guru/ru/extract-variable) (*Поместите результат выражения или его части в отдельные переменные, поясняющие суть выражения*).
* Если комментарий поясняет целый блок кода, возможно, этот блок можно извлечь в отдельный метод с помощью [**извлечения метода**](https://refactoring.guru/ru/extract-method) (*Выделите участок кода в новый метод (или функцию) и вызовите этот метод вместо старого кода.*). Название нового метода вам, скорей всего, подскажет сам комментарий.
* Если метод уже выделен, но для объяснения его действия по-прежнему нужен комментарий, дайте методу новое не требующее комментария название. Используйте для этого [**переименование метода**](https://refactoring.guru/ru/rename-method).
* Если требуется описать какие-то правила, касающиеся правильной работы метода, попробуйте рефакторинг [**введение утверждения**](https://refactoring.guru/ru/introduce-assertion) (*Замените эти предположения конкретными проверками*).